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

Dúnia Marchiori

# Root finding in code-based cryptography

Florianópolis
2022

Dúnia Marchiori

# Root finding in code-based cryptography

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.
Coorientador: Prof. Daniel Panario, Dr.

Florianópolis
2022

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

Marchiori, Dúnia
   Root finding in code-based cryptography / Dúnia
Marchiori ; orientador, Ricardo Felipe Custódio,
coorientador, Daniel Panario, 2022.
   71 p.

   Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico, Programa de Pós-Graduação em
Ciência da Computação, Florianópolis, 2022.

   Inclui referências.

   1. Ciência da Computação. 2. Criptografia pós-quântica. 3.
Criptografia baseada em códigos de correção de erros. 4.
Algoritmo para determinar raízes de polinômios. 5. Ataque de
efeito colateral. I. Custódio, Ricardo Felipe. II. Panario,
Daniel. III. Universidade Federal de Santa Catarina.
Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Dúnia Marchiori

**Root finding in code-based cryptography**

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

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.

GOV.BR
Documento assinado digitalmente
PATRICIA DELLA MEA PLENTZ
Data: 15/12/2022 13:51:51-0300
CPF: ***.243.700-**
Verifique as assinaturas em https://v.ufsc.br

Prof$^{a}$. Patricia Della Méa Plentz , Dr$^{a}$.
Coordenadora do Programa

ICP-Edu
Documento assinado digitalmente
Ricardo Felipe Custodio
Data: 15/12/2022 13:37:46-0300
CPF: ***.981.609-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Ricardo Felipe Custódio, Dr.
Orientador

Florianópolis, 2022.

À minha família

## ACKNOWLEDGEMENTS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Agradeço aos meus pais, Célio e Marlene, por todo o seu apoio, carinho e incentivo e por terem me ensinado a importância da educação. Agradeço aos meus amigos e amigas, especialmente aos meus colegas no Laboratório de Segurança em Computadores, pela companhia e pelas conversas, estando sempre presentes com palavras de encorajamento.

Agradeço aos professores Ricardo Custódio e Daniel Panario e à professora Lucia Moura, que me acompanharam e orientaram durante todo o progresso desta pesquisa. Seus ensinamentos e conselhos foram essenciais na minha formação e desenvolvimento ao longo do curso.

## RESUMO

O criptossistema de McEliece foi o primeiro criptossistema de chave pública baseado em código de correção de erros. O esquema se mantém resistente a ataques feitos nos últimos 40 anos e, por isso, é utilizado como base para criptossistemas propostos em processos de padronização em criptografia pós-quântica. Mas, apesar do criptossistema de McEliece ser considerado seguro contra ataques de adversários com computadores quânticos, ele ainda pode sofrer outros tipos de ataques. Um exemplo são ataques de efeito colateral de temporização, como um ataque que explora a variação no tempo de execução do processo de encontrar as raízes de um polinômio durante a decodificação. Portanto, é importante que os algoritmos de encontrar raízes utilizados em criptossistemas não apresentem diferenças no tempo de execução que possam ser exploradas, como diferenças influenciadas por valores que façam parte do segredo do criptossistema. Algoritmos probabilísticos de encontrar raízes de polinômios ainda não foram aplicados em criptossistemas baseados em códigos. Um obstáculo se deve ao seu tempo de execução não constante, já que variações no tempo de execução podem ser exploradas por técnicas de ataques de efeito colateral. Este trabalho tem o objetivo de criar uma abordagem de tempo constante para algoritmos probabilísticos para que criptossistemas possam se beneficiar de sua eficiência, principalmente no caso de parâmetros maiores do que os utilizados hoje. A contramedida proposta neste trabalho previne que o tempo de execução de um algoritmo probabilístico de encontrar raízes de polinômios seja dependente do grau do polinômio, tornando-o dependente apenas no valor dos parâmetros do criptossistema. O desempenho da nossa proposta é comparada com a de outros algoritmos de encontrar raízes de polinômios utilizados em criptossistemas baseados em códigos e em trabalhos correlatos. Além disso, também é avaliado que a contramedida proposta pode ser aplicada de forma eficaz em outros algoritmos de encontrar raízes. Em geral, nosso método é mais rápido que o algoritmo exaustivo utilizado no Classic McEliece, uma variante do Berlekamp Trace Algorithm, e o algoritmo additive Fast Fourier Transform para parâmetros plausíveis em corpos finitos grandes.

**Palavras-chave:** Criptografia pós-quântica. Criptografia baseada em códigos de correção de erros. Algoritmo para determinar raízes de polinômios. Algoritmos probabilísticos para determinar raízes de polinômios. Ataque de efeito colateral.

## RESUMO ESTENDIDO

### Introdução

Os sistemas de criptografia de chave pública mais populares atualmente são Rivest-Shamir-Adleman (RSA) e criptografia de curvas elípticas, os quais se baseiam em problemas que algoritmos quânticos conseguem resolver em tempo polinomial. Por este motivo, diferentes propostas de criptossistemas vêm sendo estudadas na área criptografia pós-quântica, para que as informações continuem seguras mesmo contra um adversário com um computador quântico. O esquema de McEliece é um esquema de criptografia de chave pública baseado em código de correção de erros proposto em 1978. A sua segurança se baseia no fato de que a chave pública é indistinguível de uma matriz aleatória e no problema de decodificação geral, o que leva o esquema a resistir a ataques de computadores quânticos e que o categoriza como um criptossistema pós-quântico. O esquema se mantém resistente a ataques feitos nos últimos 40 anos e, por isso, é utilizado como base para criptossistemas propostos em processos de padronização em criptografia pós-quântica. Mas, apesar do criptossistema de McEliece ser considerado seguro contra ataques de adversários com computadores quânticos, ele ainda pode sofrer outros tipos de ataques. Um exemplo são ataques de efeito colateral de temporização, como um ataque que explora a variação no tempo de execução do processo de encontrar as raízes de um polinômio durante a decodificação. Portanto, é importante que os algoritmos de encontrar raízes utilizados em criptossistemas não apresentem diferenças no tempo de execução que possam ser exploradas, como diferenças influenciadas por valores que façam parte do segredo do criptossistema. Algoritmos probabilísticos de encontrar raízes de polinômios ainda não foram aplicados em criptossistemas baseados em códigos. Um obstáculo se deve ao seu tempo de execução não constante, já que variações no tempo de execução podem ser exploradas por técnicas de ataques de efeito colateral. Com uma abordagem de tempo constante para algoritmos probabilísticos, criptossistemas baseados em códigos poderão se beneficiar de sua eficiência, principalmente no caso de parâmetros maiores do que os utilizados hoje. Este cenário de parâmetros maiores é plausível de acordo o desenvolvimento de computadores mais poderosos e o consequente aumento no poder computacional, sobretudo se considerarmos segurança a longo prazo.

### Objetivos

O principal objetivo deste trabalho é propor uma alternativa com um algoritmo probabilístico de encontrar raízes de polinômios no processo de decodificação em criptossistemas baseados em códigos de correção de erros. Para que esta alternativa seja adequada para uso em um criptossistema, a proposta não deve apresentar diferenças no tempo de execução que sejam influenciadas pelo valor de elementos que fazem parte do segredo do criptossistema, como o grau do polinômio.

### Metodologia

Para alcançar os objetivos deste trabalho, foi feita uma revisão sistemática da literatura, iniciada com uma pergunta de pesquisa que busca identificar quais algoritmos de encontrar raízes de polinômios são utilizados em criptossistemas baseados em códigos de correção de erros na literatura. Para tanto, uma revisão bibliográfica foi realizada, identificando quais trabalhos eram relevantes através dos critérios de aprovação e de reprovação previamente definidos. Após o levantamento dos principais trabalhos, foram comparados quais os principais algoritmos de encontrar raízes utilizados no contexto de criptossistemas.

**Resultados e Discussão**
A revisão bibliográfica realizada mostra que contramedidas para o uso de algoritmos probabilísticos de encontrar raízes de polinômios em criptossistemas baseados em códigos ainda não haviam sido propostas. A contramedida proposta neste trabalho previne que o tempo de execução de um algoritmo probabilístico de encontrar raízes de polinômios seja dependente do grau do polinômio, tornando-o dependente apenas do valor dos parâmetros públicos do criptossistema. Isto evita o ataque de temporização sobre criptossistemas baseados em códigos presente na literatura. A performance da nossa proposta é comparada com a de outros algoritmos de encontrar raízes de polinômios utilizados em criptossistemas baseados em códigos e em trabalhos correlatos. Além disso, também é avaliado que a contramedida proposta pode ser aplicada de forma eficaz em outros algoritmos de encontrar raízes. Em geral, nosso método é mais rápido que o algoritmo exaustivo utilizado no Classic McEliece, uma variante do Berlekamp Trace Algorithm, e o algoritmo additive Fast Fourier Transform para parâmetros plausíveis em corpos finitos grandes.

**Considerações Finais**
A contramedida proposta para um algoritmo probabilístico de encontrar raízes de polinômios como forma de prevenir ataques devido à variação do seu tempo de execução se mostrou realizável e eficiente. Além disso, foi possível aplicar a contramedida a outros algoritmos de encontrar raízes, como o BTA-it (MARTINS; BANEGAS; CUSTÓDIO, 2019). A nossa proposta de tempo constante para um algoritmo probabilístico, CTProb, exibiu um comportamento constante nos experimentos, com as suas variações ocorrendo apenas com a mudança de parâmetros do criptossistema, e não de valores que devem ser secretos, como o grau do polinômio de entrada. A implementação desta proposta pode ser realizada sem o uso de operações que podem causar vazamento de informação em ataques de temporização, resultando em um método *constant-time*. Com os resultados obtidos, observamos que o CTProb é mais eficiente que outros métodos de encontrar raízes atualmente utilizados em criptossistemas quando polinômios em corpos finitos grandes são considerados. Portanto, CTProb pode ser uma alternativa como método de encontrar raízes de polinômios quando segurança de longo prazo é considerada, por ser um método mais eficiente para parâmetros maiores que os utilizados atualmente.

**Palavras-chave:** Criptografia pós-quântica. Criptografia baseada em códigos de correção de erros. Algoritmo para determinar raízes de polinômios. Algoritmos probabilísticos para determinar raízes de polinômios. Ataque de efeito colateral.

## ABSTRACT

The McEliece cryptosystem was the first asymmetric cryptosystem based on error-correcting codes. It has resisted attacks for more than 40 years, which lead to it being the basis of some cryptosystem proposals in post-quantum cryptography standardization processes. Although the McEliece cryptosystem is believed to be secure against quantum attacks, it can still be subject to other types of attacks, such as side-channel attacks. One example is a timing side-channel attack that exploits the variations in execution time of the root-finding step during decoding. Thus, it is important that the root-finding algorithms used in a cryptosystem should not present differences in running time that could be exploited, that is, differences that can be tied to secret elements of the cryptosystem. Probabilistic algorithms for finding roots of polynomials have not been applied to code-based cryptography before. One obstacle is their non-constant execution time, since runtime variations can be exploited by side channel attack techniques. We aim to create a constant-time approach to probabilistic algorithms so cryptosystems can benefit from these algorithms' efficiency, especially when considering larger parameters. Our countermeasure prevents that the execution time of a probabilistic root-finding algorithm be dependent on the degree of the input polynomial and makes it dependent only on the cryptosystem parameters. We compare the performance of our proposal to other root-finding algorithms already used in code-based cryptography and related works. We also show that our countermeasure can be applied to other root-finding algorithms successfully. In general, our method is faster than a variant of the Berlekamp Trace Algorithm, the exhaustive algorithm in Classic McEliece and the additive Fast Fourier Transform algorithm for plausible parameters with larger finite fields.

**Keywords:** Post-quantum cryptography. Code-based cryptography. Root-finding algorithms. Probabilistic root finding. Timing side-channel attack.

# LIST OF FIGURES

# LIST OF TABLES

## LIST OF ALGORITHMS

# LIST OF SYMBOLS

| | |
|---|---|
| $\mathbb{F}_q$ | Finite field of order $q$ |
| $\mathbb{F}_q^n$ | $n$-dimensional vector space over the finite field $\mathbb{F}_q$ |
| [n,k] code | A linear code of length $n$ and dimension $k$ |
| [n,k,d] code | A linear code of length $n$, dimension $k$, and minimum distance $d$ |
| $C^{\perp}$ | Dual code of code $C$ |
| $\langle v, u \rangle$ | Inner product of vectors $v$ and $u$ |
| $c^T$ | Transpose of a vector $c$ |
| $\overrightarrow{0}$ | Vector of zeros |
| $dist(x,y)$ | Hamming distance between vectors $x$ and $y$ |
| $w(x)$ | Hamming weight of vector $x$ |
| $\Gamma(L,g)$ | Goppa code with support set $L$ and Goppa polynomial $g$ |
| $\oplus$ | Exclusive or |
| $P^{-1}$ | Inverse of matrix $P$ |
| $\mathbb{F}_q[x]$ | Ring of univariate polynomials with coefficients over the finite field of order $q$ |

**CONTENTS**

# 1 INTRODUCTION

One of the main focuses of computer security is to provide data confidentiality, ensuring that secret and private data cannot be retrieved by unauthorized accesses. Cryptosystems are suites of cryptographic algorithms used to implement security services, such as confidentiality, ensuring that an attacker cannot retrieve the original message of any ciphertext intercepted without the key necessary for decryption.

Cryptosystems are usually divided in two categories: symmetric and asymmetric. Symmetric cryptosystems denote cryptosystems which both parties use the same key for encrypting and decrypting messages. Therefore, the key is a shared secret only between the parties that are communicating. Symmetric cryptosystems provide secure communication over an insecure channel, however, a secure channel is necessary to perform the key agreement process.

Asymmetric cryptosystems, also known as public-key cryptosystems, denote cryptosystems that use a pair of keys, a public key and a private key. The pair of keys is used to perform complementary operations, such as encrypting and decrypting messages, or signing documents and verifying these signatures. The private key must be known only by its owner, but the public key can be shared, which removes the need of a secure channel for key agreement. Therefore, public-key cryptography provides secret and authenticated communication between two parties over an insecure channel.

Examples of public-key algorithms include RSA (RIVEST; SHAMIR; ADLEMAN, 1978) and ECDSA (VANSTONE, 1992). These algorithms are commonly used and their security relies on the difficulty of calculating the factorization of integers and computing discrete logarithms in polynomial time, respectively. However, with Shor's algorithm (SHOR, 1994) it is possible to factor integers and compute discrete logarithms in polynomial time in quantum computers. Consequently, current public-key encrypted information may not be safe with the evolution of quantum computers.

Post-quantum cryptography is the field where researchers aim to develop and improve cryptographic schemes whose security is based on problems that are not compromised by quantum computation. The proposals in post-quantum cryptography include cryptosystems based on hash functions, lattices, error-correcting codes, and multivariate equations (BERNSTEIN; BUCHMANN; DAHMEN, 2009). In this work we focus on a cryptosystem based on error-correcting codes, also known as code-based. Errors can occur during data transmission, which can modify the original message resulting in the receiver reading a different message than the one that was sent. Error-correcting codes can be used to detect errors in the message and correct them. This is possible due to redundant information added to the message, which aids its correction after it is sent over a noisy communication channel. In code-based cryptosystems, the errors are intentionally added to the message in order to prevent the access to the data by unwanted parties.

McEliece's scheme (MCELIECE, 1978) was the first asymmetric cryptosystem based on error-correcting codes and has resisted attacks for more than 40 years. Originally, the error-correcting code used in McEliece's proposal was Goppa codes (GOPPA, 1970). The security of McEliece's cryptosystem is based on the hardness of generic decoding, an NP-hard problem also believed to be hard on average, which so far has resisted attacks by quantum algorithms. This fact lead to McEliece's scheme being the basis of some cryptosystem proposals in post-quantum cryptography standardization processes, like the one lead by the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography Standardization (National Institute of Standards and Technology, 2022), with the Classic McEliece (ALBRECHT et al., 2020) proposal being a Round 4 candidate.

Although the McEliece cryptosystem is believed to be secure against quantum attacks, it can still be subjected to other types of attacks, such as side-channel attacks. Side-channel attacks use additional information besides the original message and the ciphertext during encryption or decryption, and must be avoided. Examples of extra information include power consumption statistics, electromagnetic radiation, sound information and the execution time of the operations. Side-channel attacks that use time information are called timing side-channel attacks. In code-based cryptography, some timing side-channel attacks on the decryption process focus on retrieving information from the error locator polynomial, especially in the process of finding its roots, which indicate the errors added to the message. An attack based on this idea is presented by Strenzke et al. and Shoufan et al. (STRENZKE et al., 2008; SHOUFAN et al., 2010) and detailed in Section 2.3.3. The attack exploits the variations in execution time of the root-finding step. Thus, it is important that the root-finding algorithms used in a cryptosystem should not present differences in running time that can be tied to secret elements. To avoid timing side-channel attacks, this paper focuses on a *constant-time implementation*, defined to be an implementation where time variations are not correlated to secret information (ALMEIDA et al., 2016).

Post-quantum cryptosystems may remove the threat over long-term security that Shor's algorithm established, but in the future, the parameters proposed now may not be enough considering the growth in research and development of more powerful computers. Consequently, wanting to keep information secure long-term usually involves larger parameters. This may be the case for post-quantum cryptosystems proposed now. Considering larger parameters, the cost of the current algorithms must be evaluated in this scenario. The most costly step during the decryption process of McEliece's cryptosystem is the root-finding step (BISWAS; HERBERT, 2009). For bigger field sizes and polynomial degrees, probabilistic root-finding algorithms would be a more reasonable choice, instead of the deterministic algorithms used currently.

Deterministic algorithms are the most common algorithms used in the root-finding process in code-based cryptography; they include the exhaustive search and the additive Fast Fourier Transform (GAO; MATEER, 2010). Both have constant-time behavior for any input polynomial for a fixed field, which indicates that any variance in the execution time of these algorithms is related only to the cryptosystem parameters, regardless of the input degree. However, when considering the evolution of quantum computers and long-term security in code-based cryptosystems, we must consider an increase in field size and number of errors. Probabilistic algorithms are more efficient than deterministic algorithms for large fields. The main drawback in this context is that the execution time of probabilistic algorithms is heavily influenced by the degree of the input polynomial, which could be easily exploited by a side-channel attack.

To the best of our knowledge, there are no proposals that focus on countermeasures for probabilistic root-finding algorithms to be applied in the context of code-based cryptography considering timing side-channel attacks with the exception of our work (MARCHIORI et al., 2021). It is the first to propose and experiment with a constant-time approach to a probabilistic algorithm for code-based cryptosystems resistant to the aforementioned side-channel attacks.

In Martins et al. (MARTINS; BANEGAS; CUSTÓDIO, 2019), some countermeasures are proposed to avoid timing side-channel attacks over the root-finding algorithm in the decoding step in code-based cryptosystems. It includes proposals for four root-finding methods: exhaustive search, linearized polynomials, the Successive Resultant Algorithm, and the Berlekamp Trace Algorithm (BTA). However, countermeasures for probabilistic algorithms are not analyzed.

The proposal for BTA by Martins et al. (MARTINS; BANEGAS; CUSTÓDIO, 2019), BTA-it, results in a less variable performance when comparing the number of CPU cycles of multiple executions of the algorithm to find the roots of polynomials of a certain degree $t$. It does not address the variation in the number of CPU cycles when receiving an input of degree $t$ versus receiving an input of degree $d < t$, as we do in this work. Our proposal focuses on a constant-time behavior when comparing the number of CPU cycles it takes to find the roots of polynomials of different degrees $d \leq t$ when expecting $t$ errors in the McEliece cryptosystem.

We observe that in code-based cryptosystems, probabilistic algorithms have been considered in two studies (SCHIPANI, 2012; SUMI; MOROZOV; TAKAGI, 2011). However, both studies do not mention timing side-channel attacks and, therefore, do not discuss how the algorithm can leak information in a code-based cryptosystem context.

The main contribution of this work is an efficient constant-time approach to a probabilistic algorithm (CTProb) for finding the roots of a polynomial. Another contribution is a general method for artificially increasing the degree of the input polynomial to a target degree expected by the cryptosystem as a countermeasure to avoid timing side-channel attacks that exploit information on the polynomial degree. The method increases the degree of the input polynomial by adding fake roots to it and then removing the fake roots of the output of the root-finding method, and also making the individual steps of the algorithm constant time.

## 1.1 OBJECTIVES

The main goal of this work is to propose a probabilistic alternative to root-finding in the decoding process of code-based cryptosystems. To be considered suitable for a code-based cryptosystem, the proposal must not present differences in running time that can be tied to secret elements, such as the error locator polynomial degree.

### 1.1.1 Specific objectives

To achieve our goal, we must complete the following specific objectives:

(i) Identify the root-finding algorithms currently used in code-based cryptography;

(ii) Select a probabilistic root-finding algorithm;

(iii) Propose a strategy to obtain a constant-time execution of a probabilistic root-finding algorithm;

(iv) Evaluate if the strategy results in a constant-time execution;

(v) Evaluate if the strategy is useful for bigger parameters in code-based cryptography, especially compared to other root-finding algorithms currently used.

## 1.2 METHODOLOGICAL APPROACH

In this section, we describe the steps of our literature review. The objects of study are the algorithms used in the root-finding step during the decoding process of linear codes in cryptosystems in order to identify what algorithms have been used and analyzed. We define the research questions, construct the search query, identify the criteria used to evaluate the studies and then analyze the selected papers. We encourage readers that are not familiar with linear codes and finite fields to read Chapter 2 and then return to this section.

### 1.2.1 Literature review

The purpose of this literature review is to identify the algorithms applied to finding roots in the error locator polynomial during the decoding process of a code-based cryptography scheme. In order to determine the algorithms and understand in which conditions they are used, the research questions are the following:

**RQ1:** Which root-finding algorithm is used in the decoding process?
**RQ2:** What are the field size and polynomial degree considered in the experiments?
**RQ3:** Does the study mention time side-channel attacks prevention?

#### *1.2.1.1 Search query and databases*

To define the search query, the major terms were derived from the population, intervention and outcome (KITCHENHAM; CHARTERS, 2007). Table 1 shows the queries for four different databases and the number of results obtained from each query.

| Search query | Characteristics | Database | Results |
|---|---|---|---|
| ("code-based" OR "mcelice") AND ("root-finding" OR "error locator polynomial") | Searches in abstracts, titles and indexing terms. Year range: 1884 - 2022 | IEEE | 1 |
| ("code-based" OR "mcelice") AND ("root-finding" OR "error locator polynomial") | Searches all content. Year range: 1986 - 2022 | Springer | 166 |
| ("code-based" OR "mcelice") AND ("root-finding" OR "error locator polynomial") | Searches all content. Year range: 1985 - 2022 | ACM | 12 |
| ("code-based" OR "mcelice") AND ("root-finding" OR "error locator polynomial") | Searches all content. Excludes patents and citations. Year range: - 2022 | Scholar | 603 |

Table 1 – Search queries and its characteristics for different databases and the number of results for each search.

#### *1.2.1.2 Inclusion and exclusion criteria*

In order to decide whether to include a paper in the study, the following inclusion criteria were adopted:

(i) The study focuses in a McEliece/Niederreiter variant;

(ii) The study discusses the root-finding step during decoding;

(iii) The study includes parameters for the context being discussed.

The exclusion criteria were the following:

(i) The study does not present a decoding algorithm that contains a root-finding step;

(ii) The study is related to hardware implementation;

(iii) The study is a presentation.

#### *1.2.1.3 Selection process*

The search queries were executed in May 2022. The search in the IEEE database returned 1 paper, ACM returned 12 papers, Springer returned 166 papers and Scholar returned 603 studies, that resulted in 782 studies in total. After that, duplicates were removed and the criteria were applied to the title and abstract of each study.

The next step was reading the full text and, again, including or excluding studies considering the criteria defined previously. This resulted in 12 studies being selected. Three NIST Post-Quantum Cryptography Standardization code-based proposals were not returned by the

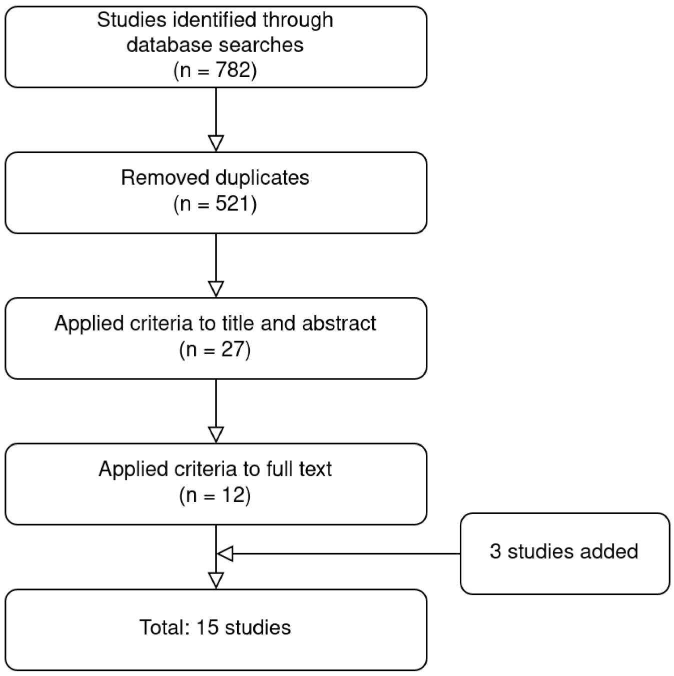
Studies identified through database searches (n = 782)
Removed duplicates (n = 521)
Applied criteria to title and abstract (n = 27)
Applied criteria to full text (n = 12)
3 studies added
Total: 15 studies

Figure 1 – Literature review execution steps.

searches, so these works were added in the final step. Therefore, at the end of the execution process, 15 studies were selected. Figure 1 illustrates the steps executed and how many studies resulted from each step.

#### *1.2.1.4 Analysis of the results*

The selected papers are now analyzed considering our three research questions. Most of the selected works use a deterministic algorithm in the root-finding step of the decoding process. In (WALTERS; ROY, 2020), (WANG, 2017), (CAYREL; HOFFMANN; PERSICHETTI, 2012) and (ALBRECHT et al., 2020), exhaustive techniques like Chien Search (CHIEN, 1964) and Horner's method for polynomial evaluation are used. In (WALTERS; ROY, 2020), the parameters tested were field size $q = 2^9$ and error correction capability of $t = 20$ errors and the implementation is designed to execute in constant time. In (WANG, 2017) the parameters tested were field size $q = 2^{10}$ and polynomials of degree $t = 280$ at maximum and does not consider timing side-channel attacks. In (CAYREL; HOFFMANN; PERSICHETTI, 2012), the field size is $q = 2^5$ and the degree of the polynomials is $t = 144$ at maximum and does not consider timing side-channel attacks either. Classic McEliece (ALBRECHT et al., 2020), a NIST post-quantum proposal, has parameters sets with field size $q = 2^{13}$ and number of errors $t = 128$ and its design had constant time execution as a priority.

The multipoint polynomial evaluation algorithm by Gao and Mateer (GAO; MATEER, 2010) is used in (WAFO-TAPA et al., 2019), (BERNSTEIN; CHOU; SCHWABE, 2013; CHOU, 2017), (MELCHOR et al., 2021) and (ALBRECHT et al., 2019). Hamming Quasi-Cyclic (HQC) (MELCHOR et al., 2021) is a NIST post-quantum proposal that uses field size $q = 2^8$

and can correct $t = 149$ errors. In (WAFO-TAPA et al., 2019), the authors identify a timing attack against HQC and define countermeasures. The countermeasures include modifications towards constant time roots computation. The authors of McBits (BERNSTEIN; CHOU; SCHWABE, 2013) propose a McEliece variant and its parameters include field size $q = 2^{13}$ and polynomials of degree $t = 119$. In (CHOU, 2017), a constant-time implementation for the scheme is given. NTS-KEM (ALBRECHT et al., 2019) is another NIST post-quantum proposal, now merged with Classic McEliece. Before being merged, its parameters sets included field size $q = 2^{13}$ and number of errors $t = 136$ and had constant-time execution.

Other deterministic algorithms appear in the selected works. In (BISWAS, 2010), a variant of the Berlekamp Trace Algorithm (BERLEKAMP, 1970) with the Zinoviev procedures (ZINOVIEV, 1996) is described. Its goal is to speed-up the Berlekamp Trace Algorithm, but does not take a constant-time execution into consideration. The parameters tested are field size $q = 2^{11}$ and polynomials of degree $t = 32$. In (STRENZKE, 2012) and (MARTINS; BANEGAS; CUSTÓDIO, 2019), exhaustive search, the Berlekamp Trace Algorithm and linearized polynomials (FEDORENKO; TRIFONOV, 2002) are analyzed and compared. In (MARTINS; BANEGAS; CUSTÓDIO, 2019) some countermeasures are proposed to avoid timing side-channel attacks over the root-finding algorithm in the decoding step. The countermeasures result in a more constant performance when comparing the number of CPU cycles of multiple executions of the algorithm to find the roots of polynomials of a certain degree $t$. It does not address the variation in the number of CPU cycles when expecting an input of degree $t$ and receiving an input of degree $d < t$, as we do in our work. The parameters tested in (STRENZKE, 2012) are field size $q = 2^9$ and polynomials of degree $t = 33$ and the parameters analyzed in (MARTINS; BANEGAS; CUSTÓDIO, 2019) are field size $q = 2^{16}$ and polynomials of degree $t = 100$.

Finding the roots of the error locator polynomial with a probabilistic algorithm is discussed in (MARCHIORI et al., 2021), (SCHIPANI, 2012) and (SUMI; MOROZOV; TAKAGI, 2011). In our work, (MARCHIORI et al., 2021), the first constant-time probabilistic root-finding algorithm is proposed. The other studies use the Cantor–Zassenhaus (CANTOR; ZASSENHAUS, 1981) algorithm, but do not mention timing side-channel attacks in their texts and, therefore, do not discuss the highly nonconstant execution time of the algorithm in a code-bases cryptosystem context. In (SCHIPANI, 2012), the modification in the algorithm made by the author has the goal of obtaining deterministic estimates on the maximum number of attempts needed to split a polynomial of a certain degree. The parameters analyzed in these works are field size $q = 2^{12}$ and number of errors $t = 100$ at maximum.

*Hypothesis.* From the analysis above, no works in the literature attempt to design a constant-time approach to a probabilistic algorithm, other than our own work. Therefore, our hypothesis is answered positively if a constant-time probabilistic root-finding algorithm is designed, which could be used in a code-based cryptosystem.

## 1.3 SCOPE OF THIS WORK

This work focuses on designing a countermeasure for probabilistic root-finding algorithms in order to avoid variations in the execution time related to any secret information of the McEliece cryptosystem. Although our work focuses on timing side-channel attacks, we avoided simple solutions that could result in a proposal susceptible to different attacks, such as power consumption attacks. However, only the timing aspect of our solution is analyzed in this work.

## 1.4 CONTRIBUTIONS AND OUTLINE OF THIS WORK

In Chapter 2, we start by introducing linear codes, focusing on Goppa codes, and the McEliece cryptosystem. Then, we proceed to present the timing side-channel attack that our proposal aims to prevent. The root-finding algorithm which our work is based upon and other root-finding algorithms used in code-based cryptography are described in Chapter 3. We propose a constant-time approach to a probabilistic root-finding algorithm in Chapter 4. The contents of this chapter were published as a peer-reviewed paper in a conference proceeding (MARCHIORI et al., 2021). We experimentally determine field sizes and target degrees for which our proposed constant-time approach to a probabilistic root-finding algorithm becomes more efficient for finding roots than other deterministic algorithms, such as an exhaustive algorithm and the additive Fast Fourier Transform algorithm, in Chapter 5. Finally, in Chapter 6, we summarize and finish our discussion with pointers to open research topics.

# 2 THEORETICAL BACKGROUND

In this chapter, we introduce the main concepts about error correcting codes and public-key cryptography in order to achieve a better understanding of this work. We describe algebraic structures, followed by an overview of error correcting codes, focusing on Goppa codes. In addition, we present the McEliece cryptosystem and describe a timing side-channel attack over the McEliece scheme.

## 2.1 ALGEBRAIC STRUCTURES

Some algebraic concepts are important to better understand the theoretical background of this work. Since our work is about root-finding algorithms in the context of error correcting codes, the main concepts are finite fields and polynomials. The elements of the linear codes described in the following sections are elements in finite fields, called the alphabet of the code.

**Definition 2.1.** A finite field, denoted $\mathbb{F}_q$, is a structure formed by $q$ elements and two operations, usually called addition and multiplication, which satisfy the axioms of associativity, closure, identity element, additive inverse element, multiplicative inverse for non-zero elements, commutativity, and distributive multiplication.

A field is a set in which we can do addition, subtraction, multiplication, and division without leaving the set (STALLINGS, 2014). Examples of fields are the rational numbers, and the real numbers.

The number of elements $q = p^m$ in a finite field $\mathbb{F}_q$ must be a power of a prime number $p$, with $m$ a positive integer (NIEDERREITER; XING, 2009). The prime number $p$ is the characteristic of the field. Binary finite fields are fields with characteristic 2.

As mentioned above, linear codes are defined over finite fields, more precisely the vector space $\mathbb{F}_q^n$.

**Definition 2.2.** A vector space $\mathbb{F}_q^n$ is an $n$-dimensional vector space over the finite field $\mathbb{F}_q$. That is, all $n$-tuples over $\mathbb{F}_q$.

In the context of this work, polynomials are also defined over finite fields.

**Definition 2.3.** A polynomial $f(x) = \sum_{i=0}^{t} g_i x^i$ is a polynomial of degree $t$. For polynomials over finite fields, $g_i \in \mathbb{F}_q$, $0 \leq i \leq t$.

**Definition 2.4.** The roots of a polynomial $f$ are elements $x_i$ such that $f(x_i) = 0$.

Some polynomials can be defined as the product of other polynomials, called factors. The error locator polynomial described in Section 2.2.2.4 is the product of polynomials of degree 1 called linear factors. Polynomials that are not formed by smaller factors are called irreducible polynomials.

**Definition 2.5.** An irreducible polynomial $f$ is a polynomial that cannot be factored into the product of two or more polynomials of positive degree strictly smaller that the degree of $f$.

**Definition 2.6.** A linear factor is a irreducible polynomial in the form $f(x) = ax + b$.

## 2.2 ERROR CORRECTING CODES

In this section, we present an overview about error correcting codes. First, we describe the characteristics of linear codes which are an important class of error correcting codes. Then, we detail Goppa codes, an error correcting code used in code-based cryptography, more specifically in the McEliece cryptosystem (detailed in Section 2.3.2) that is the focus of our work.

### 2.2.1 Linear codes

Error correcting codes detect and correct errors in messages. A linear code $C$ over $\mathbb{F}_q^n$ is defined by its dimension $k$ and length $n$, also denoted as an [n,k] code $C$. Thus, a code has $q^k$ elements, called codewords.

**Definition 2.7.** A linear code [n,k] is a subspace of dimension $k$ of the vector space $\mathbb{F}_q^n$.

Linear codes can be represented by a generator matrix or a parity check matrix (HUFFMAN; PLESS, 2003).

**Definition 2.8.** A generator matrix $G$ of an [n,k] code $C$ is a $k \times n$ matrix whose rows form a basis for $C$.

**Definition 2.9.** An $(n-k) \times n$ parity check matrix $H$ is the generator matrix of the dual code $C^{\perp}$. The dual code $C^{\perp}$ is a set of vectors $v$ such that $\{v \in \mathbb{F}_q^n \mid \langle v,u \rangle = 0, \forall u \in C\}$. That is, all vectors $v \in \mathbb{F}_q^n$ that the result of the inner product with every $u \in C$ equals 0.

The definition of a parity check matrix implies that a word $c$ is a codeword of a code $C$ if $Hc^T = \overrightarrow{0}$. Also, the vector $S_c$ resulting from $Hc^T = S_c$ is called the syndrome of the word $c$. In general, a code can have many different generator and parity check matrices.

In coding theory, there are two important metrics that help define the maximum number of errors that an error correcting code can correct. These are the Hamming distance and weight.

**Definition 2.10.** The Hamming distance $dist(x,y)$ between two vectors $x,y \in \mathbb{F}_q^n$ is the number of elements where they differ.

**Definition 2.11.** The Hamming weight $w(x)$ of a vector $x \in \mathbb{F}_q^n$ is the number of nonzero elements in $x$, that is, $w(x) = dist(x, \overrightarrow{0})$.

These metrics are used to define the minimum distance $d$ of a code $C$, or the distance of a code, that indicates the minimum distance between every two distinct codeword of $C$. Therefore, any two codewords differ in at least $d$ places. A linear code of length $n$, dimension $k$, and minimum distance $d$ is called an [n,k,d] code. The minimum distance impacts on the maximum number of errors a code can correct. A code $C$ with a minimum distance $d$ can correct up to $t = \left\lfloor \frac{(d-1)}{2} \right\rfloor$ errors (MACWILLIAMS; SLOANE, 1981, pp. 10).

### 2.2.2 Goppa Codes

Goppa codes (GOPPA, 1970) are linear codes with parameters consisting of length $n$, dimension $k$ and minimum distance $d$. A Goppa code is defined by a Goppa polynomial $g$ and a support set $L$. The polynomial $g$ is of degree $t$ and coefficients in $\mathbb{F}_{p^m}$, for $m$ a positive integer, that is, $g(x) = \sum_{j=0}^{t} g_j x^j$, where $g_j \in \mathbb{F}_{p^m}$. The set $L$ consists of $n$ elements $\alpha_i \in \mathbb{F}_{p^m}$ such that $g(\alpha_i) \neq 0$, $1 \leq i \leq n$.

A Goppa code $\Gamma(L,g)$ is formed by all vectors $c = (c_1, \ldots, c_n) \in \mathbb{F}_q^n$ that satisfy the condition

$$S_c(x) = \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \pmod{g(x)}, \tag{2.1}$$

where $x - \alpha_i$ is invertible modulo $g(x)$.

The length $n$ of the code $\Gamma(L,g)$ is the length of the codewords that is defined by the cardinality of $L$. The dimension $k$ satisfies $k \geq n - mt$ and the minimum distance $d$ satisfies $d \geq t+1$ (GOPPA, 1970). When condition (2.1) is not satisfied, the received message $c'$ contains errors, that is, $c' = c + e$, $c \in \mathbb{F}_q^n$ being a codeword and $e \in \mathbb{F}_q^n$ being an error vector with weight $w(e) \leq t$.

#### 2.2.2.1 *Parity check matrix*

The parity check matrix $H$ of a Goppa code $\Gamma(L,g)$ is used in the decoding step. A word $c = (c_1, \ldots, c_n)$ is a codeword of the code if $Hc^T = \overrightarrow{0}$. Let

$$p_i(x) \equiv \frac{1}{x - \alpha_i} \equiv -\frac{g(x) - g(\alpha_i)}{x - \alpha_i} \cdot g(\alpha_i)^{-1} (mod\ g(x)). \tag{2.2}$$

From the definition of Goppa codes we have that

$$\sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0\ (mod\ g(x)). \tag{2.3}$$

Thus, we can rewrite Equation 2.3 as

$$\sum_{i=1}^{n} c_i p_i(x) \equiv 0\ (mod\ g(x)). \tag{2.4}$$

Considering $p_i(x) = p_{i1} + p_{i2}x + \cdots + p_{it}x^{t-1}$, we have

$$\sum_{i=1}^{n} c_i p_{ij} = 0, 1 \leq j \leq t. \tag{2.5}$$

Since $g(x) = g_0 + g_1 x + \cdots + g_t x^t$ and considering $h_i = g(\alpha_i)^{-1}$, we have

$$\begin{aligned} p_i(x) &= -\frac{g_1 \cdot (x - \alpha_i) + \cdots + g_t \cdot (x^t - \alpha_i^t)}{x - \alpha_i} \cdot h_i \\ &= -(\frac{g_1 \cdot (x - \alpha_i)}{x - \alpha_i} + \cdots + \frac{g_t \cdot (x^t - \alpha_i^t)}{x - \alpha_i}) \cdot h_i \\ &= -(g_1 + g_2 \cdot (x + \alpha_i) + \ldots \\ &\quad + g_t \cdot (x^{t-1} + x^{t-2}\alpha_i + \cdots + \alpha_i^{t-1})) \cdot h_i. \end{aligned} \tag{2.6}$$

Substituting $p_i(x) = p_{i1} + p_{i2}x + p_{i3}x^2 + \cdots + p_{it}x^{t-1}$, we have the following equations for $p_{ij}$ (JOCHEMSZ, 2002)

$$\begin{cases} p_{i1} = -(g_t \alpha_i^{t-1} + g_{t-1}\alpha_i^{t-2} + \cdots + g_2 \alpha_i + g_1) \cdot h_i; \\ p_{i2} = -(g_t \alpha_i^{t-2} + g_{t-1}\alpha_i^{t-3} + \cdots + g_2) \cdot h_i; \\ \quad \vdots \\ p_{i(t-1)} = -(g_t \alpha_i + g_{t-1}) \cdot h_i; \\ p_{it} = -g_t \cdot h_i. \end{cases} \tag{2.7}$$

With the equations in (2.7), the parity check matrix $H$ of a Goppa code can be constructed such as

$$H = \begin{pmatrix} -g_t \cdot h_1 & -g_t \cdot h_2 & \ldots & -g_t \cdot h_n \\ -(g_t \alpha_1 + g_{t-1}) \cdot h_1 & -(g_t \alpha_2 + g_{t-1}) \cdot h_2 & \ldots & -(g_t \alpha_n + g_{t-1}) \cdot h_n \\ \vdots & \vdots & \vdots & \vdots \\ -\sum_{j=1}^{t} g_j \alpha_1^{j-1} \cdot h_1 & -\sum_{j=1}^{t} g_j \alpha_2^{j-1} \cdot h_2 & \ldots & -\sum_{j=1}^{t} g_j \alpha_n^{j-1} \cdot h_n \end{pmatrix}, \tag{2.8}$$

that can be row reduced to $H'$ (HUFFMAN; PLESS, 2003), where

$$H' = \begin{pmatrix} h_1 & h_2 & \ldots & h_n \\ h_1 \cdot \alpha_1 & h_2 \cdot \alpha_2 & \ldots & h_n \cdot \alpha_n \\ \vdots & \vdots & \vdots & \vdots \\ h_1 \cdot \alpha_1^{t-1} & h_2 \cdot \alpha_2^{t-1} & \ldots & h_n \cdot \alpha_n^{t-1} \end{pmatrix}. \tag{2.9}$$

#### *2.2.2.2 Generator matrix*

The generator matrix $G$ is used in the encoding step. A message $m$ is encoded like $c = mG$, resulting in a codeword $c$. Given that $Hc^T = \overrightarrow{0}$, $\forall c \in \Gamma(L, g)$, the generator matrix $G$ can be obtained from the parity check matrix $H$ like $GH^T = 0$.

#### 2.2.2.3 *Encoding*

To encode a message $m = (m_0, \dots, m_i)$, $i$ a positive integer, it must be divided in $k$-length blocks and multiplied by $G$.

$$(m_0, \dots, m_k) \cdot G = (c_1, \dots, c_n). \tag{2.10}$$

The resulting vector $c$ is a codeword of a Goppa code $\Gamma(L, g)$. Since Goppa codes are error correcting codes, even if some error are added to the codeword, the original message $m$ can be retrieved in the decoding step.

#### 2.2.2.4 *Decoding*

Let $y = (y_0, \dots, y_n)$ be a message received containing $r \leq \left\lfloor \frac{(d-1)}{2} \right\rfloor$ errors. Thus, $y$ can be written as

$$(y_1, \dots, y_n) = (m_1, \dots, m_n) + (e_1, \dots, e_n), \tag{2.11}$$

with $e_i \neq 0$ in $r$ positions, that is, the weight of the error vector $e$ is $w((e_1, \dots, e_n)) = r$. To retrieve the original message $m$ from $y$, the error vector $e$ must be constructed. To identify and correct the errors, an Error Locator Polynomial (ELP) $\sigma(x)$ is constructed; its roots indicate the positions of the errors in the received message. Let $T_e = \{i : e_i \neq 0\}$, the ELP is given by

$$\sigma(x) = \prod_{i \in T_e} (x - \alpha_i) \in \mathbb{F}_{p^m}[x]. \tag{2.12}$$

To construct the ELP and correct the errors to retrieve the codeword in the message, most commonly, the Patterson decoding algorithm (PATTERSON, 1975) is used.

##### 2.2.2.4.1 Patterson algorithm

The Patterson algorithm (PATTERSON, 1975) is used to decode Goppa codes and it is capable of correcting up to $t$ errors in a given encoded message $y$. Let $y = c + e$, to retrieve the error vector $e$, the syndrome value $S_y = Hy^T$ is calculated and used to construct the error locator polynomial $\sigma(x)$. Algorithm 1 shows all steps to retrieve the codeword $c$ from an encoded message $y$ given a Goppa code $\Gamma(L, g)$, where $L = \{\alpha_1, \dots, \alpha_n\}$.

In Step **v.** the roots of the ELP can be obtained using different methods. Four different methods are described in Chapter 3: the exhaustive method, the additive fast Fourier transform, the Berlekamp Trace Algorithm and a probabilistic root-finding algorithm. This root-finding step is the focus of our work since it can be the target of side-channel attacks as detailed in Section 2.3.3. See (PATTERSON, 1975) for details of the other steps of the algorithm.

**Algorithm 1** Patterson Algorithm(y)

```
Input: an encoded message $y$ of length $n$.
Output: the decoded codeword $c$.
i. Compute the syndrome polynomial $S_y(x) \leftarrow \sum_{i=1}^{n} \frac{y_i}{x-\alpha_i} \pmod{g(x)}$.
ii. Compute $\tau(x) \leftarrow \sqrt{S_y(x)^{-1}+x} \pmod{g(x)}$.
iii. Find two polynomials $a(x)$ and $b(x)$ so that $a(x) \equiv b(x)\tau(x) \pmod{g(x)}$ with $\deg(a) \leq \lfloor \frac{t}{2} \rfloor$ and $\deg(b) \leq \left\lfloor \frac{(t-1)}{2} \right\rfloor$.
iv. Determine the error locator polynomial $\sigma(x) \leftarrow a^2(x)+xb^2(x)$, $\deg(\sigma) \leq t$.
v. Define the error vector $e=(e_1,\ldots,e_n)$ so that $e_i \leftarrow 1$ if $\sigma(\alpha_i)=0$, otherwise $e_i \leftarrow 0$.
vi. Compute the codeword $c \leftarrow y+e$.
vii. Return $c$.
```

2.2.2.4.2 Example of a binary Goppa code

Let $m=3$ and $t=2$. The polynomial $g(x)$ must be irreducible, then $g(x)=Z^2*x^2+(Z^2+1)*x+Z^2+1$ and $L=\{0,1,Z,Z^2,Z+1,Z^2+Z,Z^2+Z+1,Z^2+1\}$. The parity check matrix of this code is

$$H=\begin{pmatrix} Z & Z^2+Z+1 & Z^2+1 & Z^2+1 & Z^2+Z & Z & Z^2+Z+1 & Z^2+Z \\ 0 & Z^2+Z+1 & 1 & Z & 1 & Z^2+Z+1 & Z+1 & Z+1 \end{pmatrix}. \tag{2.13}$$

The parity check matrix can also be represented in binary, that is

$$H=\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ -- & -- & -- & -- & -- & -- & -- & -- \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}. \tag{2.14}$$

Therefore, this Goppa code $\Gamma(L,g(x))$ is formed by the codewords $C=\{00000000, 01011101, 10101011, 11110110\}$, which are the words that result in a syndrome $S_c=0$. If some errors are added to the codeword 11110110 in the positions 4 and 7, we have $11110110+00010010=11100100$. To obtain the original codeword from the message with errors 11100100, we can use the Patterson algorithm.

Following the Patterson algorithm with $y=11100100$, we have:

1. Compute the syndrome polynomial $S_y(x) \leftarrow \sum_{i=1}^{n} \frac{y_i}{x-\alpha_i} \pmod{g(x)}$.

$$\begin{aligned}
S_y(x) &\leftarrow \sum_{i=1}^{n} \frac{y_i}{x-\alpha_i} \pmod{g(x)} \\
&= \frac{1}{x-0} + \frac{1}{x-1} + \frac{1}{x-Z} + \frac{1}{x-Z^2+Z} \\
&= ((Z+1)*x+1) + (x+Z^2+Z+1) + (Z*x+Z+1) + ((Z+1)*x) \\
S_y(x) &= (Z+1)*x+Z^2+1
\end{aligned}$$

2. Compute $\tau(x) \leftarrow \sqrt{S_y(x)^{-1}+x} \pmod{g(x)}$.

$$\begin{aligned}
S_y(x)^{-1} &= (Z+1)*x+Z^2 \\
S_y(x)^{-1}+x &= Z*x+Z^2
\end{aligned}$$

$$\begin{aligned}
\tau(x) &= \sqrt{Z*x+Z^2} \pmod{g(x)} \\
&= (Z*x+Z^2)^{2^{mt-1}} \pmod{g(x)} \\
\tau(x) &= Z^2*x+Z^2
\end{aligned}$$

3. Find two polynomials $a(x)$ and $b(x)$ so that $a(x) \equiv b(x)\tau(x) \pmod{g(x)}$ with $\deg(a) \leq \left\lfloor \frac{t}{2} \right\rfloor$ and $\deg(b) \leq \left\lfloor \frac{(t-1)}{2} \right\rfloor$.

$$\begin{aligned}
a(x) &\equiv b(x)\tau(x) \pmod{g(x)} \\
a(x) &\equiv b(x)(Z^2*x+Z^2) \pmod{g(x)} \\
Z^2*x+Z^2 &\equiv 1(Z^2*x+Z^2) \pmod{g(x)}
\end{aligned}$$

4. Determine the error locator polynomial $\sigma(x) \leftarrow a^2(x)+xb^2(x)$, $\deg(\sigma) \leq t$.

$$\begin{aligned}
\sigma(x) &\leftarrow a^2(x)+xb^2(x) \\
&= (Z^2*x+Z^2)^2+x*1^2 \\
\sigma(x) &= (Z^2+Z)*x^2+x+Z^2+Z
\end{aligned}$$

5. Define the error vector $e=(e_1,\ldots,e_n)$ so that $e_i \leftarrow 1$ if $\sigma(\alpha_i)=0$, otherwise $e_i \leftarrow 0$.

   Evaluating each $\alpha_i$ in $L=\{0,1,Z,Z^2,Z+1,Z^2+Z,Z^2+Z+1,Z^2+1\}$, we have that $\sigma(Z^2)=0$ and $\sigma(Z^2+Z+1)=0$. Therefore, the errors are in positions 4 and 7, resulting in the error vector $e=00010010$.

6. Compute the codeword $c \leftarrow y+e$.

$$\begin{aligned}
c &\leftarrow y+e \\
&= 11100100+00010010 \\
c &= 11110110
\end{aligned}$$

## 2.3 CODE-BASED CRYPTOGRAPHY

In this section, we describe public-key cryptography and detail the McEliece cryptosystem, a public-key and code-based cryptosystem. Then, we present a timing side-channel attack over the McEliece scheme, whose countermeasure is the focus of our work.

### 2.3.1 Public-key cryptography

Public key cryptography, also known as asymmetric key cryptography, is defined by the use of a pair of keys, a public key and a private one, used to make complementary operations, such as encryption and decryption, or signing a document and then verifying the signature. Any entity has access to the public key to encrypt a message but the decryption of said message can only be done with the corresponding private key.

This concept was introduced by Diffie and Hellman (DIFFIE; HELLMAN, 1976). Before, cryptosystems were designed with only one key and used mostly operations of substitution and permutation of elements in the messages whereas public key cryptosystems are based on mathematical functions. The most important and massively used public key cryptosystem is RSA (RIVEST; SHAMIR; ADLEMAN, 1978). The McEliece cryptosystem (MCELIECE, 1978) is also an example.

### 2.3.2 McEliece cryptosystem

The McEliece cryptosystem (MCELIECE, 1978) is a public key cryptosystem. Its keys are generated using an error-correcting code, originally Goppa codes. Encryption and decryption are efficient, however public key size is a drawback. The security of the cryptosystem is based on the hardness of generic decoding, an NP-hard problem also believed to be hard on average, which so far has resisted attacks by classic and quantum computers. The McEliece cryptosystem is composed of three algorithms: key generation, encryption, and decryption. In the key generation step, public and private keys are created, which can be used in the encryption and decryption of messages.

#### *2.3.2.1 Key generation*

The key generation in the original proposal (MCELIECE, 1978) uses binary Goppa codes. First, a Goppa code $\Gamma(L,g)$ is defined. A Goppa polynomial $g$ of degree $t$ and coefficients in $\mathbb{F}_{2^m}$, and a support set $L$ are selected. Then, the generator matrix $G$ is defined. After that, two random matrices are created: an invertible $k \times k$ matrix $S$ and an $n \times n$ permutation matrix $P$. The matrices are then multiplied, creating the matrix $G' = SGP$ that is part of the public key $PK = (G',t)$. The private key is formed by $SK = (g,L,S,G,P)$.

### *2.3.2.2 Encryption*

To encrypt a message $M \in \mathbb{F}_2^k$, errors are intentionally added to the message. An error vector $e$ of length $n$ with weight $w(e) = t$ is randomly selected. Then, the message $M$ can be encrypted as $c = MG' + e$, resulting in the encrypted message $c$.

### *2.3.2.3 Decryption*

The decryption consists of correcting the errors added during the encryption of the message. Let $P^{-1}$ be the inverse of matrix $P$. To decrypt a received message $c$, the receiver can compute

$$c' = cP^{-1} = MG'P^{-1} + eP^{-1} = MSGPP^{-1} + eP^{-1} = (MS)G + e', \tag{2.15}$$

where $c'$ is formed by a codeword of the Goppa code defined in the key generation step with some errors, since it is a message $M$ multiplied to the generator matrix $G$ with errors added by the vector $e'$.

The errors can be detected and corrected by a decoding algorithm, such as the Patterson decoding algorithm (PATTERSON, 1975), shown in Algorithm 1. The syndrome polynomial computed in Step **i** is used to determine the error locator polynomial $\sigma(x)$ in Step **iv**. The roots of the error locator polynomial define the error vector $e'$ used to recover the original message $M$ by computing $c' + e'$.

Therefore, after decoding $c'$, the errors are removed and we obtain $c'' = (MS)G$. Depending on the construction of the matrix $G$, we can easily obtain $c''' = MS$ from $(MS)G$. Since $S$ is invertible, we can use $S^{-1}$ to obtain the original message $M = c'''S^{-1} = MSS^{-1}$.

### *2.3.2.4 Example of McEliece's encryption and decryption processes*

We consider that the Goppa code used in this example is the one given in Section 2.2.2.4.2.

1. Key generation

   From the Goppa code definition we have $t = 2$, $g(x) = Z^2 * x^2 + (Z^2+1) * x + Z^2 + 1$, $L = \{0, 1, Z, Z^2, Z+1, Z^2+Z, Z^2+Z+1, Z^2+1\}$ and

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & Z & Z \\ 0 & 1 & 0 & 0 & 0 & 0 & Z & Z^2+Z \\ 0 & 0 & 1 & 0 & 0 & 0 & Z^2+Z+1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & Z^2+Z \\ 0 & 0 & 0 & 0 & 1 & 0 & Z^2 & Z \\ 0 & 0 & 0 & 0 & 0 & 1 & Z^2+Z+1 & Z+1 \end{pmatrix}.$$

Then, two random matrices are created: an invertible $k \times k$ matrix $S$ and an $n \times n$ permutation matrix $P$. Let

$$S = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

and

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

.

The matrices are then multiplied, creating the matrix $G' = SGP$.

$$G' = \begin{pmatrix} 1 & 0 & 1 & Z+1 & 1 & 0 & 1 & Z^2+Z \\ 0 & 0 & 0 & Z^2 & 1 & 1 & 1 & Z+1 \\ 1 & 1 & 1 & Z^2 & 0 & 1 & 1 & Z^2 \\ 1 & 1 & 1 & Z^2+1 & 1 & 1 & 1 & Z \\ 1 & 1 & 0 & Z+1 & 0 & 1 & 1 & Z^2+1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & Z+1 \end{pmatrix}$$

The public key is formed by $PK = (G', t)$ and the private key is formed by $SK = (g, L, S, G, P)$.

2. Encryption

When an entity $A$ wants to send a message $M$ to an entity $B$, $A$ can use $B$'s public key $PK_B = (G', t)$. Let $M = 010100$. $A$ must select a random error vector $e$ with weight $w(e) = t$ and compute $c = MG' + e$. Since $t = 2$, let $e = 00110000$.

$$\begin{aligned} c &= MG' + e \\ &= 11110001 + 00110000 \\ c &= 11000001 \end{aligned}$$

3. Decryption

To decrypt the received message $c$, the entity $B$ must use its private key $SK_B = (g, L, S, G, P)$. First, $c'$ is computed

$$c' = cP^{-1} = 10001001$$

After that, to detect and correct the errors in the message $c'$, Patterson algorithm is used, resulting in

$$c'' = (MS)G = 10101011$$

Since the generator matrix $G$ is in the form $G = (I_k|A)$, that is, the first $k \times k$ elements form an identity matrix, $MS$ can be easily obtained from $c''$ because $MS$ is the first $k$ elements in $c''$. Therefore, $MS = 101010$. Since S is invertible, we can obtain the original message $M$ from $M = MSS^{-1}$

$$M = MSS^{-1} = 010100$$

### 2.3.3 Timing side-channel attacks

Side-channel attacks are attacks based on extra information, other than the the original message and the ciphertext. The additional information can be power consumption statistics or the execution time of the operations. In this work, we focus on the latter.

The timing side-channel attack proposed by Strenzke et al. and Shoufan et al. (STRENZKE et al., 2008; SHOUFAN et al., 2010) exploits the time variance between different executions of the evaluation of the error locator polynomial, which reveals information resulting in the recovery of the error vector. The attack described is on the root-finding step in the decoding algorithm of a Goppa code. This attack is successful when the execution time of the decoding process varies according to the number of roots of the ELP.

In public-key cryptosystems based on the McEliece cryptosystem the attacker cannot easily construct the generator matrix from the public key but leaking information about the error vector can still be dangerous. This is not desirable given that it leaks information about the secret set $L$ of the Goppa code used to create the keys.

The attack exploits the difference in the time required in the root-finding step for a message that contains $t$ errors and a message that contains $w$ errors, for $w < t$. The number of errors in the message is also the number of roots of the error locator polynomial $\sigma(x)$. Therefore, when a message contains less than $t$ errors, the polynomial $\sigma(x)$ has $w < t$ roots, and if a message contains more than $t$ errors, $\sigma(x)$ has $w \leq t$ roots (STRENZKE, 2012).

An attacker can exploit this behavior by flipping a bit in position $i$ in a ciphertext $c$ with $t$ errors. Observing the execution time, it is possible to infer if the flipped bit added or removed an error in the ciphertext, given that the time for the evaluation of the polynomial $\sigma(x)$ will be different if the ciphertext has $t-1$ or $t+1$ errors. Performing this strategy to each bit of $c$, the attacker can discover all positions of $c$ that contain errors.

A plain implementation of the McEliece cryptosystem is susceptible to an adaptive chosen-ciphertext attack (CCA2) like the one above. Moreover, it has been established that this attack is not prevented by a CCA2 conversion (STRENZKE et al., 2008). Therefore, the implementation of a countermeasure for this attack is important for the security of the scheme. For this reason, the execution time of the root-finding step during the decryption process must be independent of the degree of the input polynomial.

Our countermeasure to this attack involves artificially raising the degree of the ELP when it is lower than $t$. The algorithm is provided in Section 4.

A constant-time implementation should prevent non-constant time operations correlated to any secret information. The types of operations to be concerned are branching (conditional jumps), memory accesses, or use of variable-time operations (integer divisions, shifts and rotations) (ALMEIDA et al., 2016). An implementation of our proposal can avoid branching and memory-access leakage, and only use operations that run in constant time.

# 3 ROOT-FINDING ALGORITHMS

In this chapter, we introduce four known root-finding algorithms: the exhaustive one that repeatedly evaluates the polynomial in search of its roots, the multipoint evaluation called additive Fast Fourier Transform, the Berlekamp Trace Algorithm and a classical probabilistic root-finding method. The latter algorithm is probabilistic, a type of algorithm not explored so far in the context of code-based cryptosystems. The other three methods, the exhaustive method, the additive Fast Fourier Transform, and the Berlekamp Trace Algorithm, are deterministic algorithms.

As mentioned, the root-finding algorithm used during the decryption process must not vary in execution time in order to not leak any information. The previously considered methods succeed in this aspect, whereas an immediate implementation of the classical probabilistic root-finding algorithms in the literature (due to Berlekamp (BERLEKAMP, 1970), Rabin (RABIN, 1980), Ben-Or (BEN-OR, 1981), Cantor and Zassenhaus (CANTOR; ZASSENHAUS, 1981), and von zur Gathen and Shoup (GATHEN; SHOUP, 1992), among others) do not. A key contribution of our work is, by providing a side-channel resistant version, to bring these well-known asymptotically fast algorithms to their potential use in code-based post-quantum cryptographic systems such as the McEliece cryptosystem and its variants.

## 3.1 EXHAUSTIVE METHOD

The simplest method to find the roots of a polynomial is to repeatedly evaluate the polynomial $\sigma(x)$ at each element in the field $\mathbb{F}_{p^m}$. In the McEliece cryptosystem, every root of the error locator polynomial is in the support set $L$ of the code. As a consequence, evaluating all elements $\alpha_i \in L$ is enough in this context to identify all the roots. We refer to this method as the exhaustive method. This approach is used in the implementation of the NIST post-quantum fourth round candidate Classic McEliece (ALBRECHT et al., 2020).

Evaluating a polynomial at $n$ points, one at a time using Horner's method, can be efficiently done for small values of $n$ and small order finite fields, like those in the Classic McEliece's parameters. We comment that Horner's method is proved to be optimal when evaluating a polynomial at a single point. However, the main drawback of its use is that its complexity deteriorates for larger values of $n$. For cryptography, the advantage of this method to find roots is that it evaluates exactly $n$ points, regardless of the polynomial, so its execution time is fixed, once the degree of the polynomial is given, regardless of the polynomial being evaluated.

Let $n$ be the number of points to be evaluated and $t$ the degree of the polynomial. The total cost $C_{\text{exh}}$ of this algorithm is

$$C_{\text{exh}} = n(t(C_{\text{add}} + C_{\text{mult}})) \tag{3.1}$$

where $C_{\text{add}}$ is the cost of addition and $C_{\text{mult}}$ is the cost of multiplication of elements in $\mathbb{F}_{p^m}$.

## 3.2 ADDITIVE FAST FOURIER TRANSFORM

A more efficient way to evaluate a polynomial is to perform a multipoint evaluation. Typical algorithms for this task are variants of the Fast Fourier Transform (FFT) method. For multipoint evaluation in characteristic 2 and in the context of code-based cryptography, the most popular method is the additive Fast Fourier Transform (GAO; MATEER, 2010). Its execution time is not tied to the polynomial degree, so it suits well the need for constant time root-finding. The NIST post-quantum second round candidate NTS-KEM (ALBRECHT et al., 2019) (now merged with Classic McEliece), as well as the alternate candidate of the fourth round HQC (MELCHOR et al., 2021) use this method in the decryption process.

Assuming that $\beta_1,\dots,\beta_m \in \mathbb{F}_{2^m}$ are linearly independent over $\mathbb{F}_2$, let $B$ be the subspace spanned by $\beta_1,\dots,\beta_m$ over $\mathbb{F}_2$, that is, $B = \langle \beta_1,\dots,\beta_m\rangle = \{a_1\beta_1+\cdots+a_m\beta_m : a_1,\dots,a_m \in \mathbb{F}_2\}$.

For any $0 \le i < 2^m$, the binary representation of $i$ is $i = a_1 + a_2\cdot 2 + \cdots + a_m \cdot 2^{m-1}$, where $a_i \in \mathbb{F}_2$. Then, the $i$-th element of $B$ is $B[i] = a_1\beta_1 + a_2\beta_2 + \cdots + a_m\beta_m$. The additive Fast Fourier Transform algorithm evaluates a given polynomial $f$ of degree smaller than $n = 2^m$ at each point in $B$, using a multipoint strategy, returning the sequence of values $f(B[0]), f(B[1]),\dots, f(B[n-1])$.

The algorithm works by recursively reducing the degree of the input polynomial $f$ by half, writing $f$ in the form $g_0(x^2+x)+xg_1(x^2+x)$, $g_0, g_1 \in \mathbb{F}_{2^m}[x]$ using a Taylor expansion; see more details in Gao & Mateer (2010).

The degree of the polynomial $g_0$ is $\lceil n/2\rceil$ and of the polynomial $g_1$ is $\lfloor n/2\rfloor$. Rewriting $f$ as $g_0(x^2+x)+xg_1(x^2+x)$ shows that evaluating $f(a)$ and $f(a+1)$ overlaps, observing that $a^2+a = (a+1)^2+(a+1)$. Let $f(a) = g_0(a^2+a)+ag_1(a^2+a)$ and $f(a+1) = g_0(a^2+a)+(a+1)g_1(a^2+a)$. After computing $f_0 = g_0(a^2+a)$ and $f_1 = g_1(a^2+a)$, $f(a)$ can be computed as $f_0+af_1$ and $f(a+1)$ can be computed as $f(a)+f_1$. This method reduces the number of multiplications and additions needed to evaluate a polynomial, especially when compared to other evaluation algorithms like the exhaustive method mentioned above.

Let $n = 2^m$. The cost $C_{\text{afft}}$ of this method is (GAO; MATEER, 2010)

$$C_{\text{afft}} = \left(\frac{n}{4}(\log_2(n))^2 + \frac{3n}{4}\log_2(n) - \frac{n}{2}\right)C_{\text{add}} + (2n\log_2(n) - 2n + 1)C_{\text{mult}}. \qquad (3.2)$$

## 3.3 BERLEKAMP'S ALGORITHMS

Our proposed probabilistic method in Section 3.3.2 has its root in the fundamental works of Berlekamp (BERLEKAMP, 1967; BERLEKAMP, 1970; BERLEKAMP, 1968, Chapter 6). In particular, the idea of using probabilistic algorithms and traces when finding roots in characteristic 2 are originated in his works. The idea of using traces also influenced other strategies for factoring polynomials, such as the Berlekamp Trace Algorithm. For a complete bibliography (up to the time of publication) about the classical problem of factoring polynomials over finite fields, see the survey (GATHEN; PANARIO, 2001). The following algorithms are factoring algorithms. However, we describe how they can be adapted to be used as root-finding algorithms.

### 3.3.1 Berlekamp Trace Algorithm

The Berlekamp Trace Algorithm (BTA) is a recursive deterministic factoring algorithm. The algorithm is based on properties of the trace function, defined as $\mathrm{Tr}_{\mathbb{F}_{2^m}/\mathbb{F}_2}(\alpha) = \alpha + \alpha^2 + \alpha^{2^2} + \cdots + \alpha^{2^{m-1}}$. As usual, we denote $\mathrm{Tr}_{\mathbb{F}_{2^m}/\mathbb{F}_2}(\alpha)$ by $\mathrm{Tr}(\alpha)$. Let $B = \{\beta_1, \ldots, \beta_m\}$ be a basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. Every element $\alpha \in \mathbb{F}_{2^m}$ is uniquely represented by the binary $m$-tuple $(\mathrm{Tr}(\beta_1 \cdot \alpha), \ldots, \mathrm{Tr}(\beta_m \cdot \alpha))$ (BISWAS; HERBERT, 2009).

A polynomial $f \in \mathbb{F}_{2^m}[x]$ such that $f(x) | (x^{2^m} - x)$ is split into two polynomials $f_0(x) = \gcd(f(x), \mathrm{Tr}(\beta_i \cdot x))$ and $f_1(x) = \gcd(f(x), \mathrm{Tr}(\beta_i \cdot x) - 1) = f(x)/f_0(x)$. Computing iteratively on $\beta_i \in B$ and recursively factoring on $f$, the input polynomial $f$ can be factored into linear factors that can be adapted to find the roots of $f$. Algorithm 2 shows the Berlekamp Trace Algorithm.

**Algorithm 2** Berlekamp Trace Algorithm — `BTA(f,i)`

```
Input: a polynomial $f \in \mathbb{F}_{2^m}$ of degree $t$ and an integer $i$, $1 \leq i \leq m$.
Output: the list of irreducible factors of degree 1 of $f$.

    Let $B = \{\beta_1, \ldots, \beta_m\}$ be a basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$.
  i. If $\deg(f) = 1$, then return $\{f\}$; if $\deg(f) = 0$ return $\{\}$.
 ii. Compute $f_0 \leftarrow \gcd(f, \mathrm{Tr}(\beta_i \cdot x))$.
iii. Compute $f_1 \leftarrow f/f_0$.
 iv. Return BTA($f_0, i+1$) $\cup$ BTA($f_1, i+1$).
```

The cost of this method is $O(mt^2)$ operations in $\mathbb{F}_{2^m}$ (BISWAS; HERBERT, 2009). Therefore, the execution time of the algorithm is correlated to the degree of the input polynomial. Figure 2 shows the execution time of the BTA-it algorithm, an iterative implementation of the Berlekamp Trace Algorithm, proposed in (MARTINS; BANEGAS; CUSTÓDIO, 2019, Section 3.3), for input degrees 2 to 128, with 2500 measures for each point. The boxplots display the minimum, first quartile, median, third quartile, and maximum values for each point in the x-axis. The BTA-it algorithm is a countermeasure to the Berlekamp Trace Algorithm in order to obtain a less variable execution time when the degree of the input polynomials are equal to the target degree $t$. Therefore, the countermeasure does not addresses cases when the input degree may be different than $t$ and the variation in execution time it entails. Consequently, BTA-it is still vulnerable to the timing side-channel attack described in Section 2.3.3.

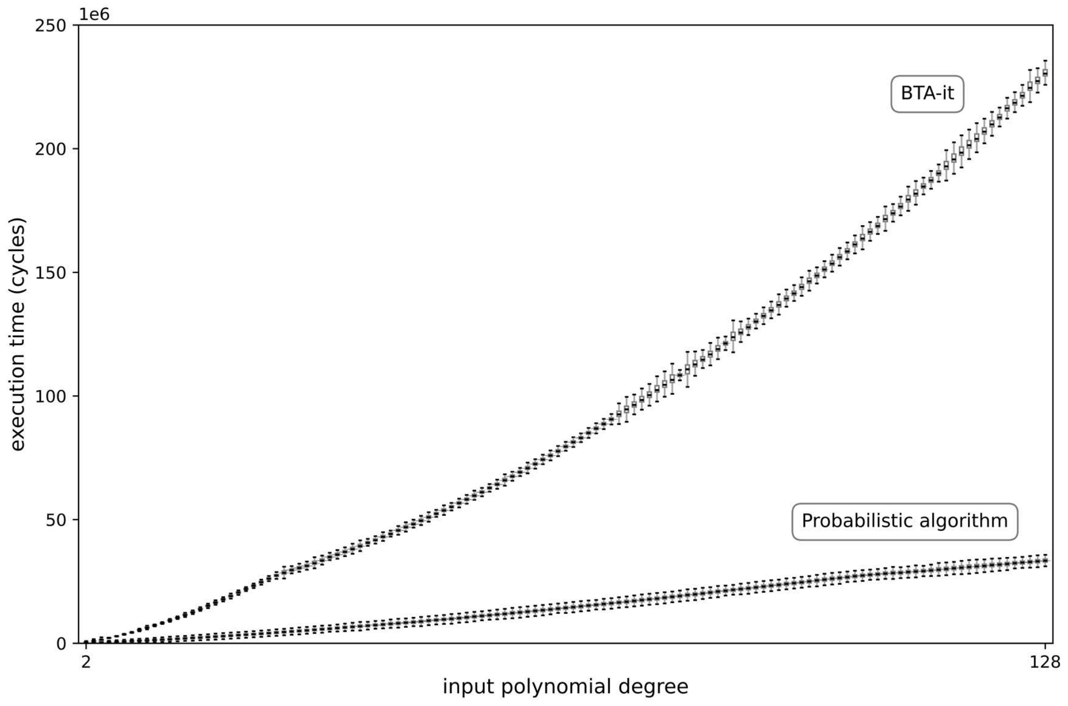

Figure 2 – Execution time in CPU cycles of BTA-it and equal-degree probabilistic algorithm for degrees 2 to 128 with polynomials in $\mathbb{F}_{2^{13}}[x]$.

### 3.3.2 A probabilistic root-finding algorithm

A general factoring strategy uses three stages: the squarefree factorization (that removes repeated factors), the distinct-degree factorization (that partially factors the polynomial into factors of the same degree), and the equal-degree factorization (that finally obtains the irreducible polynomials for each of the polynomials produced in the second stage). In the case of finding roots of interest here, we just need to run the equal-degree factorization for factors of degree $d = 1$.

The asymptotically fastest algorithms for equal-degree factorization over finite fields are probabilistic; see (GATHEN; PANARIO, 2001). Indeed, even for middle size polynomial degrees or finite field orders, these probabilistic methods behave better than deterministic algorithms. Algorithm 3 shows the steps of the equal-degree factorization for a polynomial over $\mathbb{F}_q$, $q = 2^m$, and the common degree of the irreducible factors equal to $d$. In this work we are interested in $d = 1$.

A method for factoring polynomials can be easily adapted to compute roots. The error locator polynomial is composed only of linear factors, so $d = 1$ in this case. In Step **i**, when $\deg(f) = d$, it means that $\deg(f) = 1$, that is, $f(x) = x - \gamma$, for some $\gamma$. Hence, the roots of the original input polynomial $f$ are the list of coefficients $\gamma$.

Step **iii** implements the trace computation of $g$, while Step **iv** attempts to improve the factorization towards the complete factorization in roots. These are ideas from Berlekamp and our starting point.

**Algorithm 3** Equal-degree factorization algorithm for polynomials in $\mathbb{F}_{2^m}[x]$ — `EDF(d,f)`

```
Input: $d \in \mathbb{N}$, a monic polynomial $f \in \mathbb{F}_{2^m}$ of degree $n = rd$.
Output: the list of $r$ monic irreducible factors of degree $d$ of $f$.

  i. If $\deg(f) = d$, then return $\{f\}$; if $\deg(f) = 0$ return $\{\}$.
 ii. Pick $g \in \mathbb{F}_q[x]$ at random with degree smaller than $n$.
iii. Compute
         $h \leftarrow \sum_{i=0}^{m-1} g^{2^i}.$
 iv. Compute $p_1 \leftarrow \gcd(h, f)$ and $p_2 \leftarrow f/p_1$.
  v. Return $\mathrm{EDF}(d, p_1) \cup \mathrm{EDF}(d, p_2)$.
```

The random polynomial choice adds a probability of failure to the algorithm since the chosen polynomial may not deliver satisfying results when computing Step **iv**. This probability is $1-(1/2^{t-1})$, where $t$ is the polynomial degree; see (GATHEN; PANARIO, 2001). This shows that the chance of having to retry Steps **ii**, **iii**, **iv** of Algorithm 3 in order to obtain a valid factor is very small unless the input polynomial has a very small degree.

Although the number of extra times that Steps **ii**, **iii**, **iv** are executed can vary with the random choices, in our experiments for $q = 2^{13}$, $t = 10, 96, 128, 200, 260$, 512, and with 2500 measures, it is on average $c = 0.44t$ for every $t$. Moreover, we observe that the most common size for the two factors is about half the original size. Indeed, the probability of a polynomial splitting into two factors, one of small degree and the other of large degree, is much smaller than the probability of breaking into two factors of about the same size. These facts have also been confirmed in our experiments.

The execution time of Algorithm 3 can be derived as a special case of the analysis of the polynomial factorization process formed by the three stages discussed above. The complete cost analysis is delicate (see (FLAJOLET; GOURDON; PANARIO, 1996, Section 7) and (FLAJOLET; GOURDON; PANARIO, 2001, Section 5)), but the special case of Algorithm 3 simplifies considerably since we only need the third stage, and only for degree 1; see (FLAJOLET; GOURDON; PANARIO, 2001, Table 1). Also, the execution time is extremely tied to the degree of the input polynomial, as shown in Figure 2 for degrees 2 to 128, with 2500 measures for each point.

In the case of using Algorithm 3 in a code-based cryptosystem, an attacker can easily see a difference in the duration of the decryption process when decoding a word with $t$ errors and one with $t' = t - \delta$ errors, where $\delta$ is a positive integer smaller than $t$. The countermeasure we propose in Section 4 assures that the variation in the execution time is not related to any secret information, like the input polynomial degree, preventing attacks such as those discussed in Section 2.3.3.

# 4 CONSTANT-TIME APPROACH TO A PROBABILISTIC ROOT-FINDING ALGORITHM

In this chapter, we propose a constant-time approach to a probabilistic root-finding algorithm (CTProb). The execution time of the root-finding algorithm detailed in Section 3.3.2 is tied to the degree of the input polynomial. As commented above, this property can be exploited in a side-channel attack, such as the one explained in Section 2.3.3. In order to prevent this attack, a constant-time root-finding algorithm must be used in the decryption process in a code-based cryptosystem. We remark that the notion of constant-time we are using in this work is that different inputs may yield variations in time, however, any variation in the execution time of the root-finding algorithm must not be tied to any secret information. In the context of the McEliece cryptosystem, the degree of the Error Locator Polynomial is important information that must be kept a secret in situations which it is different than the expected by cryptosystem parameters.

**Algorithm 4** Constant-time probabilistic root-finding algorithm (`f`, $\zeta$)

**Input:** a polynomial $f \in \mathbb{F}_{2^m}[x]$ of degree $1 \leq n \leq t$, $\zeta \in \mathbb{F}_{2^m}$ that is not part of the support set $L$ of the Goppa code.
**Output:** a list of roots $R$ of size $t$, with $n$ roots and $t-n$ elements $\zeta$.

```
  i. Pick a random number $0 < j < (\texttt{maximum field element}/t)$.
 ii. $g \leftarrow$ AddRoots(f,j);
iii. $R \leftarrow$ IterativeRootFinding(g);
 iv. $R \leftarrow$ RemoveRoots(R,t,n,j,$\zeta$);
  v. Return $R$.
```

We propose a countermeasure, shown in Algorithm 4, that aims to prevent the aforementioned time side-channel attack. It results in a constant-time probabilistic root-finding algorithm. The idea is to artificially increase the degree of the input polynomial when its degree is lower than the target value $t$. The target degree in this context is the number of errors that the cryptosystem can correct and that is added to the message. Thus, the root-finding algorithm is always going to find the same number of roots for any polynomial. First, fake roots are added to the polynomial so its degree is increased to the target number. Then, after finding the roots of the modified polynomial, the fake roots are removed from the set.

In the algorithms that we propose, the branches taken and memory addresses accessed are independent of any secret inputs, and operations that can leak information, like integer divisions, are avoided. The operands of the division in Step **i** in Algorithm 4 are public cryptosystem parameters so that no secret can be leaked from this operation. These properties are important in order to achieve a constant-time probabilistic root-finding algorithm.

First, in order to create different fake roots on each execution, a random number $j$ is picked. This value is used as the increment between each fake root added to the input polynomial in Step **ii** of Algorithm 4. The creation and addition of fake roots are shown in Algorithm 5. The goal is to increase the degree of the input polynomial $f$ to the target value $t$. Since the degree of the input polynomial must be $n \geq 1$, at most $t-1$ fake roots $\rho$ are needed so that, after adding the fake roots, the degree of the resulting polynomial is equal $t$. We denote addition in finite fields with $\oplus$. To add the fake roots, the polynomial is multiplied with $t-1$ factors. The factors are of the form $ax+b$. When the degree of the input polynomial is $n<t$, then $t-n$ factors are created with $a=1$ and $b=\rho$, $\rho \in \mathbb{F}_{2^m}$, $j \oplus 1 \leq \rho \leq ((t-n)*j) \oplus 1$, and $n-1$ factors are created with $a=0$ and $b=1$, resulting in $t-1$ factors. Consequently, when $n=t$, fake roots are not necessary. Hence, the $t-1$ factors are created with $a=0$ and $b=1$ so that the same amount of work is done for any $n$, that is, for any input $t-1$ factors are always multiplied.

**Algorithm 5** Adding fake roots — `AddRoots(f,j)`

**Input:** a polynomial $f \in \mathbb{F}_{2^m}[x]$ of degree $1 \leq n \leq t$, the value $j$ of the increment between each fake root.
**Output:** a polynomial of degree $t$.

```
i.   factors ← CreateFactors(t,n,j);
ii.  $\deg(p) \leftarrow t-1$; $p \leftarrow 1$;
iii. MultiplyFactors(p,t-1,factors,t-1);
iv.  Pad $f$ with t-n zero coefficients to get to $t+1$ coefficients.
v.   $g \leftarrow f * p$;
vi.  Truncate polynomial $g$ to keep $t+1$ coefficients.
vii. Return $g$.
```

The creation of the $t-1$ factors is shown in Algorithm 6. For each factor, in Step **ii.a** it is defined the value of the leading coefficient, that is either 0 or 1. The leading coefficient is 1 for $t-n$ factors. In Step **ii.c** the value of the fake root is calculated. The variable `added` indicates how many fake roots were already created. This value is used to create the multiples of the given value $j$. Also, because the value of the fake root must be 1 when the leading coefficient is 0, Step **ii.c** is formed by $1 \oplus (\texttt{addRoot} * (\texttt{value of the fake root}))$, that is, when the leading coefficient is 0, it results in $1 \oplus 0$. Therefore, when the leading factor is 0, the value of the fake root is 1, whereas when the leading factor is 1, the fake root is defined as $1 \oplus (\texttt{added} * j)$, $1 \leq \texttt{added} \leq t-n$. Finally, in Step **ii.d** the factor is created.

The factors are then multiplied, creating the polynomial $p$, with degree $t-1$. This is shown in Algorithm 7. This multiplication is done recursively. The loop divides the input polynomial of degree $d$ into two other polynomials $p_1, p_2$ of degree $\lceil d/2 \rceil$ and $\lfloor d/2 \rfloor$ that recursively feed Algorithm 7. The base step is reached when $d<3$. Then, when $d=2$ two factors are multiplied and returned, or when $d=1$ one factor is returned. This step is needed for cases when $t-1$ is an odd number. When returning from the recursion, $p_1$ and $p_2$ are multiplied and returned. This process can be illustrated in a structure similar to a binary tree.

**Algorithm 6** Creating fake roots — `CreateFactors(t,n,j)`

**Input:** the target number of errors $t$, the degree $n$ of the input polynomial, the value $j$ of the increment between each fake root.
**Output:** a list $F$ of $t-1$ factors.

```
i. Initialize diff $\leftarrow t-n$, addRoot $\leftarrow 1$, added $\leftarrow 0$.
ii. For index in $[1, t-1]$
    a. addRoot $\leftarrow$ addRoot $-$ not(index $-$ (diff + 1));
    b. added $\leftarrow$ added $+$ addRoot;
    c. fakeRoot $\leftarrow 1 \oplus$ (addRoot $*$ (added $* j$));
    d. $F$[index $-$ 1] $\leftarrow$ (addRoot)$x$ $-$ fakeRoot;
iii. Return $F$.
```

An example for when $t = 10$ is shown in Figure 3. Each internal node represents a call of Algorithm 7 and the number in the nodes represent the degree of the polynomial in each call. The value in the root is $t-1$ since it is created by multiplying $t-1$ factors, therefore resulting in a polynomial $p$ of degree $t-1$.

**Algorithm 7** Multiplying factors — `MultiplyFactors(p,d,factors,index)`

**Input:** a polynomial $p$ that will be overwritten, the degree $d$ of the polynomial, a list `factors` of $t-1$ factors, and the current index in the list.
**Output:** the input polynomial $p$ will be overwritten.

```
i. If $d < 3$, then
    a. If $d = 1$, then
        i. $p \leftarrow$ factors[index];
        ii. index $\leftarrow$ index $-$ 1;
    b. Else
        i. $p \leftarrow$ factors[index] $*$ factors[index $-$ 1];
        ii. index $\leftarrow$ index $-$ 2;
    c. Return.
ii. MultiplyFactors($p_1$, $\lceil d/2 \rceil$, factors, index);
iii. MultiplyFactors($p_2$, $\lfloor d/2 \rfloor$, factors, index);
iv. $p \leftarrow p_1 * p_2$.
```

Then, the input polynomial $f$ is multiplied by $p$, resulting in the polynomial $g$ of degree $2t-1$, with the higher $t-1$ coefficients being 0 due to the padding of the input polynomial and the multiplication with factors of the form $0x-1$. Thus, $g$ can be truncated resulting in a polynomial of degree $t$, with $n$ original roots and $t-n$ fake roots. The polynomial $g$ is the input of the root-finding algorithm in Step **iii** in Algorithm 4. In order to compute similar operations

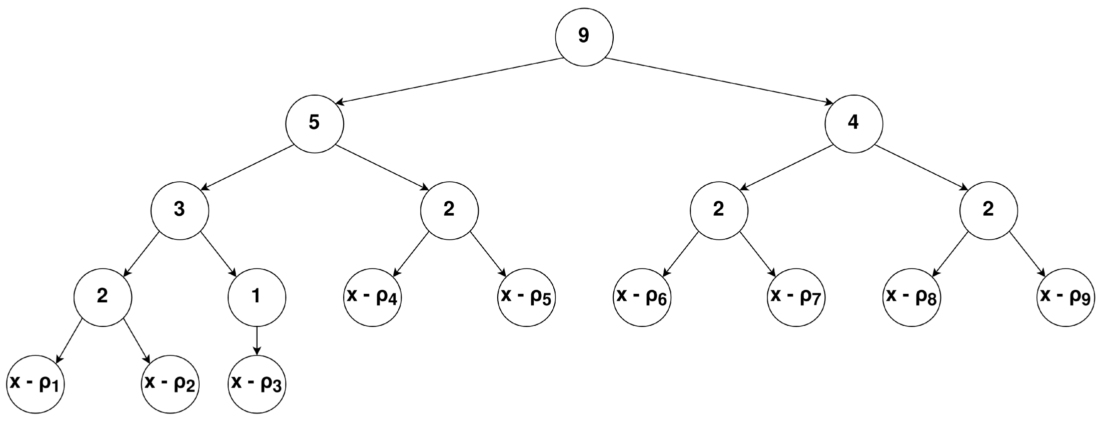

Figure 3 – Illustration of the degree of the input polynomial during the recursive calls of Algorithm 7 when $t = 10$.

regardless of the input degree $n$, the input polynomial $f$ is padded and always multiplied by a polynomial $p$ of degree $t-1$. When creating $p$, the same number of multiplications is made regardless of the value of $n$. This is shown in Steps **iv**, **v**, and **vi** of Algorithm 5.

The cost of Algorithm 5 $C_{\text{add\_roots}}$ is composed of the costs of Algorithms 6 and 7, and the cost of Step **iv**, that is, the additions while creating the fake roots in Algorithm 5, the polynomial multiplications in Algorithm 7, and the multiplications while combining the fake roots into the input polynomial in Step **iv**. The process of multiplying the factors in Algorithm 7 results in a structure similar to a binary tree, considering that the value in the nodes is the polynomial degree. At each level, the value is approximately half the degree in the upper node. Therefore, the cost of Algorithm 7 $C_{\text{multiply\_factors}}(x)$ is the cost of multiplying $x$ factors.

$$C_{\text{multiply\_factors}}(x) = \begin{cases} 0 & \text{if } x = 1 \\ 4 & \text{if } x = 2 \\ \left(\left(\left\lceil \frac{x}{2} \right\rceil + 1\right) * \left(\left\lfloor \frac{x}{2} \right\rfloor + 1\right)\right) + & \text{otherwise} \\ C_{\text{multiply\_factors}}\left(\left\lceil \frac{x}{2} \right\rceil\right) + C_{\text{multiply\_factors}}\left(\left\lfloor \frac{x}{2} \right\rfloor\right), & \end{cases} \tag{4.1}$$

Moreover, the additions when creating the fake roots in Algorithm 6 are done $t-1$ times, and Step **iv** is the multiplication of a polynomial of degree $t$ (that is, $t+1$ coefficients) and a polynomial of degree $t-1$, resulting in

$$C_{\text{add\_roots}} = (t-1)C_{\text{add}} + (C_{\text{multiply\_factors}}(t-1) + ((t+1)*t))C_{\text{mult}}. \tag{4.2}$$

After adding the fake roots into the input polynomial, the root-finding algorithm receives the new $g$ of degree $t$ as input. For CTProb, the root-finding algorithm is Algorithm 8, and it is an iterative adaptation of Algorithm 3 that returns the roots of a polynomial instead of its monic factors. These modifications do not change the final cost of the algorithm. The root-finding algorithm can be replaced with other root-finding algorithms. In Section 5, we substitute Algorithm 8 with BTA-it, proposed by Martins et al. (MARTINS; BANEGAS; CUSTÓDIO, 2019). The GCD computation can be done in constant-time (BERNSTEIN; YANG, 2019).

**Algorithm 8** Iterative probabilistic root-finding — `IterativeRootFinding(f)`

**Input:** a monic polynomial $f \in \mathbb{F}_{2^m}$ of degree $t$.
**Output:** the list of $t$ roots of $f$.

```
i.   Initialize a stack $S$ and push $f$.
ii.  Initialize a list $R$ of size $t$.
iii. For index in $[1, 2t-1]$
       a. Pop a polynomial $f_1$ from the stack.
       b. If $\deg(f_1) = 1$, then include the constant coefficient of $f_1$ in $R$, continue.
       c. Do:
            1. Pick $g \in \mathbb{F}_q[x]/(f_1)$ at random.
            2. Compute
                 $h \leftarrow \sum_{i=0}^{m-1} g^{2^i} \pmod{f_1}$.
            3. Compute $p_1 \leftarrow \gcd(h, f_1)$.
          While $\deg(p_1) \leq 0$ or $\deg(p_1) = \deg(f_1)$.
       d. Compute $p_2 \leftarrow f_1/p_1$.
       e. Push $p_1$ and $p_2$ to the stack.
iv.  Return $R$.
```

After finding the roots of the polynomial $g$, the fake roots must be removed from the list. The removal is shown in Algorithm 9. The goal is to erase the fake roots in the list $R$ returned by Algorithm 8. The list $R$ is copied onto the list $R_1$ of size $t$ and then the list $R_1$ is sorted. Hence, the fake roots in the list are in increasing order. Therefore we can remove the fake roots in $R_1$ by comparing the elements in the list with the values of the fake roots that were added to the polynomial $f$. This is done in Step **iv**. When an element $R_1$[`index`] is equal to the current value being searched for and there are still fake roots to be found, it means that the element $R_1$[`index`] is a fake root of $f$. When this condition is satisfied, the value of $R_1$[`index`] is set to the maximum possible value for a coefficient. When an element of the list is different from the current fake root being searched for, it means that the element is an original root, so its value is maintained.

If a fake root added to the polynomial $f$ is equal to an original root of $f$, the root appears twice in the list $R_1$. Algorithm 9 only removes the first one of the duplicates in this case because when a fake root is found in the list, the next fake root value is calculated. Thus, the original root is not removed. After all elements of $R_1$ are evaluated and all the fake roots receive the maximum coefficient value, the list is sorted again. Consequently. the original roots are ordered in the first $n$ elements of the list and the last $t-n$ elements are the ones with the maximum coefficient value. These elements are fake roots added to the input polynomial, therefore, they receive the given value $\zeta$ in Step **vii**.

**Algorithm 9** Removing fake roots — `RemoveRoots(R,t,n,step,`$\zeta$`)`

**Input:** a list $R$ of $t$ roots returned by Algorithm 8, including fake roots; the target degree $t$; the degree $n$ of the original polynomial; the value `step` of the increment between each fake root; the element $\zeta$ that is used to indicate the fake roots in $R$.

**Output:** a list $R_1$ of $t$ elements, with the $n$ roots of the original polynomial and $t-n$ elements $\zeta$.

```
i. Initialize a list $R_1$ of size $t$ and copy $R$ to $R_1$.
ii. Sort $R_1$.
iii. Initialize diff $\leftarrow t-n$, keepCoeff $\leftarrow 1$, numberFakeRoots $\leftarrow$ diff, noFakeRoots $\leftarrow$
     not(diff), maxCoeff $\leftarrow$ maximum value of a coefficient, fakeRoot $\leftarrow$ step $\oplus 1$.
iv. For index in $[0, t-1]$
    a. $r \leftarrow R_1$[index];
    b. eq $\leftarrow$ not($r \oplus$ fakeRoot);
    c. keepCoeff $\leftarrow$ not(not(noFakeRoots) and eq);
    d. $r \leftarrow$ keepCoeff $* r$;
    e. $R_1$[index] $\leftarrow r \oplus$ (not(keepCoeff) $*$ maxCoeff);
    f. fakeRoot $\leftarrow$ (keepCoeff $*$ fakeRoot) $\oplus$
       ((not keepCoeff) $*$ ($1 \oplus$ ((diff $-$ numberFakeRoots $+ 2$) $*$ step)));
    g. numberFakeRoots $\leftarrow$ numberFakeRoots $-$ not(keepCoeff);
    h. noFakeRoots $\leftarrow$ not(numberFakeRoots);
v. Sort $R_1$.
vi. Set keepCoeff $\leftarrow 1$.
vii. For index in $[0, t-1]$
    a. $r \leftarrow R_1$[index];
    b. keepCoeff $\leftarrow$ keepCoeff $-$ not(index $- n$);
    c. $R_1$[index] $\leftarrow r \oplus$ (not(keepCoeff) $*$ (maxCoeff $\oplus \zeta$));
viii. return $R_1$.
```

The output is a list of $t$ elements, with $n$ sorted roots in the first $n$ elements followed by $t-n$ elements $\zeta$. In the context of the McEliece cryptosystem, the value $\zeta$ must be an element that is not in the support set of the Goppa code, so the error vector created from the returned list of roots $R_1$ is generated correctly.

Algorithm 9 has its cost based on the sorting algorithm and the addition of coefficients. The number of iterations is $t$ for each “for" step, resulting in

$$C_{\text{remove\_roots}} = 2C_{\text{sort}} + t(6C_{\text{add}}) + C_{\text{add}}, \tag{4.3}$$

where $C_{\text{sort}}$ is the cost of the used sorting algorithm. The sorting Steps **ii.** and **v.** can be done in constant time using the *djbsort* constant-time sorting library (BERNSTEIN, 2019).

# 5 EXPERIMENTS AND RESULTS

In this section we compare several root-finding algorithms: the exhaustive method detailed in Section 3.1; the additive fast Fourier transform detailed in Section 3.2; the probabilistic method, showed in Algorithm 8; the iterative Berlekamp Trace Algorithm proposed by Martins et al. (MARTINS; BANEGAS; CUSTÓDIO, 2019) (BTA-it); the constant-time probabilistic algorithm (CTProb), showed in Algorithm 4; and the constant-time BTA-it (CTBTA-it), that is similar to CTProb but substitutes Algorithm 8 with BTA-it. The exhaustive method and the additive Fast Fourier Transform algorithm are deterministic algorithms used in the implementation of NIST post-quantum proposals.

Considering these algorithms, we focus on the following questions:

(i) Which of the root-finding algorithms above behave in practice as a constant-time algorithm? In other words, which algorithms will have approximately the same running time behavior regardless of the input degree for a fixed target degree?

(ii) Does the method of adding fake roots work well in practice to transform different algorithms that are not constant time into constant time algorithms when the input degree varies for a fixed target degree?

(iii) Which impact in time do different methods of adding fake roots have? More specifically, we consider two methods: (M1) adding fake roots one by one, via successive polynomial multiplication (method used in our extended abstract (MARCHIORI et al., 2021)); and (M2) adding fake roots using a multiplication tree as in Figure 3 (method used for all experiments in this thesis, showed in Algorithm 7).

(iv) Which of the constant time algorithms is more efficient for different ranges of parameters (finite field size $q$ and target degree $t$)?

In our experiments, we used an Intel® Core™ i5-3317U CPU @ 1.70GHz and the code was compiled with GCC version 10.2.0 with the compilation flags "`-march=native -mtune=native -O2 -Wall`". These results are from experiments with a constant-time sorting method and a constant-time implementation of the GCD algorithm.

## 5.1 ANALYSES OF THE EXECUTION TIME OF CTPROB AND CTBTA-IT

In this section we give answers to questions (i) and (ii). Figure 4 shows the box plot of the execution time in CPU cycles of CTProb and CTBTA-it when the target degree is 128 and the input are polynomials of degree 2 to 128. For each degree, 2500 random polynomials in $\mathbb{F}_{2^{13}}[x]$ were used as input. In this figure, we observe that the running time of both algorithms is insensitive to the degree of the input polynomial, contrasting with Figure 2. When the target

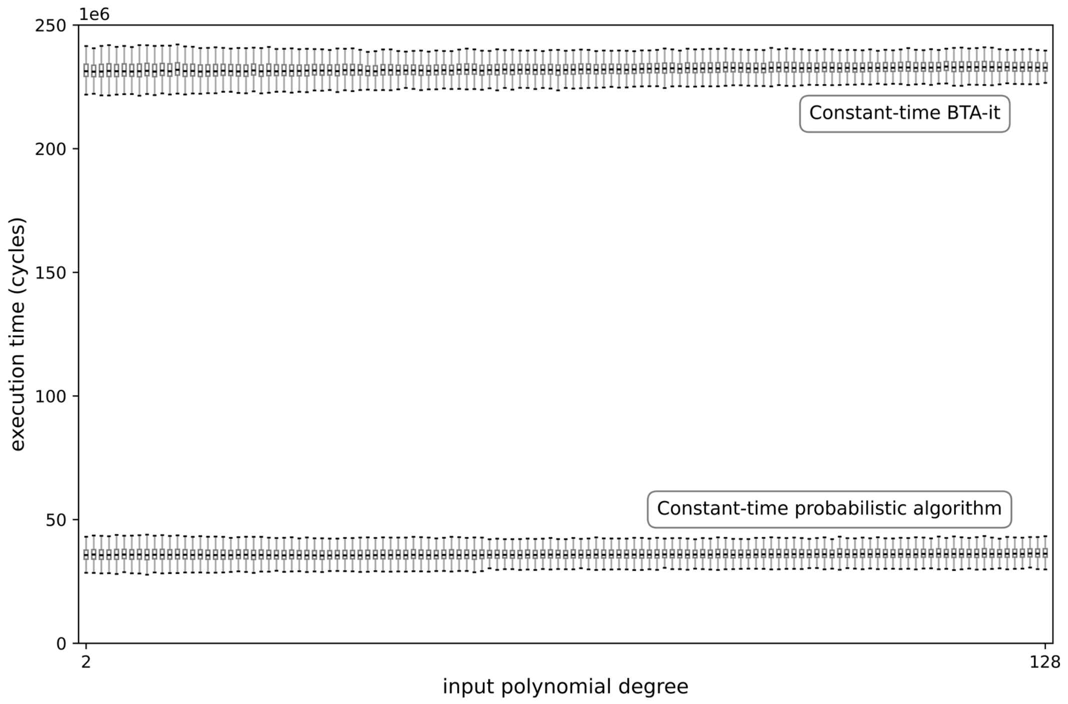

Figure 4 – Execution time in CPU cycles of the constant-time probabilistic method (CTProb) and CTBTA-it for input degrees 2 to 128 with target degree 128 and polynomials in $\mathbb{F}_{2^{13}}[x]$.

degree $t$ is fixed, the degree of the input polynomial does not impact the execution time of the algorithms.

Table 2 shows the number of CPU cycles for six different polynomial root-finding methods: the exhaustive method, the additive Fast Fourier Transform, BTA-it (MARTINS; BANEGAS; CUSTÓDIO, 2019), CTBTA-it, the probabilistic method, and CTProb. Each point shows the mean and margin of error for an interval of confidence at the 95% level calculated with a sample of 2500 measures. The field size considered in these executions is $\mathbb{F}_{2^{13}}$ and the set size is $n = 6688$, which are parameters suggested in a NIST post-quantum proposal (ALBRECHT et al., 2020). The values show the CPU cycles for each method with target degree $t = 128$ and degree of the input polynomial $d = \{64, 96, 128\}$.

Analyzing the values in Table 2 we can answer question (i). We observe that the exhaustive method, the additive FFT, CTBTA-it and CTProb show approximately the same running time behavior regardless of the input degree for a fixed target degree. That is, these algorithms behave in practice as a constant-time algorithm.

The exhaustive method evaluates $n$ points separately. Thus, for the small field considered in these tests ($q = 2^{13}$), its execution is slightly more efficient than CTProb. Its advantage is that it evaluates $n$ points regardless of the polynomial, so its execution time is not tied to the input polynomial degree. The additive FFT algorithm performs multipoint evaluation, resulting in a more efficient method than the exhaustive one. In small finite fields such as the one used in these experiments, it has the best performance of all methods analyzed; this changes for slightly larger fields, as we show later.

| **Target degree** | **128** | | | | | |
|---|---|---|---|---|---|---|
| **Input degree** | **64** | | **96** | | **128** | |
| **Algorithm** | $\mu$ | $\pm\varepsilon$ | $\mu$ | $\pm\varepsilon$ | $\mu$ | $\pm\varepsilon$ |
| Exhaustive | 32.0427 | 0.0294 | 32.1338 | 0.0304 | 32.3462 | 0.0361 |
| Additive FFT | 16.5810 | 0.1584 | 17.0102 | 0.1544 | 17.1382 | 0.1535 |
| BTA-it | 77.0296 | 0.1978 | 143.9288 | 0.2783 | 230.9652 | 0.2939 |
| Probabilistic | 13.5337 | 0.0957 | 23.8441 | 0.1446 | 32.6855 | 0.1976 |
| CTBTA-it | 231.7684 | 0.1993 | 232.3113 | 0.2426 | 233.0854 | 0.2250 |
| CTProb | 35.4109 | 0.2039 | 35.3324 | 0.3029 | 35.5263 | 0.3140 |

Table 2 – Mean ($\mu$) and margin of error ($\varepsilon$) for an interval with 95% of confidence of the execution time in $10^6$ cycles of CTProb and other root-finding methods for polynomials in $\mathbb{F}_{2^{13}}[x]$.

The iterative Berlekamp Trace Algorithm (BTA-it) and the original probabilistic algorithm show a significant variation with the increase of the input polynomial degree. With the use of our constant-time countermeasure, CTBTA-it and CTProb show less variable results. Therefore, answering question (ii), our method of adding fake roots works well in practice to transform different algorithms that are not constant time into constant time algorithms. However, the execution times of CTBTA-it and CTProb is increased when compared to the execution times of BTA-it and the original probabilistic algorithm because of the extra work introduced by adding and removing roots. Indeed, for the values in Table 2 in column 128, when comparing the execution time between the probabilistic method and CTProb, and BTA-it and CTBTA-it, they show approximately the same difference.

CTProb is more efficient than CTBTA-it even though the finite field is small, and its performance is comparable to the exhaustive method in these parameters. CTBTA-it shows the worse performance between the analyzed constant-time methods, even for such small parameters the method is costly.

Therefore, we observe that our hypothesis is answered positively since the probabilistic root-finding method we designed in Chapter 4, CTProb, is a constant-time probabilistic algorithm. Additionally, it performs better in bigger fields than other root-finding algorithms currently used in code-based cryptography, as detailed in Section 5.3.

## 5.2 ANALYSES OF DIFFERENT MULTIPLICATION METHODS WHEN ADDING FAKE ROOTS

In this section we answer question (iii). Table 3 shows the number of CPU cycles when adding fake roots using methods M1 and M2. We recall that M1 is the method that adds fake roots via successive polynomial multiplication (method used in our extended abstract (MARCHIORI et al., 2021)); and M2 is the method that adds fake roots using a multiplication tree (Algorithm 7). The mean and margin of error are presented for an interval of confidence at the 95% level calculated with a sample of 2500 measures. In addition, the percentage of improvement of the mean obtained by M2 over M1 is shown.

| $q$ | $t$ | **Method** | $\mu$ | $\pm\varepsilon$ | **Difference** |
|---|---|---|---|---|---|
| $2^{13}$ | 128 | M1 | 34.5328 | 0.2056 | $-2.9\%$ |
| | | M2 | 35.5263 | 0.3140 | |
| $2^{16}$ | 130 | M1 | 39.8711 | 0.3029 | 2.5% |
| | | M2 | 38.8918 | 0.3162 | |
| $2^{20}$ | 500 | M1 | 563.3925 | 1.5341 | 4.9% |
| | | M2 | 535.8684 | 0.8438 | |

Table 3 – Mean ($\mu$), margin of error ($\varepsilon$) and the difference of the mean between methods M1 and M2 in CTProb for an interval with 95% of confidence of the execution time in $10^6$ cycles.

We observe in Table 3 that, in smaller fields, such as field size $2^{13}$, M2 performs worse than M1, increasing the number of CPU cycles by 2.9%. However, the number of CPU cycles of M2 is 2.5% lower for field size $2^{16}$, and 4.9% lower for field size $2^{20}$. In conclusion, for smaller fields, adding fake roots using method M1 results in an improvement on the overall execution time. But using method M2 is recommended in bigger fields, given that it reduces the overhead caused by the addition of fake roots in CTProb in these situations.

## 5.3 COMPARISON OF CTPROB WITH OTHER ROOT-FINDING METHODS IN BIGGER FIELDS

In this section we answer question (iv). Although the degree of the input polynomial does not increase the execution time of the additive FFT algorithm, the size of the field does. Therefore additive FFT may not be the fastest algorithm in bigger fields. Also, in bigger fields, the value of $n$ increases and the performance of the exhaustive method may not be similar to CTProb for every parameter set. Since probabilistic factorization algorithms tend to perform well in bigger fields, we measured the execution times of CTProb, the additive FFT algorithm, and the exhaustive method in fields of size $q = 2^m, m = 13, 16, 20$ and identified the point in which the curves cross.

Comparing BTA-it to the probabilistic method in Figure 2, we observe that the higher the degree, the greater the advantage of the probabilistic method over BTA-it. Even for small fields, the probabilistic method is much faster than BTA-it, likewise, CTProb is much faster than CTBTA-it (see Table 2). Figure 2 shows that the gap between the execution time of the probabilistic method and BTA-it tends to widen as the degree $t$ increases. Therefore, as CTProb is always more efficient than CTBTA-it, it remains to compare CTProb with the exhaustive and additive FFT algorithms in bigger fields.

First, we analyze if there may be practical advantage of using CTProb for real code-based cryptosystems. Currently, no code-based cryptosystem has parameters for a field of sizes $2^{16}$ and $2^{20}$. However, considering the main attacks and the NIST security levels, we can outline some plausible values for $t$ and $n$ for bigger fields. To achieve NIST security level 5 the following values can be considered: for a field of size $2^{16}$, $t \sim 130$ and $n \sim 9000$; and for a field of size $2^{20}$, $t \sim 500$ and $n \sim 50000$. Therefore, these values for $t$ and $n$ were considered for the measurements.

Figures 5, 6, and 7 show the average number of CPU cycles of CTProb, the additive FFT algorithm, and the exhaustive method for some input and target degrees for field size $2^{13}$, $2^{16}$, and $2^{20}$, respectively. We observe that in field size $q = 2^{13}$, CTProb does not perform better than the additive FFT and the exhaustive method for the current parameters sets being used in code-based cryptography. However, for field size $q = 2^{16}$, when considering the aforementioned parameters set with $t = 130$, CTProb performs better than both other algorithms. The same occurs when $q = 2^{20}$ and $t = 500$.

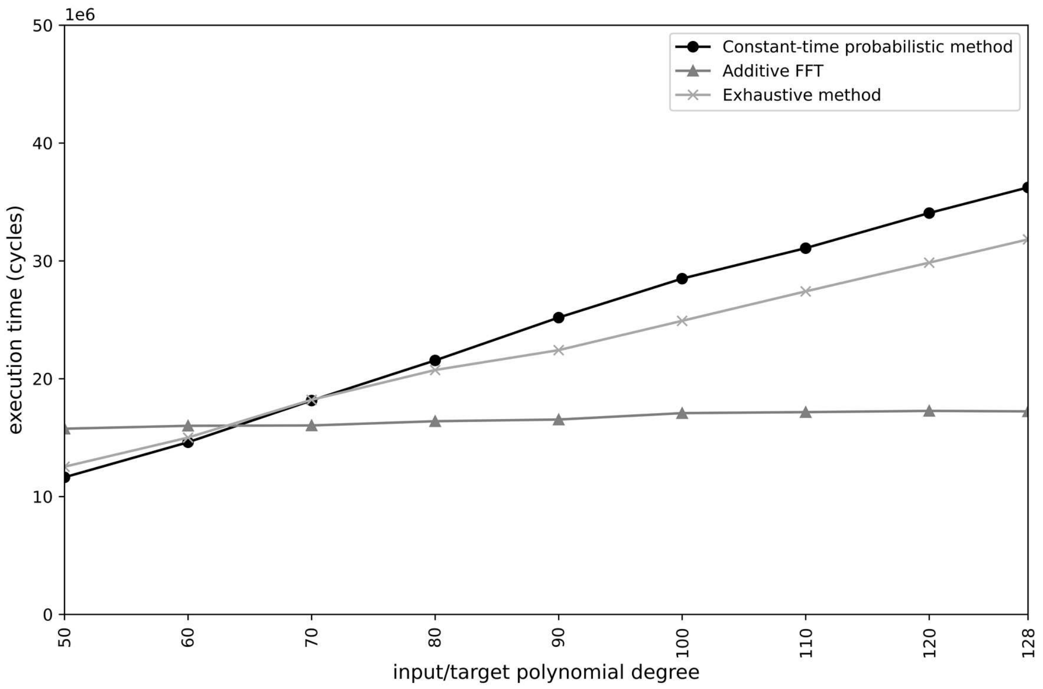

Figure 5 – Average number of CPU cycles of the constant-time probabilistic method (CTProb), the additive FFT algorithm, and the exhaustive method for polynomials in $\mathbb{F}_{2^{13}}[x]$.

These results show that the constant-time probabilistic algorithm proposed is a viable alternative to be used as a root-finding algorithm in a code-based cryptosystem in large fields. CTProb performs better than the additive FFT algorithm and the exhaustive method for values before the curves intersect, which includes the reasonable parameters aforementioned. For target degree values higher than the intersection, the additive FFT algorithm is a better alternative.

Close-ups on the intersection points are shown in Figure 8 for field size $q = 2^{13}$, Figures 9, 10 for field size $q = 2^{16}$ comparing with the additive FFT algorithm and the exhaustive

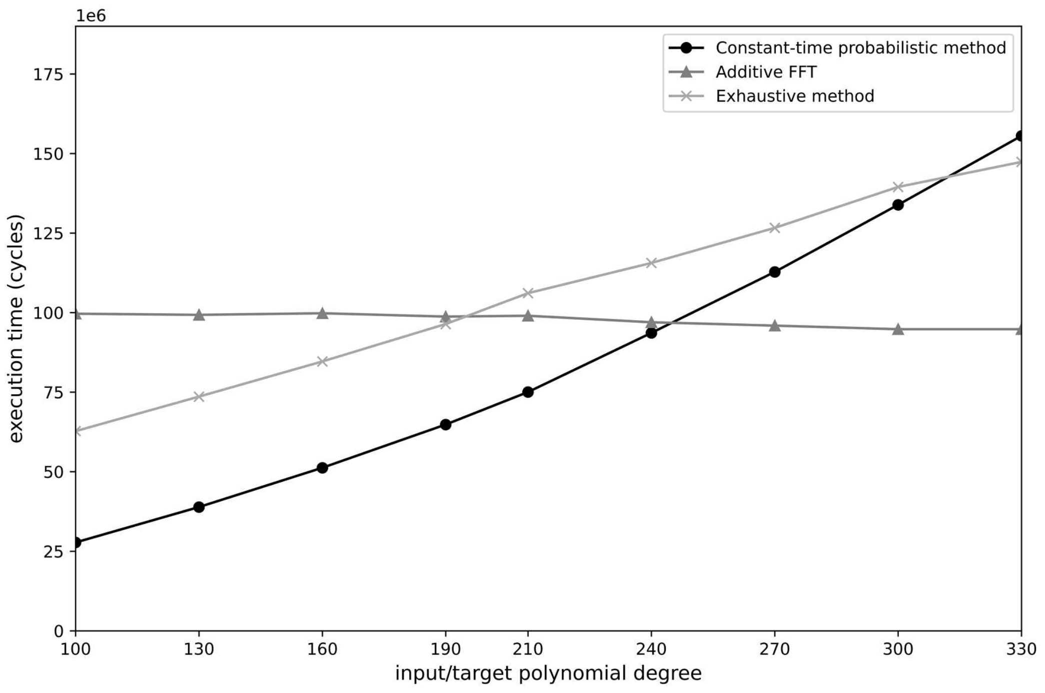

Figure 6 – Average number of CPU cycles of the constant-time probabilistic method (CTProb), the additive FFT algorithm, and the exhaustive method for polynomials in $\mathbb{F}_{2^{16}}[x]$.

method, respectively, and Figures 11, 12 for $q = 2^{20}$. The points in the x-axis represent the value of the input and target degree. The number of CPU cycles for CTProb increases when the target degree $t$ increases.

We observe that the point of intersection of the number of CPU cycles of CTProb and the other algorithms increases as the field size increases. Considering the median value and observing the interquartile range, in a field size $q = 2^{13}$, the intersection with the additive FFT occurs for degree 66, indicating that CTProb is faster in the range [1..65]. In $q = 2^{16}$, CTProb is faster in the range [1..253], and in $q = 2^{20}$, the range is [1..923]. For the exhaustive algorithm, in $q = 2^{13}$, CTProb is faster in the range [1..66]. In $q = 2^{16}$, the range is [1..328], and in $q = 2^{20}$, it is [1..1479]. In conclusion, we observe that for higher field sizes, CTProb is a great alternative to obtain faster but still secure root finding.

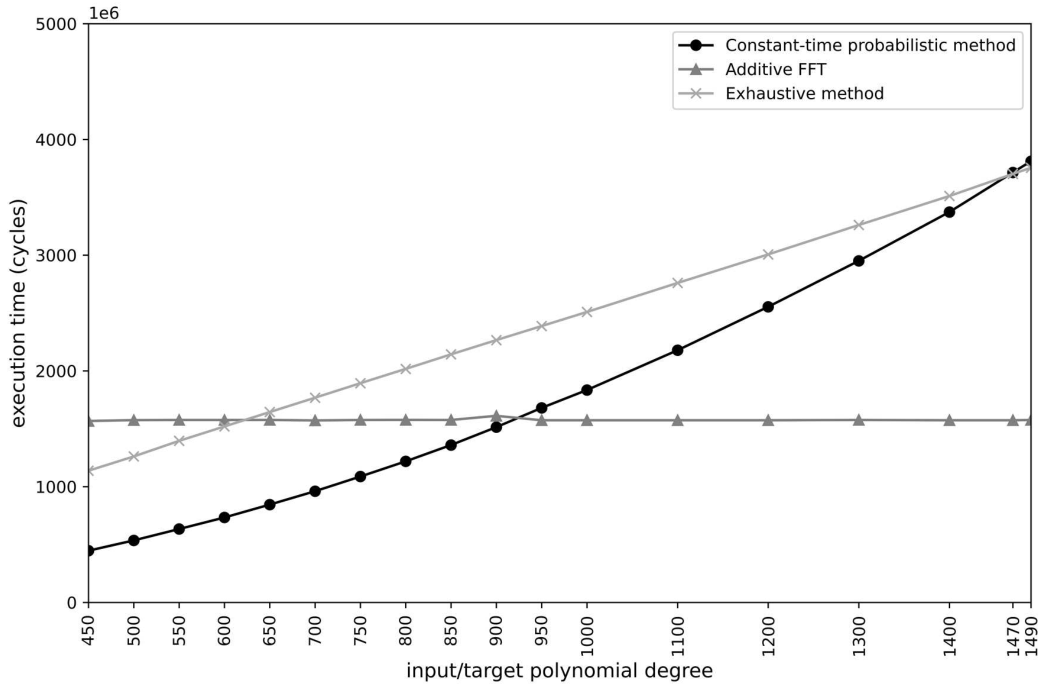

Figure 7 – Average number of CPU cycles of the constant-time probabilistic method (CTProb), the additive FFT algorithm, and the exhaustive method for polynomials in $\mathbb{F}_{2^{20}}[x]$.

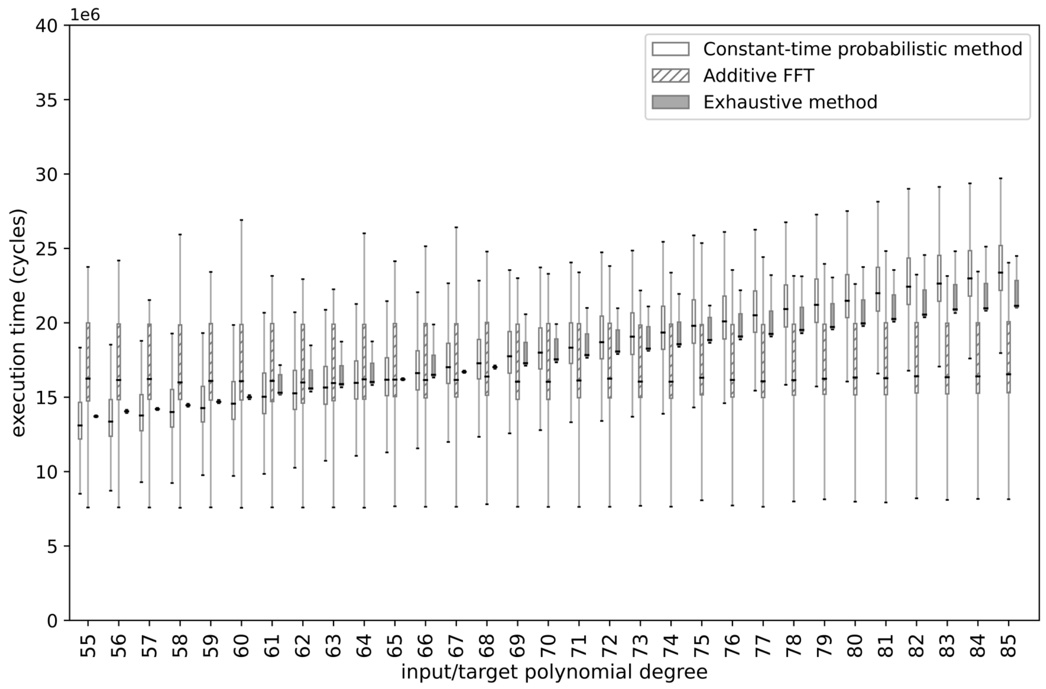

Figure 8 – Comparison between the number of CPU cycles of the constant-time probabilistic method (CTProb), the additive FFT, and the exhaustive method for finding roots of polynomials in $\mathbb{F}_{2^{13}}[x]$.

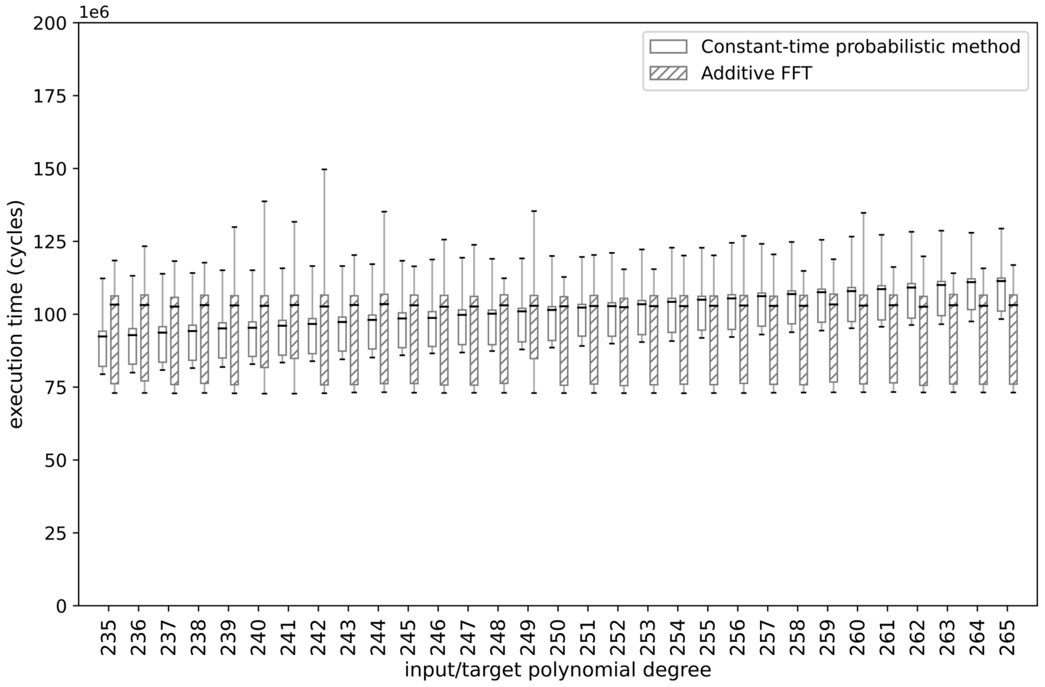

Figure 9 – Comparison between the number of CPU cycles of the constant-time probabilistic method (CTProb) and the additive FFT for finding roots of polynomials in $\mathbb{F}_{2^{16}}[x]$.

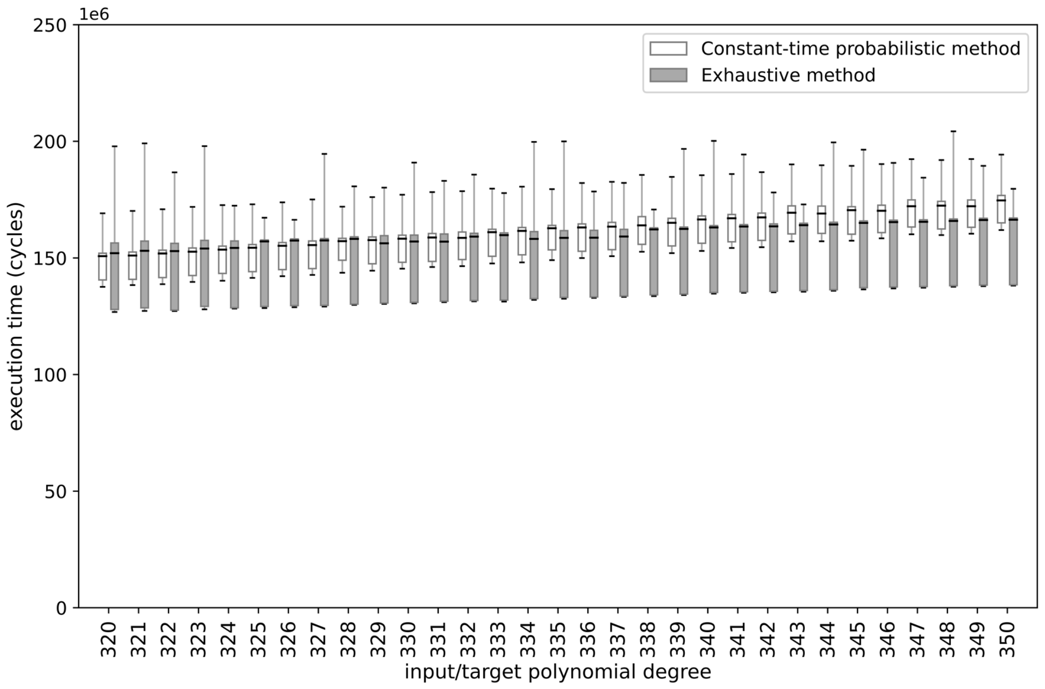

Figure 10 – Comparison between the number of CPU cycles of the constant-time probabilistic method (CTProb) and the exhaustive method for finding roots of polynomials in $\mathbb{F}_{2^{16}}[x]$.

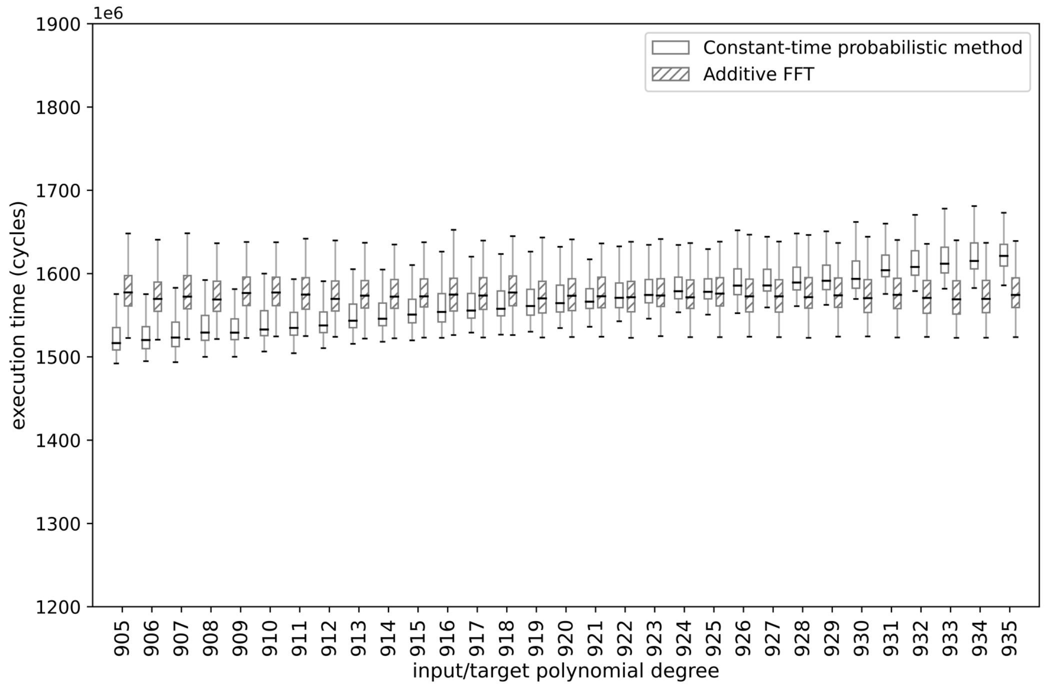

Figure 11 – Comparison between the number of CPU cycles of the constant-time probabilistic method (CTProb) and the additive FFT for finding roots of polynomials in $\mathbb{F}_{2^{20}}[x]$.

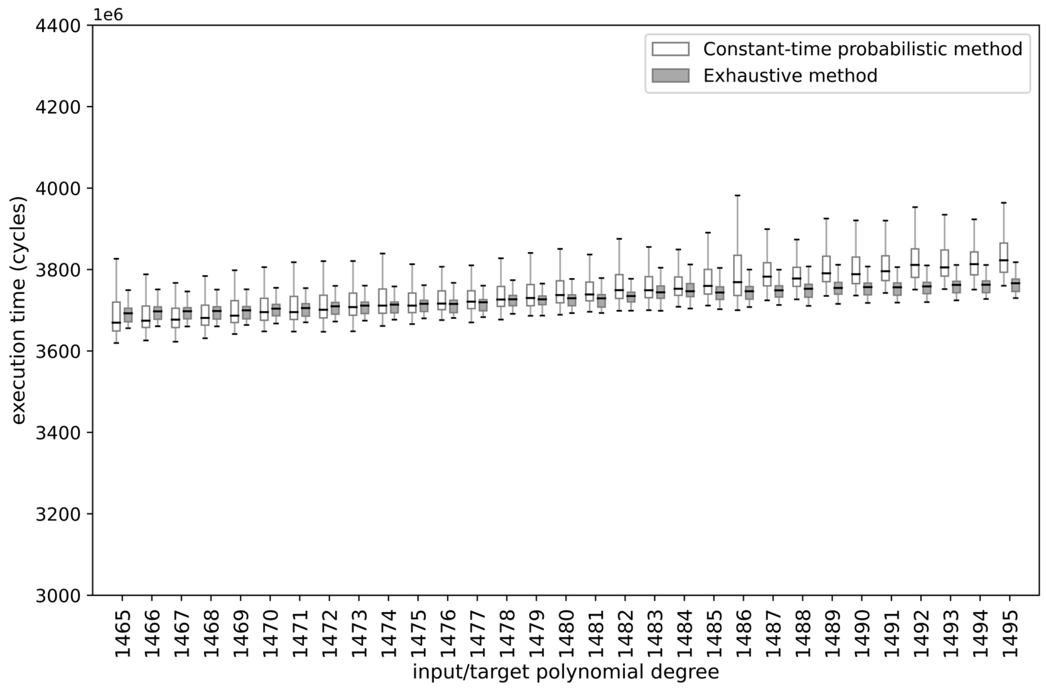

Figure 12 – Comparison between the number of CPU cycles of the constant-time probabilistic method (CTProb) and the exhaustive method for finding roots of polynomials in $\mathbb{F}_{2^{20}}[x]$.

# 6 CONCLUSION

In this thesis, we propose a countermeasure to a probabilistic root-finding algorithm to prevent the leakage of information due to the variation of execution time according to different polynomial degrees. Our approach is a general method for artificially increasing the degree of the input polynomial to a target degree expected by the cryptosystem as a countermeasure to avoid timing side-channel attacks that exploit information on the input polynomial degree. The method increases the degree of the input polynomial by adding fake roots to it and then removing the fake roots of the output of the root-finding method, and also making the individual steps of the algorithm constant time. The implementation can be done without branching, memory-access and non-constant operand leakage, resulting in a constant-time method.

These countermeasure is shown to work with other root-finding algorithms as well, like BTA-it (MARTINS; BANEGAS; CUSTÓDIO, 2019). We verify the practical effect in the total running time not only for a probabilistic root-finding algorithm but also for BTA-it. Our constant-time probabilistic proposal, CTProb, shows a constant behavior in our experiments when expecting an input polynomial of degree $t$ and receiving a polynomial of degree $n \leq t$, as well as our experiment using this technique for making BTA-it constant time (CTBTA-it).

Considering the efficiency of CTProb in large field sizes, we experimentally determine field sizes and target degrees for which CTProb becomes more efficient for finding roots than other deterministic algorithms, such as an exhaustive algorithm and the additive Fast Fourier Transform algorithm. Also, we analyze how the new constant-time algorithms (CTProb and CTBTA-it) compare with existing algorithms when the field size increases, where we observe that CTProb is always faster than CTBTA-it. Additionally, we give examples of plausible parameters that can be considered for code-based cryptography in larger fields, where we observe that CTProb is much faster than other deterministic root-finding methods.

Moreover, the impact in execution time of different multiplication methods for adding fake roots is analyzed. We observe that each method improves the efficiency for adding fake roots for different fields sizes. Therefore, the multiplication method must be chosen wisely according to the cryptosystem parameters.

Concluding, CTProb is faster than the exhaustive method used in Classic McEliece (ALBRECHT et al., 2020), the additive Fast Fourier Transform algorithm, and BTA-it (MARTINS; BANEGAS; CUSTÓDIO, 2019) in the specified ranges for field sizes $2^{16}$ and $2^{20}$. These ranges include plausible parameters for larger fields in code-based cryptography, for which CTProb is the most efficient method when compared to other root-finding algorithms currently used in code-based cryptosystems. Therefore, CTProb can be a useful alternative for root finding when considering long-term security, since it provides some savings for larger parameters.

## 6.1 FUTURE WORK

We remark that our work does not include some optimizations for operations in finite fields, like vectorization and bitslicing, which may improve the execution time of our algorithms. Moreover, all experiments were done in only one computer architecture, and testing on different architectures could be interesting to verify that our implementation remains constant in many platforms.

In order to achieve a safer implementation, different attack scenarios must be taken into consideration. Our focus was on timing side-channel attacks, that are one of the most common side-channel attacks applied to code-based cryptosystems in the literature. However, it would be interesting to analyze other types of side-channel attacks, like power analysis attacks.

Additionally, other approaches can be explored in our proposal, such as the use of a fixed value for the variable $j$ in Algorithm 4 or other different methods for the multiplication of the fake roots.

# BIBLIOGRAPHY

ALBRECHT, M. et al. NTS-KEM. **NIST PQC Round**, v. 2, 2019.

ALBRECHT, M. R. et al. Classic McEliece: conservative code-based cryptography. **NIST PQC Round**, v. 3, 2020.

ALMEIDA, J. B. et al. Verifying constant-time implementations. In: **25th USENIX Security Symposium (USENIX Security 16)**. [S.l.: s.n.], 2016. p. 53–70.

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

BERLEKAMP, E. R. Factoring polynomials over finite fields. **Bell System Tech. J.**, v. 46, p. 1853–1859, 1967. ISSN 0005-8580.

BERLEKAMP, E. R. **Algebraic Coding Theory**. New York: McGraw-Hill Book Co., 1968. xiv+466 p.

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

BERNSTEIN, D. J. **djbsort library**. 2019. Https://sorting.cr.yp.to/index.html.

BERNSTEIN, D. J.; BUCHMANN, J.; DAHMEN, E. **Post-quantum cryptography**. 1. ed. Berlin, Heidelberg: Springer, 2009.

BERNSTEIN, D. J.; CHOU, T.; SCHWABE, P. Mcbits: Fast constant-time code-based cryptography. In: BERTONI, G.; CORON, J.-S. (Ed.). **Cryptographic Hardware and Embedded Systems - CHES 2013**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 250–272.

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

BISWAS, B. **Implementational aspects of code-based cryptography**. Tese (Doutorado) — Ecole Polytechnique X, 2010.

BISWAS, B.; HERBERT, V. **Efficient Root Finding of Polynomials over Fields of Characteristic 2.** 2009. Hal-00626997.

CANTOR, D. G.; ZASSENHAUS, H. A new algorithm for factoring polynomials over finite fields. **Mathematics of Computation**, v. 36, n. 154, p. 587–592, 1981.

CAYREL, P.-L.; HOFFMANN, G.; PERSICHETTI, E. Efficient implementation of a CCA2-secure variant of McEliece using generalized Srivastava codes. In: SPRINGER. **International Workshop on Public Key Cryptography**. [S.l.], 2012. p. 138–155.

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

CHOU, T. Mcbits revisited. In: SPRINGER. **International Conference on Cryptographic Hardware and Embedded Systems**. [S.l.], 2017. p. 213–231.

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

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

FLAJOLET, P.; GOURDON, X.; PANARIO, D. Random polynomials and polynomial factorization. In: MEYER, F.; MONIEN, B. (Ed.). **Automata, Languages and Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. p. 232–243. ISBN 978-3-540-68580-7.

FLAJOLET, P.; GOURDON, X.; PANARIO, D. The complete analysis of a polynomial factorization algorithm over finite fields. **J. Algorithms**, v. 40, n. 1, p. 37–81, 2001. ISSN 0196-6774.

GAO, S.; MATEER, T. Additive fast Fourier transforms over finite fields. **IEEE Transactions on Information Theory**, IEEE, v. 56, n. 12, p. 6265–6272, 2010.

GATHEN, J. von zur; PANARIO, D. Factoring polynomials over finite fields: a survey. **J. Symbolic Comput.**, v. 31, n. 1-2, p. 3–17, 2001. ISSN 0747-7171. Computational algebra and number theory (Milwaukee, WI, 1996).

GATHEN, J. von zur; SHOUP, V. Computing Frobenius maps and factoring polynomials. **Computational complexity**, Springer, v. 2, n. 3, p. 187–224, 1992.

GOPPA, V. D. A new class of linear correcting codes. **Problemy Peredachi Informatsii**, Russian Academy of Sciences, Branch of Informatics, Computer Equipment and Automatization, v. 6, n. 3, p. 24–30, 1970.

HUFFMAN, W. C.; PLESS, V. **Fundamentals of Error-Correcting Codes**. United States of America: Cambridge University Press, 2003.

JOCHEMSZ, E. Goppa codes & the McEliece cryptosystem. **Vrije Universiteit Amsterdam**, p. 63, 2002.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering, EBSE technical report EBSE-2007-01. **Keele University, Keele, UK**, 2007.

MACWILLIAMS, F.; SLOANE, N. **The Theory of Error-Correcting Codes**. 3. ed. Netherlands: North-Holland, 1981. v. 16.

MARCHIORI, D. et al. Towards constant-time probabilistic root finding for code-based cryptography. In: **Anais do XXI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2021. p. 155–168. ISSN 0000-0000.

MARTINS, D.; BANEGAS, G.; CUSTÓDIO, R. Don't forget your roots: Constant-time root finding over $\mathbb{F}_{2^m}$. In: SCHWABE, P.; THÉRIAULT, N. (Ed.). **Progress in Cryptology – LATINCRYPT 2019**. [S.l.]: Springer International Publishing, 2019. p. 109–129.

MCELIECE, R. J. A public-key cryptosystem based on algebraic coding theory. **The Deep Space Network Progress Report**, v. 42-44, p. 114–116, 1978.

MELCHOR, C. A. et al. Hamming Quasi-Cyclic (HQC). **NIST PQC Round**, v. 3, 2021.

National Institute of Standards and Technology. **Post-Quantum Cryptography**. 2022. Https://csrc.nist.gov/Projects/post-quantum-cryptography.

NIEDERREITER, H.; XING, C. **Algebraic geometry in coding theory and cryptography**. 1. ed. EUA: Princeton University Press, 2009.

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

RABIN, M. 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**, ACM, v. 21, n. 2, p. 120–126, 1978.

SCHIPANI, D. **Efficient decoding of cyclic codes and applications in cryptography**. Tese (Doutorado) — University of Zurich, 2012.

SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IEEE. **Proceedings 35th Annual Symposium on Foundations of Computer Science**. [S.l.], 1994. p. 124–134.

SHOUFAN, A. et al. A timing attack against patterson algorithm in the McEliece PKC. In: SPRINGER. **Information, Security and Cryptology – ICISC 2009**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 161–175.

STALLINGS, W. **Cryptography and network security**. 6. ed. Estados Unidos: Pearson Education, 2014.

STRENZKE, F. Fast and secure root finding for code-based cryptosystems. In: SPRINGER. **Cryptology and Network Security**. Berlin, Heidelberg, 2012. p. 232–246.

STRENZKE, F. et al. Side channels in the McEliece PKC. In: SPRINGER. **Post-Quantum Cryptography**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 216–229.

SUMI, T.; MOROZOV, K.; TAKAGI, T. Efficient implementation of the McEliece cryptosystem. In: **Computer Security Symposium 2011 (CSS2011)**. [S.l.: s.n.], 2011. p. 582–587.

VANSTONE, S. Responses to NIST's proposal. **Communications of the ACM**, ACM, v. 35, n. 7, 1992.

WAFO-TAPA, G. et al. A practicable timing attack against HQC and its countermeasure. **Advances in Mathematics of Communications**, 2019. Disponível em: https://eprint.iacr.org/2019/909.pdf.

WALTERS, M.; ROY, S. S. Constant-time BCH error-correcting code. In: **2020 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2020. p. 1–5.

WANG, Y. Decoding generalized Reed-Solomon codes and its application to RLCE encryption schemes. **arXiv preprint arXiv:1702.07737**, 2017.

ZINOVIEV, V. On the solution of equations of degree $\leq 10$ over finite fields GF($2^q$). **Rapports de recherche- INRIA**, Institut national de recherche en informatique et en automatique, 1996.