Towards constant-time probabilistic root finding for code-based cryptography

Dúnia Marchiori[1], Ricardo Custódio[1], Daniel Panario[2], Lucia Moura[3]

[1]Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Santa Catarina, Florianópolis, Brazil

[2]School of Mathematics and Statistics
Carleton University, Ottawa, Canada

[3]School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada

`dunia.marchiori@posgrad.ufsc.br, ricardo.custodio@ufsc.br,`

`daniel@math.carleton.ca, lmoura@uottawa.ca`

Abstract. *In code-based cryptography, deterministic algorithms are used in the root-finding step of the decryption process. However, probabilistic algorithms are more time efficient than deterministic ones for large fields. These algorithms can be useful for long-term security where larger parameters are relevant. Still, current probabilistic root-finding algorithms suffer from time variations making them susceptible to timing side-channel attacks. To prevent these attacks, we propose a countermeasure to a probabilistic root-finding algorithm so that its execution time does not depend on the degree of the input polynomial but on the cryptosystem parameters. We compare the performance of our proposed algorithm to other root-finding algorithms already used in code-based cryptography. In general, our method is faster than the straightforward algorithm in Classic McEliece. The results also show the range of degrees in larger finite fields where our proposed algorithm is faster than the Additive Fast Fourier Transform algorithm.*

1. Introduction

Shor's algorithm [Shor 1994] was responsible for a change in what we currently consider digitally secure. It shows that quantum computers can calculate the factorization of integers in polynomial time, which is the problem that the security of RSA [Rivest et al. 1978] is based upon, the most common asymmetric algorithm used until now. Shor also shows a similar result for computing discrete logarithms in polynomial time; the security of several cryptosystems, including the ones based on elliptic curves [Cohen et al. 2006], rely on this operation.

Considering the growth in research and development of quantum computers, data encrypted today may not be secure in the future, when an adversary can have access to a quantum computer. As a way of preventing that, researchers in the field of post-quantum cryptography study cryptographic schemes in which their security is based on problems that are not affected by quantum computation. The proposals in post-quantum cryptography include cryptosystems based on hash functions, lattices, and error-correcting codes, among other areas.

McEliece's scheme [McEliece 1978] was the first asymmetric cryptosystem based on error-correcting codes and has resisted attacks for more than 40 years, thus being a great candidate for a post-quantum cryptosystem. While the security of the cryptosystem is based on the hardness of generic decoding, an NP-hard problem, the scheme implementation may not be secure. Side-channel attacks use additional information besides the original message and the ciphertext during encryption or decryption, and must be avoided. The extra information can be power consumption statistics or the execution time of the operations. Side-channel attacks that use time information are called timing side-channel attacks.

In code-based cryptography, timing side-channel attacks on the decryption process mostly 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 in [Strenzke et al. 2008, Shoufan et al. 2010] and detailed in Section 2.3. Considering this type of attacks, 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. In other words, we aim to obtain a method that allows a so-called *constant-time implementation*, which is an implementation where time variations are not correlated to secret information [Pornin 2018].

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 and Mateer 2010]. Both have constant-time behavior for any input polynomial, regardless of the input degree. In [Martins et al. 2019] some countermeasures are proposed for other deterministic algorithms. However, countermeasures for probabilistic algorithms are not included.

When considering the evolution of quantum computers and long-term security in code-based cryptosystems, we must consider an increase in field size and polynomial degrees. 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.

In code-based cryptosystems, probabilistic algorithms appear in [Schipani 2012] and [Sumi et al. 2011], but 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. Our work is the first to propose and experiment with a constant-time probabilistic algorithm with the goal of it being considered suitable for a code-based cryptosystem and resistant to the side-channel attacks mentioned.

Contributions of this paper: The main result of this paper is the proposal of a constant-time probabilistic algorithm for finding the roots of a polynomial. We compare our proposal with other root-finding algorithms used in code-based cryptography and show parameters in which the algorithm proposed require fewer CPU cycles than other deterministic constant-time algorithms. In general, our method is always faster than an exhaustive algorithm and, in the given ranges, it is faster than the Additive Fast Fourier Transform algorithm in larger finite fields.

Structure of this paper: In Section 2, we give a brief description of Goppa codes and the McEliece cryptosystem, and discuss a timing side-channel attack. In Section 3, we present the probabilistic algorithm which our proposal is based upon. In Section 4, we describe our proposal, showing the algorithms and their costs. In Section 5, we give experiments comparing our proposed algorithm against other algorithms currently used. Finally, in Section 6 we conclude our work and mention open problems.

2. Preliminaries

In this section, we provide a brief description of Goppa codes [Goppa 1970] and the McEliece cryptosystem, focusing on aspects that are relevant to our work. A timing side-channel attack over the decryption process of the McEliece cryptosystem is described, which our proposal in Section 4 prevents.

2.1. Goppa codes

A Goppa polynomial $g(x)$ is a polynomial of degree t and coefficients over $\mathbb{F}_{q^m}$ for m a positive integer, that is, $g(x) = \sum_{j=0}^{t} g_j x^j$, where $g_j \in \mathbb{F}_{q^m}$. Let $L = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_{q^m}^n$ 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

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

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

Goppa codes are linear codes with parameters consisting of length n, dimension k and minimum distance d. Here, n is the length of the codewords that is defined by L. The dimension k satisfies $k \geq n - mt$ and the minimum distance d satisfies $d \geq t + 1$ [Goppa 1970]. When condition (1) is not satisfied, the received message c' contains errors, that is, $c' = c \oplus e$, $c \in \mathbb{F}_q^n$ being a codeword and $e \in \mathbb{F}_q^n$ being a vector with weight $w(e) \leq t$. To identify and corrects the errors, the Error Locator Polynomial (ELP) $\sigma(x)$ plays a central role. Let $T_e = \{i : e_i \neq 0\}$, the ELP is given by

$$\sigma(x) = \prod_{i \in T_e} (x - \alpha_i) \in \mathbb{F}_{q^m}[x]. \tag{2}$$

The roots of the error locator polynomial indicate the position of the errors in the received message. To construct the ELP and correct the errors to retrieve the codeword in the message, the Patterson decoding algorithm [Patterson 1975] is used.

2.2. McEliece cryptosystem

In this section, we give a brief description of each algorithm that forms the McEliece cryptosystem: key generation, encryption, and decryption [McEliece 1978].

Key generation: In the original proposal, binary Goppa codes are used to construct the keys. First, a Goppa polynomial $g(x)$ of degree t and coefficients in $\mathbb{F}_{2^m}$ is 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. Finally, the matrices are multiplied creating the matrix $G' = SGP$ that is part of the public key $PK = (G', t)$. The private key is formed by $SK = (g, L, S, G, P)$.

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

Decryption: The decryption consists of correcting the errors added during the encryption of the message. Patterson decoding algorithm [Patterson 1975], shown in Algorithm 1, is used in this process. 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 $c \oplus e$.

Algorithm 1 Patterson Algorithm

```
Input: a ciphertext $c$ of length $n$, private key $PR = (g, L, S, G, P)$.
Output: message $M$.
  i. Compute the syndrome polynomial $S_c(x) \leftarrow \sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} \pmod{g(x)}$.
  ii. Compute $\tau(x) \leftarrow \sqrt{S(x)^{-1} + x} \pmod{g(x)}$.
  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 $e$ from the roots of $\sigma(x)$ as $e \leftarrow (\sigma(\alpha_1), \ldots, \sigma(\alpha_n)) \oplus (1, \ldots, 1)$, $\alpha_i \in L$.
  vi. Compute the plaintext $M \leftarrow c \oplus e$.
  vii. Return $M$.
```

In Step **v.** the roots of the ELP can be obtained using different methods. This root-finding step is the focus of our work. Our main contribution is a constant-time probabilistic algorithm for the problem based on a variant of the seminal work of Berlekamp [Berlekamp 1970]. We consider and analyze this variant and compare it against an exhaustive search that evaluates the polynomial repeatedly, and against the additive Fast Fourier Transform (FFT). These are some of the most popular algorithms in cryptography used for this task.

2.3. Timing side-channel attacks

In general, side-channel attacks use additional information besides the original message and the ciphertext to attack the system. The extra 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 in [Strenzke et al. 2008, Shoufan et al. 2010] exploits how the time variance between different executions of the evaluation of the error locator polynomial can reveal the original message. The attack described is for the root-finding step in the decoding algorithm of a Goppa code. The goal is to retrieve the original message by obtaining the error vector. This can be possible if the execution time of the decoding process varies according to the number of roots of the ELP.

The attack explores the difference between the execution time of the root-finding step of a message that contains t errors and other that contains w errors. If a message contains less than t errors, the error locator polynomial $\sigma(x)$ has $w < t$ roots, and if a message contains more than t errors, the polynomial $\sigma(x)$ has $w > t$ roots.

As a result of this property, the attack is designed as follows: in possession of a ciphertext c with t errors, the attacker inverts one bit of c, obtaining c'. Observing the execution time during the decryption of c', it is possible to infer if c' has $t-1$ or $t+1$ errors. Given that information, when the polynomial $\sigma(x)$ has $t-1$ roots, it can result in a faster execution. Performing this strategy to each bit of c, the attacker can discover which positions of c contain errors, making it possible to retrieve the original message in c.

A plain implementation of the McEliece cryptosystem is susceptible to an adaptive chosen-ciphertext attack (CCA2) like the one above. It has been established that this attack is not prevented by a CCA2 conversion [Strenzke et al. 2008]. Therefore, to maintain the security of the scheme, 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. Based on this idea, we propose the first constant-time probabilistic algorithm for finding the roots of a polynomial. This algorithm is provided in Section 4. A constant-time implementation should prevent non-constant time operations correlated to the secret. The types of operations to be concerned are branching (conditional jumps), memory accesses, or use of variable-time operations (integer divisions, shifts and rotations) [Pornin 2018]. These type of operations must not be used in connection with secret information. An implementation of our proposal can avoid branching and memory-access leakage and only use basic operations that run in constant time.

In [Martins et al. 2019], proposals are given to avoid timing side-channel attacks over the root-finding algorithm in the decoding step. The work describes countermeasures for four methods: exhaustive search, linearized polynomials, Berlekamp Trace Algorithm (BTA), and the Successive Resultant Algorithm. Our proposal is based on the works of Berlekamp as is the Berlekamp Trace Algorithm. The proposal for BTA in [Martins et al. 2019] results 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 receiving an input of degree t and receiving an input of degree $d < t$, as we do in this work. Our proposal focuses on a more constant behavior when comparing the number of CPU cycles it takes to find the roots of polynomials of different degrees $d < t$ when expecting t errors in the McEliece cryptosystem.

3. Root-finding algorithms

In this section, we introduce the probabilistic algorithm that our proposal is based upon. Like the exhaustive method and the Additive Fast Fourier Transform, other methods used currently are deterministic algorithms. As mentioned, the root-finding algorithm used during the decryption process must not present variations in execution time that may leak information related to any secrets. The previously considered methods succeed in this aspect, whereas an immediate implementation of the classical probabilistic root-finding algorithms in the literature [Berlekamp 1970, Rabin 1980, Cantor and Zassenhaus 1981, von zur Gathen and Shoup 1992] 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. A probabilistic root-finding algorithm

As it is well documented, the asymptotically fastest algorithms for factoring polynomials over finite fields are probabilistic. Indeed, even for middle size polynomial degrees or finite field orders, these probabilistic methods behave better. For a complete bibliography (up to the time of publication) in the classical problem of factoring polynomials over finite fields see the survey [von zur Gathen and Panario 2001].

Our proposed probabilistic method has its root in the fundamental works of Berlekamp [Berlekamp 1968, Chapter 6; Berlekamp 1967, Berlekamp 1970]. In particular, the idea of using probabilistic algorithms and traces when finding roots in characteristic 2 are originated in his works. The general factoring strategy is based on 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 $d = 1$. Algorithm 2 shows the steps of the equal-degree factorization for a polynomial over $\mathbb{F}_q$, $q = 2^m$, and common degree of the irreducible factors equal to $d = 1$. These are the parameters we are interested in this paper.

Algorithm 2 Equal-degree factorization algorithm for polynomials in $\mathbb{F}_{2^m}$ and degree $d = 1$ for the irreducible factors

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 $g \in \mathbb{F}_q[x]/(f)$ 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.   Recursively factor $p_1$ and $p_2$ and return the union of these two lists of factors.
```

The execution time of Algorithm 2 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 et al. 1996, Section 7] and [Flajolet et al. 2001, Section 5]), but the special case of Algorithm 2 simplifies considerably since we only need the third stage, and only for degree 1; see [Flajolet et al. 2001, Table 1].

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

We recall that the trace of an element $\alpha \in \mathbb{F}_{2^m}$ over $\mathbb{F}_2$ is $\mathrm{Tr}_{\mathbb{F}_{2^m}/\mathbb{F}_2}(\alpha) = \alpha + \alpha^2 + \alpha^{2^2} + \cdots + \alpha^{2^{m-1}}$. Step **iii** implements the trace computation of α, while Step **iv** attempts to improve the factorization towards the complete factorization in roots. These are ideas from Berlekamp and our starting point.

The random polynomial choice adds a probability of failure to the algorithm since a random polynomial can potentially result in a trivial factor of the input polynomial. This probability is $1-(1/2^{t-1})$, where t is the polynomial degree [von zur Gathen and Panario 2001]. This shows that the chance of having to retry Steps **ii.**, **iii.**, **iv.** of Algorithm 2 in order to obtain a valid factor (root) is very small unless the input polynomial has a very small degree.

The execution time of Algorithm 2 is extremely tied to the degree of the input polynomial. In this case, if the original algorithm is used in a code-based cryptosystem, an attacker can 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 δ is a positive integer smaller than t. We propose a countermeasure in Section 4 in order to achieve a constant-time algorithm, so that the variation in the execution time is not related to any secret information, like the polynomial degree.

4. Constant-time probabilistic root-finding algorithm

In order to prevent the attack mentioned in Section 2.3, an implementation of a root-finding algorithm in the decryption process must be constant-time. Therefore, countermeasures for the implementation of the standard equal-degree algorithm were evaluated so that its execution time is not related to any secret information. This includes avoiding branching, memory-access and operand leakages.

As commented above, the execution time of the original algorithm detailed in Algorithm 2 is tied to the degree of the input polynomial. When decoding a word with $t-\delta$ errors, the error locator polynomial is of degree $t-\delta$. The ideal behavior would be that the root-finding process of a polynomial of degree t and of degree $t-\delta$ lasts about the same time so there are no variations in the duration of the decoding process that an attacker can exploit.

We propose a countermeasure that, when receiving a polynomial of degree smaller than expected, fake roots are added to the polynomial so that the algorithm always takes the time that is expected to. The root-finding algorithm receives the polynomial with fake roots as input, and after computing all the roots, the fake ones are removed from the set of roots. This is shown in Algorithm 3. The fake roots added are created with multiples of `step`. In Step **i.**, the value of `step` is defined considering that the maximum value that a fake root can take is $((t-1) * \texttt{step}) \oplus 1$. In our implementation, the coefficient values are treated as integers. Step **ii.** is detailed in Algorithm 4 and is responsible for adding the fake roots to the polynomial. Step **iii.** is shown in Algorithm 5 and is an adaptation of Algorithm 2. The removal of the fake roots is done in Step **iv.**. The list returned in Step **iii.** is analysed and the fake roots are substituted by the given element ζ. The returned list is sorted and all the n original roots are in the first n positions of the list, followed by $t-n$ positions containing ζ. This is detailed in Algorithm 6.

In order for these processes to occur in constant time, the branches taken and memory addresses accessed must be independent of any secret inputs, and operations that can leak information, like integer divisions, must be avoided. The operands of the division in Step **i.** in Algorithm 3 are public cryptosystem parameters, so no secret can be leaked from this operation. Also, in this context, the same amount of operations must be done on every possible input, regardless of the input degree, since it is secret information. Even when receiving a polynomial of the expected degree, extra work must be done so every execution lasts about the same. The expected degree in this context is the number of errors that the cryptosystem can correct and that is added to the message.

Algorithm 3 Constant-time probabilistic root-finding algorithm(f, ζ)

Input: 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 $t - n$ zeros.

```
i.   step ← ((maximum field element)/$t$) − 1;
ii.  $g$ ← AddRoots(f,step);
iii. $R$ ← IterativeRootFinding(g);
iv.  $R$ ← RemoveRoots(R,t,n,step,$\zeta$);
v.   Return $R$.
```

The creation and addition of fake roots are shown in Algorithm 4. The goal is to multiply fake linear factors to f so its degree is increased to t, which is the expected degree by the cryptosystem parameters. The input polynomial f of degree n is multiplied by $t - n$ linear factors of the form $x - \rho$, $\texttt{step} \oplus 1 \leq \rho \leq ((t-n) * \texttt{step}) \oplus 1$, in the first $t - n$ iterations. Thus, each coefficient ρ will be included in the list of roots by the root-finding algorithm. In Step **iii.c.** the value of the fake root is created. The multiplications of multiples of `step` are done with integer operations in our implementation. If a root of the original polynomial is equal to a fake root, the root appears twice in the returned list.

Algorithm 4 Adding fake roots — `AddRoots(f,step)`

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

Output: a polynomial of degree t.

```
i.   Initialize diff ← $t - n$, fakeRoot ← 1, addRoot ← 1, added ← 0, $g$ ← $f$.
ii.  Pad $g$ with diff zero coefficients to get to $t + 1$ coefficients.
iii. For index in $[1, t-1]$
        a. addRoot ← addRoot − not(index − (diff + 1));
        b. added ← added + addRoot;
        c. fakeRoot ← 1 ⊕ (addRoot ∗ (added ∗ step));
        d. $g$ ← $g$ ∗ ((addRoot)$x$ − fakeRoot);
iv.  Truncate polynomial $g$ to keep $t + 1$ coefficients, removing the artificial ones.
v.   Return $g$.
```

In order to compute similar operations regardless of the degree n of the input polynomial, t multiplications are made. After multiplying f by $t - n$ linear factors, f is multiplied n times by $0x + 1$, so the resulting polynomial g is of degree t.

The main cost of Algorithm 4 is composed by the polynomial multiplications and the addition operations in finite fields for the creation of the value of the fake roots. The multiplications and additions are done $t-1$ times, resulting in

$$C_{add\_roots} = (t-1)(C_{add} + C_{poly\_mult}) \tag{3}$$

where C_{add} is the cost of the addition of elements in $\mathbb{F}_{q^m}$ and C_{poly_mult} is the cost of the multiplication between two polynomials.

After this, the root-finding algorithm showed in Algorithm 5 is executed, receiving the new g of degree t as input. Algorithm 5 is an iterative adaptation of Algorithm 2, that returns the roots of f instead of its monic factors. These modifications do not change the final cost of the algorithm. The GCD computation in Step **iii.c.3** can be done in constant-time as described in [Bernstein and Yang 2019].

Algorithm 5 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 $R$.
```

The removal of the fake roots from the list is shown in Algorithm 6. The goal is to erase the fake roots in the list R returned by Algorithm 5. The list R_1 of size t receives a copy of the list R and sorts it. Thus, the fake roots in the list will be in increasing order. Then, the elements from the list R_1 are compared with the values of the fake roots added to the polynomial f. When an element $R_1[\texttt{index}]$ is equal to the current value being searched for and there are still fake roots to be found, it means that the element $R_1[\texttt{index}]$ is a fake root of f. In this case, the value of $R_1[\texttt{index}]$ is set to the maximum value possible for a coefficient. When an element $R_1[\texttt{index}]$ is different from the current value of fake roots it means that $R_1[\texttt{index}]$ is an original root, so its value is maintained.

The list R_1 can contain duplicates when a fake added root is equal to an original root of f. When a fake root is identified the next fake root value is calculated, meaning that only one of the duplicates will be removed, so the original root stays on the returned

Algorithm 6 Removing fake roots — `RemoveRoots(R,t,n,step,ζ)`

Input: a list R of t roots returned by Algorithm 5, including fake roots; the expected degree t; the degree n of the original polynomial; the value `step` of the increment between each fake root; the element ζ 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 ζ.

```
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 $R_1$.
```

list. After all elements of R_1 were evaluated and the fake root received the maximum coefficient value, the list R_1 is sorted again. Therefore, all the original roots will be ordered and in the first n elements of the list. The other $t - n$ elements are the ones with the maximum value at the end of the list. These elements are fake roots added to the input polynomial and, consequently, receive the given value ζ in Step **vii.**.

The output is a list of t elements, with n sorted roots in the first n elements followed by $t - n$ elements ζ. In a code-based context, the value ζ 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 not generated incorrectly.

Algorithm 6 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

$$C_{remove\_roots} = 2C_{sort} + t(6C_{add}) + C_{add} \quad (4)$$

where C_{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. Results

In our experiments, we used an Intel® Core™ i5-3317U CPU @ 1.70GHz and the code[1] was compiled with GCC version 11.1.0 with the compilation flags "`-march=native -mtune=native -Wall`".

When comparing the number of CPU cycles of our proposed algorithm against three other algorithms for polynomial root-finding, we note that our proposal is advantageous, even though is not the best for very small fields and degrees. Table 1 shows this for $\mathbb{F}_{2^{13}}$ with the mean and margin of error for an interval of confidence at the 95% level. Each point was calculated with a sample of 2500 measures. The set size considered in these executions is $n = 6688$ and the polynomials are in $\mathbb{F}_{2^{13}}$, that are parameters suggested in NIST post-quantum proposals. The values show the CPU cycles for each method when expecting a polynomial of degree $t = 128$ but receiving as input a polynomial of degree $d = \{64, 96, 128\}$. The exhaustive method and the Additive Fast Fourier Transform algorithm are deterministic algorithms used in the implementation of NIST post-quantum proposals.

Degree	**64**		**96**		**128**	
Algorithm	μ	$\pm\epsilon$	μ	$\pm\epsilon$	μ	$\pm\epsilon$
Exhaustive	82.2004	0.4165	82.1148	0.5214	82.2850	0.3819
Additive FFT	31.2994	0.3006	31.4474	0.2797	31.4957	0.2805
Probabilistic method	25.7721	0.1255	40.2116	0.1953	57.9839	0.1881
Constant-time probabilistic	62.2050	0.1674	62.4064	0.1743	62.6301	0.1624

Table 1. Mean (μ) and margin of error (ϵ) for an interval with 95% of confidence of the execution time in 10^6 cycles of our proposal and other root-finding methods for polynomials in $\mathbb{F}_{2^{13}}$.

The exhaustive method evaluates n points separately, and even for the small field considered in these tests ($q = 2^{13}$), its execution is not the most efficient. Its advantage is that it evaluates n points regardless of the polynomial, so its execution time is not tied to the polynomial degree. The Additive FFT algorithm performs multipoint evaluation, thus it is more efficient than the exhaustive method. 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.

The values of the original probabilistic algorithm show a significant variation in execution time, that increases according to the polynomial degree. Our proposed constant-time probabilistic algorithm shows great results in this aspect, achieving the desired constant time. The improvement, compared to the original numbers, is apparent. It is more efficient than the exhaustive method even though the finite field is small, reducing the number of CPU cycles by half. These results are from experiments with a constant-time sorting method and a constant-time implementation of the GCD algorithm.

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, in bigger fields, Additive FFT may not be the fastest algorithm. Since probabilistic algorithms tend to perform well in bigger fields, we measured the execution times of our proposed method and

[1] `https://github.com/DuniaMarchiori/probabilistic-root-finding`

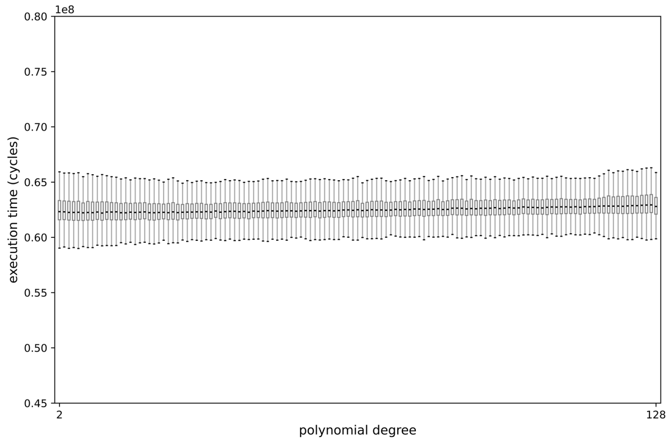

Figure 1. Execution time in CPU cycles of our proposal for degrees 2 to 128 with polynomials in $\mathbb{F}_{2^{13}}$ when expecting a polynomial of degree 128.

the Additive FFT algorithm in fields of size $q = 2^m, m = [13, 16, 18, 20]$ and identified the point in which the curves cross. For values before the curves intersect, our proposed algorithm performs better than Additive FFT, but when the expected degree is higher than the intersection, the Additive FFT algorithm is a better alternative.

The point of intersection of the number of CPU cycles of our proposed algorithm and the additive FFT algorithm increases as the field size increases. Considering the value of the interval for a confidence of 95%, in a field size $q = 2^{13}$, the intersection occurs for degree 75, indicating that our proposal is faster in the range $[1, 74]$. In $q = 2^{16}$, our experiments showed that our proposal is faster in the range $[1, 213]$. In $q = 2^{18}$, the range is $[1, 383]$ and in $q = 2^{20}$, it is $[1, 848]$.

Figure 1 shows the box plot of the execution time in CPU cycles of our proposed algorithm when expecting a polynomial of degree 128 and receiving a polynomial of degree 2 to 128. For each degree, 2500 random polynomials in $\mathbb{F}_{2^{13}}$ were used as input. The measures were taken from a implementation of our proposal with constant-time sorting and GCD algorithms. In this figure, we observe that the execution time is not tied to the degree of the input polynomial. When expecting a polynomial of degree t, the degree of the input polynomial does not impact the execution time of the algorithm. However, the execution time is increased because of the extra work introduced by adding and removing roots.

The 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. It does not leak information when applying the timing side-channel attack mentioned in Section 2.3, and it is faster than the additive Fast Fourier Transform algorithm until large polynomial degrees relative to the chosen finite field size. At the same time, in code-based cryptography a large number of errors is not necessary to achieve a sufficient security level, indicating that our proposal may be the best alternative in larger fields than the ones used in current cryptosystems.

6. Conclusion

In this work, 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 proposed algorithm shows a constant behavior in our experiments when expecting an input polynomial of degree t and receiving a polynomial of degree $n \leq t$. The implementation can be done without branching, memory-access and non-constant operand leakage, resulting in a constant-time method. It is faster than the exhaustive method used in the proposal [Bernstein et al. 2020] and also faster, in the specified ranges, than the additive Fast Fourier Transform algorithm. This can be beneficial for larger fields than the ones used so far in code-based cryptography. While the additive FFT algorithm may be considered a more complex algorithm, our proposed algorithm is relatively simpler, and can offer a simple alternative to the exhaustive method, providing some savings.

6.1. Future work

Future work includes doing a more thorough experimental analysis, using tools that verify that our implementation remains constant in different computer architectures. We could also investigate more thoroughly the exact intersection point between our method and the additive FFT for different field sizes. The addition of optimizations for operations in finite fields, like vectorization and bitslicing, may improve the execution time of our algorithm. Finally, other security analyses can be made, like power analysis side-channel attacks.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and in part by NSERC of Canada.

References

Berlekamp, E. R. (1967). Factoring polynomials over finite fields. *Bell System Tech. J.*, 46:1853–1859.

Berlekamp, E. R. (1968). *Algebraic Coding Theory*. McGraw-Hill Book Co., New York.

Berlekamp, E. R. (1970). Factoring polynomials over large finite fields. *Mathematics of computation*, 24(111):713–735.

Bernstein, D. J. (2019). djbsort library. https://sorting.cr.yp.to/index.html.

Bernstein, D. J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., et al. (2020). Classic McEliece: conservative code-based cryptography. *NIST PQC Round*, 3.

Bernstein, D. J. and Yang, B.-Y. (2019). Fast constant-time GCD computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 340–398.

Cantor, D. G. and Zassenhaus, H. (1981). A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–592.

Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., and Vercauteren, F., editors (2006). *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, Boca Raton, FL.

Flajolet, P., Gourdon, X., and Panario, D. (1996). Random polynomials and polynomial factorization. In Meyer, F. and Monien, B., editors, *Automata, Languages and Programming*, pages 232–243, Berlin, Heidelberg. Springer Berlin Heidelberg.

Flajolet, P., Gourdon, X., and Panario, D. (2001). The complete analysis of a polynomial factorization algorithm over finite fields. *J. Algorithms*, 40(1):37–81.

Gao, S. and Mateer, T. (2010). Additive fast Fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272.

von zur Gathen, J. and Panario, D. (2001). Factoring polynomials over finite fields: a survey. *J. Symbolic Comput.*, 31(1-2):3–17. Computational algebra and number theory (Milwaukee, WI, 1996).

von zur Gathen, J. and Shoup, V. (1992). Computing Frobenius maps and factoring polynomials. *Computational complexity*, 2(3):187–224.

Goppa, V. D. (1970). A new class of linear correcting codes. *Problemy Peredachi Informatsii*, 6(3):24–30.

Martins, D., Banegas, G., and Custódio, R. (2019). Don't forget your roots: Constant-time root finding over $\mathbb{F}_{2^m}$. In *International Conference on Cryptology and Information Security in Latin America*, pages 109–129. Springer.

McEliece, R. J. (1978). A public-key cryptosystem based on algebraic coding theory. *The Deep Space Network Progress Report*, 42-44:114–116.

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

Pornin, T. (2018). Constant-time crypto. https://bearssl.org/constanttime.html.

Rabin, M. (1980). Probabilistic algorithms in finite fields. *SIAM Journal on Computing*, 9(2):273–280.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Schipani, D. (2012). *Efficient decoding of cyclic codes and applications in cryptography*. PhD thesis, University of Zurich.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE.

Shoufan, A., Strenzke, F., Molter, H. G., and Stöttinger, M. (2010). A timing attack against patterson algorithm in the McEliece PKC. In *International Conference on Information Security and Cryptology*, pages 161–175. Springer.

Strenzke, F., Tews, E., Molter, H. G., Overbeck, R., and Shoufan, A. (2008). Side channels in the McEliece PKC. In *International Workshop on Post-Quantum Cryptography*, pages 216–229. Springer.

Sumi, T., Morozov, K., and Takagi, T. (2011). Efficient implementation of the McEliece cryptosystem. In *Computer Security Symposium 2011*.