Don't forget your roots: constant-time root finding over $\mathbb{F}_{2^m}$

Douglas Martins[1], Gustavo Banegas[2,3], and Ricardo Custódio[1]

[1] Departamento de Informática e Estatística,
Universidade Federal de Santa Catarina
Florianópolis, SC, 88040-900, Brasil
marcelino.douglas@posgrad.ufsc.br, ricardo.custodio@ufsc.br
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[3] Chalmers University of Technology
Gothenburg, Sweden
gustavo@cryptme.in

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

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

1 Introduction

In recent years, the area of post-quantum cryptography has received considerable attention, mainly because of the call by the National Institute of Standards and Technology (NIST) for the standardization of post-quantum schemes. On this call, NIST did not restrict to specific hard problems. However, most schemes for the Key Encapsulation Mechanism (KEM) are lattice-based and code-based. The latter type is based on coding theory and includes one of the oldest unbroken cryptosystems, namely the McEliece cryptosystem [15].

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; through the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161; and by Sweden through the WASP expedition project Massive, Secure, and Low-Latency Connectivity for IoT Applications.

One of the requirements for those proposals is that they are resistant to all known cryptanalysis methods. However, even if a scheme is immune to such attacks, it may be subject to attacks related to its implementation. In particular, submissions need to avoid side-channel attacks.

There are different ways to apply side-channel attacks to a cryptosystem. As an example, an attacker can measure the execution time of the operations performed by an algorithm and, based on these measures, estimate some secret information of the scheme. This approach is thriving even in a data communication network environment. Daniel J. Bernstein, for instance, demonstrated how to recover AES keys by doing timing attacks on the cache "access speed" [5].

In code-based cryptography, timing attacks on the decryption process are mostly done during the retrieval of the Error Locator Polynomial (ELP) as shown by [20]. The attack is usually done during the polynomial evaluation process, while computing its roots. This attack was demonstrated first in [20] and later in an improved version in [10].

[21] demonstrates algorithms to find roots efficiently in code-based cryptosystems. However, the author shows only timings in different types of implementations and selects the one that has the least timing variability. In other words, the author does not present an algorithm to find the roots in constant time and eliminate a remote timing attack as remarked in Section 6 of [22]. In our work, we use strategies to make the execution time of those algorithms constant. The first and most important one is to write the algorithms iteratively, eliminating all recursions. We also use permutations and simulated operations to uncouple possible measurements of the side effects of the data being measured. The implementation for finding roots in [12] uses Fast Fourier Transform (FFT), which is efficient, but is built and optimized for $\mathbb{F}_{2^{13}}$. In this paper, we aim at developing a more generic implementation that does not require specific optimization in the finite field arithmetic.

Contributions of this paper: In this paper, we show how to perform a timing attack on a code-based key encapsulation mechanism called BIGQUAKE, which was submitted to NIST [2]. The attack was based on timing leakage on root finding process on the decoding step. The original implementation submitted to NIST uses a variation of the Berlekamp Trace Algorithm (BTA) to find roots in the ELP. We provide other methods to find roots and implement them avoiding timing attacks. Moreover, we make a comparison between methods, showing the number of CPU cycles required for our implementation.

Structure of this paper: In Section 2, we give a brief description of Goppa codes, the McEliece cryptosystem and BIGQUAKE for an understanding about how the cryptosystems work and the basic notation used in this paper. In Subsection 2.4, we show how to use a timing attack for recovering the error vector in BIGQUAKE. In Section 3, the core of the paper, we present four methods for finding roots over $\mathbb{F}_{2^m}$. We also include countermeasures for avoiding timing attacks. Section 4 provides a comparison of the number of cycles of the origi-

nal implementation and the implementation with countermeasures. At last, we conclude and discuss open problems.

2 Preliminaries

In this section, we briefly introduce key concepts about Goppa codes and the McEliece cryptosystem [15], relevant for this paper. For more details about algebraic codes, see [3]. After that, we introduce the BIGQUAKE submission, which is the focus of a timing attack presented in Subsection 2.4.

Our focus is on binary Goppa codes since BIGQUAKE [2] and other McEliece schemes use them in their constructions. Moreover, Goppa codes are being used in other submissions in the Second Round of the NIST standardization process.

2.1 Goppa codes

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

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

where $\frac{1}{z+\alpha_i}$ is the unique polynomial with $(z + \alpha_i)\frac{1}{z+\alpha_i} \equiv 1 \mod g(z)$.

Definition 1. *The binary Goppa code $\Gamma(L, g(z))$ consists of all vectors $c \in \mathbb{F}_2^n$ such that*

$$S_c(z) \equiv 0 \bmod g(z). \tag{2}$$

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

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

$$S_c(z) \equiv \frac{w(z)}{\sigma(z)} \mod g(z), \tag{3}$$

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

A binary Goppa code can correct a codeword $c \in \mathbb{F}_2^n$, which is obscured by an error vector $e \in \mathbb{F}_2^n$ with Hamming weight $w_h(e)$ up to t, i.e., the numbers

of non-zero entries in e is at most t. The way to correct errors is using a decoding algorithm. For irreducible binary Goppa codes, we have three alternatives: Extended Euclidean Algorithm (EEA), Berlekamp-Massey algorithm and Patterson algorithm [17]. The first two are out of the scope of this paper since they need a parity-check matrix that has twice more rows than columns. The Patterson algorithm, which is the focus of this paper, can correct up to t errors with a smaller structure.

2.2 McEliece Cryptosystem

In this section, we describe the three important algorithms of the McEliece cryptosystem [15], i.e., key generation, message encryption, and message decryption. To give a practical explanation, we describe the McEliece scheme based on binary Goppa codes. However, it can be used with any q-ary Goppa codes or Generalized Srivastava codes with small modifications as shown by [16] and [1].

Algorithm 1 is the key generation of McEliece. First, it starts by generating a binary Goppa polynomial $g(z)$ of degree t, which can be an irreducible Goppa polynomial. Second, it generates the support L as an ordered subset of $\mathbb{F}_{2^m}$ satisfying the root condition. Third, it is the computation of the systematic form of $\hat{H}$ is done using the Gauss-Jordan elimination algorithm. Steps four, five, and six compute the generator matrix from the previous systematic matrix and return secret and public key. Algorithm 2 shows the encryption process of

Algorithm 1: McEliece key generation.

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

McEliece. The process is simple and efficient, requiring only a random vector e with $w_h(e) \leq t$ and a multiplication of a vector by a matrix.

Algorithm 2: McEliece encryption.

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

Algorithm 3 gives the decryption part of McEliece. This algorithm consists of the removal of the applied errors using a decoding algorithm. First, we compute the syndrome polynomial $S_c(z)$. Second, we recover the error vector e from the syndrome polynomial. Finally, we can recover the plaintext m computing $c \oplus e$, i.e., the exclusive-or of the ciphertext and the error vector. Note that in modern KEM versions of McEliece, $m \in \mathbb{F}_2^n$ is a random bit string used to compute a session key using a hash function. Hence, there is no intelligible information in seeing the first k positions of m with almost no error.

Algorithm 3: McEliece decryption.

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

In the decryption algorithm, steps 2-5 are the description of Patterson's algorithm [17]. This same strategy can be used in schemes that make use of the Niederreiter cryptosystem [11]. These schemes differ in their public-key structure, encryption, and decryption step, but both of them, in the decryption steps, decode the message from the syndrome.

The roots of the ELP can be acquired with different methods. Although these methods can be implemented with different forms, it is essential that the implementations do not leak any timing information about their execution. This leakage can lead to a side-channel attack using time differences in the decryption algorithm, as we explore in a scheme in Subsection 2.4.

2.3 BIGQUAKE Key Encapsulation Mechanism

BIGQUAKE (BInary Goppa QUAsi-cyclic Key Encapsulation) [2] uses binary Quasi-cyclic (QC) Goppa codes in order to accomplish a KEM between two distinct parts. Instead of using binary Goppa codes, BIGQUAKE uses QC Goppa codes, which have the same properties as Goppa codes but allow smaller keys. Furthermore, BIGQUAKE aims to be IND-CCA [6], which makes the attack scenario in Section 2.4 meaningful.

Let us suppose that Alice and Bob (A and B respectively) want to share a session secret key K using BIGQUAKE. Then Bob needs to publishes his public key and Alice needs to follow the encapsulation mechanism. $\mathcal{F}$ is a function that maps an arbitrary binary string as input and returns a word of weight t, i.e $\mathcal{F} : \{0,1\}^* \to \{x \in \mathbb{F}_2^n | w_h(x) = t\}$. The detailed construction of the function $\mathcal{F}$

can be found at subsection 3.4.4 in [2]. $\mathcal{H} : \{0,1\}^k \to \{0,1\}^s$ is a hash function. The function $\mathcal{H}$ in the original implementation is SHA-3. The encapsulation mechanism can be described as:

1. A generates a random $m \in \mathbb{F}_2^s$;
2. Generate $e \leftarrow \mathcal{F}(m)$;
3. A sends $c \leftarrow (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ to B;
4. The session key is defined as: $K \leftarrow \mathcal{H}(m, c)$.

After Bob receives c from Alice, he initiates the decapsulation process:

1. B receives $c = (c_1, c_2, c_3)$;
2. Using the secret key, Bob decodes c_2 to e' with $w_h(e') \leq t$ such that $c_2 = H \cdot e'^T$;
3. B computes $m' \leftarrow c_1 \oplus \mathcal{H}(e')$;
4. B computes $e'' \leftarrow \mathcal{F}(m')$;
5. If $e'' \neq e'$ or $\mathcal{H}(m') \neq c_3$ then B aborts.
6. Else, B computes the session key: $K \leftarrow \mathcal{H}(m', c)$.

After Bob executes the decapsulation process successfully, both parties of the protocol agree on the same session secret key K.

2.4 Attack Description

In [20], the attack exploits the fact that flipping a bit of the error e changes the Hamming weight w and per consequence the timing for its decryption. If we flip a position that contains an error ($e_i = 1$) then the error will be removed and the time of computation will be shorter. However, if we flip a bit in a wrong position ($e_i = 0$) then it will add another error, and it will increase the decryption time. The attack described in [10] exploits the root finding in the polynomial ELP. It takes advantage of sending ciphertexts with fewer errors than expected, which generate an ELP with degree less than t, resulting in less time for finding roots. We explore both ideas applied to the implementation of BIGQUAKE.

Algorithm 4 is the direct implementation of the attack proposed in [20]. We reused the attack presented to show that the attack still works in current implementations such as BIGQUAKE when the root finding procedure is vulnerable to remote timing attacks.

After finding the position of the errors, one needs to verify if the error e' found is the correct one, and then recover the message m. In order to verify for correctness, one can check e' by computing $\mathcal{H}(e) \oplus \mathcal{H}(e') \oplus m = m'$ and if c_3 is equal to $\mathcal{H}(m')$. As mentioned in Subsection 2.3, the ciphertext is composed by $c = (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ or $c = (c_1, c_2, c_3)$.

Algorithm 4: Attack on ELP.

Data: n-bit ciphertext c, t as the number of errors and precision parameter M
Result: Attempt to obtain an error vector e hidden in c.

$e \leftarrow [0, \ldots, 0]$;
for $i \leftarrow 0$ **to** $n-1$ **do**
  $T \leftarrow 0$;
  $c' \leftarrow c \oplus \text{setBit}(n, i)$;
  $time_m \leftarrow 0$;
  for $j \leftarrow 0$ **to** M **do**
    $time_s \leftarrow time()$;
    decrypt(c');
    $time_e \leftarrow time()$;
    $time_m \leftarrow time_m + (time_e - time_s)$;
  end
  $T \leftarrow time_m / M$;
  $L \leftarrow (T, i)$;
end
Sort L in descending order of T;
for $k \leftarrow 0$ **to** $t-1$ **do**
  $index \leftarrow L[k].i$;
  $e[index] \leftarrow 1$;
end
return e;

2.5 Constant-time $\mathbb{F}_{2^m}$ operations

In our analysis, we noticed that the original implementation of BIGQUAKE uses log and antilog tables for computing multiplications and inversions. These look-up tables give a speedup in those operations. However, this approach is subject to cache attacks in a variation of [9], where the attacker tries to induce cache misses and infer the data.

Since we want to avoid the use of look-up tables, we made a constant time implementation for multiplication and inversion, using a similar approach as [12]. In order to illustrate that, Listing 1.1 shows the multiplication in constant-time between two elements over $\mathbf{F}_{2^{12}}$ followed by the reduction of the result by the irreducible polynomial $f(x) = x^{12} + x^6 + x^4 + x + 1$. The inversion in finite fields can be computed by raising an element a to the power $2^m - 2$, i.e., a^{2^m-2}, as shown in Listing 1.1.

3 Root finding methods

As argued, the leading cause of information leakage in the decoding algorithm is the process of finding the roots of the ELP. In general, the time needed to find these roots varies, often depending on the roots themselves. Thus, an attacker who has access to the decoding time can infer these roots, and hence get the

vector of errors e. Next, we propose modifications in four of these algorithms to avoid the attack presented in Subsection 2.4.

Strenzke [21] presents an algorithm analysis for fast and secure root finding for code-based cryptosystems. He uses as a basis for his results the implementation of "Hymes " [7]. Some of that implementation uses, for instance, log and antilog tables for some operations in finite fields, which are known to be vulnerable. Given that, we rewrote those operations without tables and analyzed each line of code from the original implementations, taking care of modifying them in order to eliminate processing that could indicate root-dependent execution time. The adjustments were made in the following algorithms to find roots: exhaustive search, linearized polynomials, Berlekamp trace algorithm (BTA), and successive resultant algorithm (SRA).

In this work, we use the following notation: given a univariate polynomial f, with degree d and coefficients over $\mathbb{F}_{p^n}$, one needs to find its roots. In our case, we are concerned about binary fields, i.e., $p = 2$. Additionally, we assume that all the factors of f are linear and distinct.

3.1 Exhaustive search

The exhaustive search is a direct method, in which the evaluation of f for all the elements in $\mathbb{F}_{2^m}$ is performed. A root is found whenever the evaluation result is zero. This method is acceptable for small fields and can be made efficient with a parallel implementation. Algorithm 5 describes this method.

As can be seen in Algorithm 5, this method leaks information. This is because whenever a root is found, i.e., $dummy = 0$, an extra operation is performed. In this way, the attacker can infer from this additional time that a root was found, thus providing ways to obtain data that should be secret.

Algorithm 5: Exhaustive search algorithm for finding roots of a univariate polynomial over $\mathbb{F}_{2^m}$.

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

```
R ← ∅;
for i ← 0 to n − 1 do
    dummy ← p(A[i]);
    if dummy == 0 then
        R.add(A[i]);
    end
end
return R;
```

One solution to avoid this leakage is to permute the elements of vector A. Using this technique, an attacker can identify the extra operation, but without learning any secret information. In our case, we use the Fisher-Yates shuffle [8] for shuffling the elements of vector A. In [25], the authors show an implementation

of the shuffling algorithm safe against timing attacks. Algorithm 6 shows the permutation of the elements and the computation of the roots.

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

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

```
permute(A);
R ← ∅;
for i ← 0 to n − 1 do
    dummy ← p(A[i]);
    if dummy == 0 then
        R.add(A[i]);
    end
end
return R;
```

Using this approach, we add one extra step to the algorithm. However, this permutation blurs the sensitive information of the algorithm, making the usage of Algorithm 6 slightly harder for the attacker to acquire timing leakage.

The main costs for Algorithm 5 and Algorithm 6 are the polynomial evaluation and we define as C_{pol_eval}. Since we need to evaluate each element in A, it is safer to assume that the total cost is:

$$C_{exh} = n(C_{pol\_eval}). \tag{4}$$

We can go further and express the cost for one polynomial evaluation by the number of operations in finite fields. In our implementation[4] the cost is determined by the degree d of the polynomial and basic finite field operations such as addition and multiplication. As a result, the cost for one polynomial evaluation is:

$$C_{pol\_eval} = d(C_{gf\_add} + C_{gf\_mul}). \tag{5}$$

3.2 Linearized polynomials

The second countermeasure proposed is based on linearized polynomials. The authors in [14] propose a method to compute the roots of a polynomial over $\mathbb{F}_{2^m}$, using a particular class of polynomials, called linearized polynomials. In [21], this approach is a recursive algorithm which the author calls "dcmp-rf". In our solution, however, we present an iterative algorithm. We define linearized polynomials as follows:

[4] available in https://git.dags-project.org/gustavo/roots_finding

Definition 2. *A polynomial $\ell(y)$ over $\mathbb{F}_{2^m}$ is called a linearized polynomial if*

$$\ell(y) = \sum_i c_i y^{2^i}, \tag{6}$$

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

In addition, from [24], we have Lemma 1 that describes the main property of linearized polynomials for finding roots.

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

$$y = \sum_{k=0}^{m-1} y_k \alpha^k, y_k \in \mathbb{F}_2 \tag{7}$$

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

$$\ell(y) = \sum_{k=0}^{m-1} y_k \ell(\alpha^k). \tag{8}$$

We call $A(y)$ over $\mathbb{F}_{2^m}$ as an affine polynomial if $A(y) = \ell(y) + \beta$ for $\beta \in \mathbb{F}_{2^m}$, where $\ell(y)$ is a linearized polynomial.

We can illustrate a toy example to understand the idea behind finding roots using linearized polynomials.

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

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

or

$$y^2 + (\alpha^2+1)y = (\alpha^2+\alpha+1)y^0 \tag{9}$$

We can point that on the left hand side of Equation 9, $\ell(y) = y^2 + (\alpha^2+1)y$ is a linearized polynomial over $\mathbb{F}_{2^3}$ and Equation 9 can be expressed just as

$$\ell(y) = \alpha^2 + \alpha + 1 \tag{10}$$

If $y = y_2\alpha^2 + y_1\alpha + y_0 \in \mathbb{F}_{2^3}$ then, according to Lemma 1, Equation 10 becomes

$$y_2\ell(\alpha^2) + y_1\ell(\alpha) + y_0\ell(\alpha^0) = \alpha^2 + \alpha + 1 \tag{11}$$

We can compute $\ell(\alpha^0), \ell(\alpha)$ and $\ell(\alpha^2)$ using the left hand side of Equation 9 and we have the following values

$$\begin{aligned} \ell(\alpha^0) &= (\alpha^0)^2 + (\alpha^2+1)(\alpha^0) = \alpha^2 + 1 + 1 = \alpha^2 \\ \ell(\alpha) &= (\alpha)^2 + (\alpha^2+1)(\alpha) = \alpha^2 + \alpha^2 + \alpha + 1 = \alpha + 1 \\ \ell(\alpha^2) &= (\alpha^2)^2 + (\alpha^2+1)(\alpha^2) = \alpha^4 + \alpha^4 + \alpha^2 = \alpha^2. \end{aligned} \tag{12}$$

A substitution of Equation 12 into Equation 11 gives us

$$(y_2 + y_0)\alpha^2 + (y_1)\alpha + (y_0)\alpha^0 = \alpha^2 + \alpha + 1 \tag{13}$$

Equation 13 can be expressed as a matrix in the form

$$\begin{bmatrix} y_2 & y_1 & y_0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}. \tag{14}$$

If one solves simultaneously the linear system in Equation 14 then the results are the roots of the polynomial given in Equation 9. From Equation 13, one observes that the solutions are $y = 110$ and $y = 011$, which can be translated to $\alpha + 1$ and $\alpha^2 + \alpha$.

Fortunately, the authors in [14] provide a generic decomposition for finding affine polynomials. In their work, each polynomial in the form $F(y) = \sum_{j=0}^{t} f_j y^j$ for $f_j \in \mathbb{F}_{2^m}$ can be represented as

$$F(y) = f_3 y^3 + \sum_{i=0}^{\lceil (t-4)/5 \rceil} y^{5i} (f_{5i} + \sum_{j=0}^{3} f_{5i+2^j} y^{2^j}) \tag{15}$$

After that, we can summarize all the steps as Algorithm 7. The function "generate(m)" refers to the generation of the elements in $\mathbb{F}_{2^m}$ using Gray codes, see [19] for more details about Gray codes.

Algorithm 7 presents a countermeasure in the last steps of the algorithm, i.e., we added a dummy operation for blinding if $X[j]$ is a root of polynomial $F(x)$.

Using Algorithm 7, the predominant cost for its implementation is:

$$C_{lin} = m(C_{gf\_pow} + C_{pol\_eval}) + 2^m(C_{gf\_pow} + 2C_{gf\_mul}) \tag{16}$$

3.3 Berlekamp Trace Algorithm – BTA

In [4], Berlekamp presents an efficient algorithm to factor a polynomial, which can be used to find its roots. We call this algorithm *Berlekamp trace algorithm* since it works with a trace function defined as $Tr(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{m-1}}$. It is possible to change BTA for finding roots of a polynomial $p(x)$ using $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ as a standard basis of $\mathbb{F}_{2^m}$, and then computing the greatest common divisor between $p(x)$ and $Tr(\beta_0 \cdot x)$. After that, it starts a recursion where BTA performs two recursive calls; one with the result of gcd algorithm and the other with the remainder of the division $p(x)/\gcd(p(x), Tr(\beta_i \cdot x))$. The base case is when the degree of the input polynomial is smaller than one. In this case, BTA returns the root, by getting the independent term of the polynomial. In summary, the BTA is a divide and conquer like algorithm since it splits the task of computing the roots of a polynomial $p(x)$ into the roots of two smalls polynomials. The description of BTA algorithm is presented in Algorithm 8.

Algorithm 7: Linearized polynomials for finding roots over $\mathbb{F}_{2^m}$.

Data: $F(x)$ as a univariate polynomial over $\mathbb{F}_{2^m}$ with degree t and m as the extension field degree.
Result: R as a set of roots of $p(x)$.

$\ell_i^k \leftarrow \emptyset$; $\ell_{is} \leftarrow \emptyset$; $A_k^j \leftarrow \emptyset$; $R \leftarrow \emptyset$; $dummy \leftarrow \emptyset$;
if $f_0 == 0$ **then**
  $R.append(0)$;
end
for $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
  $\ell_i(x) \leftarrow 0$;
  for $j \leftarrow 0$ **to** 3 **do**
    $\ell_i(x) \leftarrow \ell_i(x) + f_{5i+2^j} x^{2^j}$;
  end
  $\ell_{is}[i] \leftarrow \ell_i(x)$;
end
for $k \leftarrow 0$ **to** $m-1$ **do**
  for $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
    $\ell_i^k \leftarrow \ell_{is}(\alpha^k)$;
  end
end
$A_i^0 \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
  $A_i^0 \leftarrow f_{5i}$;
end
$X \leftarrow \text{generate}(m)$;
for $j \leftarrow 1$ **to** $2^m - 1$ **do**
  for $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
    $A \leftarrow A_i^{j-1}$;
    $A \leftarrow A + \ell_i^{\delta(X[j], X[j-1])}$;
    $A_i^j \leftarrow A$;
  end
end
for $j \leftarrow 1$ **to** $2^m - 1$ **do**
  $result \leftarrow 0$;
  for $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
    $result = result + (X[j])^{5i} A_i^j$;
  end
  $eval = result + f_3 (X[j])^3$;
  if $eval == 0$ **then**
    $R.append(X[j])$;
  else
    $dummy.append(X[j])$;
  end
end
return R;

Algorithm 8: Berlekamp Trace Algorithm [21] – $BTA(p(x), i) - rf$.

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

As we can see, a direct implementation of Algorithm 8 has no constant execution time. The recursive behavior may leak information about the characteristics of roots in a side-channel attack. Additionally, in our experiments, we noted that the behavior of the gcd with the trace function may result in a polynomial with the same degree. Therefore, BTA will divide this input polynomial in a future call with a different basis. Consequently, there is no guarantee of a constant number of executions.

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

This new approach allows us to define a fixed number of iterations for our version of BTA. Since we always divide into two small instances, we need $t-1$ iterations to split a polynomial of degree t in t polynomials of degree 1. Algorithm 9 presents this approach.

Algorithm 9 extracts a root of the polynomial when the variable *current* has a polynomial with degree equal to one. If this degree is greater than one, then the algorithm needs to continue dividing the polynomial until it finds a root. The algorithm does that by adding the polynomial in a stack and reusing this polynomial in a division.

As presented in the previous methods, the overall cost of Algorithm 9 is:

$$C_{BTA-it} = t(mC_{gcd} + C_{QuoRem}). \tag{17}$$

where C_{gcd} is the cost of computing the gcd of two polynomials, with d the higher degree of those polynomials. In our implementation, the cost of C_{gcd} is:

$$C_{gcd} = d(C_{gf\_inv} + 3C_{gf\_mul}), \tag{18}$$

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

Data: $p(x)$ as an univariate polynomial over $\mathbb{F}_{2^m}$, t as number of expected roots.
Result: The set of roots of $p(x)$.

$g \leftarrow \{p(x)\}$; `// The set of polynomials to be computed`
for $k \leftarrow 0$ **to** t **do**
  $current = g.pop()$;
  Compute $candidates = gcd(current, Tr(\beta_i \cdot x))\ \forall\ \beta_i \in \beta$;
  Select $p_0 \in candidates$ such as $p_0.degree \simeq \frac{current}{2}$;
  $p_1(x) \leftarrow current/p_0(x)$;
  if $p_0.degree == 1$ **then**
    $R.add$(root of p_0)
  end
  else
    $g.add(p_0)$;
  end
  if $p_1.degree == 1$ **then**
    $R.add$(root of p_1)
  end
  else
    $g.add(p_1)$;
  end
end
return R

and C_{QuoRem} is the cost for computing the quotient and remainder between two polynomials. The cost for this computation is:

$$C_{QuoRem} = d(C_{gf\_inv} + (d+1)C_{gf\_mul} + C_{gf\_add}). \tag{19}$$

3.4 Successive Resultant Algorithm

In [18], the authors present an alternative method for finding roots in $\mathbb{F}_{p^m}$. Later on, the authors better explain the method in [13]. The Successive Resultant Algorithm (SRA) relies on the fact that it is possible to find roots exploiting properties of an ordered set of rational mappings.

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

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

The functions ℓ_j are examples of linearized polynomials, as previously defined in subsection 3.2. Our next step is to present the theorems from [18]. Check original work for the proofs.

Theorem 1. *a) Each polynomial ℓ_i is split and its roots are all elements of the vector space generated by $\{v_1, \ldots, v_i\}$. In particular, we have $\ell_n(z) = z^{p^m} - z$.*
b) We have $\ell_i(z) = \ell_{i-1}(z)^p - a_i \ell_{i-1}(z)$ where $a := (\ell_{i-1}(v_i))^{p-1}$.
c) If we identify $\mathbb{F}_{p^m}$ with the vector space $(\mathbb{F}_p)^m$, then each ℓ_i is a p-to-1 linear map of $\ell_{i-1}(z)$ and a p^i to 1 linear map of z.

From Theorem 1 and its properties, we can reach the following polynomial system:

$$\begin{cases} f(x_1) = 0 \\ x_j^p = a_j x_j = x_{j+1} \quad j = 1, \ldots, m-1 \\ x_n^p - a_n x_n = 0 \end{cases} \tag{20}$$

where the $a_i \in \mathbb{F}_{p^n}$ are defined as in Theorem 1. Any solution of this system provides us with a root of f by the first equation, and the n last equations together imply this root belongs to $\mathbb{F}_{p^n}$. From this system of equations, [18] derives Theorem 2.

Theorem 2. *Let $(x_1, x_2, \ldots, x_m)$ be a solution of the equations in Equation 20. Then $x_1 \in \mathbb{F}_{p^m}$ is a solution of f. Conversely, given a solution $x_1 \in \mathbb{F}_{p^m}$ of f, we can reconstruct a solution of all equations in Equation 20 by setting $x_2 = x_1^p - a_1 x_1$, etc.*

In [18], the authors present an algorithm for solving the system in Equation 20 using resultants. The solutions of the system are the roots of polynomial $f(x)$. We implemented the method presented in [18] using SAGE Math [23] due to the lack of libraries in C that work with multivariate polynomials over finite fields. It is worth remarking that this algorithm is almost constant-time and hence we just need to protect the branches presented on it. The countermeasure adopted was to add dummy operations, as presented in Subsection 3.2.

4 Comparison

In this section, we present the results of the execution of each of the methods presented in Section 3. We used an Intel® Core(TM) i5-5300U CPU @ 2.30GHz. The code was compiled with GCC version 8.3.0 and the following compilation flags "*-O3 -g3 -Wall -march=native -mtune=native -fomit-frame-pointer -ffast-math*". We ran 100 times the code and got the average number of cycles. Table 1 shows the number of cycles of root finding methods without countermeasures, while Table 2 shows the number of cycles when there is a countermeasure. In both cases, we used $d = \{55, 65, 100\}$ where d is the number of roots. We remark that the operations in the tables are over $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{16}}$. We used two different finite fields for showing the generality of our implementations and the costs for a small field and a larger field.

Nr. Roots	Field	Exhaustive Search	Linearized polynomials	BTA-rf	SRA
55	$\mathbb{F}_{2^{12}}$	10.152	45.697	9.801	2,301.663
	$\mathbb{F}_{2^{16}}$	117.307	425.494	72.766	2,333.519
65	$\mathbb{F}_{2^{12}}$	12.103	56.270	11.933	2,711.318
	$\mathbb{F}_{2^{16}}$	139.506	522.208	80.687	2,782.838
100	$\mathbb{F}_{2^{12}}$	18.994	84.076	17.322	3,555.221
	$\mathbb{F}_{2^{16}}$	213.503	863.063	133.487	3,735.954

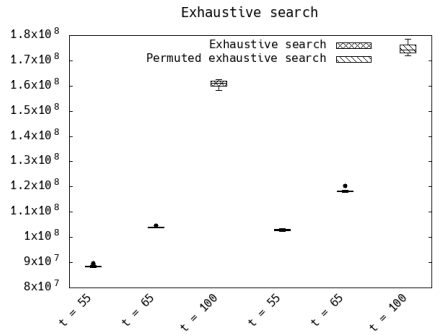
Table 1: Number of cycles divided by 10^6 for each method of finding roots without countermeasures.

Nr. Roots	Field	Exhaustive Search	Linearized polynomials	BTA-it	SRA
55	$\mathbb{F}_{2^{12}}$	11.741	45.467	11.489	2,410.410
	$\mathbb{F}_{2^{16}}$	142.774	433.645	75.467	2,660.052
65	$\mathbb{F}_{2^{12}}$	13.497	55.908	14.864	2,855.899
	$\mathbb{F}_{2^{16}}$	164.951	533.946	86.869	2,929.608
100	$\mathbb{F}_{2^{12}}$	20.287	89.118	20.215	4.211.459
	$\mathbb{F}_{2^{16}}$	238.950	882.101	138.956	4,212.493

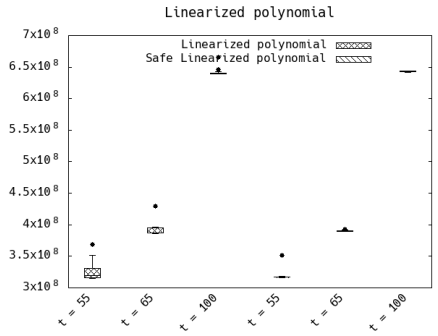
Table 2: Number of cycles divided by 10^6 for each method of finding roots with countermeasures.

Figure 1 shows the number of cycles for random polynomials with degree 55, 65 and 100 and all the operations are over $\mathbb{F}_{2^{16}}$. Figure 1a shows a time variation in the execution time of the exhaustive method, as expected, the average time was increased. In 1b, note a variation of time when we did not add the countermeasures, but when we add them, we see a constant behavior. In 1c, it is possible to see a nonconstant behavior of $BTA-rf$. However, this is different for $BTA-it$, which shows a constant behavior.

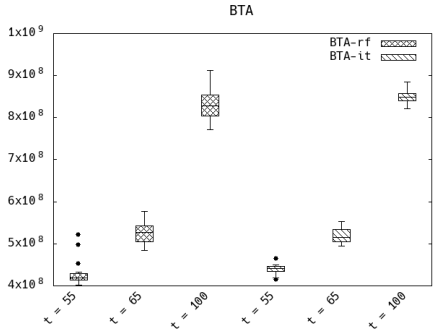
The main focus of our proposal was to find alternatives to compute roots of ELP that has constant execution time. Figure 2 presents an overview between the original implementations and the implementations with countermeasures. It is possible that when a countermeasure is present on Linearized and on BTA approach, the number of cycles increases. However, the variance of time decreases. Additionally, Figure 2 shows an improvement in time variance for SCA method, without a huge increase on the average time. We remark that the "points" out of range can be ignored since we did not run the code under a separated environment, and as such it could be that some process in our environment influenced the result.



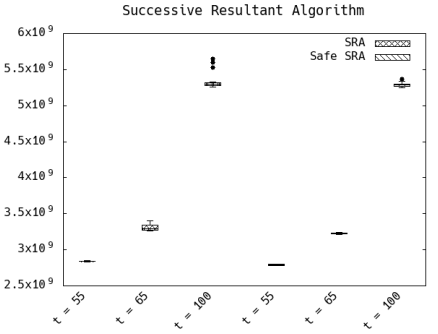
(a) Comparison between exhaustive search with and without countermeasures.



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



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



(d) Comparison between SRA and Safe SRA executions.

Fig. 1: Plots of measurements cycles for methods presented in Section 3. Our evaluation of SRA was made using a Python implementation and cycles measurement with C. In our tests, the drawback of calling a Python module from C has behavior bordering to constant.

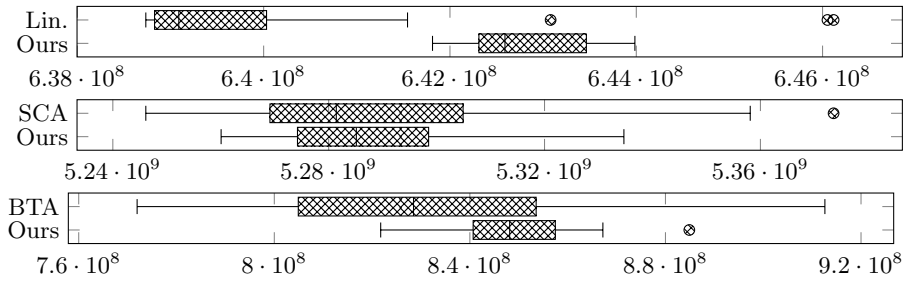


Fig. 2: Comparison of original implementation and our proposal for Linearized, Successive resultant algorithm and Berlekamp trace algorithm with $t = 100$.

5 Conclusion

In our study, we demonstrated countermeasures that can be used to avoid remote timing attacks. In our empirical analysis, i.e, the results in Table 2, BTA-it shows an advantage in the number of cycles which makes it a more efficient and safer choice. However, the exhaustive search with shuffling shows the smallest variation of time, which can be an alternative for usage. Still, the problem for this method is that if the field is large, then it becomes costly to shuffle and iterate all elements.

5.1 Open problems

We bring to the attention of the reader that we did not use any optimization in our implementations, i.e., we did not use vectorization or bit slicing techniques or any specific instructions such as Intel® IPP Cryptography for finite field arithmetic in our code. Therefore, these techniques and instructions can improve the finite fields operations and speed up our algorithms.

We remark that for achieving a safer implementation, one needs to improve the security analysis, by removing conditional memory access and protecting memory access of instructions. Moreover, one can analyze the security of the implementations, by considering different attack scenarios and performing an in-depth analysis of hardware side-channel attacks.

6 Acknowledgments

We want to thank the reviewers for the thoughtful comments on this work. We would also like to thank Tanja Lange for her valuable feedback. We want to extend the acknowledgments to Sonia Belaïd from Cryptoexperts for the discussions about timing attacks.

References

1. Banegas, G., Barreto, P.S., Boidje, B.O., Cayrel, P.L., Dione, G.N., Gaj, K., Gueye, C.T., Haeussler, R., Klamti, J.B., Ndiaye, O., Nguyen, D.T., Persichetti, E., Ricardini, J.E.: DAGS: Key encapsulation using dyadic GS codes. Journal of Mathematical Cryptology **12**(4), 221–239 (2018)
2. Bardet, M., Barelli, E., Blazy, O., Torres, R.C., Couvreur, A., Gaborit, P., Otmani, A., Sendrier, N., Jean-Pierre, T.: BIG QUAKE BInary Goppa QUAsi–cyclic Key Encapsulation. Tech. rep., National Institute of Standards and Technology (NIST) (2017)
3. Berlekamp, E.: Algebraic coding theory. World Scientific (2015)
4. Berlekamp, E.R.: Factoring polynomials over large finite fields. Mathematics of computation **24**(111), 713–735 (1970)
5. Bernstein, D.J.: Cache-timing attacks on AES (2005), https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

6. Bernstein, D.J., Persichetti, E.: Towards KEM unification. Cryptology ePrint Archive, Report 2018/526 (2018), https://eprint.iacr.org/2018/526
7. Biswas, B., Sendrier, N.: HyMES - an open source implementation of the McEliece cryptosystem (2008), http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hyme
8. Black, P.E.: Fisher-Yates shuffle. Dictionary of algorithms and data structures **19** (2005)
9. Bruinderink, L.G., Hülsing, A., Lange, T., Yarom, Y.: Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In: Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings. pp. 323–345 (2016). https://doi.org/10.1007/978-3-662-53140-2_16
10. Bucerzan, D., Cayrel, P.L., Drağoi, V., Richmond, T.: Improved timing attacks against the secret permutation in the McEliece PKC. International Journal of Computers Communications & Control **12**(1), 7–25 (2017)
11. Chor, B., Rivest, R.L.: A knapsack-type public key cryptosystem based on arithmetic in finite fields. IEEE Transactions on Information Theory **34**(5), 901–909 (1988)
12. Chou, T.: Mcbits revisited. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. pp. 213–231 (2017). https://doi.org/10.1007/978-3-319-66787-4_11
13. Davenport, J.H., Petit, C., Pring, B.: A Generalised Successive Resultants Algorithm. In: Duquesne, S., Petkova-Nikova, S. (eds.) Arithmetic of Finite Fields. pp. 105–124. Springer International Publishing, Cham (2016)
14. Fedorenko, S.V., Trifonov, P.V.: Finding roots of polynomials over finite fields. IEEE Transactions on communications **50**(11), 1709–1711 (2002)
15. McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (Jan 1978)
16. Misoczki, R., Barreto, P.S.L.M.: Compact McEliece keys from Goppa codes. In: Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers. pp. 376–392 (2009). https://doi.org/10.1007/978-3-642-05445-7_24
17. Patterson, N.: The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory **21**(2), 203–207 (1975)
18. Petit, C.: Finding roots in GF(p^n) with the successive resultant algorithm. IACR Cryptology ePrint Archive **2014**, 506 (2014)
19. Savage, C.: A survey of combinatorial Gray codes. SIAM review **39**(4), 605–629 (1997)
20. Shoufan, A., Strenzke, F., Molter, H.G., Stöttinger, M.: A timing attack against patterson algorithm in the McEliece PKC. In: Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers. pp. 161–175 (2009). https://doi.org/10.1007/978-3-642-14423-3_12
21. Strenzke, F.: Fast and secure root finding for code-based cryptosystems. In: Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings. pp. 232–246 (2012). https://doi.org/10.1007/978-3-642-35404-5_18
22. Strenzke, F.: Efficiency and implementation security of code-based cryptosystems. Ph.D. thesis, Technische Universität (2013)

23. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 8.7) (2019), `https://www.sagemath.org`
24. Truong, T.K., Jeng, J.H., Reed, I.S.: Fast algorithm for computing the roots of error locator polynomials up to degree 11 in Reed-Solomon decoders. IEEE Transactions on Communications **49**(5), 779–783 (2001)
25. Wang, W., Szefer, J., Niederhagen, R.: FPGA-based Niederreiter cryptosystem using binary goppa codes. In: Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings. pp. 77–98 (2018). https://doi.org/10.1007/978-3-319-79063-3_4

A Implementation Code

```
#include <stdint.h>
typedef uint16_t gf;
gf gf_q_m_mult(gf in0, gf in1) {
    uint64_t i, tmp, t0 = in0, t1 = in1;
    //Multiplication
    tmp = t0 * (t1 & 1);
    for (i = 1; i < 12; i++)
        tmp ^= (t0 * (t1 & (1 << i)));
    //reduction
    tmp = tmp & 0x7FFFF;
    //first step of reduction
    gf reduction = (tmp >> 12);
    tmp = tmp & 0xFFF;
    tmp = tmp ^ (reduction << 6);
    tmp = tmp ^ (reduction << 4);
    tmp = tmp ^ reduction << 1;
    tmp = tmp ^ reduction;
    //second step of reduction
    reduction = (tmp >> 12);
    tmp = tmp ^ (reduction << 6);
    tmp = tmp ^ (reduction << 4);
    tmp = tmp ^ reduction << 1;
    tmp = tmp ^ reduction;
    tmp = tmp & 0xFFF;
    return tmp;
}
gf gf_inv(gf in) {
    gf tmp_11 = 0;
    gf tmp_1111 = 0;
    gf out = in;
    out = gf_sq(out); //a^2
    tmp_11 = gf_mult(out, in); //a^2*a = a^3
    out = gf_sq(tmp_11); //(a^3)^2 = a^6
    out = gf_sq(out); // (a^6)^2 = a^12
    tmp_1111 = gf_mult(out, tmp_11); //a^12*a^3 = a^15
    out = gf_sq(tmp_1111); //(a^15)^2 = a^30
    out = gf_sq(out); //(a^30)^2 = a^60
    out = gf_sq(out); //(a^60)^2 = a^120
    out = gf_sq(out); //(a^120)^2 = a^240
    out = gf_mult(out, tmp_1111); //a^240*a^15 = a^255
    out = gf_sq(out); // (a^255)^2 = 510
    out = gf_sq(out); //(a^510)^2 =  1020
    out = gf_mult(out, tmp_11); //a^1020*a^3 = 1023
    out = gf_sq(out); //(a^1023)^2 = 2046
    out = gf_mult(out, in); //a^2046*a = 2047
    out = gf_sq(out); //(a^2047)^2 = 4094
    return out;
}
```

Listing 1.1: Multiplication of two elements in $\mathbb{F}_{2^{12}}$ and inversion of an element in $\mathbb{F}_{2^{12}}$