Optimized and secure implementation of ROLLO-I

Lina Mortajine[1,3], Othman Benchaalal[1], Pierre-Louis Cayrel[2], Nadia El Mrabet[3], and Jérôme Lablanche[1]

[1] Wisekey, Arteparc de Bachasson, Bâtiment A, 13590 Meyreuil
{jlablanche,lmortajine,obenchaalal}@wisekey.com
[2] Laboratoire Hubert Curien, UMR CNRS 5516,
Bâtiment F 18 rue du Benoît Lauras, 42000 Saint-Etienne
pierre.louis.cayrel@univ-st-etienne.fr
[3] Mines Saint-Etienne, CEA-Tech, Centre CMP, Departement SAS,
F - 13541 Gardanne France
nadia.el-mrabet@emse.fr

Abstract. This paper presents our contribution regarding two implementations of the ROLLO-I algorithm, a code-based candidate for the NIST PQC project. The first part focuses on the two implementations of the ROLLO-I algorithm, and the second part analyzes a side-channel attack and the associated countermeasures. The first implementation utilizes existing hardware by using a crypto co-processor to speed-up operations in $\mathbb{F}_{2^m}$. The second one is a full software implementation (not using the crypto co-processor), running on the same hardware and is publicly available on GitHub. Finally, the side-channel attack allows us to recover the private key with only 79 ciphertexts for ROLLO-I-128. We propose counter-measures in order to protect future implementations.

Keywords: post-quantum cryptography, side-channel attacks, ROLLO-I cryptosystem

Introduction

Today, 26 candidates are still under study for the standardization campaign launched by the National Institute of Standards and Technology (NIST) in 2016. Among the candidates that were submitted are 8 signature schemes based on lattices and multivariate. Also submitted were 17 public-key encryption schemes, key-encapsulation mechanisms (KEMs), that base their security on codes, lattices, or isogenies. In addition, one more signature scheme based on a zero-knowledge proof system has also been submitted.
In this paper, we focus our analysis on the submissions based on codes. The first cryptosystems based on codes (e.g. McEliece cryptostem) uses keys far too large to be usable by the industry. The development of new cryptosystems based on different codes as well as the introduction of codes embedded with the rank

metric has resulted in a considerable reduction of key sizes and thus reaches key sizes comparable to those used in lattice-based cryptography. Despite the evolution of research in this field, some post-quantum cryptosystems submitted to the NIST PQC project require a large number of resources. Notably regarding the memory which becomes binding when we have to implement the algorithms into constrained environments such that microcontrollers. It is then hardly conceivable to imagine that these cryptosystems may replace the ones used nowadays on chips. In that purpose, we decided to study the real cost of a code-based cryptosystem implementation. This study is essential to prepare the transition to post-quantum cryptography. For this study, we decided to perform two implementations on microcontroller, the first one using only software and the second one using the crypto co-processor featuring in the microcontroller.
One of the main criteria for the selection of the cryptosystem has been the RAM available on the microcontroller to run cryptographic protocols. We first decided to compare the size of elements manipulated in submitted code-based cryptosystems. The respective sizes are reported in Table 1. Three other code-based cryptosystems in round 2; Classic McEliece, LEDAcrypt, and NTS-KEM use much larger keys and, thus were not taken into account in our study and not listed in Table 1.

Parameter \ Algorithm	BIKE			HQC	RQC	ROLLO		
scheme number	I	II	III			**I**	II	III
public key	8,188	4,094	9,033	14,754	3,510	**947**	2,493	2,196
secret key	548	548	532	532	3,510	**1,894**	4,986	2,196
ciphertext	8,188	4,094	9,033	14,818	3,574	**947**	2,621	2,196

Table 1. Size of elements in bytes for code-based cryptosystems (security level 5)

The selection of a microcontroller with only 4 kB of RAM that can be found on the market led us to choose the ROLLO-I submission. As seen in Table 1, the total size of the parameters is the smallest one when we choose ROLLO-I and consequently, this is the algorithm that needs the smallest amount of RAM. Since operations on ROLLO-II and ROLLO-III are similar, they should be integrated quickly.
However, embedded implementations can lead to vulnerabilities that can be exploited by a side-channel attacker who gathers information about private data by exploiting physical measurements. Some side-channel attacks have already been performed on code-based cryptosystems [1], [2]. Then, to provide a first secure implementation of ROLLO-I, we propose the countermeasures against the side-channel attack that we introduced.

Our contribution. In this paper, we present two practical implementations of ROLLO-I in a microcontroller in which 4 kB of RAM is dedicated to cryptographic data. The first one consisting in full software implementation and the second one uses the crypto co-processor featuring in the microcontroller.

We finally give a first study on the security of ROLLO-I against side-channel attacks and implement countermeasures against the attack that we have found.

Organization of this paper. This paper is organized as follows: we start with some preliminary definitions and present the ROLLO-I cryptosystem in Section 1, then we present in Section 2 the memory-optimized implementations and in Section 3.1, we finally demonstrate a first side-channel attack on ROLLO-I and present associated countermeasures.

1 Background

In this section, we give some definitions to explain the Low-Rank Parity Check (LRPC) codes which have been first introduced in [3]. For more details, the reader is referred to [4,5]. For fixed prime numbers m and n, we denote by:

q	a power of a prime number p, where p is prime
$\mathbb{F}_q$	the finite field of q elements
$\mathbb{F}_{q^m}$	the vector space that is isomorphic to $\mathbb{F}_q[x]/(P_m)$, with P_m an irreducible polynomial of degree m over $\mathbb{F}_q$
$\mathbb{F}_{q^m}^n$	a vector space isomorphic to $\mathbb{F}_{q^m}[X]/(P_n)$, with P_n an irreducible polynomial of degree n over $\mathbb{F}_q$
$\mathbf{v}$	an element of $\mathbb{F}_{q^m}^n$
$M(\mathbf{v})$	the matrix $(v_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq m}}$ corresponding to the element $\mathbf{v}$.

Let k, and n be two integers such that $k \geq n$. A linear code $\mathcal{C}$ over $\mathbb{F}_{q^m}$ of length n and dimension k is a subspace of $\mathbb{F}_{q^m}^n$. It is denoted by $[n,k]_{q^m}$.
A linear code $\mathcal{C}$ of parameters $[n,k]_{q^m}$ can be represented by a generator matrix $\mathbf{G} \in \mathbb{F}_{q^m}^{k\times n}$ such that

$$\mathcal{C} := \{\mathbf{x}.\mathbf{G}, \mathbf{x} \in \mathbb{F}_{q^m}^k\}.$$

The code $\mathcal{C}$ can also be given by its parity-check matrix $\mathbf{H} \in \mathbb{F}_{q^m}^{(n-k)\times n}$ such that

$$\mathcal{C} := \{\mathbf{x} \in \mathbb{F}_{q^m}^n, \mathbf{H}.\mathbf{x}^T = 0\}.$$

The vector $\mathbf{s}_x = \mathbf{H}.\mathbf{x}^T$ is called the syndrome of $\mathbf{x}$.
ROLLO cryptosystem is based on codes in rank metric over $\mathbb{F}_{q^m}^n$. In rank metric, the distance between two words $\mathbf{x} = (x_1, \cdots, x_n)$ and $\mathbf{y} = (y_1, \cdots, y_n)$ in $\mathbb{F}_{q^m}^n$ is defined by

$$d(\mathbf{x},\mathbf{y}) := \|\mathbf{x}-\mathbf{y}\| = \|\mathbf{v}\| = \text{Rank } M(\mathbf{v}),$$

where $M(\mathbf{v}) = (v_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq m}}$ and $\|\mathbf{v}\|$ is called the rank weight of the word $\mathbf{v} = \mathbf{x}-\mathbf{y}$.
The rank of a word $\mathbf{x} = (x_1, \cdots, x_n)$ can also be seen as the dimension of its support $\text{Supp}(\mathbf{x}) \subset \mathbb{F}_{q^m}$ spanned by the basis of $\mathbf{x}$. In other words, the support of $\mathbf{x}$ is given by

$$\text{Supp}(\mathbf{x}) = \langle x_1, \cdots, x_n\rangle_{\mathbb{F}_q}.$$

The authors of [5] introduced the family of ideal codes that allows them to reduce the size of the code's representation, the associated generator matrix is based on ideal matrices.
Given a polynomial $P \in \mathbb{F}_q[X]$ of degree n and a vector $\mathbf{v} \in \mathbb{F}_{q^m}^n$, an ideal matrix generated by $\mathbf{v}$ is an $n \times n$ square matrix defined by

$$\mathcal{IM}(\mathbf{v}) = \begin{pmatrix} \mathbf{v} \\ X\mathbf{v} \bmod P \\ \vdots \\ X^{n-1}\mathbf{v} \bmod P \end{pmatrix}.$$

An $[ns, nt]_{q^m}$-code $\mathcal{C}$, generated by the vectors $(\mathbf{g}_{i,j})_{\substack{i\in[1,\cdots,s-t]\\ j\in[1,\cdots,t]}} \in \mathbb{F}_{q^m}^n$, is an ideal code if its generator matrix in systematic form is of the form

$$\mathbf{G} = \begin{pmatrix} & \mathcal{IM}(\mathbf{g_{1,1}}) & \cdots & \mathcal{IM}(\mathbf{g_{1,s-t}}) \\ \mathbf{I}_{nt} & \vdots & \ddots & \vdots \\ & \mathcal{IM}(\mathbf{g_{t,1}}) & \cdots & \mathcal{IM}(\mathbf{g_{t,s-t}}) \end{pmatrix}.$$

In [5], the authors restrain the definition of ideal LRPC (Low-Rank Parity Check) codes to $(2, 1)$-ideal LRPC codes that they used in ROLLO cryptosystems.
Let F be a $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ such that $dim(F) = d$. Let $(\mathbf{h}_1, \mathbf{h}_2)$ be two vectors of $\mathbb{F}_{q^m}^n$, such that $\mathrm{Supp}(\mathbf{h}_1, \mathbf{h}_2) = F$, and $P \in \mathbb{F}_q[X]$ be a polynomial of degree n. A $[2n, n]_{q^m}$-code $\mathcal{C}$ is an ideal LRPC code if it has a parity-check matrix of the form

$$\mathbf{H} = \begin{pmatrix} \mathcal{IM}(\mathbf{h}_1)^T & \mathcal{IM}(\mathbf{h}_2)^T \end{pmatrix}.$$

Hereafter, we will focus on ROLLO-I submission, which has smaller parameters than ROLLO-II and ROLLO-III (see Table 1).

ROLLO-I scheme

The submission of ROLLO-I is a Key Encapsulation Mechanism (KEM) composed of three probabilistic algorithms: the Key generation (Keygen), Encapsulation (Encap), and Decapsulation (Decap) are detailed in Table 4. During the decapsulation process, the syndrome of the received ciphertext c is computed, then the Rank Support Recovery (RSR) algorithm is performed to recover the error's support. The latter is explained in [5].

The fixed parameter sets given in Table 3 allow to achieve respectively 128, 192, and 256-bit security level according to NIST's security strength categories 1, 3, and 5 [6]. As described in Section 1, the parameters n and m correspond respectively to the degrees of irreducible polynomials P_n and P_m implied in the fields $\mathbb{F}_q[x]/(P_m)$ and $F_{q^m}[X]/(P_n)$. Note that for the three security levels, $q = 2$. The parameters d and r correspond respectively to the private key and error's ranks.

Algo. \ Param.	d	r	P_n	P_m	Security level (bits)
ROLLO-I-128	6	5	$X^{47}+X^5+1$	$x^{79}+x^9+1$	128
ROLLO-I-192	7	6	$X^{53}+X^6+X^2+X+1$	$x^{89}+x^{38}+1$	192
ROLLO-I-256	8	7	$X^{67}+X^5+X^2+X+1$	$x^{113}+x^9+1$	256

Table 3. ROLLO-I parameters for each security level

Alice		Bob
KeyGen		
Generate a support F of rank d		
Generate the private key		
$\mathbf{sk} = (\mathbf{x}, \mathbf{y})$ from the support F		
Compute the public key		
$\mathbf{h} = \mathbf{x}^{-1} \cdot \mathbf{y} \mod P_n$	$\xrightarrow{\mathbf{h}}$	**Encapsulation**
		Generate a support E of rank r
		Pick randomly two elements
		$(\mathbf{e_1}, \mathbf{e_2})$ from the support E
		Compute the ciphertext
		$\mathbf{c} = \mathbf{e_2} + \mathbf{e_1} \cdot \mathbf{h} \mod P_n$
		Derive the shared secret
Decapsulation	$\xleftarrow{\mathbf{c}}$	$K = \text{Hash}(E)$
Compute the syndrome		
$\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \mod P_n = x.\mathbf{e_2} + y.\mathbf{e_1} \mod P_n$		
Recover the error's support		
$E = \text{RSR}(F, \mathbf{s}, r)$		
Compute the shared secret		
$K = \text{Hash}(E)$		

Table 4. ROLLO-I KEM protocol

2 ROLLO-I implementations

In this section we detail the algorithms in the rings $\mathbb{F}_2[x]/(P_m)$ and $\mathbb{F}_{2^m}[X]/(P_n)$ required in ROLLO-I cryptosystem. The implementations are performed on 32-bit architecture systems.

2.1 Operations in $\mathbb{F}_2[x]/(P_m)$

The addition in $\mathbb{F}_2[x]/(P_m)$ consists in xoring 32-bit words. Thus, the three main operations to implement are the multiplication, the modular reduction, and the inversion. For the inversion in $\mathbb{F}_2[x]/(P_m)$, we use the extended Euclidean algorithm for binary polynomials as given in [7, Algo. 2.48].

2.1.1 Multiplication

Regarding the multiplication between two polynomials $a, b \in \mathbb{F}_2[x]/(P_m)$, we use the left-to-right comb method with windows of width $w = 4$ as described in [7, Algo. 2.36]. For any polynomial $a \in \mathbb{F}_2[x]/(P_m)$, we associate the vector $A = (A_0, \ldots, A_{\lceil m/32 \rceil - 1})$ where A_j is the jth 32-bit word and we note $A_{j,i}$ the ith block of four coefficients in A_j. First we pre-compute the product $u(x) \times b(x)$ for all polynomials u of degree less than 4 (16 elements are stored in a table T). Let $\hat{u}$ denote the binary representation of the coefficients of the polynomial $u(x)$ (i.e $u(x) = 0 \leftrightarrow \hat{u} = 0, u(x) = 1 \leftrightarrow \hat{u} = 1, u(x) = x \leftrightarrow \hat{u} = 2, \cdots, u(x) = x^3 + x^2 + x + 1 \leftrightarrow \hat{u} = 15$). Thus, we have $T_{\hat{u}} = b(x) \times u(x)$.

Then, for $0 \leq j < \lceil m/32 \rceil$, we add to the result $R_j = (R_j, \ldots, R_n)$, the element $T_{\hat{u}}$ where $\hat{u}$ is the integer associated to $A_{j,i}$, for each i. If i is non zero, we multiply the polynomial R by x^4, which is equivalent to a shift of 32-bit words.

Algorithm 1: Polynomial multiplication using the left-to-right method with a width window $w = 4$

Input: Two polynomials $a, b \in \mathbb{F}_2[x]/(P_m)$
Output: $r(x) = a(x) \times b(x)$

For all polynomials $u(x)$ of degree at most $w - 1$, compute $T_{\hat{u}} = b(x) \times u(x)$
$R \leftarrow 0$
for *i from* 7 *downto* 0 **do**
    for *j from 0 to* $\lceil m/32 \rceil - 1$ **do**
        Let $\hat{u} = u_3 u_2 u_1 u_0$ where u_k is the bit $wi + k$ of A_j.
        $R_j \leftarrow R_j \oplus T_{\hat{u}}$
    if $i \neq 0$ **then**
        $R(x) \leftarrow R(x) \times x^4$
return R

2.1.2 Modular reduction

Several modular reductions with parse polynomials are being performing in ROLLO-I cryptosystem. We decide to use the same technique explained in [7, Sec. 2.3.5].
Let us take the example of ROLLO-I-128 and consider an element $\mathbf{c} = (c_0, \cdots, c_{156})$ obtained after a multiplication in $\mathbb{F}_2[x]/(P_{79})$. The modular reduction is performed on each 32-bit word composing $C = (C_0, C_1, C_2, C_3, C_4)$ as in Algorithm 2.

Allow us detail the method for the reduction modulo $P_m(x) = x^{79} + x^9 + 1$ of the 4th word of C which corresponds to the polynomial $c_{96}x^{96} + c_{97}x^{97} + \cdots + c_{127}x^{127}$.

We have:

$$\begin{aligned} x^{96} &\equiv x^{17} + x^{26} \mod P_m \\ &\vdots \\ x^{127} &\equiv x^{48} + x^{57} \mod P_m \end{aligned}$$

Given those congruences, the reduction of C_3 is operated by adding two times C_3 to C as shown in Figure 1.

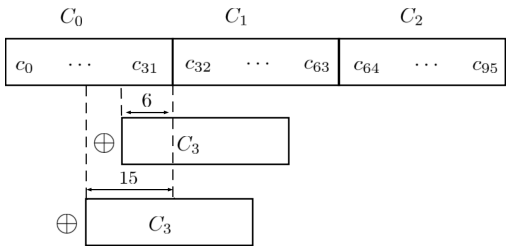

Fig. 1. Reduction of the 32-bit word C_3 modulo $P_m(x) = x^{79} + x^9 + 1$

Algorithm 2: Reduction modulo $P_m(x) = x^{79} + x^9 + 1$

Input: polynomial $c(x)$ of degree at most 156
Output: $c(x) \mod P_m(x)$

$C_2 \leftarrow (C_4 \gg 6) \oplus (C_4 \gg 15)$
$C_1 \leftarrow (C_4 \ll 17) \oplus (C_4 \ll 26) \oplus (C_3 \gg 6) \oplus (C_3 \gg 15)$
$C_0 \leftarrow (C_3 \ll 17) \oplus (C_4 \ll 26)$
$T \leftarrow C_2$ & `0xFFFF8000`
$C_0 \leftarrow C_0 \oplus (T \gg 15)$
$C_1 \leftarrow C_1 \oplus (T \gg 6)$
$C_2 \leftarrow C_2 \oplus (T \ll 22)$
$C_2 \leftarrow C_2$ & `0x7FFF`
$C_3, C_4 \leftarrow 0$
return C

2.2 Operations and memory costs issues in $\mathbb{F}_{2^m}[X]/(P_n)$

In this section, m_b represents the length in bytes of one coefficient in $\mathbb{F}_{2^m}$.

2.2.1 Multiplication

The multiplication in $\mathbb{F}_{2^m}[X]/(P_n)$ is one of the most used operations of this cryptosystem: it is involved in the computation of the public key, the ciphertext and the syndrome.

For example, let $P(X) = p_0 + p_1X$ and $Q(X) = q_0 + q_1X$ be two polynomials of degree 1 in a given polynomial ring. The result of the product is

$$P(X) \times Q(X) = p_0q_0 + (p_0q_1 + p_1q_0)X + p_1q_1X^2.$$

Naively, we have four multiplications and one addition over the coefficients. Thus, the schoolbook multiplication [8] requires n^2 multiplications in $\mathbb{F}_{2^m}$. The Karatsuba algorithm uses the following equation

$$(p_0q_1 + p_1q_0) = (p_0 + p_1)(q_0 + q_1) - p_0q_0 - p_1q_1,$$

and $P(X)\times Q(X)$ requires only three multiplications and four additions/subtractions over the coefficients. To reduce the number of multiplications in $\mathbb{F}_{2^m}$, we implement a combination of Schoolbook multiplication and Karatsuba method [9] as described in Algorithm 8.

Algorithm 3: Karatsuba multiplication

Input: two polynomials $\mathbf{f}$ and $\mathbf{g} \in \mathbb{F}_{2^m}^n$ and N the number of coefficients of $\mathbf{f}$ and $\mathbf{g}$
Output: $\mathbf{f} \cdot \mathbf{g}$ in $\mathbb{F}_{2^m}^n$

1 **if** N *odd* **then**
2 $\quad$ *result* $\leftarrow$ Schoolbook$(\mathbf{f}, \mathbf{g}, N)$
3 $\quad$ **return** *result*
4 $N^{'} \leftarrow N/2$
5 Let $\mathbf{f}(x) = \mathbf{f}_0(x) + \mathbf{f}_1(x)x^{N^{'}}$
6 Let $\mathbf{g}(x) = \mathbf{g}_0(x) + \mathbf{g}_1(x)x^{N^{'}}$
7 $R_1 \leftarrow$ Karatsuba$(\mathbf{f}_0, \mathbf{g}_0, N^{'})$ // Compute recursively $\mathbf{f}_0\mathbf{g}_0$
8 $R_2 \leftarrow$ Karatsuba$(\mathbf{f}_1, \mathbf{g}_1, N^{'})$ // Compute recursively $\mathbf{f}_1\mathbf{g}_1$
9 $R_3 \leftarrow \mathbf{f}_0 + \mathbf{f}_1$
10 $R_4 \leftarrow \mathbf{g}_0 + \mathbf{g}_1$
11 $R_5 \leftarrow$ Karatsuba$(R_3, R_4, N^{'})$ // Compute recursively R_3R_4
12 $R_6 \leftarrow R_5 - R_1 - R_2$
13 **return** $R_1 + R_6x^{N'} + R_2x^{2N}$

In line 4 (Algorithm 8), we divide the polynomial's length N by 2. Consequently, we need to add a padding to the input polynomials with zero coefficients to obtain N even. In Figure 2, we observe that the cycles' number is not strictly increasing due to the division by 2.
Depending on the memory available for a multiplication in $\mathbb{F}_{2^m}[X]/(P_n)$, we can add more or less padding. For example, in ROLLO-I-128 with $n = 47$, we decide to add one zero coefficient which allows us to reduce considerably the number of cycles; however, in ROLLO-I-192 with $n = 53$ we have two possibilities: pad the polynomials with 3 or 11 coefficients. The second possibility is about 10% faster but requires an additional memory cost of $11 \times \lceil 89/32 \rceil \times 4 = 132$ bytes per polynomial. That is why the first choice represents a good balance between memory and execution time.

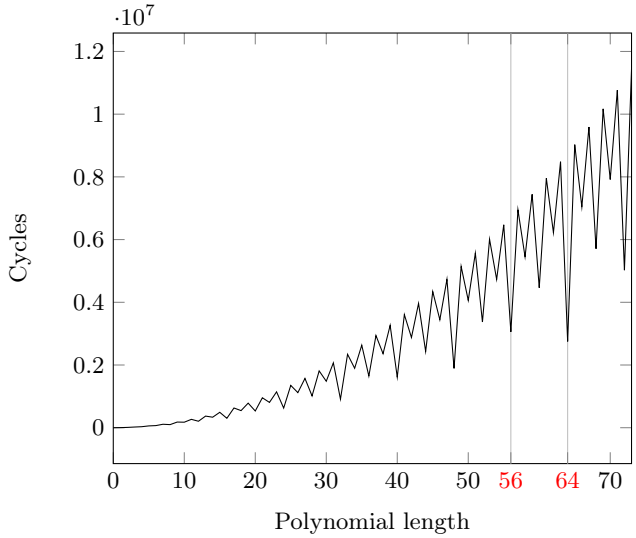

Fig. 2. Number of cycles required by Karatsuba combined with schoolbook multiplication depending on the polynomial length

2.2.2 Inversion

For the inversion in $\mathbb{F}_{2^m}[X]/(P_n)$, we adjust extended Euclidean algorithm given in [7, Algo. 2.48] to the ring $\mathbb{F}_{2^m}[X]/(P_n)$ as described i Algorithm 4.
During the execution of the extended Euclidean algorithm, we have in memory:

- the polynomial to be inverted Q;
- a copy of Q (in order to keep it in memory);
- the dividend;
- the two Bézout coefficients;
- three buffers used to perform intermediates operations (swap between polynomials, results of multiplications).

A way to implement it is to allocate the maximum memory size for each element. As each element can be composed of n coefficients in $\mathbb{F}_{2^m}$, the computation of the inverse in $\mathbb{F}_{2^m}[X]/(P_n)$ requires $8 \times n \times m_b$ bytes. Considering the parameters of ROLLO-I-128, ROLLO-I-192 and ROLLO-I-256 the memory usage represents respectively $4,512$, $5,088$, and $8,576$ bytes, thus exceeding the memory size available on the target microcontroller for all parameters sets. However, during the algorithm we notice that:

- the degree of the polynomial Q is at most $n-1$ and the degree of the dividend is n at the beginning of the process, both decrease during the execution;
- the degrees of the two Bézout coefficients are 0 at the beginning and increase during the process.

Thus, we decide to perform a dynamic memory allocation by setting the necessary memory space for each element at each step of the inversion process.

Algorithm 4: Inversion in $\mathbb{F}_{2^m}[X]/(P_n)$

Input: Q a polynomial in $\mathbb{F}_{2^m}[X]/(P_n)$
Output: $Q^{-1} \mod P_n$

$U \leftarrow Q$, $V \leftarrow P_n$
$G_1 \leftarrow 1$, $G_2 \leftarrow 0$
while $U \neq 1$ **do**
  $j \longleftarrow deg(U) - deg(V)$
  if $j < 0$ **then**
    $U \leftrightarrow V$
    $G_1 \leftrightarrow G_2$
    $j \leftarrow -j$
  $lc\_V \leftarrow V_{deg(V)-1}$ `// leading coefficient of` V
  $U \leftarrow U + X^j.(lc\_V)^{-1}.V$
  $lc\_G_2 \leftarrow G_{2_{deg(G_2)-1}}$ `// leading coefficient of` G_2
  $G_1 \leftarrow G_1 + X^j.(lc\_G_2)^{-1}.G_2$
return G_1

The memory usage is reduced to $2,590$, $2,904$ and $4,864$ bytes respectively for ROLLO-I-128, ROLLO-I-192 and ROLLO-I-256.

2.2.3 Rank Support Recovery (RSR) algorithm

The main memory issue in the RSR algorithm [5, Algo. 1] is the multiple intersections between sub-spaces over $\mathbb{F}_{2^m}^n$. Considering two sub-spaces $U = \langle u_0, u_1, \cdots, u_{n-1} \rangle$ and $V = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ and their associated vectors $\mathbf{u} = (u_0, u_1, \cdots, u_{n-1})$ and $\mathbf{v} = (v_0, v_1, \cdots, v_{n-1})$ in $\mathbb{F}_{2^m}^n$. The intersection $\mathcal{I}_{U,V} = U \cap V$ is computed by following the Zassenhaus algorithm [10], as described below:

- Create the block matrix $\mathcal{Z}_{\mathbf{U},\mathbf{V}} = \begin{pmatrix} M(\mathbf{u}) & M(\mathbf{u}) \\ M(\mathbf{v}) & 0 \end{pmatrix}$;
- Apply the Gaussian elimination on $\mathcal{Z}_{\mathbf{U},\mathbf{V}}$ to obtain a row echelon form matrix;
- The resulting matrix has the following shape: $\begin{pmatrix} M(\mathbf{c}) & * \\ 0 & \mathcal{I}_{U,V} \\ 0 & 0 \end{pmatrix}$, where $\mathbf{c} \in \mathbb{F}_{2^m}^n$.

In the initial RSR algorithm, some pre-computations are performed to avoid additional operations on data. First, we pre-compute $S_i = f_i^{-1} S$, for $1 \geq i \geq d$, where f_i are the elements of the support F. As each S_i is composed of $r \times d$ coefficients in $\mathbb{F}_{2^m}$. then, $r \times d \times d \times m_b$ bytes are needed for the S_i pre-computations. Then, the intersections $S_{i,i+1} = S_i \cap S_{i+1}$, for $1 \geq i \geq d-1$, are each composed of r elements in $\mathbb{F}_{2^m}$. For the pre-computations of the intersections $S_{i,i+1}$, we also need to consider the memory usage induced by the Zassenhaus algorithm. It requires writing in memory four S_i, in other words $4 \times r \times d \times m_b$ bytes.

Furthermore, for these pre-computations, the private key's support F (d coefficients) and the support of the syndrome S ($r \times d$ coefficients) are needed. Thus, the average memory cost of all these pre-computations is:

$$\text{Memory}_{pre-computed} = (r \times d \times (d+5) + (d-3) \times r + d) \times m_b.$$

With this formula, we can predict that ROLLO-I-128 requires $4,212$ bytes to store the pre-computations which is too high for our chosen microcontroller. In order to reduce the memory cost, we store in memory at most three S_i and directly compute the two associated intersections as framed in Algorithm 5.

Algorithm 5: RSR (Rank Support Recover)

Input: $F = \langle f_1, \cdots, f_d \rangle$ an $\mathbb{F}_q$-subspace of $\mathbb{F}_{2^m}$, $s = (s_1, \cdots, s_n) \in \mathbb{F}_{2^m}^n$ syndrome of an error e and r the rank's weight of e
Output: Vector subspace E
Compute $S = \langle s_1, \cdots, s_n \rangle$
// Recall that $S_i = f_i^{-1} S$
$tmp_1 \leftarrow S_1$
$tmp_2 \leftarrow S_2$
$tmp_3 \leftarrow S_3$
Compute $S_{1,2} = tmp_1 \cap tmp_2$
for *i from 1 to* $d-2$ **do**
  Compute $S_{i+1,i+2} = tmp_{i+1} \cap tmp_{i+2}$
  Compute $S_{i,i+2} = tmp_i \cap tmp_{i+2}$
  $tmp_{(i-1)\%3+1} \leftarrow S_{i+3}$
for *i from 1 to d-2* **do**
  $tmp \leftarrow S + F \cdot (S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})$
  if $\dim(tmp) \leq rd$ **then**
    $S \leftarrow tmp$;
$E \leftarrow \bigcap_{1 \leq i \leq d} f_i^{-1} \cdot S$
return E

After the modifications, the total memory cost is:

$$\text{Memory}_{pre-computed} = (8 \times r \times d + (d-3) \times r + d) \times m_b.$$

This method allows us to save $(d-3) \times r \times d \times m_b$ bytes. The gains in memory for each security level are presented in the Table 5.

Algorithm	Save bytes
ROLLO-I-128	1080
ROLLO-I-192	2016
ROLLO-I-256	4480

Table 5. Memory gains with the modified RSR algorithm

2.3 Performance evaluation

The two implementations are implemented in C on a microcontroller, based on a widely used 32-bit SecurCore® SC300™, which has an embedded 32-bit mathematical crypto co-processor to perform operations in $GF(p)$ and $GF(2^m)$ and a True Random Number Generator (TRNG). Among the 24kB of RAM featuring on the microcontroller only 4 kB are available for cryptographic computations. For performance measurements, we use IAR compiler C/C++ with high-speed optimization level and we count the number of cycles with the debugging functionality of the IAR Embedded Workbench IDE [11].
An element in $GF(q^m)^n$ is represented by $n \times \lceil m/32 \rceil \times 4$ bytes. For ROLLO-I-128, $m = 79$ and for ROLLO-I-192, $m = 89$, we obtain $\lceil 79/32 \rceil = \lceil 89/32 \rceil = 3$ 32-bit words. Thus, the memory usages for ROLLO-I-128 and ROLLO-I-192 only differ according to n. Nevertheless, for ROLLO-I-256, $\lceil 113/32 \rceil = 4$, which explains the important difference of memory usage between the higher security and the two lowers. To compute the memory usage, we differentiate the full software implementations and ones using the crypto co-processor.

For the full software implementations, we take into account the space required for keys. However, we notice in ROLLO-I cryptosystem (Table 4) that the part $\mathbf{y}$ of the secret key is only used on the Key Generation process. We suppose that this part is used, with the public key $\mathbf{h}$, to prove the integrity of the other part $\mathbf{x}$. Thus, to reduce the memory used, we decide to store the Cyclic Redundancy Check (CRC) of $(\mathbf{x}, \mathbf{y})$, instead of the part $\mathbf{y}$, leading us to keep the proof of integrity of $\mathbf{x}$ and thus without reducing the security. The result of CRC is stored in a 32-bit word.
For the implementations using the crypto co-processor, the memory usage refers to the RAM required to perform the cryptosystem, the keys being stored in the EEPROM (Electrically Erasable Programmable Read-Only memory). As we can see in Table 6, ROLLO-I-256 cannot be implemented in our target because its memory usage exceeds significantly the 4kB of RAM.

	Full software			With co-processor		
Algo. / Security	GenKey	Encap	Decap	GenKey	Encap	Decap
ROLLO-I-128	3,520	3,592	3,964	2,940	2,940	3,320
ROLLO-I-192	4,120	4,188	5,096	3,448	3,432	4,334
ROLLO-I-256	7,440	7,152	8,992	6,288	5,872	7,776

Table 6. Memory usage for ROLLO-I (in bytes)

All the operations in $GF(2^m)$ take advantage of the crypto co-processor, leading the implementations using the crypto co-processor of ROLLO-I-128 and ROLLO-I-192 to be faster than their full software versions. We can see on Table 7 the number of cycles and the time in milliseconds required by ROLLO-I for the different security levels with the microcontroller running at 50MHz. We do not

compare our implementations with others implementations as they do not fit into the target microcontroller.

		Full software on SC300			On SC300 with co-processor		
Security		GenKey	Encap	Decap	GenKey	Encap	Decap
ROLLO-I-128	cycles ($\times 10^6$)	15.47	1.99	4.31	8.68	0.55	3.75
	ms	309	40.8	86.3	173.6	11	75
ROLLO-I-192	cycles ($\times 10^6$)	21.31	3.38	7.8	11.11	0.8	6.63
	ms	426	67.6	156	222.2	16	132.6
ROLLO-I-256	cycles ($\times 10^6$)	39.92	6.62	15.54	ND	ND	ND
	ms	798.5	132.5	310.8	ND	ND	ND

Table 7. Execution time of ROLLO-I

To see if ROLLO-I can be a realistic alternative to the current key exchange schemes, we compare in Table 8 the full software implementations with Elliptic Curve Diffie-Hellman key Exchange (ECDH) [12] implemented on the same platform.
For ROLLO-I, the key agreement takes into account the Encapsulation and Decapsulation processes. As a remainder, for ECDH, two entities compute two scalars multiplication over $E(\mathbb{F}_q)$ in parallel to establish a shared secret. Thus, for its cost's estimation, we only consider the two scalar multiplications.

Security	Algorithm	Clock cycle ($\times 10^6$)
128	ROLLO-I-128	6.3
	ECDH Curve 256	3.49
192	ROLLO-I-192	11.18
	ECDH Curve 384	8.45

Table 8. Performance comparison between ROLLO-I and ECDH for two different security levels.

We observe that the two implementations are of the same order of magnitude.

3 Side-channel attack on ROLLO-I

Side-channel attacks were first introduced by Kocher in 1996 [13]. Some of these attacks exploit the leakage information coming from a device executing a cryptographic protocol. An adversary extracts these information without having to tamper with the device.
In this section, we deal with chosen-ciphertext Simple Power Analysis (SPA) attack. With the observation of the power traces, SPA attack consists of identifying sequences of an algorithm to recover the key.

3.1 Attack

ROLLO-I does not require the use of ephemeral keys. The generation of keys is generally performed once in the life cycle of a component. Whereas encapsula-

tion and decapsulation processes are performed several times with the same key pair $((\mathbf{x}, \mathbf{y}, \mathbf{F}), \mathbf{h})$. The SPA attack leads us to recover the private key, used to establish the shared secret between two entities.
The decapsulation process is a good target for side-channel attacks because it involves the secret key $\mathbf{x}$ during the syndrome computation

$$\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \bmod P_n.$$

Then, the aim of the attack is to recover the syndrome. The syndrome's support computation S applies Gaussian elimination algorithm to the matrix associated to the syndrome $\mathbf{s}$. The standard Gaussian elimination on a binary matrix is given in Algorithm 6.

Algorithm 6: Gaussian elimination algorithm

Input: Matrix $M \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Output: Matrix M under row echelon form and the rank $rank$ of the matrix

```
1  Rank ← 0
2  for i = 0 to m − 1 do
3      for j = i to n − 1 do
4          if M_{j,i} = 1 then
5              // The line j is a pivot
6              line i ↔ line j
7              rank ← rank +1
8              break
9      for k = line i + 1 to n do
10         if M_{k,i} = 1 then
11             line k ← line k + line i
12 return (M, rank)
```

The first non-zero coefficient (i.e. 1) in the column is the pivot. With the first for loop (line 3 – Algorithm 6) we scan each coefficient in the column to find the pivot. Then we exchange the current line of the founded coefficient with the pivot line. The time required to determine the pivot indicates the number of coefficients processed and allows us to recover the pivot line.
With the second loop for (line 9 - Algorithm 6), we remove the other coefficients 1 in the column.
Specifically, two different treatments are performed on each coefficients:

1. If the coefficient is 0 then no operation is performed.
2. If the coefficient is 1 then an addition in $GF(2^m)$ is performed between the pivot row and the one processed.

This difference of treatment leads us to determine the rows where the coefficients are 1. The syndrome's rank is at most r.d, thus, at the end of the process, we obtain a matrix Ms in row echelon form with the first column known by the

attacker.

$$M_{\boldsymbol{s}} = \begin{pmatrix} \boldsymbol{s_{0,0}} & * & * & * & * * \\ \boldsymbol{s_{1,0}} & s_{1,1} & * & * & * * \\ \vdots & \vdots & \ddots & * & * * \\ \boldsymbol{s_{n-1,0}} & s_{n-1,1} & \cdots & s_{n-1,r\times d-1} & * * \end{pmatrix}$$

That is why we only consider the first column for the attack. To recover the syndrome, we perform m rotations of the matrix $M_{\boldsymbol{s}}$ modulo P_m with the use of the initial ciphertext. Specifically, we multiply the ciphertext by x^i in $\mathbb{F}_2[x]/(P_m)$, with $0 \leq i < m$.
However, we have to consider the modular rotation during the recovering of the columns' syndrome matrix. For example, with ROLLO-I-128 parameters given in Table 3, multiplying the ciphertext by x modulo $(x^{79} + x^9 + 1)$ implies that the last column of the matrix syndrome is xored with the columns 0 and 9 as depicted in Figure 3.

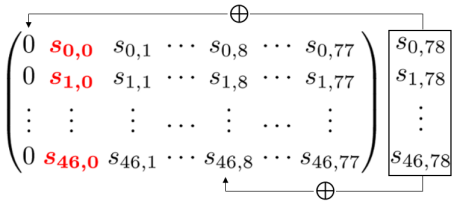

Fig. 3. Example of modular rotation for the syndrome's matrix for ROLLO-I-128

The column 78 is recovered as explained above and to recover the column 9 xored with column 78, we multiply the ciphertext by x^{69} modulo $P_m(x)$. In ROLLO-I-128 and ROLLO-I-256, we need to keep in mind the xor when recovering the columns 1 to 8. For ROLLO-I-192, columns 1 to 38 are concerned.

To develop this attack, we target the implementation using the crypto co-processor. For the experiment, we consider the parameters of ROLLO-I-128, namely n= 47, and m= 79. The secret key **x** and the ciphertext **c** involved in the syndrome computation are generated during the Key Generation and Encapsulation processes. ROLLO-I-128 traces are captured with a Lecroy SDA 725Zi-A oscilloscope. We observe in Figure 4 the difference of patterns between the treatment of the bits 1 and 0. This trace allows us to recover the first column of the syndrome's matrix corresponding to

$$10110101110111010001010111001111001001110010110.$$

We use the same techniques to recover all the columns after the matrix rotation and finally the syndrome.

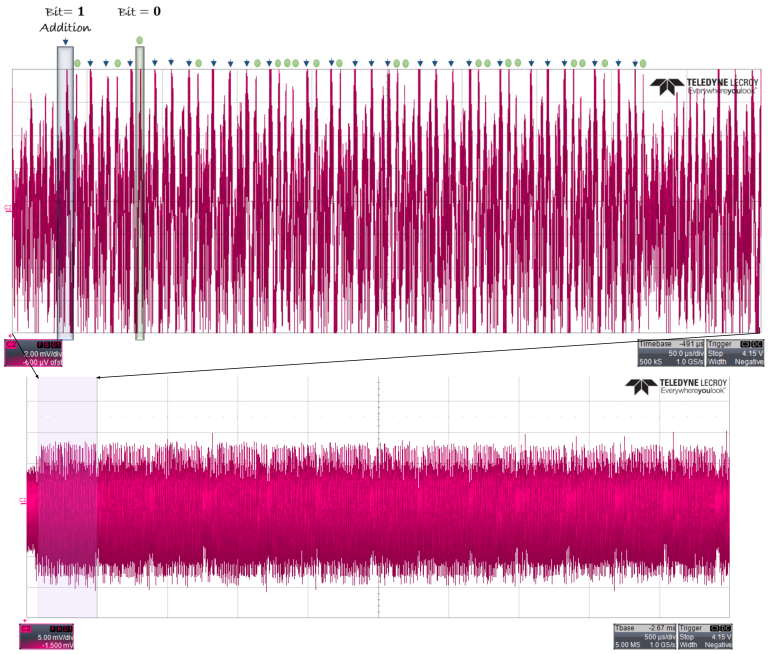

Fig. 4. SPA performed on the first column during Gaussian elimination process

3.2 Countermeasures

Let us discuss solutions to secure the cryptosystem against the attack explained previously in Section 3.1. Several solutions are available to protect the Gaussian elimination against SPA attacks.

A first one is to randomize the treatment of coefficients in each column as described in Algorithm 7.
An attacker is not able to recover the indices of the pivot and the processed rows. Considering the first column, the attacker has n possibilities for the pivot and $m!$ possibilities for the row treatment. Thus the complexity of the SPA attack is $(n!)^m$. For example, with ROLLO-I-128, the complexity is $(47!)^{79}$ which corresponds to about $2^{15,591}$ possibilities.

Let us go back to the previous experiment. With the same parameters, Figure 5 provides the trace of the execution of Gaussian elimination with the countermeasure presented in Algorithm 7.
Although we can still set the coefficients 0 and 1, the order of the elements in each column is completely random so we can not exploit this information any more longer.

Two other solutions consist of:

- adding dummy operations when processing the coefficients 0 as described in Algorithm 8;

Algorithm 7: Gaussian elimination with countermeasures

Input: Matrix $M \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Output: Matrix under row echelon form

```
1  Rank ← 0
2  for i = 0 to m − 1 do
3      for j = i to n − 1 do
4          j_rand = (j + random()) mod (n − i)
5          if M_{j_rand,i} = 1 then
6              // The line j_rand is a pivot
7              line i ↔ line j_rand
8              Rank ← Rank +1
9              break
10     for k = line i + 1 to n do
11         k_rand = (k + random()) mod (n − k)
12         line k ↔ line k_rand
13         if M_{k,i} = 1 then
14             line k_rand ← line k_rand + line i
15 return (M, Rank)
```

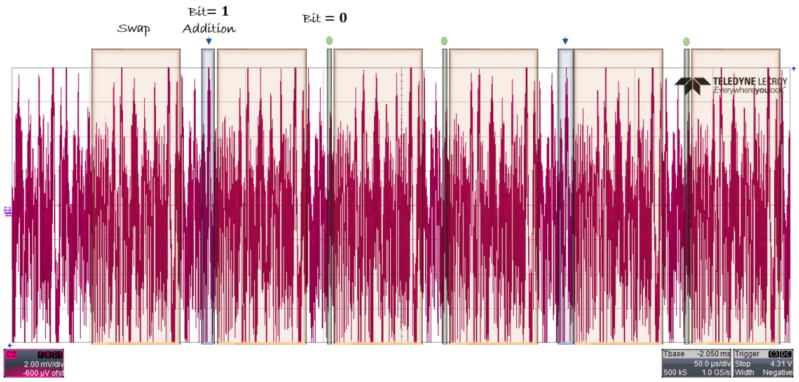

Fig. 5. Trace of the first column in Gaussian elimination process after application of randomization

- implementing a constant-time algorithm in which the additions in $\mathbb{F}_2[x]/(P_m)$ are independent from the processed coefficients as presented in [14].

These solutions require an additional element in $GF(2^m)$.
With these solutions an attacker is no longer able to exploit patterns according to the processed bit.
However, these solutions are subjected to others side-channel attacks that are not covered in this paper such as Differential Power Analysis (DPA) attacks. .

With the implementation of the second countermeasure, we observe in Figure 6 a uniformization of the trace due to the added noise.

Algorithm 8: Gaussian elimination algorithm

Input: Matrix $M \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Output: Matrix M in row echelon form and the rank $rank$ of the matrix

```
1  Dim ← 0
2  Temp ← 0
3  for i = 0 to m − 1 do
4      for j = i to n − 1 do
5          if M_{j,i} = 1 then
6              // The row j is a pivot
7              row i ↔ row j
8              rank ← rank +1
9              break
10     for k = row i + 1 to n do
11         if M_{k,i} = 1 then
12             row k ← row k + row i
13         else
14             Temp ← row k + row i
15 return (M, rank)
```

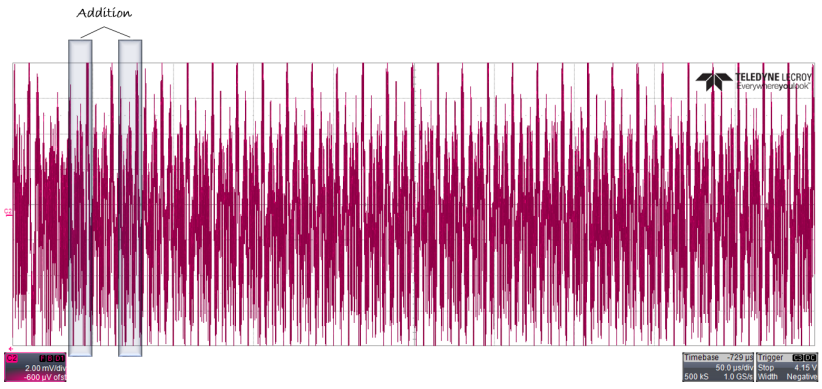

Fig. 6. Trace of the first column in Gaussian elimination with dummy operations.

As we can see in Table 9, regarding the first countermeasure, exchanging two rows at each iteration has a significant impact on the execution time of decapsulation process, increasing it by about 50%. The second countermeasure impacts the execution time by about 40%.

Conclusion

In this paper, we have highlighted that ROLLO-I can be implemented in a constrained environment and the structure used allows the cryptosystem to benefit

		Decapsulation		
Security		With randomization	With dummy operation	Without countermeasures
128	cycles ($\times 10^6$)	8.09	5.84	4.31
	ms	161.8	116.6	86.3
192	cycles ($\times 10^6$)	17.01	11.23	7.8
	ms	340.2	224.6	156
256	cycles ($\times 10^6$)	32.45	21.62	15.54
	ms	649	432.4	310.8

Table 9. Executing time of ROLLO-I decapsulation with countermeasures.

from the current crypto co-processor. We have also shown that in comparison with existing algorithms such as ECDH, our implementation's performances were compelling.
Moreover, we have provided a first side-channel attack on ROLLO-I as well as countermeasures against the proposed attack.
For future works, it will be interesting to look up some optimizations in time for operations in $\mathbb{F}_{q^m}[X]/(P_n)$ and extend the study to ROLLO-II and ROLLO-III.

References

1. Ingo Maurich and Tim Güneysu. Towards side-channel resistant implementations of qc-mdpc mceliece encryption on constrained devices. volume 8772, pages 266–282, 10 2014.
2. Tania Richmond, Martin Petrvalsky, and Milos Drutarovsky. A Side-Channel Attack Against the Secret Permutation on an Embedded McEliece Cryptosystem. In *3rd Workshop on trustworthy manufacturing and utilization of secure devices - TRUDEVICE 2015*, Grenoble, France, March 2015.
3. Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zémor. Low rank parity check codes and their application to cryptography. 04 2013.
4. Nicolas Aragon, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, and Gilles Zémor. Low Rank Parity Check Codes: New Decoding Algorithms and Applications to Cryptography. *CoRR*, abs/1904.00357, 2019.
5. Carlos Aguilar Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Adrien Hauteville Philippe Gaborit, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC second round submisssion : ROLLO - Rank-Ouroboros, LAKE & LOCKER, 2017.
6. National Instute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, 2016.
7. Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, Berlin, Heidelberg, 2003.
8. H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Discrete Mathematics and Its Applications. CRC Press, 2005.
9. André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations, 2006. aweimerskirch@escrypt.com 13331 received 2 Jul 2006.

10. Eugene Luks, Ferenc Rakoczi, and Charles Wright. Some Algorithms for Nilpotent Permutation Groups. *J. Symb. Comput.*, 23:335–354, 04 1997.
11. IAR Embedded Workbench.
12. SEC 1. Standards for Efficient Cryptography Group: Elliptic Curve Cryptography - version 2.0, 2009.
13. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
14. Florian Caullery Rusydi H. Makarim Marc Manzano Chiara Marcolla Carlos Aguilar-Melchor, Emanuele Bellini and Victor Mateu. Constant-timealgorithmsforROLLO. 2019.
15. Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Olivier Blazy Loïc Bidoux, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, and Valentin Vasseur. NIST PQC submisssion : BIKE - Bit Flipping Key Encapsulation, 2017.
16. Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Olivier Blazy Loïc Bidoux, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. NIST PQC submisssion : Hamming Quasi-Cyclic (HQC), 2017.
17. Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Olivier Blazy Loïc Bidoux, Jean-Christophe Deneuville, Philippe Gaborit, Gilles Zémor, Alain Couvreur, and Adrien Hauteville. NIST PQC submisssion : Rank Quasi-Cyclic (RQC), 2017.
18. Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Adrien Hauteville Philippe Gaborit, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC first round submisssion : LAKE - Low rAnk parity check codes Key Exchange, 2017.
19. Nicolas Aragon, Olivier Blazy, Slim Bettaieb, Loïc Bidoux, Jean-Christophe Deneuville, Adrien Hauteville Philippe Gaborit, Olivier Ruatta, and Gilles Zémor. NIST PQC first round submisssion : LOCKER - LOw rank parity ChecK codes EncRyption , 2017.
20. Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Adrien Hauteville Philippe Gaborit, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC first round submisssion : Ouroboros-R , 2017.
21. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.
22. Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
23. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange—A New Hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, 2016. USENIX Association.
24. Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $F_{2m}[x]$ on Cortex-M4 to Speed up NIST PQC candidates. In *ACNS*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2019.
25. Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):243–266, 2018.
26. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

27. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
28. Philippe Delsarte. Bilinear forms over a finite field, with applications to coding theory. *J. Comb. Theory, Ser. A*, 25:226–241, 1978.
29. R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

Appendix A Algorithm

In this section, we present in Algorithm 9 the inversion of binary polynomials in $\mathbb{F}_2[x]/(P_m)$ that we implemented and raised in Section 2.

Algorithm 9: Inversion in $\mathbb{F}_2[x]/(P_m)$

Input: a a non zero binary polynomial of degree at most $m-1$
Output: $a^{-1} \mod P_m$

$u \leftarrow a$, $v \leftarrow P_m$
$g_1 \leftarrow 1$, $g_2 \leftarrow 0$
while $u \neq 1$ **do**
  $j \longleftarrow deg(u) - deg(v)$
  if $j < 0$ **then**
    $u \leftrightarrow v$
    $g_1 \leftrightarrow g_2$
    $j \leftarrow -j$
  $u \leftarrow u + x^j v$
  $g_1 \leftarrow g_1 + x^j g_2$
return g_1