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 analyses a side-channel attack and the associated countermeasures. The first implementation benefits from 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 that is publicly available on GitHub. Finally, the side-channel attack allows us to recover the 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. Amongst these submitted candidates, 8 signature schemes based on lattices and multivariate and 17 public-key encryption schemes or key-encapsulation mechanisms (KEMs) are basing their security on codes, lattices, and isogenies. In addition to that, one more signature scheme based on a zero-knowledge proof system has also been submitted.
In this paper, we focus our analysis on submissions based on codes. The firsts 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 have resulted in a considerable reduction of key sizes and thus reach 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 concerning the memory which becomes binding when we have to implement the algorithms into constrained environments as in microcontrollers. It is then hardly conceivable to imagine that these cryptosystems may replace the ones used today in 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, the first one on a microcontroller featuring a crypto co-processor and a second one which is full software.
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 others code-based cryptosystems in round 2 Classic McEliece, LEDAcrypt and NTS-KEM, not listed in Table 1, use very large key sizes and thus were not taken into account in our study.

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. As operations on ROLLO-II and ROLLO-III are similar, they should be integrated quickly. To provide a first secure implementation of ROLLO-I, we propose countermeasures against the side-channel attack we introduced.

Our contribution. In this paper, we present two practical implementations of ROLLO-I, the first one consisting in full software implementation and the second one is embedded in a microcontroller in which 4 kB of RAM are dedicated to cryptographic data.
We finally give a first study on the security of ROLLO-I against side-channel attacks and implement countermeasures against this attack.

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 [1]. For more details, the reader is referred to [2,3]. For fixed prime numbers m and n, we denote by:

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

Let k, n be two integers such that $k \geq n$. A linear code 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 can be represented by its generator matrix $\mathbf{G} \in \mathbb{F}_{q^m}^{k\times n}$ as

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

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

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

The matrix $\mathbf{s}_x = \mathbf{H}.\mathbf{x}^T$ is called the syndrome of $\mathbf{x}$.
ROLLO cryptosystem is based on codes embedded with 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

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

with $M(\mathbf{v}) = (v_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq m}}$ 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 [3] 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 $P \in \mathbb{F}_q[X]$ 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 $(\mathbf{g}_{i,j})_{i\in[1,\cdots,s-t]} \in \mathbb{F}_{q^m}^n$, is an ideal code if its generator matrix under systematic form is given by

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

In [3], they restrain the definition of ideal LRPC (Low-Rank Parity Check) codes to $(2, 1)$-ideal LRPC codes that they used in ROLLO cryptosystem.
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 $\text{Supp}(\mathbf{h}_1, \mathbf{h}_2) = F$, and $P \in \mathbb{F}_q[X]$ be a polynomial of degree n. An $[2n, n]_{q^m}$ code $\mathcal{C}$ is an ideal LRPC code if its parity check matrix is 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 that presents small parameter sizes compared to 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) detailed in Table 4. During the decapsulation process, the syndrome of the received ciphertext $\mathbf{c}$ is first computed, then the Rank Support Recovery (RSR) algorithm is performed to recover the error's support. The latter is explained in Appendix 9 .

The fixed parameter sets given in Table 3 allow achieving respectively 128, 192, and 256 bits level of security according to NIST's security strength categories 1, 3, and 5 [4]. 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 $F_{q^m}[X]/(P_n)$. We note that for the three security levels, q is set to 2. The parameters d and r correspond respectively to the private key and error's rank.

Param. / Algo.	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 cipher
		$\mathbf{c} = \mathbf{e_2} + \mathbf{e_1} \cdot \mathbf{h} \mod P_n$
		Derive the shared secret
Decapsulation	$\xleftarrow{\mathbf{c}}$	$K = \text{Hash}(E)$
Compute the syndrome		
$\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \mod P_n$		
Recover 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

We give in this section some details on the implementation of operations in the rings $\mathbb{F}_2[x]/(P_m)$ and $\mathbb{F}_{2^m}[X]/(P_n)$ required in ROLLO-I cryptosystem. The implementations have been performed on 32-bit architecture systems.

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

The addition in $\mathbb{F}_2[x]/(P_m)$ has been implemented with XORs between 32-bit words. Thus, the three main operations to implement were the multiplication, the modular reduction, and the inversion. For the latter, we chose the Euclidean extended algorithm for binary polynomials described in [5] and given in Appendix A.

2.1.1 Modular reduction

Several modular reductions with parse polynomials are performed in the cryptosystem, we then decided to use the same technique explained in [5] concerning the reduction of binary trinomial with middle terms close to each other. Let us take the example of ROLLO-I-128 and the modular reduction of an element $\mathbf{C} = (c_0, \cdots, c_{156})$ obtained after a multiplication in $\mathbb{F}_2[x]/(P_m)$. The reduction can then be performed on each 32-bit word composing $\mathbf{C}$. Considering

the reduction of the 4th word of $\mathbf{C}$, $\mathbf{C}[3]$ that 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 the above congruences, we notice that the reduction of $\mathbf{C}[3]$ can be operated by adding two times $\mathbf{C}[3]$ to $\mathbf{C}$ as shown in Figure 1.

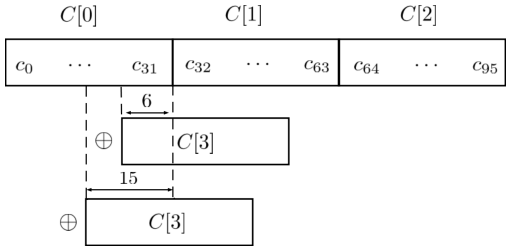

Fig. 1. Reduction of the 32-bit word $\mathbf{C}[3]$ modulo $P_m(x) = x^{79} + x^9 + 1$

Based on this method, we implemented fast not generic modular reduction algorithms (e.g Algorithm 1) for each security level.

Algorithm 1: 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

2.1.2 Multiplication

Finally, for the multiplication, we used the left-to-right comb method with windows of width 4 as described in [5, Algo. 2.36]. The left-to-right comb method is based on the fact that if $b(x).x^i$ has already been computed for $0 \leq i \leq w - 1$,

with w a chosen width window, then $b(x).x^{wj+i}$ can be determined by adding j zero w-bit words to the right of the vector representation of b. For each polynomial $a \in \mathbb{F}_q[x]/(P_m)$, we have the associated vector form $A = (a_0, a_1, \cdots, a_{m-1})$, as we work in a 32-bit architecture, let A_i denotes the ith 32-bit word of A.
Considering the multiplication between two polynomials $a, b \in \mathbb{F}_q[x]/(P_m)$, the first step consists in precomputing the products $u(x) \times b(x)$ for all polynomials u of degree at most $w-1$, in our case, 16 elements have been stored in a table T. Let $\hat{u}$ denotes the integer associated to the polynomial $u(x)$ ($\hat{u} = 0 \leftrightarrow u(x) = 0, \hat{u} = 1 \leftrightarrow u(x) = 1, \hat{u} = 2 \leftrightarrow u(x) = x, \cdots, \hat{u} = 15 \leftrightarrow u(x) = x^3 + x^2 + x + 1$), we have then $T_{\hat{u}} = b(x) \times u(x)$.
In the second step, each word A_i is first divided in t blocks of w coefficients, $A_{i,j}$ denotes then the jth block of 4 coefficients in A_i. Then, the element $T_{\hat{u}}$ is added to the result element R_j.
Finally, if i is non zero, we multiply the polynomial R by x^w, this amounts to a shift of 32-bit words.

Algorithm 2: Polynomial multiplication using the left-to-right method with a width window w

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

```
// Step 1
For all polynomials u(x) of degree at most w − 1, compute T_û = b(x) × u(x)
R ← 0
for i from (32/w) − 1 to 0 do
    for j from 0 to ⌈m/32⌉ do
        Let û = u_{w−1} ··· u_0 with u_k the bit wi + k of A_j.
        R_j ← R_j ⊕ T_û
    if i ≠ 0 then
        R ← R.x^w
return R
```

2.2 Operations 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 Inversion

For the inversion, we adjusted Extended Euclidean algorithm given in Appendix A to the ring $\mathbb{F}_{2^m}[X]/(P_n)$ as presented in Algorithm 3.
However, the implementation of this operation can be quite expansive in terms of memory usage. 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 performed intermediates operations (swap between polynomials, results of multiplications).

Algorithm 3: 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

A simple way to implement the Extended Euclidean algorithm is to allocate the maximum memory size for each element. As each element can be composed of n coefficients in $\mathbb{F}_{2^m}$, we should have $8 \times n \times m_b$ bytes required to compute the inverse of a polynomial in $\mathbb{F}_{2^m}[X]/(P_n)$. Considering the parameters of ROLLO-I-128, ROLLO-I-192 and ROLLO-I-256 the memory usage represents respectively 4512, 5088, and 8576 bytes, thus exceeding the memory size available on the target microcontroller.
However, during the Extended Euclidean 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 and decrease during the algorithm execution.
- the degrees of two Bézout coefficients are 0 at the beginning and increase during the process.

Thus, we decided to perform a dynamic memory allocation by allocating the necessary memory space to each element and moving them during the inversion process allowing us to reduce the memory usage to 2590, 2904 and 4864 bytes respectively for ROLLO-I-128, ROLLO-I-192 and ROLLO-I-256.

2.2.2 Multiplication

The multiplication in $\mathbb{F}_{2^m}^n$ is one of the most used operations of this cryptosystem: it is involved in the computation of the public key, the cipher and the

syndrome. The Schoolbook multiplication requires n^2 multiplications in $\mathbb{F}_{2^m}$, this can be reduced by implementing a combination of Schoolbook multiplication and Karatsuba method [6] as presented in Algorithm 7.

Let $P = p_0 + p_1 X$ and $Q = q_0 + q_1 X$ be two polynomials of degree 1. The result of the product is

$$P \cdot Q = p_0 q_0 + (p_0 q_1 + p_1 q_0) X + p_1 q_1 X^2.$$

Naively, we have to compute 4 multiplications and 1 addition. The Karatsuba algorithm is based on the fact that:

$$(p_0 q_1 + p_1 q_0) = (p_0 + p_1)(q_0 + q_1) - p_0 q_0 - p_1 q_1.$$

The Karatsuba algorithm takes advantage of this method which leads the computation of PQ to require only 3 multiplications and 4 additions.

Algorithm 4: 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** $R_1 + R_6 x^{N^{'}} + R_2 x^{2N}$

The fourth step (line 4 - Algorithm 7) requires to divide the polynomial's length by 2, as consequence, we have to add a padding to the polynomials involved in the multiplications with zero coefficients to make the number of coefficients of the polynomials even.
Figure 2 provides the number of cycles depending on the polynomial length required by the combination of Karatsuba and schoolbook method, we observe that the cycles' number is not strictly increasing, this is due to the division by 2 of the polynomial's length involved in the method. Depending on the memory available for a multiplication in $\mathbb{F}_{2^m}[X]/(P_n)$, we can then choose to add more or less padding. For example, in ROLLO-I-128 with $n = 47$, we decided to add one zero coefficient which induces 48 coefficients and allow us to reduce

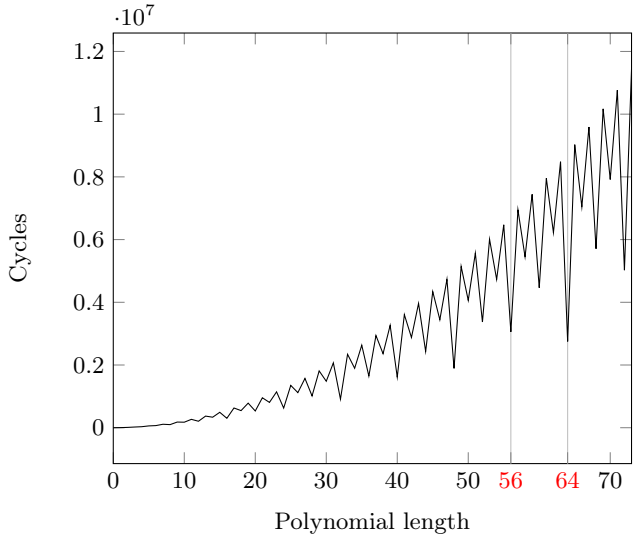

Fig. 2. Number of cycles required by Karatsuba combined with schoolbook multiplication depending on the polynomial length

considerably the number of cycles; however, in ROLLO-I-192 with $n = 53$ we have two possibilities:

- Pad the polynomials with 3 coefficients to reach 56 coefficients.
- Pad with 11 coefficients to lower the cost of multiplications in $\mathbb{F}_{2^m}$.

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. The first choice represents then a good balance between memory and execution time.

2.2.3 Rank Support Recovery (RSR) algorithm

The RSR algorithm involves the computation of intersections between two subspaces 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 two 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 $\mathcal{I}_{U,V} = U \cap V$ can be computed by following the Zassenhaus algorithm [7], described with the below steps:

- 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: $\begin{pmatrix} M(\mathbf{c}) & * \\ 0 & \mathcal{I}_{U,V} \\ 0 & 0 \end{pmatrix}$, with $\mathbf{c} = (c_0, \cdots, c_{n-1}) \in \mathbb{F}_{2^m}^n$.

In the initial RSR algorithm given in Appendix 9, some pre-computations have been performed to avoid the recalculation of some data. We can estimate the average memory cost of these pre-computations: each S_i is composed of rd coefficients in $\mathbb{F}_{2^m}$, thus, for the S_i pre-computations, $rd \times d \times m_b$ bytes are needed. Concerning the pre-computation of intersections, each composed of r coefficients, we need to consider the memory usage induced by the Zassenhaus algorithm described below. To be performed, the latter requires the writing in memory of four S_i, in other words $4rd \times m_b$ bytes. Furthermore, for these pre-computations, the private key's support (d coefficients) and the syndrome (rd coefficients) are needed. Thus, supposing that we computed the last intersection between two S_i, the total memory cost should be:

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

With this formula, we can predict that ROLLO-I-128 should required 4212 bytes to store the pre-computations which is too high. In order to reduce the memory cost, we grouped the two pre-computations by storing in memory at most three S_i and then directly computing two intersections as framed in Algorithm 5.

Algorithm 5: RSR (Rank Support Recover)

Input: $F = \langle f_1, \cdots, f_d \rangle$ a $\mathbb{F}_q$ vector 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 $S_{i,i+2} = tmp_i \cap tmp_{i+2}$
    $tmp_{i\%3} \leftarrow S_{i+3}$
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

By repeating the same above reasoning, the total memory cost should be:

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

In this case, ROLLO-I-128 should required 3132 bytes. Indeed, 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 Results

In this section, we present the performance evaluation of proposed implementations regarding memory usage and execution time. Our implementations were implemented in C. For performance measurements, we used IAR compiler C/C++ with high-speed optimization level and counted the cycles with the debugging functionality of the IAR Embedded Workbench IDE [8].
ROLLO-I-128 and ROLLO-I-192 have also been implemented on an FPGA Xilinx Virtex-II. The microcontroller which is based on a widely used 32-bit ARM® SecurCore® SC300™and has an embedded 32-bit mathematical crypto co-processor to perform operations in $GF(p)$ and $GF(2^m)$. The microcontroller features 24 kB of RAM whose 4 kB are available to the cryptographic computations, and a True Random Number Generator (TRNG). In our embedded implementations, all the operations in $GF(2^m)$ take then advantage of the crypto co-processor leading the embedded implementations ROLLO-I-128 and ROLLO-I-192 to be faster than their full software versions as we can seen on Table 6 that provides the number of cycles required by ROLLO-I for the different security levels. The microcontroller running to 50 MHz, we also provided the time in milliseconds.

		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 6. Executing time of ROLLO-I

Table 7 gives the memory usage to perform the key encapsulation mechanism for the three levels of security. The table highlights that the ROLLO-I-256 implementation exceed significantly the 4kB of RAM, so it could not be implemented in our target.
To compute the memory usage, we also have to difference the full software implementations and ones embedded in the microcontroller. For the latter, the memory usage referring to the RAM required to perform the cryptosystem, the keys being stored in the EEPROM (Electrically Erasable Programmable Read-Only memory). In contrast, for the full software implementations that can be

processed on any platform, the space required for keys have been taken into account in the computation of the memory usage.

For the ROLLO-I implementation on a 32-bit architecture, an element in $GF(q^m)^n$ will be represented as $n \times \lceil m/32 \rceil \times 4$ bytes. Considering this fact, the memory usage of ROLLO-I-128 and ROLLO-I-192 will only differ according to n, indeed for ROLLO-I-128, $m = 79$ and for ROLLO-I-192, $m = 89$, we thus obtain $\lceil 79/32 \rceil = \lceil 89/32 \rceil = 3$ 32-bit words. Nevertheless, each element in $GF(q^m)$ for ROLLO-I-256 requires one additional 32-bit word explaining the important difference of memory usage between the higher security and the two lowers.
In order to reduce the memory used in the full software implementation, we decided to not keep the two parts of the secret key. Indeed, as we notice in the ROLLO-I cryptosystem (Table 4), the part $\mathbf{y}$ of the secret key is only used on the key generation process. We can supposed that this part could be used to prove the integrity of the other part $\mathbf{x}$ from the public key $\mathbf{h}$. Thus, instead of storing the part $\mathbf{y}$, we decided to store the cyclic redundancy check (CRC) of $(\mathbf{x}, \mathbf{y})$ leading us to keep the proof of integrity of $\mathbf{x}$. The result of CRC is then stored in a 32-bit word.

	Full software			Embedded		
Security \ Algo.	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 7. Memory usage for ROLLO-I (in bytes)

To give a rough idea of ROLLO-I cryptosystem's place in cryptography used today, we decided to compare our implementation full software of ROLLO-I with the Elliptic Curve Diffie-Hellman key exchange (ECDH) [9] implemented in the same platform. To establish a shared secret between two entities, the ECDH protocol required 2 scalars multiplications over $E(\mathbb{F}_q)$ that are executed in parallel by these two entities. In the case of ROLLO-I, the key agreement takes into account the Encapsulation and Decapsulation processes.
Thus, Table 8 gives the performances of a key agreement for ECDH and ROLLO-I. For the cost's estimation of ECDH, we only considered 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.

Table 8 highlights that ROLLO-I could be a realistic alternative to the current key exchange schemes.

3 Side-channel attack on ROLLO-I

Side-channel attacks, were first introduced by Kocher in 1996 [10]. Amongst these attacks, some of them are based on the exploitation of the leakage information coming from a device executing a cryptographic protocol. An adversary is then able to extract this information without having to tamper with the device.
In this section, we deal with chosen-ciphertext Simple Power Analysis (SPA) attack which aims to reveal the key by identifying sequences of an algorithm thanks to the observation of its power traces.

3.1 Attack explanation

ROLLO-I submission does not require the use of ephemeral keys, this means that the encapsulation and decapsulation processes can be performed several times using the same key pair $((\mathbf{x}, \mathbf{y}, \mathbf{F}), \mathbf{h})$. The attack presented in this section leads us to recover the private key $\mathbf{sk} = (\mathbf{x}, \mathbf{y})$ that is used to establish the shared secret between two entities.
Decapsulation process is a good target for side-channel attacks, indeed it involves the secret key $\mathbf{x}$ during the syndrome computation

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

The second part of the secret key $\mathbf{y}$ may be recover by computing

$$\mathbf{y} = \mathbf{x} \cdot \mathbf{h} \bmod P_n,$$

with $\mathbf{h}$ the public key.
We note that ROLLO-I cryptosystem has not been proven IND-CCA2 implying that we can give to the decapsulation process chosen ciphertexts. We then targeted the syndrome's support computation $\mathbf{S}$ that occurred in the RSR algorithm described in Algorithm 5. The operation is performed by applying the Gaussian elimination algorithm to the matrix associated to the syndrome $\mathbf{s}$ as described in Algorithm 6 that highlights the points that could be seen as sources of leakage. Indeed, the two "if" conditions allow us to recover some bits by operating a different treatment when the bit is 1 or 0.
The rank of the syndrome is at most $r \times d$. In other words, at the end of the process, we must have a matrix in row echelon form:

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

Algorithm 6: Gaussian elimination algorithm

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

```
1  Dim ← 0
2  for i = 1 to m do
3      for j = 1 to n do
4          if M_{j,i} = 1 then
5              // The line j is a pivot
6              line i ↔ line j
7              Dim ← Dim +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, Dim)
```

However, we can only consider the first column, recovering the other columns should required some suppositions on coefficients, that are not processed during the Gaussian elimination, to solve equation systems.
Thus, the attack can be realized by m rotations of the initial ciphertext that will involve m rotations of the matrix $M_{\boldsymbol{s}}$ modulo P_m and thus allow us to recover the syndrome. Specifically, we take a ciphertext that has been sent to the decapsulation, after the ith decapsulation, we multiply the ciphertext by x^i modulo P_m.
It is straightforward to determine the first column since each pattern allows us to directly recover the coefficient on each row of the matrix $M_{\mathbf{s}}$.
Determining the first pivot amounts to find the first coefficient to 1 in the column. During the Gaussian elimination process, each coefficient in the column is scanned and operation is performed or not according to the coefficient value. We can distinguish two cases:

1. The first coefficient processed is 1, it is then the pivot, no operation is performed and we can directly go to the second loop for (line 9 - Algorithm 6). We can deduce that the first coefficient is 1.
2. The first coefficient is 0, so the algorithm consists in finding the first non-zero coefficient in the column. Once the coefficient to 1 is found, the corresponding line is exchanged with the pivot line. The time required to determine the pivot indicates the number of coefficient processed and allows us to recover the coefficient to 0 in the previous rows as well as the pivot line. In this case, we go to the second loop **for** after the permutation of the two rows.

The second loop **for** (line 9 - Algorithm 6) allows to remove the other coefficients to 1 in the column. Specifically, all the column is scanned from the pivot line and two different treatments are performed on the other coefficients:

1. The coefficient is 0 and then no operation is performed.

2. The coefficient is 1 and then an addition in $\mathbb{F}_{2^m}$ is performed between the pivot row and the one processed.

This difference of treatment led us to determine on which row the coefficient is 1 or 0. At this stage, we obtained a matrix $M_{\mathbf{s},0}$ in which we knew the first column.
As explained before, the remaining coefficients can be recovered by a rotation of the matrix before the Gaussian elimination process. This rotation is performed by multiplying the ciphertext by $x^i \in \mathbb{F}_2[x]/(P_m)$, with $0 < i < m$, inducing m different matrices $M_{\mathbf{s},i}$ that could be viewed as a rotation of the matrix $M_{\mathbf{s},0}$ modulo P_m. The modular rotation has to be taken into account during the recovering of the columns' syndrome matrix. For example, considering the ROLLO-I-128 parameters given in Table 3, multiplying the ciphertext by x modulo $P_m = x^{79} + x^9 + 1$ implies that the last column of the matrix syndrome is xored with the columns indexed by 0 and 9 as presented in Figure 3.

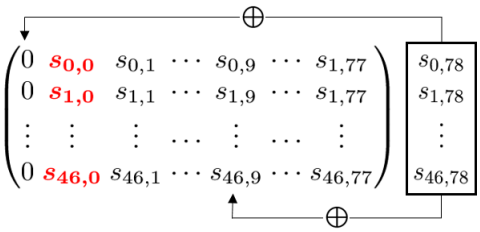

Fig. 3. Example of modular rotation for the syndrome's matrix for ROLLO-I-128

The column 78 is recovered by applying the SPA as previously explained. To recover the column 9, we had to multiply the ciphertext by x^{69} and be aware that the recovering column corresponded to column 9 xored with column 78 that is already known at this step of the attack. In the case of ROLLO-I-128, this consideration concerns columns indexed by i with $1 \leq i \leq 9$. The same is true for ROLLO-I-256 and for ROLLO-I-192, columns indexed by i with $1 \leq i \leq 38$ are concerned. The attack results are set out and discussed in the section 3.3.

3.2 Countermeasures

In this section, we focus on solutions allowing the cryptosystem to be secured against the attack explained in the Section 3.1. The use of ephemeral keys or an INC-CCA2 scheme would allow ROLLO-I to be resistant against the proposed attack. But ROLLO-I has not be proven IND-CCA2 and the generation of keys being generally performed once in the life cycle of a component then it is complicated to consider the use of ephemeral keys on cryptosystems. We then focus on protecting the Gaussian elimination algorithm by adding noise to the leakage making it independent of the manipulating data. Two countermeasures, as presented in Algorithm 7, can be adopted:

- Treat randomly the research of pivot in a column and the processed line for the transformation in row echelon form.

- Add dummies operations in the case in which the processed bit is 0.

Algorithm 7: Gaussian elimination with countermeasures

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

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

For the first countermeasure, an attacker will not be able to recover the indices of the pivot and the first processed line. For example, for ROLLO-I-128, the attacker will have 47 possibilities for the pivot and 46 possibilities for the index of the first line processed for each columns. Then the complexity of the SPA attack is $(47 \times 46)^{79}$ which corresponds to about 2^{869} operations.
The second countermeasure complicates the SPA attack, in any case, an addition in $\mathbb{F}_2[x]/(P_m)$ is performed, then the attacker will not be able to distinguish the treatment pattern of a bit to 1 or 0. The countermeasures required an additional 32-bit word to store the result in the case of a bit to 0.
We notice that side-channel attacks can be performed on the two countermeasures implemented separately. Indeed, in the first case, an attacker can replay the same ciphertexts and recover the order of elements in a column and in the second case, a Correlation Power Analysis (CPA) should be performed on the addresses of registers in order to recover the treatment of a bit to 1 or 0. However, the combination of both complicates any attack, an attacker should not be able to distinguish any pattern and the randomization added in the search of the pivot and processed line disturbs the alignment of traces making the CPA attack harder.

3.3 Results

To develop this attack, we used the implementation on the microcontroller in which all the operations in $GF(2^m)$ take advantage of the crypto co-processor. For the experiment, we consider parameters of ROLLO-I-128, $n = 47$, $m = 79$. The secret key $\mathbf{x}$ and ciphertext $\mathbf{c}$ involved in the syndrome computation have been generated during the Key Generation and Encapsulation processes. As expected, we observe in the trace given in Figure 4 the difference of patterns between the treatment of a bit to 1 and a bit to 0 that led us to recover the first column of the syndrome's matrix corresponding to

$$10110101110100010101110011110010011100101 10.$$

ROLLO-I-128 traces have been captured with a Lecroy SDA 725Zi-A oscilloscope.

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

The same patterns appeared on the other columns after the matrix rotation allowing us to recover the syndrome.
Figure 5 provides the trace of Gaussian elimination process with the implementation of countermeasures presented in Algorithm 7. We can observe that no pattern can be distinguished and we are not able to know if the first operation corresponds to the first row of the matrix.

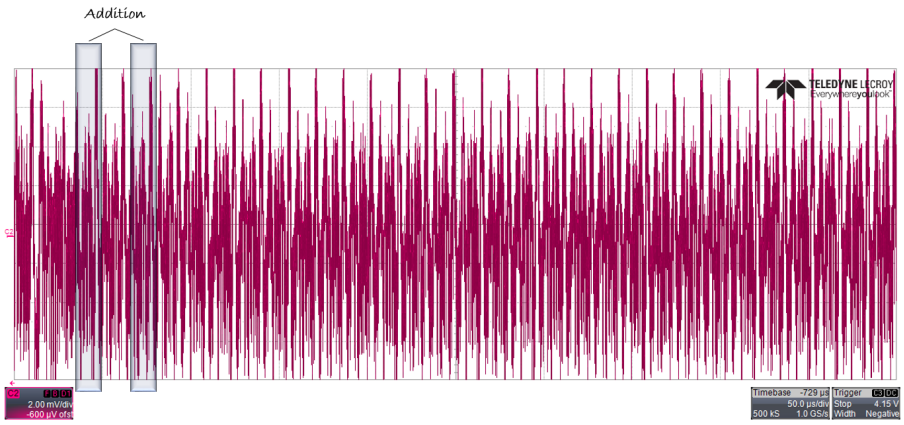

Fig. 5. Trace of the first column in Gaussian elimination process after application of countermeasures

The countermeasures only impact the decapsulation process by increasing its execution time as we can see on Table 9 that gives the number of cycles for the three security level of ROLLO-I. Then, we can see in the Table that the countermeasures' implementation increases by about 30% the execution time of the decapsulation process but it stays reasonable given the attack.

		Decap	
Security		With countermeasures	Without countermeasures
ROLLO-I-128	cycles ($\times 10^6$)	6.43	4.31
	ms	128.6	86.3
ROLLO-I-192	cycles ($\times 10^6$)	12.54	7.8
	ms	250.8	156
ROLLO-I-256	cycles ($\times 10^6$)	23.92	15.54
	ms	478.4	310.8

Table 9. Executing time of ROLLO-I with countermeasures

Conclusion

In this paper, we highlighted that ROLLO-I can be implemented in a constrained environment and by the structure used in the cryptosystem, the latter can even benefit from actual crypto co-processor. We also shown that our implementation can compete in terms of performances with existing algorithms such as ECDH. Moreover we provided a first side-channel attack on ROLLO-I as well as countermeasures against the proposed attack.
For future work, it will be interesting to look up some optimizations in time concerning operations in $\mathbb{F}_{q^m}[X]/(P_n)$ and also continue the study with ROLLO-II and ROLLO-III.

References

1. Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zémor. Low rank parity check codes and their application to cryptography. 04 2013.
2. 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.
3. 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.
4. National Instute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, 2016.
5. Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003.
6. André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations, 2006. aweimerskirch@escrypt.com 13331 received 2 Jul 2006.
7. Eugene Luks, Ferenc Rakoczi, and Charles Wright. Some Algorithms for Nilpotent Permutation Groups. *J. Symb. Comput.*, 23:335–354, 04 1997.
8. IAR Embedded Workbench.
9. SEC 1. Standards for Efficient Cryptography Group: Elliptic Curve Cryptography - version 2.0, 2009.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.

17. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.
18. 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.
19. 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.
20. Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $\mathbb{F}_{2m}[x]$ on Cortex-M4 to Speed up NIST PQC candidates. In *ACNS*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2019.
21. 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.
22. 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`.
23. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
24. Philippe Delsarte. Bilinear forms over a finite field, with applications to coding theory. *J. Comb. Theory, Ser. A*, 25:226–241, 1978.
25. 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 8 the inversion of binary polynomials in $\mathbb{F}_2[x]/(P_m)$ that we implemented and raised in Section 2.

Algorithm 8: 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

Appendix B Rank Support Recovery algorithm

In Algorithm 9, the support S is a subspace of EF given by:

$$EF = \langle \{ef, e \in E \text{ and } f \in F\} \rangle ,$$

with $\text{Rank}(E) = r$ and $\text{Rank}(F) = d$, then $\dim(S) \leq rd$.
In the RSR algorithm, the loop **for** (line **4** - Algorithm 9) allows to recover the whole vector space EF in case of $\dim(S) < rd$.
For the failure analysis, we let the lecturer refers to [3].
Once $S = \langle EF \rangle = \langle e_1 f_1, \cdots, e_r f_1, \cdots, e_1 f_i, \cdots e_r f_i, \cdots, e_r f_d \rangle$, since $S_i = f_i^{-1} S$, we have for all $1 \leq i \leq d$,

$$E \subset S_i \Rightarrow E = \bigcap_{1 \leq i \leq d} S_i.$$

In rank metric code-based cryptography, the support recovery is considered as a hard problem, ROLLO bases a part of its security proof on the **2-Ideal Rank Support Recovery** (2-IRSR) [3] problem that consists in, given a polynomial $P \in \mathbb{F}_q[X]$ of degree n, vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{F}_{q^m}^n$, and a syndrome $\mathbf{s}$, recovering the support E of $(\mathbf{e_1}, \mathbf{e_2})$ with $\dim(E) \leq r$ and such that:

$$\mathbf{e_1 x} + \mathbf{e_2 y} = \mathbf{s} \mod P.$$

Algorithm 9: Rank Support Recovery (RSR) algorithm

Input: A $\mathbb{F}_q$-subspace $F = \langle f_1, \cdots, f_d \rangle$, $\mathbf{s} = (s_1, \cdots, s_n)$ a syndrome of an error $\mathbf{e}$, r the error's rank weight
Output: A candidate E for the support of $\mathbf{e}$

1 Compute the support S of the syndrome $\mathbf{s}$
2 Precompute every $S_i = f_i^{-1} S$ for $i = 1$ to d
3 Precompute every $S_{i,i+1} = S_i \bigcap S_{i+1}$ for $i = 1$ to $d - 1$
4 **for** $i = 1$ *to* $d - 2$ **do**
5 $\quad tmp \leftarrow S + F(S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})$
6 $\quad$ **if** $\dim(tmp) \leq rd$ **then**
7 $\quad\quad S \leftarrow tmp$
8 $\quad$ **end**
9 **end**
10 $E \leftarrow \bigcap_{1 \leq i \leq d} f_i^{-1} S$
11 **return** E