Optimized implementation of the NIST PQC submission ROLLO on microcontroller

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

[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. We present in this paper an efficient implementation of the code-based cryptosystem ROLLO, a candidate to the NIST PQC project, on a device available on the market. This implementation benefits of the existing hardware by using crypto co-processor for ECC contained in a microcontroller to speed-up operations in $\mathbb{F}_{2^m}$. Optimizations are then made on operations in $\mathbb{F}_{2^m}^n$. Finally, the cryptosystem outperforms the public key exchange protocol ECDH for a security level of 192 bits showing then the possibility of the integration of this new cryptosystem in current chips. According to our implementation, the CPA-secure ROLLO-I-128 submission takes 173,6 ms for key generation, 12 ms for encapsulation and 79.4 ms for decapsulation on an embedded system including featured with a Cortex M3 core running at 50 MHz.

Keywords: post-quantum cryptography, optimization, embedded system, ROLLO

1 Introduction

In 2016, the National Institute of Standards and Technology (NIST) issued a report announcing the launch of an international process in order to propose new cryptographic schemes, resistant to a quantum computer. Since 2017, 69 proposals were accepted to the NIST post-quantum project.
After a year of study, withdrawn and merge schemes, the NIST reduced the candidates' list by announcing the second round and the 26 accepted submissions. Among these candidates, 8 signature schemes and 17 public-key encryption schemes or key-encapsulation mechanisms (KEMs) based their security on hard mathematical problems in codes, lattices, isogenies or multivariate. In addition to that, one more signature scheme based on a zero-knowledge proof system has been submitted.
In this paper, we focus on submissions based on codes. Code-based cryptography

was introduced by R. McEliece in 1978 [17] but the McEliece cryptosystem did not interest the cryptography community due to the use of large key size.
Cryptosystems based on lattices that use reasonable key size and are also quantum-resistant, have often been preferred over codes for practical implementations: in 2016, Google decides to integrate the lattice-based cryptosystem New Hope [3] to safeguard Chrome from quantum computers and NTRUEncrypt cryptosystem [11] was accepted as the X9.98 standard for financial transaction protection in 2011.
However, the development of news cryptosystems based on different codes from those used in the McEliece cryptosystem as well as the introduction of codes embedded with the rank metric have resulted in a considerable reduction of key sizes of code-based cryptosystems 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 required a large number of resources notably concerning the memory which becomes binding when we have to implement them into constraints environments as micro-controllers.
It is then hardly conceivable to imagine that these cryptosystems may replace the ones in use today in chips. In that sense, we decided to study the real cost of a code-based cryptosystem implementation. This study seems to be essential to prepare the transition to post-quantum cryptography.
For our study, we chose an embedded commercialized system to implement the targeted cryptosystem.
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 observe the memory required to store elements for different cryptosystems. The respective sizes are reported in Table 1. As we have only 4 kB of RAM, we implement ROLLO submission and more specifically ROLLO-I. Indeed, for this cryptosystem, the size of the public key and the ciphertext are by far the smallest. It is not the case for the secret key, but the cryptosystems in Table 1 with a small secret key have very big public key and ciphertext. Operations on ROLLO-II and ROLLO-III being similar, they could be integrated quickly.

Algorithm / Parameter	BIKE			HQC	RQC	ROLLO		
scheme number	I	II	III			**I**	II	III
public key	8188	4094	9033	14754	3510	**947**	2493	2196
secret key	548	548	532	532	3510	**1894**	4986	2196
ciphertext	8188	4094	9033	14818	3574	**947**	2621	2196

Table 1. Size of parameters in bytes to store in RAM for some code-based cryptosystems with security level 5

In view of Table 1, for practical implementation, we then decided to study the KEM cryptosystem ROLLO-I [18] which parameter sizes are well-balanced.
The second round submission ROLLO is a merge of the first round submis-

sions LAKE [6] (renamed ROLLO-I), Locker [5] (renamed ROLLO-II), and Ourobouros-R [7] (renamed ROLLO-III).

Our contribution. In this paper, we present two practical software implementations of ROLLO-I. In the best of our knowledge, it is the first implementations of this cryptosystem on an embedded system. The chosen target features a CORTEX-M3 processor and 4 kB of RAM are dedicated to cryptographic data. The first implementation is optimized depending on the memory and the second is optimized in time.
We finally prove the practicability of such an algorithm on real products by comparing the execution time with an Elliptic Curve Diffie-Hellman (ECDH) key exchange that is widely used today and implemented in the same target.

Organization of this paper. This paper is organized as follows. We start with some preliminary definitions in Section 2. We then present the ROLLO cryptosystem in Section 3 and our optimized implementations in Section 4. Finally, we expose our results in Section 5.

2 Background

We use the same notations as in [18]. For fixed prime numbers m and n, we denote by:

q	a prime number
$\mathbb{F}_q$	the finite field with q elements
$\mathbb{F}_{q^m}$	the finite field with q^m elements
$\mathbb{F}_{q^m}^n$	the vector space that can be identified with the ring $\mathbb{F}_{q^m}[X]/(P)$, with P a polynomial of degree n
$\mathbf{v}$	an element of $\mathbb{F}_{q^m}^n$
$M(\mathbf{v})$	the matrix $(v_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq m}}$

In this section, we recall some generalities on codes, more specifically rank metric codes. For more details, we let the reader refers to [8,18].

Let k, n 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 $\mathbb{F}_{q^m}^n$ of dimension k. It is denoted $[n, k]_{q^m}$.
A linear code can be represented by its generator matrix $\mathbf{G} \in \mathbb{F}_{q^m}^{k\times n}$ as:

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

If the generator matrix is of the form $\mathbf{G} = \big(I_k \mid A\big)$ with A an $(n-k)\times k$ matrix, we denote that $\mathbf{G}$ is under the systematic form.
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:

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

Thus, $\mathbf{s}_x = \mathbf{H}.\mathbf{x}^T$ is called the syndrome of $\mathbf{x}$.
In code-based cryptography, codes can be embedded with two different metrics: Hamming **??** and rank [9]. As ROLLO cryptosystem is based on codes embedded with rank metric over $\mathbb{F}_{q^m}^n$, we will leave aside the Hamming metric for the rest of this paper.
The metric allows us to define distance in codes. 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 as

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

with $M(\mathbf{v}) = (v_{i,j})_{\substack{1 \le i \le n \\ 1 \le j \le m}}$ and $\|\mathbf{v}\|$ is called the rank weight of the word $\mathbf{v} = \mathbf{x} - \mathbf{y}$.
The rank of a word $\mathbf{x}$ can also be seen as the dimension of its support given by

$$\text{Supp}(\mathbf{x}) = \langle x_1, \cdots x_n \rangle_{\mathbb{F}_q}. \quad (1)$$

In order to define codes used in ROLLO cryptosystem, we first need to define circulant and double circulant matrices.
An $n \times n$ circulant matrix is defined as a matrix where each row is rotated one element to the right depending on the preceding row:

$$\begin{pmatrix} a_0 & a_1 & \cdots & a_{n-2} & a_{n-1} \\ a_{n-1} & a_0 & \cdots & a_{n-3} & a_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 & a_2 & \cdots & a_{n-1} & a_0 \end{pmatrix}.$$

We denote the set of circulant matrices of size n over $\mathbb{F}_{q^m}$ as $Circ_n(\mathbb{F}_{q^m}) \subset \mathcal{M}_n(\mathbb{F}_{q^m})$. Thus, there exists an isomorphism

$$\begin{aligned} \phi : \ \mathbb{F}_{q^m}[X]/(X^n - 1) &\longrightarrow \ Circ_n(\mathbb{F}_{q^m}), \\ \sum_{i=0}^{n} a_i X^i &\longmapsto \begin{pmatrix} a_0 & a_1 & \cdots & a_{n-2} & a_{n-1} \\ a_{n-1} & a_0 & \cdots & a_{n-3} & a_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 & a_2 & \cdots & a_{n-1} & a_0 \end{pmatrix}. \end{aligned}$$

A $[2n, n]_{q^m}$ linear code $\mathcal{C}$ is called double circulant if its generator matrix $\mathbf{G}$ is of the form $\mathbf{G} = (\mathbf{A}_1 | \mathbf{A}_2)$ with $\mathbf{A}_1$ and $\mathbf{A}_2$ two $n \times n$ circulant matrices.

The authors of [18] introduced the family of ideal codes that allows them to reduce the size of the code's representation. The generator matrix of this family 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 as:

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

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 [18], they restraint the definition of ideal LRPC (Low Rank Parity Check) codes to $(2,1)$-ideal LRPC codes that they used to construct 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:

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

The decoding algorithm of LRPC codes presented in Algorithm 1 is described in [18]. Let E and F be two $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ with respectively basis $(e_1, \cdots, e_r)$ and $(f_1, \cdots, f_d)$. Let $\mathbf{s} = (s_1, \cdots, s_n) \in \mathbb{F}_{q^m}^n$ be a syndrome of an error $\mathbf{e}$ of weight r and such that $\text{Supp}(\mathbf{e}) = E$.

Given F and $\mathbf{s}$, the RSR algorithm, presented in Algorithm 1, recovers the support E of the error $\mathbf{e}$.

Algorithm 1: 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      $tmp \leftarrow S + F(S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})$
6      if $\dim(tmp) \leq rd$ then
7          $S \leftarrow tmp$
8      end
9  end
10 $E \leftarrow \bigcap_{1\leq i\leq d} f_i^{-1} S$
11 return $E$
```

In Algorithm 1, the support S is a subspace of EF given by:

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

thus, $\dim(S) \leq rd$.
In the RSR algorithm, the loop **for** (line **4** - Algorithm 1) allows to recover the whole vector space EF in case $\dim(S) < rd$. It is possible that the algorithm fails and at the end of the loop, S is different of EF, for the failure analysis, we let the lecturer refers to [18].
In the case $S = \langle EF \rangle = \langle e_1 f_1, \cdots, e_r f_1, \cdots, e_1 f_i, \cdots e_r f_i, \cdots, e_r f_d \rangle$, 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. Indeed, ROLLO cryptosystem bases a part of its security proof on the **2-Ideal Rank Support Recovery** (2-IRSR) [18] 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:

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

Hereafter, we will focus on ROLLO-I submission that presents small parameter sizes compared to ROLLO-II and ROLLO-III (see Table 1).

3 Target cryptosystem: ROLLO-I

3.1 Presentation

The algorithms described on this section resume the description of the ROLLO cryptosystem presented in [18].

ROLLO-I is a Key Encapsulation Mechanism (KEM) defined by three probabilistic algorithms: Keygen, Encap and Decap.
This cryptosystem uses a rank-based code in $\mathbb{F}_{q^m}$ (LRPC codes) with polynomial operations, we first have to define the two parameters n, m and the two irreducible polynomials associated P, P_m such that: $\begin{cases} \deg(P_m) = m \\ \deg(P) \quad = n \end{cases}$.
As seen in the previous section, this KEM mainly depends on the hardness to retrieve the error's support used to generate a couple of vectors $(\mathbf{e_1}, \mathbf{e_2})$, if this support dimension is lower than r.

Algorithm 2: KeyGen

```
  Input: $n$ and $m$ to define the code, $d$ the private key's rank weight
  Output: public key $\mathbf{pk} = \mathbf{h}$ and private key $\mathbf{sk} = (\mathbf{x}, \mathbf{y})$
1 Generate a random support $F$ of rank $d$.
2 Create one random element $\mathbf{sk} = (\mathbf{x}, \mathbf{y}) \in \mathbb{F}_{2^m}^{2n}$ from the support $F$.
3 Compute $\mathbf{h} = \mathbf{x}^{-1} \cdot \mathbf{y} \mod P$
4 return $\mathbf{pk}, \mathbf{sk}$
```

The random generation of support and the generation of an element from a support are given respectively by Algorithm 5 and 6.
The KeyGen algorithm presented in Algorithm 2 creates randomly the secret key used in Decap, and the public key used to hide the shared secret.

Algorithm 3: Encap

```
Input: $n$ and $m$ to define the code, $\mathbf{h}$ the public key, $r$ the error's rank weight
Output: ciphertext $\mathbf{c}$ and shared secret $K$
1 Generate a random support $E$ of rank $r$.
2 Create two random elements $(\mathbf{e_1}, \mathbf{e_2}) \in \mathbb{F}_{2^m}^{2n}$ from the support $E$.
3 Compute $\mathbf{c} = \mathbf{e_2} + \mathbf{e_1} \cdot \mathbf{h} \mod P$
4 Derive the shared secret $K = \text{Hash}(E)$
5 return $\mathbf{c}$,$K$
```

The Encap algorithm presented in Algorithm 3 randomly creates two vectors $\mathbf{e_1}$ and $\mathbf{e_2}$ depending on one support E used to derive the shared secret using a hash function.

Algorithm 4: Decap

```
Input: $(\mathbf{x}, \mathbf{y})$ the private key, $r$ the error's rank weight, $d$ the private key's rank
       weight, $\mathbf{c}$ the ciphertext
Output: shared secret $K$
1 Compute $\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \mod P$
2 Retrieve error's support: $E = \text{RSR}(F, \mathbf{s}, r)$
3 Derive the shared secret: $K = \text{Hash}(E)$
4 return $K$
```

The Decap algorithm presented in Algorithm 4 first computes the syndrome of the received ciphertext c and then uses the Rank Support Recovery Algorithm presented in Algorithm 1 to retrieve the error's support.

3.2 Operations

Support generation: As defined in Equation (1) the support of an element $\mathbf{x} = (x_0, x_1, \cdots, x_{n-1})$ is the basis of the vector subspace generated by the coordinates of $\mathbf{x}$. Therefore, to generate a support in $\mathbb{F}_{2^m}^n$ of a given dimension d we have to generate at random d linearly independent vectors in $\mathbb{F}_{2^m}$.

In order to generate a support of dimension d in $\mathbb{F}_{2^m}^n$, we use Algorithm 5.

Element generation from a support: An element $\mathbf{x} \in \mathbb{F}_{2^m}^n$ in rank metric is characterized by its dimension and its induced support. From a given random support of dimension d we can get a random element of the same dimension

Algorithm 5: Random generation of support

```
Input: Dimension $d$
Output: $S \in \mathbb{F}_{2^m}^d$ support of dimension $d$
1 for $i$ from 0 to $d-1$ do
2     for $j$ from 0 to $\lceil \frac{m}{8} \rceil$ do
3         $S_{i,j} \leftarrow$ random byte (using a TRNG)
4     end
5     Clear $8 \times \lceil \frac{m}{8} \rceil - m$ most significant bits by applying a mask
6 end
7 Apply a row echelon algorithm to the resulting support.
8 return $S$
```

by computing n coefficients as linear combinations of the vectors defining the support.
Thus, an element in $\mathbb{F}_{2^m}^n$ can be generated from a support of dimension d with Algorithm 6.

Algorithm 6: Element generation from support

```
Input: $S \in \mathbb{F}_{2^m}^d$ support of dimension $d$
Output: $x \in \mathbb{F}_{2^m}^n$ generated from support $S$
1 for $i$ from 0 to $n-1$ do
2     Pick a random integer $r$ in $[2, d-1]$
3     $x_i$ = Add at random $r$ coefficients in $S$
4 end
5 return $x$
```

Intersection of two sub-spaces:
Let $U = \langle u_0, u_1, \cdots, u_{n-1} \rangle$ and $V = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ be two sub-spaces over $\mathbb{F}_{2^m}^n$. Considering the two vectors $\mathbf{u} = (u_0, u_1, \cdots, u_{n-1})$ and $\mathbf{v} = (v_0, v_1, \cdots, v_{n-1})$, elements in $\mathbb{F}_{2^m}^n$, the intersection $\mathcal{I}_{U,V} = U \cap V$ can be computed by following the Zassenhaus algorithm [16], described with the above 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 $\mathbf{c} = (c_0, \cdots, c_{n-1}) \in \mathbb{F}_{2^m}^n$.

3.3 Parameters

The submission of this scheme contains three sets of parameters ROLLO-I-128, ROLLO-I-192, ROLLO-I-256 corresponding to three different levels of security achieving respectively 128, 192 and 256 bits of security according to NIST's security strength categories 1, 3 and 5 [21], we recall them in Table 3. As described in Section 3, the parameters n and m correspond respectively to the degrees of irreducible polynomials P and P_m used to construct the field $\mathbb{F}_{q^m}^n$ and the parameters d and r correspond respectively to the rank of the secret key and the error.

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

Therefore, the size of the public key, secret key and ciphertext involved by this parameters are given in Table 4.

Param. / Algo.	Public key	Private key	Ciphertext	Shared secret
ROLLO-I-128	465	930	465	64
ROLLO-I-192	590	1.180	590	64
ROLLO-I-256	947	1.894	947	64

Table 4. Parameter size (bytes)

4 Implementation

4.1 Target platform and memory usage for implementation

For the implementation, we wanted to demonstrate that ROLLO cryptosystem can be deployed in an existing product, we have chosen an 32-bit secure ARM Cortex-M3 processor (ARM® SecurCore® SC300™) due to the fact that it is widely used in the industry. For this reason, we selected the Wisekey MS6001 microcontroller because it was perfectly fitting our needs. This product features 24 kB of RAM and 4 kB are dedicated to cryptographic data, it is equipped with a Random Number Generator (RNG) and a 32-bit mathematical co-processor able to perform operations in $\mathbb{F}_p$ and $\mathbb{F}_{q^m}$, in practice used for public key cryptography as RSA or ECC. In our implementations, all operations in $\mathbb{F}_{2^m}$ are

then performed using this crypto co-processor in order to speed up them.
Even if low-level implementations cannot be described in this paper since no standard is available, the algorithms described in Sections 4.3 and 4.4 are implementable on microcontroller containing a crypto co-processor for ECC or by implementing basic operations in $\mathbb{F}_{2^m}$, consisting in multiplication, addition, inversion and modular reduction.
As the reference implementation, we compute the shared secret by applying the hash function SHA-512 to the support of the error .

4.2 Target operations for optimization

Firstly, we try to find which operations can be optimized. In this paper, we do not consider operations over $\mathbb{F}_{2^m}$ as they are already implemented in the crypto-processor. Thus, we focused on optimizing the operations over $\mathbb{F}_{2^m}^n$.

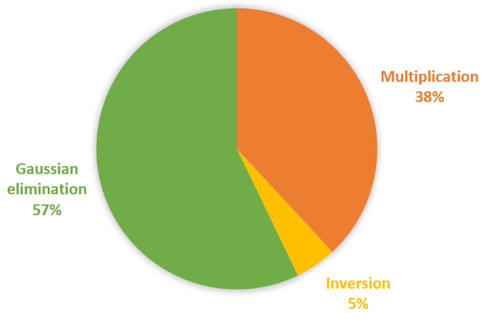

Fig. 1. Number of operations in $\mathbb{F}_{2^m}^n$ featured on ROLLO cryptosystem.

Figure 1 shows that Gaussian elimination and multiplication in $\mathbb{F}_{2^m}^n$ are often performed in the ROLLO-I cryptosystem.
Gaussian elimination is principally involved in the RSR Algorithm to compute to the intersection and sum between two vector spaces, and it is performed on big matrices. Thus, reduce the call of the Gaussian elimination method seems to be a good optimization. We then decided to focus on the multiplication and the RSR algorithm in order to optimize them.
The inversion is only performed during the key generation process and does not favour ephemeral keys. In this way, even if the extended Euclidean algorithm [15], used to compute the inverse in $\mathbb{F}_{2^m}^n$, is very costly, it is not considered in this paper.

For the rest of this section, we focus on the optimization of the two main operations in ROLLO cryptosystem: the multiplication in $\mathbb{F}_{2^m}^n$ and the RSR algorithm. We will then present in Section 4.3 an implementation optimized following the memory and in Section 4.4 an implementation optimized in time.

4.3 Optimized implementation depending on the memory

4.3.1 Multiplication in $\mathbb{F}_{2^m}^n$

Let P be an irreducible polynomial of degree n, and $\mathbf{x}, \mathbf{y} \in \mathbb{F}_{2^m}/(P)$ be two polynomials in $\mathbb{F}_{2^m}$ of degree strictly lower than n.
The result $\mathbf{r}$ of the polynomial product $\mathbf{x} \cdot \mathbf{y}$ is of degree lower than $2n - 1$. In order to reduce the result, we then have to apply a modular reduction on $\mathbf{r}$ by P. These two consecutive operations imply a major issue in terms of memory usage: $\mathbf{x}$ and $\mathbf{y}$ require $n \cdot \lceil m/32 \rceil \cdot 4$ bytes each, and the result $\mathbf{r}$ needs the double of this value.
Therefore, we decided to merge the two algorithms to get one which directly returns the result after the modular reduction, thus dividing the length of the result by 2. However, this choice requires the use of a simple polynomial multiplication. We choose the Schoolbook multiplication algorithm that involves $\mathcal{O}(n^2)$ multiplications in $\mathbb{F}_{2^m}$. Even if the complexity of this algorithm is quadratic, as the modulo P is parse, it is straightforward to combine modular reduction and Schoolbook multiplication, as presented in Algorithm 7, and then save memory usage.

Algorithm 7: Multiplication

```
   Input: $\mathbf{x}, \mathbf{y} \in \mathbb{F}_{2^m}^n$, $P_m$ modulo for $\mathbb{F}_{2^m}$, $P = (p_1, \cdots, p_n)$ modulo for $\mathbb{F}_{2^m}^n$
   Output: $\mathbf{r} = \mathbf{x} \cdot \mathbf{y}$
1  for i from 0 to n-1 do
2      for j from 0 to n-1 do
3          $tmp \leftarrow x_i \cdot y_j \mod P_m$
4          if $i + j \geq n$ then
5              for each $p_k \neq 0$ and $k < n$ do
6                  if $(i + j \mod n) + k \geq n$ then
7                      for each $p_l \neq 0$ and $l < n$ do
8                          $r_{(i+j+k+l) \mod n} \leftarrow r_{(i+j+k+l) \mod n} + tmp$
9                  else
10                     $r_{(i+j+k) \mod n} \leftarrow r_{(i+j+k) \mod n} + tmp$
11         else
12             $r_{i+j} \leftarrow r_{i+j} + tmp$
13 return r
```

4.3.2 Rank Support Recovery algorithm

The RSR algorithm, see Algorithm 1, defined in the ROLLO submission needs to pre-compute some values which be re-used multiple times without having to re-compute them. The RSR algorithm is by far the most costly operation of the decapsulation process.
We can compute the average memory cost as follows:

1. Compute the S_i will lead us to store $r \cdot d \cdot m$ bits at most.

2. Compute the $S_{i,i+1}$ will lead us to store $(d-1) \cdot r \cdot m$ bits on average.

We decided to completely remove the pre-computation phase steps to save memory, but the average resulting timing was heavily increased.

4.4 Optimized implementation in time

4.4.1 Multiplication in $\mathbb{F}_{2^m}^n$

The multiplication in $\mathbb{F}_{2^m}^n$ is one of the most used operations of this cryptosystem: it's 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 [22] as presented in Algorithm 8. Let $P = p_0 + p_1X$ and $Q = q_0 + q_1X$ be two polynomials of degree 1. The result of the product is

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

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

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

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

Algorithm 8: 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: $\mathbf{f} \cdot g$ in $\mathbb{F}_{2^m}^n$
1  if $N$ odd then
2      result ← Schoolbook($\mathbf{f}, \mathbf{g}, N$)
3      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 $R_1 + R_6x^{N'} + R_2x^{2N}$
```

The fourth step requires to divide the polynomial's degree by 2 and when the number of polynomial coefficients is odd, the algorithm computes the multiplication with the Schoolbook method, otherwise, continue to split. 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.
Finally, we will have a total memory cost of $4 \cdot (n + k) \cdot \lceil m/32 \rceil \cdot 8$ bytes with k the number of zero coefficients added to the polynomials.
We compare the number of multiplications required by the Schoolbook algorithm and Karatsuba in Figure 2.

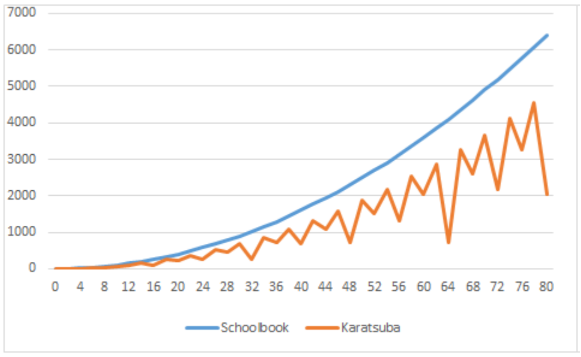

Fig. 2. Number of multiplications in $\mathbb{F}_{2^m}$ in function of the degree

We can see in Figure 2 that the number of multiplications required in the Karatsuba method is not strictly increasing, this is due to the polynomial division by 2 involved in the method. Thus, when the number of coefficients is equal to a power of two or a number that we can divide recursively by two, the number of multiplications is smaller.

Under the memory available for a multiplication in $\mathbb{F}_{2^m}^n$, we can thus choose to add more or less padding. For example, in ROLLO-I-128 with $n = 47$ we have to add one zero coefficient to reach a degree 47 (which induces 48 coefficients); however, in ROLLO-I-192 with $n = 53$ we have two possibilities:

- Pad the polynomials with 3 coefficients which leads to 1323 multiplications in $\mathbb{F}_{2^m}$.
- Pad with 11 coefficients to lower the cost to 729 multiplications in $\mathbb{F}_{2^m}$.

The second possibility presents 45% fewer multiplications but requires memory additional cost of $11 \times \lceil 79/32 \rceil \times 4 = 132$ bytes per polynomial, including the different buffers, the additional cost is 528 bytes: regarding the total cost, this number can be really important and thus we prefer the first choice to balance between memory and speed.

4.4.2 Rank Support Recovery algorithm

Compared to the initial Rank Support Recover algorithm, we chose to remove the pre-computation part for the benefit of memory usage: this method yields a higher computation cost.
For our optimized implementation, we decided to perform some pre-computations taking account of the memory cost constraint. In the Algorithm 9, these steps are framed and correspond to lines 2 to 10. This algorithm is running in constant time.

Algorithm 9: RSR (Rank Support Recover)

```
1   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 $E$
2   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
3       Compute $S_{i+1,i+2} = tmp_{i+1} \cap tmp_{i+2}$
4       Compute $S_{i,i+2} = tmp_i \cap tmp_{i+2}$
5       $tmp_{i\%3} \leftarrow S_{i+3}$
6   end
7   for $i$ from 1 to d-2 do
8       $tmp \leftarrow S + F \cdot (S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})$
9       if $\dim(tmp) \leq rd$ then
10          $S \leftarrow tmp$;
11      end
12  end
13  $E \leftarrow \bigcap_{1 \leq i \leq d} f_i^{-1} \cdot S$
14  return $E$
```

We can estimate the average memory cost of this pre-computation: we will have at most three S_i and $1+2\cdot(d-2)$ intersections as well as the private key's support and the error's syndrome. By adding the matrix induced by the Zassenhaus algorithm [16], we have a total memory cost of:

$$RAM_{pre-compute} = (8rd + d + 1 + 2(d-2)) \times m_b,$$

with m_b the length in bytes for one coefficient in $\mathbb{F}_{2^m}$.

The pre-computation part of the RSR algorithm included in the ROLLO submission had only one issue for our constraints: the memory cost required to store every S_i is too high, our solution is to have at most 3 of this S_i in memory and compute every intersection. As consequence, we save $(d-3) \cdot r \cdot d \cdot m_b$ bytes at most: it represents 1080, 2016 and 4480 bytes for respectively ROLLO-I-128, ROLLO-I-192, and ROLLO-I-256. Moreover, this modification does not stop the algorithm from being executed in constant time: it always computes the same number of intersections.

5 Results and comparison

In this section, we present the performance evaluation of proposed implementations regarding memory usage and speed. Our implementations were implemented in C. For performance measurements, we used IAR compiler C/C++ with high-speed optimization level and count the cycles with the debugging functionality of the IAR Embedded Workbench IDE [1].

	Memory optimised			Speed optimised		
Security \ Algo.	GenKey	Encap	Decap	GenKey	Encap	Decap
ROLLO-I-128	2640	1928	2168	3148	3376	3060
ROLLO-I-192	2972	2156	2748	3520	3508	4248
ROLLO-I-256	4850	3328	4832	5792	4424	7504

Table 5. Memory usage for ROLLO-I (in bytes)

Table 5 provides memory usage during the key encapsulation mechanism according to the optimization in terms of memory or in terms of time.

The core is a 32 bits multiplier, it means that every element in $\mathbb{F}_{q^m}$ has to be represented on $\lceil m/32 \rceil \cdot 4$ bytes: for this implementation, an element from $\mathbb{F}_{q^m}^n$ will be represented as $n \cdot \lceil m/32 \rceil \cdot 4$ bytes. Considering this fact, the memory usage of ROLLO-I-128 and ROLLO-I-192 will only differ because of n, indeed for ROLLO-I-128, $m = 79$ and for ROLLO-I-192, $m = 89$, we thus obtain $\lceil 79/32 \rceil = \lceil 89/32 \rceil = 3$. In contrast, every element in $\mathbb{F}_{q^m}$ for ROLLO-I-256 requires one more 32 bits word, it explains the huge difference of memory usage between the higher security and the two lowers. Moreover, Table 5 highlights the fact that the implementation of ROLLO-I-256 needs more than 4kB of RAM, so it cannot be implemented in our target.
Table 6 provides the number of cycles required by ROLLO-I-128 and ROLLO-I-192.

	Memory optimised			Speed optimised		
Algo. / Security	GenKey	Encap	Decap	GenKey	Encap	Decap
ROLLO-I-128	9.7	1.51	10.17	8.68	0.6	3.97
ROLLO-I-192	12.7	2.39	15.29	11.11	0.8	6.63

Table 6. Cycles counts ($\times 10^6$) for ROLLO-I

The key generation process performances are not really impacted by the speed optimization given that the multiplication is the only operation that has been changed between the implementations.

Table 7 presents the execution time in milliseconds for ROLLO-I at the 128 and 192 bits security level.

	Memory optimised			Speed optimised		
Algo. / Security	GenKey	Encap	Decap	GenKey	Encap	Decap
ROLLO-I-128	194	30.2	203.4	173.6	12	79.4
ROLLO-I-192	254	47.8	305.8	222.2	16	132.6

Table 7. Timings in miliseconds for ROLLO-I

Figure 3 and Figure 4 provide the cycle counts for different operations performed in ROLLO-I-128 and ROLLO-I-192 that consist of the cipher and syndrome computation and RSR algorithm. We do not take into account the cycle counts for operations in $\mathbb{F}_{q^m}$ or for SHA-512 that are performed by using the existing hardware and cryptographic library and they are not been optimized in this work.

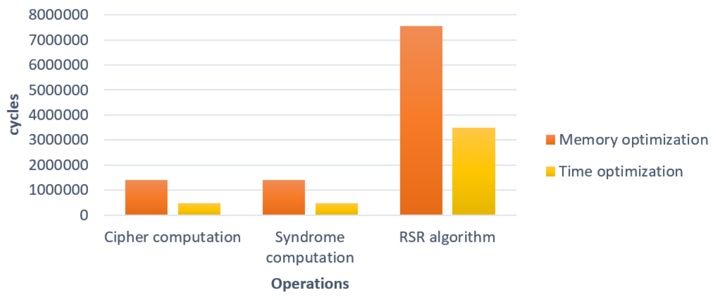

Fig. 3. Operations in ROLLO-I-128 according to memory optimization and time optimization

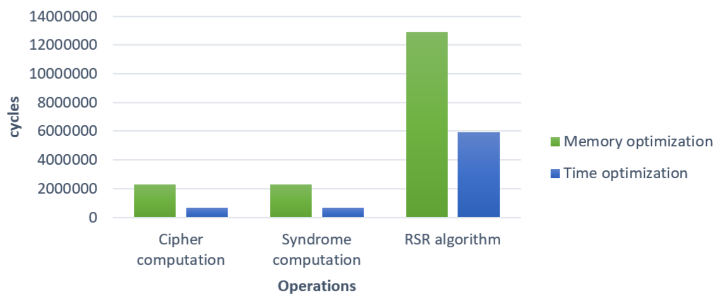

Fig. 4. Operations in ROLLO-I-192 according to memory optimization and time optimization 2

To give a rough idea, we decided to compare ROLLO-I implementation with a widely used key exchange cryptosystem Elliptic Curve Diffie-Hellman (ECDH) [2]. The elliptic curve multiplication in $\mathbb{Z}_p$ has already been implemented on the same platform and also benefits of the crypto co-processor for ECC.
To establish a shared secret between two entities, the ECDH protocol required 2 multiplications in $\mathbb{Z}_p$ that are executed in parallel by these two entities.
Thus, Table 8 gives the performances of a key agreement for ECDH and ROLLO-I cryptosystems according to the security level. For the estimation cost of ECDH, we only consider the two scalar multiplication, we do not into account the necessary time for the exchange of intermediate computation.

Security	Algorithm	Clock cycle ($\times 10^6$)
128	ROLLO-I-128	4.52
	ECDH Curve 256	3.49
192	ROLLO-I-192	7.43
	ECDH Curve 384	8.45

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

This shows that in case ROLLO-I submission should be standardized, it could be seen as a good alternative to cryptography in use today on embedded devices.

Finally, Table 9 gives indicatively performances in terms of memory and speed for others post-quantum cryptosystems, candidates to the NIST PQC project. However, the comparison with our implementation is not possible due to a large number of parameters that we have to take into account to compute the cycle counts. Indeed, value provided in Table 9 derive from implementation on different platforms and thus different architecture with different clock frequencies. Plus, the level of security of cryptosystems quoted is not the same.
Nevertheless, we decided to only give cycle counts and memory usage of cryp-

tosystems implemented on ARM CORTEX-M4 microcontrollers in order to reduce the architecture differences and give a rough idea of the cost of other post-quantum cryptosystems.

	Key generation		Encapsulation		Decapsulation	
Schemes	speed	memory	speed	memory	speed	memory
ROLLO-I-192 (memory)	12,700k	2,972	2,390k	2,156	15,290k	2,748
ROLLO-I-192 (speed)	11,110k	3,520	800k	3,508	6,630k	4,248
Saber [14]	1,165k	6,931	1,530k	7,019	1,635k	8,115
Saber [12]	895k	13,248	1,161k	15,528	1,204k	16,624
Kyber768 [13]	1,200k	10,544	1,446k	13,720	1,477k	14,880
NewHope [13]	1,246k	11,160	1,966k	17,456	1,977k	19,656
NTRU-HRSS [12]	145,963k	23,396	404k	19,492	819k	22,140

Table 9. Speed (cycles) and memory(bytes) performances for other NIST submissions on CORTEX-M4.

Conclusion

In this paper, we have shown that a post-quantum code-based cryptosystem could be implemented in an existing chip with existing hardware. For this practical implementation, ROLLO-I submission that presents reasonable parameter sizes has been seen as a good candidate to be implemented on a constraint device disposing of 4 kB of RAM for cryptographic data. We then provided two implementations, one optimized in memory usage and one in time that leads us to provide a comparison with the ECDH protocol that is implemented on the same target platform and benefits of the same crypto co-processor. This comparison has shown the good performances of ROLLO-I making it as a good alternative to replace public key exchange cryptosystems.
For future work, it would be interesting to look out the performances of ROLLO-I on a full hardware implementation. It would be also interesting to compare this implementation with other post-quantum cryptosystems implemented on the same target platform.

References

1. IAR Embedded Workbench. URL: `https://www.iar.com/`.
2. SEC 1. Standards for Efficient Cryptography Group: Elliptic Curve Cryptography - version 2.0, 2009. URL: `https://www.secg.org/sec1-v2.pdf`.
3. 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. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

4. 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.
5. 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.
6. 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.
7. 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.
8. 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. URL: http://arxiv.org/abs/1904.00357, arXiv:1904.00357.
9. Philippe Delsarte. Bilinear forms over a finite field, with applications to coding theory. *J. Comb. Theory, Ser. A*, 25:226–241, 1978.
10. R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950. doi:10.1002/j.1538-7305.1950.tb00463.x.
11. 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.
12. 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.
13. 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.
14. 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.
15. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
16. Eugene Luks, Ferenc Rakoczi, and Charles Wright. Some Algorithms for Nilpotent Permutation Groups. *J. Symb. Comput.*, 23:335–354, 04 1997. doi:10.1006/jsco.1996.0092.
17. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.
18. 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.

19. 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.
20. 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.
21. National Instute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, 2016. URL: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`.
22. André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations, 2006. aweimerskirch@escrypt.com 13331 received 2 Jul 2006. URL: `http://eprint.iacr.org/2006/224`.