The Rank-Based Cryptography Library

Nicolas Aragon[1], Slim Bettaieb[2], Loïc Bidoux[2,3], Yann Connan[1,2], Jérémie Coulaud[2], Philippe Gaborit[1], and Anaïs Kominiarz[2]

[1]University of Limoges, [2]Worldline, [3]Technology Innovation Institute

Abstract. Rank-based cryptography provides cryptosystems that aim to be secure against both classical and quantum computers. In the past few years, the interest for code-based cryptography in the rank metric setting has tremendously increased notably since the beginning of the NIST post-quantum cryptography standardization process. This paper introduces RBC a library dedicated to Rank-Based Cryptography and details its design and architecture. The performances of RBC are illustrated against comparable state of the art librairies. RBC greatly outperforms those libraries as it is 2 to 5 times faster than NTL and 40 to 138 times faster than $\mathtt{mp}\mathbb{F}_q$ on the multiplication and inversion over $\mathbb{F}_{q^m}^n$ which are the most critical operations when it comes to rank-based cryptography performances. In addition, the performances of ROLLO and RQC two rank-based cryptosystems provided by the library are reported for two platforms: a desktop computer equipped with an Intel Skylake-X CPU and an ARM Cortex-M4 microcontroller.

Keywords: RBC · Rank Metric · Library · Code-Based Cryptography

Introduction

Post-quantum cryptography aims at proposing schemes that provide security against adversaries having access to both classical and quantum computers. Since the seminal work of McEliece in 1978 [McE78], code-based cryptography using the Hamming metric has established itself as a serious alternative to classical cryptography. It is based on the difficulty of the syndrome decoding (SD) problem which has been proven NP-complete [BMVT78]. Many code-based cryptosystems have been proposed over the years culminating during the NIST post-quantum standardization process [Nis16] whose round 3 features three code-based key encapsulation mechanism (KEM) using the Hamming metric [ABC+20,AMAB+20b,AMAB+20a]. Introduced in 1985 [Gab85], the rank-metric constitutes a promising avenue for code-based cryptography. The security of rank-based cryptography relies on the rank syndrome decoding (RSD) problem which is the rank analogue of the syndrome decoding problem. One of the main benefits of the rank metric is that the time complexity of the best known attacks against the RSD grows faster with respect to the size of parameters than for the Hamming metric. As a consequence, rank-based cryptosystems feature smaller ciphertext and key sizes than their Hamming counterpart for identical security level. Rank-based schemes have also been considered in the NIST post-quantum standardization process whose round 2 includes two KEM namely ROLLO [AAB+18,AAB+19a,AAB+20a] and RQC [AAB+17b,AAB+19b,AAB+20b].

In this paper, we introduce RBC [AB+] a new C library dedicated to rank-based cryptography which aims to promote and foster community efforts on code-based cryptography in the rank metric setting. Rank-based cryptography relies on binary field arithmetic for which there already exist several libraries in the literature. Nevertheless, none of these libraries is entirely suitable for our purpose as they don't provide all the functionalities required by rank-based cryptography. Indeed, in order to implement rank-based schemes, one needs functions performing arithmetic of $\mathbb{F}_{q^m}$ elements, arithmetic of polynomials and vector spaces over $\mathbb{F}_{q^m}$ as well as specific functions dedicated to the notion of rank weight. In addition, existing libraries are not satisfactory when it comes to performances. Some libraries are really efficient for arithmetic in $\mathbb{F}_{q^m}$ while other really shine on arithmetic in $\mathbb{F}_{q^m}^n$ unfortunately no existing library is clearly superior to another one when the whole spectrum of rank-based cryptography is considered. Besides, some libraries relies on algorithms that are not the most efficient ones for the values of m and n typically used in rank-based cryptography as they target other applications. All these considerations have motivated the design and release of the RBC library.

Paper organization. In Section 1, we introduce the rank metric, Gabidulin and LRPC codes as well as the ROLLO and RQC cryptosystems. Next in Section 2, we describe the design and the architecture of our new library. We also detail some of the algorithms provided by the library focusing on the most critical ones with respect to performances. In Section 3, we present the performances of our library by comparing it to the $\mathtt{mp}\mathbb{F}_q$, NTL and RELIC libraries. We also showcase its performances by reporting the execution timing of ROLLO and RQC on two platforms: a desktop computer equipped with a Skylake-X CPU and an ARM Cortex-M4 microcontroller. To finish, ongoing and future works related to the RBC library are discussed.

1 Preliminaries

In this section, we present some preliminaries regarding the rank metric (Section 1.1), Gabidulin and LRPC codes (Section 1.2) as well as the ROLLO and RQC schemes (Section 1.3).

1.1 Rank metric overview

The rank metric has been introduced by Gabidulin in 1985 [Gab85]. Let q be a power of a prime p, m be an integer, $\mathbb{F}_{q^m}$ a finite field, $\mathcal{B} = \{\beta_1, \ldots, \beta_m\}$ a basis of $\mathbb{F}_{q^m}$ viewed as a m-dimensional vector space over $\mathbb{F}_q$ and $\mathcal{V}$ a n-dimensional vector space over $\mathbb{F}_{q^m}$. One can express the coordinates of $\mathbf{x} \in \mathcal{V}$

in $\mathcal{B}$ thus defining the matrix $\mathbf{M_x} \in \mathcal{M}_{m,n}(\mathbb{F}_q)$ where $\mathbf{M_x} = (x_{i,j})$ such that $x_j = \sum_{i=1}^{m} x_{i,j}\beta_i$ for all $j \in [\![0, n-1]\!]$.

$$
\begin{array}{cccc}
\mathbf{M}: & \mathbb{F}_{q^m}^n & \simeq & \mathcal{M}_{m,n}(\mathbb{F}_q) \\
 & \mathbf{x} = (x_0, \ldots, x_{n-1}) & \mapsto & \mathbf{M_x} = \begin{array}{c c} \begin{pmatrix} x_{1,0} & \ldots & x_{1,n-1} \\ x_{2,0} & \ldots & x_{2,n-1} \\ \vdots & & \vdots \\ x_{m,0} & \ldots & x_{m,n-1} \end{pmatrix} & \begin{array}{c} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{array} \end{array}
\end{array}
$$

Let $P \in \mathbb{F}_q[X]$ be a polynomial of degree n, one can also identify the vector space $\mathcal{V}$ to the commutative ring $\mathbb{F}_{q^m}[X]/\langle P \rangle$ where $\langle P \rangle$ denotes the ideal of $\mathbb{F}_{q^m}[X]$ generated by P.

$$
\begin{array}{cccc}
\Psi: & \mathbb{F}_{q^m}^n & \simeq & \mathbb{F}_{q^m}[X]/\langle P \rangle \\
 & \mathbf{x} = (x_0, \ldots, x_{n-1}) & \mapsto & \Psi(\mathbf{x}) = \sum_{i=0}^{n-1} x_i X^i
\end{array}
$$

For $\mathbf{x}, \mathbf{y} \in \mathcal{V}$, the product $\mathbf{z} = \mathbf{x} \cdot \mathbf{y}$ is defined using the polynomial multiplication in $\mathbb{F}_{q^m}[X]/\langle P \rangle$ namely $\mathbf{z}$ is the only vector such that $\Psi(\mathbf{z}) = \Psi(\mathbf{x}) \cdot \Psi(\mathbf{y})$. To finish, we introduce the support and rank weight of $\mathbf{x} \in \mathcal{V}$ which are two core notions in rank-based cryptography.

Definition 1 (Support). *The support of* $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathcal{V}$, *denoted* $\mathsf{Supp}(\mathbf{x})$, *is the* $\mathbb{F}_q$*-subspace of* $\mathbb{F}_{q^m}$ *generated by the coordinates of* $\mathbf{x}$ *namely* $\mathsf{Supp}(\mathbf{x}) = \langle x_0, \ldots, x_{n-1} \rangle_{\mathbb{F}_q}$.

Definition 2 (Rank weight). *The rank weight of* $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathcal{V}$, *denoted* $\|\mathbf{x}\|$, *is defined as the dimension of* $\mathsf{Supp}(\mathbf{x})$ *or equivalently as the rank of the matrix* $\mathbf{M_x}$.

1.2 Rank metric codes

There are two main families of codes in rank metric. Gabidulin codes [Gab85] are analogue to the Reed-Solomon codes and can be thought as the evaluation of q-polynomials [Ore33] of bounded degree on the coordinates of a vector over $\mathbb{F}_{q^m}$. Gabidulin codes are $\mathbb{F}_{q^m}$-linear codes that can deterministically decode up to $\left\lfloor \frac{n-k}{2} \right\rfloor$ errors. Low Rank Parity Check (LRPC) codes [GMRZ13] are $\mathbb{F}_{q^m}$-linear codes whose parity check matrix coefficients belong to a space of small dimension. Unlike Gabidulin codes, LRPC codes are probabilistic and as such they feature a non-zero decoding failure probability.

Definition 3 ($\mathbb{F}_{q^m}$-linear code). *An* $\mathbb{F}_{q^m}$*-linear code* $\mathcal{C}$ *of dimension* k *and length* n, *denoted* $[n,k]_{q^m}$, *is a subspace of* $\mathbb{F}_{q^m}^n$ *of dimension* k.

Definition 4 (Generator matrix). *A matrix* $\mathbf{G} \in \mathbb{F}_{q^m}^{m \times n}$ *is a generator matrix for the* $[n,k]_{q^m}$ *code* $\mathcal{C}$ *if* $\mathcal{C} = \{\mathbf{xG} \mid \mathbf{x} \in \mathbb{F}_{q^m}^k\}$.

Definition 5 (Parity-check matrix). *A matrix* $\mathbf{H} \in \mathbb{F}_{q^m}^{(n-k)\times n}$ *is a parity-check matrix for the* $[n,k]_{q^m}$ *code* $\mathcal{C}$ *if* $\mathcal{C} = \{\mathbf{x} \in \mathbb{F}_{q^m}^n \mid \mathbf{H}\mathbf{x}^\top = 0\}$. *The vector* $\mathbf{H}\mathbf{x}^\top \in \mathbb{F}_{q^m}^{n-k}$ *is called the syndrome of* $\mathbf{x}$.

Definition 6 (q-polynomials). *The set of q-polynomials over* $\mathbb{F}_{q^m}$ *is the set of polynomials with the following shape:* $\{P(X) = \sum_{i=0}^{r} p_i X^{q^i} \mid p_i \in \mathbb{F}_{q^m}, p_r \neq 0\}$. *The q-degree of a q-polynomial* P *is defined as* $deg_q(P) = r$.

Definition 7 (Gabidulin codes). *Let* $k, n, m \in \mathbb{N}$ *such that* $k \leqslant n \leqslant m$. *Let* $\mathbf{g} = (g_0, \ldots, g_{n-1})$ *be a* $\mathbb{F}_q$*-linearly independent family of vectors of* $\mathbb{F}_{q^m}$. *The Gabidulin code* $\mathcal{G}_g(n,k,m)$ *is the code defined as* $\{P(\mathbf{g}) \mid deg_q(P) < k\}$ *where* $P(\mathbf{g}) := (P(g_1), \ldots, P(g_n))$.

Definition 8 (LRPC codes). *Let* $\mathbf{H} = (h_{ij})_{i\in[\![1,n-k]\!],\ j\in[\![1,n]\!]} \in \mathbb{F}_{q^m}^{(n-k)\times n}$ *be a full-rank matrix such that its coefficients generate an* $\mathbb{F}_q$*-subspace* F *of small dimension* d, *i.e.* $F = \langle h_{ij}\rangle_{\mathbb{F}_q}$ *and* $d = dim(F)$. *Let* $\mathcal{C}$ *be the code with parity-check matrix* $\mathbf{H}$, $\mathcal{C}$ *is called an* $[n,k]_{q^m}$ *LRPC code.*

1.3 The ROLLO and RQC schemes

ROLLO. ROLLO is the merge of the three cryptosystems LAKE [ABD+17a], LOCKER [ABD+17b] and Rank-Ouroboros [AAB+17a] which all share the same decryption algorithm for LRPC codes. Following [AAB+20a], we only consider ROLLO-I (formerly LAKE) and ROLLO-II (formerly LOCKER) in the remaining of this paper. ROLLO-I is an IND-CPA KEM whereas ROLLO-II is an IND-CCA2 public key encryption (PKE) scheme. They are respectively described in Figure 1 and Figure 2 from Appendix A ; we defer the interested reader to [AAB+20a] for additional details.

RQC. RQC is an IND-CCA2 KEM build from an IND-CPA PKE construction on top of which the HHK transform [HHK17] is performed. Unlike many other code-based cryptosystems, the security of RQC does not rely on any code indistinguishability assumption following the approach introduced by Alekhnovich [Ale03]. We only describe the PKE version of RQC for simplicity (see Appendix A, Figure 3) and defer the reader to [AAB+20b] for additional details.

2 The RBC library

In this section, we describe the design and the architecture of our new library (Sections 2.1 and 2.2). We also detail some algorithms provided by the library focusing on the most critical ones with respect to performances (Section 2.3).

2.1 RBC library overview

RBC [AB+] is a C library dedicated to rank-based cryptography that focuses on performances without sacrificing usability. It is released under the LGPL license and can be retrieved at `https://rbc-lib.org`. It currently features:

- A *core layer* providing arithmetic for elements, vectors and polynomials over $\mathbb{F}_{2^m}$ with some utility functions tailored to rank-based cryptography ;
- A *code layer* providing implementations for the main codes used in rank-based cryptography namely Gabidulin codes and LRPC codes ;
- A *scheme layer* providing implementations for ROLLO and RQC, two rank-based cryptosystems submitted to the NIST PQC standardization process.

Dual API. The RBC library API can be thought as a dual API targeting two different audiences. We refer as *end users* people who are mainly concerned with using the schemes provided by the library (for instance to include ROLLO in a software, benchmark rank-metric based cryptosystems...) and we refer as *advanced users* people who want to use rank-based cryptography functionalities that are not limited to the schemes provided by the library (for instance to implement a new rank-metric based cryptosystem, contribute to the library...). End users should consider that the RBC library API is limited to the scheme layer functions while advanced users should use the whole API namely functions from the core, code and scheme layers.

Design choice regarding finite fields. The RBC library currently only supports finite fields of the form $\mathbb{F}_{q^m}$ with $q = 2$ which are the most commonly used finite fields in rank-metric cryptography. Regarding implementation of finite field arithmetic, one can either provide generic algorithms suited for any value of m or provide specific algorithms tailored for each value of m. While the first approach is superior with respect to simplicity and usability, the RBC library uses the second approach which is better when it comes to performances. This has no impact on usability for end users but adds some complexity for advanced users which is partly mitigated thanks to our preprocessing and build system.

Preprocessing and build system. RBC library preprocessing and build system is a set of python scripts facilitating development for advanced users and allowing build customization for all users. It features a templating system for the core layer that generates optimized code for each finite field while avoiding code redundancy. In addition, it provides automatic source code specialization for the code and scheme layers allowing users to write generic code that will be automatically instantiated with finite fields specified in a configuration file. Doing so, one can write generic code while keeping the possibility to use several instantiations of its code at once. For instance, writing only one ROLLO-I implementation and creating a program that call both ROLLO-I-128 and ROLLO-I-192 instantiated respectively with $\mathbb{F}_{2^{67}}^{83}$ and $\mathbb{F}_{2^{79}}^{97}$ while avoiding any code redundancy. In addition, the RBC preprocessing and build system allows users to customize the build of the library by specifying several options in a configuration file. Users may choose the targeted architecture amongst x86, x64 and x64 along with CLMUL and AVX2 support. The preprocessing and build system will generate code accordingly by choosing the best available algorithms for the specified architecture. Users may also choose which cryptosystems from the scheme layer they want to include in their build thus offering the possibility to minimize the size of the generated library files.

Tests, documentation and examples. In order to ease the use of the RBC library, a documentation is available. In addition, working examples and benchmark tools are provided for the cryptosystems included in the library. Unit-tests are available for the core layer functions and KAT tests are provided for the code and scheme layers.

Third-party implementations. The RBC library relies on several cryptographic primitives that are outside the scope of rank-based cryptography such as a pseudorandom number generator, a seedexpander, SHA2, FIPS202 and AES. Implementations for theses primitives are retrieved from the BearSSL [Por16], OpenSSL [Ope], PQClean [PQC], SUPERCOP [Sup] projects and [Nis16,Gue10]. In addition, the Minunit framework [Min] and the mp$\mathbb{F}_q$ library [GT07,GT08] are used to provide unit-tests against the library.

2.2 RBC library architecture

The RBC library introduces several structures and types corresponding to mathematical objects manipulated in rank-based cryptography. They are easily identified thanks to their common `rbc` prefix.

The following structures constitute the core layer of the RBC library:

- `rbc_elt` implementing an `element` of $\mathbb{F}_{q^m}$;
- `rbc_vec` implementing a `vector` over $\mathbb{F}_{q^m}$;
- `rbc_vspace` implementing a `vector space` over $\mathbb{F}_{q^m}$;
- `rbc_poly` implementing a `polynomial` over $\mathbb{F}_{q^m}$;
- `rbc_qre` implementing an `element of the quotient ring` $\mathbb{F}_{q^m}[X]/\langle P\rangle$ where $\langle P\rangle$ denotes the ideal of $\mathbb{F}_{q^m}[X]$ generated by P.

These types have various dependencies one to each other. For instance, `rbc_vec` are constructed from `rbc_elt` while `rbc_vspace` and `rbc_poly` are based on `rbc_vec`. In addition, the `rbc_qre` type is built from the `rbc_poly` one. For each of the aforementioned types, the library provides arithmetic operations, generation of random elements, serialization as well as utility functions tailored to rank-based cryptography.

Additional types and functions are defined within RBC code layer:

- `rbc_qpoly` implementing a `q-polynomial` over $\mathbb{F}_{q^m}$;

- rbc_gabidulin implementing a Gabidulin code ;
- rbc_lrpc_RSR() providing LRPC decoding.

The rbc_gabidulin type relies on rbc_qpoly in order to provide encoding and decoding algorithms for Gabidulin codes. As LRPC encoding is generally performed using rbc_qre arithmetic, we provide LRPC decoding using only the rbc_lrpc_RSR() function.

The scheme layer follows a different convention where the rbc prefix is replaced by schemeName_securityLevel for convenience. For instance, ROLLO-I-128 can be instantiated using the following functions: rolloI_128_kem_keygen(), rolloI_128_kem_encaps() and rolloI_128_kem_decaps().

2.3 RBC library algorithms

In this section, we detail some of the algorithms implemented in the RBC library focusing on the most critical ones with respect to performances. As arithmetic over $\mathbb{F}_{q^m}^n$ is of paramount importance in rank metric, we have selected algorithms that are well suited for the values of m and n typically used in rank-based cryptography. Hereafter, we denote by the *RBC supported instructions sets* the CLMUL and AVX2 instruction sets. For some operations, we provide two implementations depending on whether the RBC supported instruction sets can be used or not. These instructions are leveraged using Intel intrinsics therefore we refer to them with the name of the corresponding intrinsics instruction.

Algorithms related to rbc_elt

The RBC library uses polynomial representation for the rbc_elt therefore elements $e \in \mathbb{F}_{2^m}$ are represented as vectors $(e_0, \ldots, e_{m-1})$ of size m over $\mathbb{F}_2$. Operations in $\mathbb{F}_{2^m}$ are performed using polynomial arithmetic modulo Π where Π is the sparse irreducible polynomial used to define $\mathbb{F}_{2^m}$ as $\mathbb{F}_2[X]/\langle \Pi \rangle$. As such, many operations on rbc_elt generate unreduced elements (represented by the rbc_elt_ur type) that can be transformed to rbc_elt by performing reduction modulo Π.

Multiplication. The rbc_elt_mul() function encompasses a polynomial multiplication followed by a modular reduction. Two polynomial multiplication algorithms are provided depending on whether the RBC supported instruction sets can be used or not. If the aforementioned instruction sets are supported, a textbook polynomial multiplication accelerated by the _mm_clmulepi64() intrinsics instruction is performed. Otherwise, the multiplication is implemented using the left-to-right comb method with preprocessing ; see Algorithm 2.36 of [HMV06] for additional details.

Inversion. The rbc_elt_inv() function is implemented using a version of the Euclidean algorithm tailored for binary fields.

Squaring. The `rbc_elt_sqr()` function inserts several zeros within the representation of an element $e = (e_0, e_1, \ldots, e_{m-1})$ in order to obtain an unreduced element $e' = (e_0, 0, e_1, 0, \ldots, e_{m-1})$ which correspond to squaring in $\mathbb{F}_{2^m}$ after modular reduction of e'. If the RBC supported instruction sets are available, this is done using the interleaving intrinsics instructions `_mm_unpacklo_epi8()` and `_mm_unpackhi_epi8()` along with preprocessing ; see Algorithm 1 of [ALH10]. A similar but less efficient algorithm is used if the aforementioned instruction sets are not supported.

Modular reduction. The `rbc_elt_reduce()` function uses an algorithm that exploits the sparse structure of the polynomial Π by performing reduction over $\mathbb{F}_{2^m}$ one word at a time. This algorithm is tailored to each considered finite field as Π differs for each value of m ; see Figure 2.9 and Algorithm 2.41 of [HMV06] for an example over $\mathbb{F}_{2^{163}}$.

Algorithms related to `rbc_vec`

The `rbc_vec` is an utility type mainly used to construct the `rbc_poly` and `rbc_vspace` types nevertheless it provides some core functionalities for rank-based cryptography such as random vectors generation and rank weight computation. It is implemented as a pointer of `rbc_elt` whose size is fixed at initialization without any resize function provided.

Random vectors generation. Three ways of generating random vectors over $\mathbb{F}_{q^m}$ are provided in the RBC library:

1. The `rbc_vec_set_random()` function generates a vector purely at random by sampling each of its coordinate randomly in $\mathbb{F}_{q^m}$;
2. The `rbc_vec_set_random_full_rank()` function generates a full rank vector. To do so, each coordinates of the vector is sampled randomly in $\mathbb{F}_{q^m}$ then the rank weight of the vector is computed. This process is repeated until the vector is of full rank ;
3. The `rbc_vec_set_random_from_support()` function generates a vector randomly with each coordinate sampled from a support F of dimension d. First, the generating family of F is copied at random positions of the vector then the remaining coordinates are filled with random linear combinations of the generating family of F.

Rank weight. The `rbc_vec_get_rank()` function determines the rank weight of a vector $\mathbf{x} \in \mathbb{F}_{2^m}^n$ by computing the rank of its associated matrix $\mathbf{M_x}$ using the Gauss algorithm.

Algorithms related to rbc_poly

The rbc_poly type is implemented as a structure containing a rbc_vec element used to store the coefficients of the polynomial, the current degree of the polynomial and a max_degree value that keeps track of the size of the underlying rbc_vec element.

Multiplication. The rbc_poly_mul() function implements a recursive Karatsuba algorithm. Each level of recursion splits each of the polynomials in half and an hardcoded multiplication is used when the degrees of both polynomials is at most one. Our implementation is inspired from the NTL library [S+01].

Inversion. The rbc_poly_inv() function implements polynomial inversion using the extended Euclidean algorithm.

Algorithms related to rbc_vspace

Vector spaces are represented using generating families therefore the rbc_vspace type is simply a rbc_vec and the corresponding subspace of $\mathbb{F}_{q^m}$ is the vector space generated by the elements stored within the rbc_vec.

Direct sum. The rbc_vspace_directsum() function computes the direct sum of two vector spaces A and B by concatenating their generating families.

Product. Given a vector spaces A and B of generating families $(A_0, \dots, A_{d-1})$ and $(B_0, \dots, B_{r-1})$, the rbc_vspace_product() function calculates their product C of generating family $(C_{0,0}, \dots, C_{r-1,d-1})$ by computing the following elements: $C_{i,j} = A_i \times B_j$ for $i \in [0, d-1]$ and $j \in [0, r-1]$.

Intersection. The rbc_vspace_intersection() function computes the intersection of two vector spaces A and B by using the Zassenhaus algorithm.

Canonical basis. Some cryptosystems use vector spaces as inputs to hash functions therefore one needs to be able to represent vector spaces in a non ambiguous way. Given a vector space V represented by a rbc_vec $\mathbf{v}$, one can compute the row echelon form of the matrix $\mathbf{M_v}$ associated to $\mathbf{v}$ by calling the rbc_vec_echelonize() function thus obtaining a canonical basis of V.

Algorithms related to Gabidulin and LRPC codes

Gabidulin. The rbc_gabidulin_encode() function performs Gabidulin codes encoding using a classical vector / matrix multiplication. Gabidulin codes decoding is realized by the rbc_gabidulin_decode() function using the algorithm proposed by Loidreau in [Loi05] and later improved in [ALR18]. This algorithm uses the q-polynomial reconstruction method and as such relies extensively on the arithmetic of the ring of q-polynomials over $\mathbb{F}_{2^m}$ which is provided by the `rbc_qpoly` structure. More precisely, the RBC library implement the variant described in [BBGM19] along with the "Polynomials with lower degree" optimization from section 4.4.2 of [ALR18].

LRPC. No specific structure for LRPC codes have been provided as LRPC encoding is generally performed throught `rbc_qre` arithmetic. LRPC decoding is performed by the `rbc_lrpc_RSR()` function that implements the Rank Support Recover algorithm ; see Algorithm 1 of [AAB$^+$18]. This algorithm is similar to the standard LRPC codes decoding algorithm described in [GMRZ13] except that it stops after recovering the support E of the error vector $\mathbf{e}$.

3 RBC library performances

In this section, we discuss the performances of the RBC library by comparing it to the mp$\mathbb{F}_q$, NTL and RELIC libraries (Section 3.1). Next, we report the performances of RQC and ROLLO as implemented in the library for two platforms: a desktop computer equipped with a Skylake-X CPU (Section 3.2) and a Cortex-M4 microcontroller (Section 3.3).

3.1 Comparison with the NTL, mp$\mathbb{F}_q$ and RELIC libraries

The benchmarks have been performed on a machine that has 16GB of memory and an Intel® Core™ i7-7820X (Skylake-X) CPU @ 3.6GHz for which the Hyper-Threading, Turbo Boost and SpeedStep features were disabled. The following libraries have been used: NTL [S$^+$01] (version 11.4.3) along with GF2X (version 1.3.0) and GMP (version 6.2.0), mp$\mathbb{F}_q$ [GT07,GT08] (version 1.1) and RELIC [AGM$^+$] (version 0.5.0). The benchmarks have been compiled with GCC (version 10.1.0) using the `-O3 -flto -mavx2 -mpclmul -msse4.2 -maes` flags. The results have been obtained by computing the average running time from 1000 random instances. In order to minimize biases from background tasks running on the benchmark platform, each instance have been repeated 100 times and averaged. Our benchmark is focused on the finite fields corresponding to the different parameters sets of ROLLO and RQC. The RELIC library provides several implementations for each arithmetic operation ; we have tested all implementations while reporting only the most efficient one.

Multiplication and inversion over $\mathbb{F}_{q^m}^n$ are the most critical operations when it comes to rank-based cryptography performances. One can see from Table 1 bellow (as well as Appendix B, Tables 2 to 10) that RBC greatly outperforms other libraries on these operations as it is 2 to 5 times faster than NTL and 40 to 138 times faster than mp$\mathbb{F}_q$. Overall, the RBC library is more efficient than NTL and RELIC on all the considered operations nevertheless mp$\mathbb{F}_q$ sometimes outperforms RBC on arithmetic operations over $\mathbb{F}_{q^m}$. Indeed, one can see that inversion over $\mathbb{F}_{q^m}$ is about 20% faster in mp$\mathbb{F}_q$ than in RBC. Multiplication and squaring over $\mathbb{F}_{q^m}$ feature similar performances in RBC and mp$\mathbb{F}_q$ althought

$\mathtt{mp}\mathbb{F}_q$ seems more efficient than RBC when the polynomial used for reduction is a pentanomial. However, whenever $m \geq 128$, RBC outperforms $\mathtt{mp}\mathbb{F}_q$ for both multiplication and squaring. This highlights some minor room for improvement within the RBC library that will be explored in future work.

Operation	RBC	$\mathtt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{127}}$	32	32	223	1 118
Inversion over $\mathbb{F}_{2^{127}}$	5 320	3 924	7 296	7 822
Squaring over $\mathbb{F}_{2^{127}}$	32	90	161	166
Multiplication over $\mathbb{F}_{2^{127}}^{113}$	88 221	8 868 521	453 234	-
Inversion over $\mathbb{F}_{2^{127}}^{113}$	1 548 059	-	5 604 693	-

Table 1: Performances in CPU cycles for $\mathbb{F}_{2^{127}}^{113}$ (RQC-128 parameters)

3.2 Performances of ROLLO and RQC on Intel Skylake-X

The benchmarks have been performed on a machine that has 16GB of memory and an Intel® Core™ i7-7820X (Skylake-X) CPU @ 3.6GHz for which the Hyper-Threading, Turbo Boost and SpeedStep features were disabled. The schemes have been compiled with GCC (version 10.1.0) using the `-O3 -flto -mavx2 -mpclmul -msse4.2 -maes -std=c99` flags. The OpenSSL library (version 1.1.1.g) have been used as a provider for SHA2. The results have been obtained by computing the average running time from 1000 random instances. In order to minimize biases from background tasks running on the benchmark platform, each instance have been repeated 100 times and averaged.

One can see from Appendix B, Tables 11 to 13 that ROLLO and RQC are both efficient on the x64 architecture. Indeed, one can compute the Keygen, Encaps and Decaps operations of ROLLO-I-128 and ROLLO-I-256 in respectively less than 0.5 ms and 1 ms on our benchmark machine. ROLLO-II is slightly less efficient as an inversion over $\mathbb{F}_{q^m}^n$ have to be performed during the Keygen. Nonetheless all the operations of ROLLO-II-128 and ROLLO-II-256 can be computed in respectively less than 1.5 ms and 2 ms on our benchmark machine. RQC-128 is also fairly efficient as the Keygen, Encaps and Decaps can be computed in less than 1 ms on the considered machine. However, Gabidulin decoding become costly for bigger parameters therefore one need up to 3.5 ms to compute the Keygen, Encaps and Decaps of RQC-256 on our benchmark machine.

3.3 Performances of ROLLO and RQC on ARM Cortex-M4

In this section, we present the performances of ROLLO and RQC as implemented within the RBC library on microcontroller. Several implementations have been reported in the litterature. The first one provide an implementation of ROLLO-I leveraging the ARM SecurCore SC300 crypto co-processor [LMB+19] while the second one studies the Encaps operation of both ROLLO and RQC on the ARM Cortex-M0 microcontroller [ABC+19]. Hereafter, we focus on the ARM Cortex-M4 microcontroller as suggested by the NIST and therefore compare our results to those of the pqm4 project [KRSS20] that aims to provide a post-quantum cryptography library for the Cortex-M4.

The benchmarks have been performed on a STM32F4 discovery board featuring a 32-bit ARM-Cortex-M4 processor, 1 MByte flash memory and 196 KByte RAM. Our tests use the pqm4 benchmark scripts and as such follow the methodology described in [KRSS20]. In particular, all cycle counts are obtained at 24 MHz. For each scheme, 100 executions have been performed using `arm-none-eabi-gcc` in version 10.1.0. The mean running times for ROLLO-I, ROLLO-II and RQC are presented in Appendix C, Tables 14 to 16. No value is reported for RQC-256 as the current implementation exceeds the available memory of the targeted platform. We defer to future work the design of a memory optimized implementation of RQC-256. In order to contextualize these results, the Table 17 depicts the performances of some post-quantum KEM included in pqm4 focusing on C implementations targeting 128 bits of security (*i.e.* comparable to ROLLO-I-128, ROLLO-II-128 and RQC-128). Out of fairness for projects that have released implementations with Cortex-M4 specific optimizations (which we did not do), we have also reported their performances in Appendix C, Table 18.

Implementations from the pqm4 project are based on the implementations targeting the 64-bit architecture submitted to the NIST PQC standardization process. Hereafter, we report improvements over this work using our new implementations targeting 32-bit architectures. The observed running timings for ROLLO and RQC are up to twice as fast as the ones currently reported in pqm4.

Ongoing and future work

The first version of the RBC library constitutes a solid basis to support people implementing rank-based cryptography. Nonetheless, the RBC library is still in its infancy and will be improved over time. In the short term, our priority is to provide a better treatment of constant-time within the library. While some functionalities have received some attention with respect to constant-time, the library currently contains several functions that are not implemented in a constant-time way. Our future releases will include improvements with respect to constant-time within the library (by considering results from [AMADG21,ABC$^+$] for example).

Some avenues worth exploring for future work include (somewhat sorted by priority): (i) integrating additional rank-based cryptosystems such as Durandal [ABG$^+$19], (ii) integrating additional finite fields to RBC as the library currently only provides the ones used by ROLLO and RQC, (iii) exploring the algorithmic improvements mentioned in Section 3.1 as well as (iv) exploring potential algorithmic improvements using other representations for $\mathbb{F}_{q^m}$ elements such as normal bases. The RBC library aims to promote community efforts on rank-based cryptography and as such contributions are welcomed. People interested to contribute are invited to contact the library authors.

References

[AAB+17a] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Ouroboros-R. *NIST Post-Quantum Cryptography Standardization Project (Round 1)*, 2017. `https://pqc-ouroborosr.org`.

[AAB+17b] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Rank Quasi-Cyclic (RQC). *NIST Post-Quantum Cryptography Standardization Project (Round 1)*, 2017. `https://pqc-rqc.org`.

[AAB+18] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. ROLLO - Rank-Ouroboros, LAKE & LOCKER. *NIST Post-Quantum Cryptography Standardization Project (Round 1)*, 2018. `https://pqc-rollo.org`.

[AAB+19a] Carlos Aguilar Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. ROLLO - Rank-Ouroboros, LAKE & LOCKER. *NIST Post-Quantum Cryptography Standardization Project (Round 2)*, 2019. `https://pqc-rollo.org`.

[AAB+19b] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Alain Couvreur, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Rank Quasi-Cyclic (RQC). *NIST Post-Quantum Cryptography Standardization Project (Round 2)*, 2019. `https://pqc-rqc.org`.

[AAB+20a] Carlos Aguilar Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. ROLLO - Rank-Ouroboros, LAKE & LOCKER. *NIST Post-Quantum Cryptography Standardization Project (Round 2)*, 2020. `https://pqc-rollo.org`.

[AAB+20b] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Maxime Bros, Alain Couvreur, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Rank Quasi-Cyclic (RQC). *NIST Post-Quantum Cryptography Standardization Project (Round 2)*, 2020. `https://pqc-rqc.org`.

[AB+] Nicolas Aragon, Loïc Bidoux, et al. RBC Library. Version 1.0. `https://rbc-lib.org`.

[ABC+] Carlos Aguilar Melchor, Emanuele Bellini, Florian Caullery, Rusydi Hasan Makarim, Marc Manzano, Chiara Marcolla, and Victor Mateu. Constant-time algorithms for ROLLO.

[ABC+19] Ameera Salem Al Abdouli, Emanuele Bellini, Florian Caullery, Marcos Manzano, and Victor Mateu. Rank-metric Encryption on Arm-Cortex M0: Porting code-based cryptography to lightweight devices. In *Proceedings of the 6th ASIA Public-Key Cryptography Workshop*, 2019.

[ABC+20] Martin R. Albrecht, Daniel J Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Patterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Marlin Tomlinson, and Wen Wang. Classic McEliece. *NIST Post-Quantum Cryptography Standardization Project (Round 3)*, 2020. `https://classic.mceliece.org`.

[ABD+17a] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LAKE-Low rAnk parity check codes Key Exchange. *NIST Post-Quantum Cryptography Standardization Project (Round 1)*, 2017.

[ABD+17b] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LOCKER–LOw rank parity ChecK codes EncRyption. *NIST Post-Quantum Cryptography Standardization Project (Round 1)*, 2017.

[ABG+19] Nicolas Aragon, Olivier Blazy, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. Durandal: A Rank Metric Based Signature Scheme. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 728–758. Springer, 2019.

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[Ale03] Michael Alekhnovich. More on average case vs approximation complexity. In *44th Annual IEEE Symposium on Foundations of Computer Science, Proceedings*, pages 298–307, 2003.

[ALH10] Diego De Freitas Aranha, Julio López, and Darrel Hankerson. Efficient software implementation of binary field arithmetic using vector instruction sets. In *International Conference on Cryptology and Information Security in Latin America*, pages 144–161. Springer, 2010.

[ALR18] Daniel Augot, Pierre Loidreau, and Gwezheneg Robert. Generalized Gabidulin codes over fields of any characteristic. *Designs, Codes and Cryptography*, 86(8):1807–1848, 2018.

[AMAB+20a] Carlos Aguilar Melchor, Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation. *NIST Post-Quantum Cryptography Standardization Project (Round 3)*, 2020. `https://bikesuite.org`.

[AMAB+20b] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérome Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic (HQC). *NIST Post-Quantum Cryptography Standardization Project (Round 3)*, 2020. `https://pqc-hqc.org`.

[AMADG21] Carlos Aguilar Melchor, Nicolas Aragon, Nicolas Dyseryn, and Philippe Gaborit. Fast and Secure Key Generation for Low Rank Parity Check Codes Cryptosystems. In *To be published*, 2021.

[BBGM19] Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. Preventing timing attacks against RQC using constant time decoding of Gabidulin codes. In *International Conference on Post-Quantum Cryptography*, pages 371–386. Springer, 2019.

[BMVT78] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[Gab85] Ernest Mukhamedovich Gabidulin. Theory of codes with maximum rank distance. *Problemy Peredachi Informatsii*, 21(1):3–16, 1985.

[GMRZ13] Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zémor. Low rank parity check codes and their application to cryptography. In *Proceedings of the Workshop on Coding and Cryptography (WCC)*, 2013.

[GT07] Pierrick Gaudry and Emmanuel Thomé. The MPFQ library and implementing curve-based key exchanges. 2007.

[GT08] Pierrick Gaudry and Emmanuel Thomé. MPFQ, a finite field library, 2008. `https://mpfq.gitlabpages.inria.fr/`.

[Gue10] Shay Gueron. Intel Advanced Encryption Standard (AES) new instructions set, 2010.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017.

[HMV06] Darrel Hankerson, Alfred John Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[KRSS20] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4, 2020. `https://github.com/mupq/pqm4`.

[LMB+19] Jérôme Lablanche, Lina Mortajine, Othman Benchaalal, Pierre-Louis Cayrel, and Nadia El Mrabet. Optimized implementation of the NIST PQC submission ROLLO on microcontroller. *IACR Cryptol. ePrint Arch.*, 2019:787, 2019.

[Loi05] Pierre Loidreau. A Welch–Berlekamp like algorithm for decoding Gabidulin codes. In *International Workshop on Coding and Cryptography*, pages 36–45. Springer, 2005.

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *NASA*, 1978.

[Min] Minunit, a minimal unit testing framework for C/C++. `https://github.com/siu/minunit`.

[Nis16] *NIST Post-Quantum Standardization Process*, 2016. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`.

[Ope] OpenSSL. `https://www.openssl.org/`.

[Ore33] Oystein Ore. On a special class of polynomials. *Transactions of the American Mathematical Society*, 35(3):559–584, 1933.

[Por16] Thomas Pornin. BearSSL: A smaller SSL/TLS library, 2016. `https://bearssl.org/`.

[PQC] PQClean: Clean, portable, tested implementations of post-quantum cryptography. `https://github.com/PQClean/PQClean`.

[S+01] Victor Shoup et al. NTL: A library for doing number theory, 2001. `https://www.shoup.net/ntl/`.

[Sup] SUPERCOP, measuring the performance of cryptographic software. `https://bench.cr.yp.to/supercop.html`.

Appendix A ROLLO and RQC

This appendix describes the ROLLO and RQC schemes. Let $\mathcal{S}_w^n(\mathbb{F}_{q^m})$, $\mathcal{S}_{1,w}^n(\mathbb{F}_{q^m})$ and $\mathcal{S}_{(w_1,w_2)}^{3n}(\mathbb{F}_{q^m})$ be defined as:

$$\begin{aligned}
\mathcal{S}_w^n(\mathbb{F}_{q^m}) &= \{\mathbf{x} \in \mathbb{F}_{q^m}^n : \|\mathbf{x}\| = w\} \\
\mathcal{S}_{1,w}^n(\mathbb{F}_{q^m}) &= \{\mathbf{x} \in \mathbb{F}_{q^m}^n : \|\mathbf{x}\| = w, 1 \in \mathsf{Supp}(\mathbf{x})\} \\
\mathcal{S}_{(w_1,w_2)}^{3n}(\mathbb{F}_{q^m}) &= \{\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \in \mathbb{F}_{q^m}^{3n} : \|(\mathbf{x}_1, \mathbf{x}_3)\| = w_1, \|\mathbf{x}_2\| = w_1 + w_2, \\
&\qquad \mathsf{Supp}(\mathbf{x}_1, \mathbf{x}_3) \subset \mathsf{Supp}(\mathbf{x}_2)\}
\end{aligned}$$

- ◇ $\mathsf{Setup}(1^\lambda)$: generates and outputs the global parameters $\mathsf{param} = (n, m, d, r, P)$ where $P \in \mathbb{F}_q[X]$ is an irreducible polynomial of degree n.
- ◇ $\mathsf{KeyGen}(\mathsf{param})$: Picks $(\mathbf{x}, \mathbf{y}) \xleftarrow{\$} \mathcal{S}_d^{2n}(\mathbb{F}_{q^m})$. Sets $\mathbf{h} = \mathbf{x}^{-1}\mathbf{y} \mod P$ and returns $\mathsf{pk} = \mathbf{h}$ and $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$.
- ◇ $\mathsf{Encaps}(\mathsf{pk})$: Picks $(\mathbf{e}_1, \mathbf{e}_2) \xleftarrow{\$} \mathcal{S}_r^{2n}(\mathbb{F}_{q^m})$, sets $E = \mathsf{Supp}(\mathbf{e}_1, \mathbf{e}_2)$, $\mathbf{c} = \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{h} \mod P$. Computes $K = \mathsf{Hash}(E)$ and returns $\mathbf{c}$.
- ◇ $\mathsf{Decaps}(\mathsf{sk}, \mathbf{c})$: Sets $\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \mod P$, $F = \mathsf{Supp}(\mathbf{x}, \mathbf{y})$ and $E = \mathsf{RSR}(F, \mathbf{s}, r)$. Computes $K = \mathsf{Hash}(E)$.

Fig. 1: Description of ROLLO-I [AAB+20a]

- ◇ $\mathsf{Setup}(1^\lambda)$: generates and outputs the global parameters $\mathsf{param} = (n, m, d, r, P)$ where $P \in \mathbb{F}_q[X]$ is an irreducible polynomial of degree n.
- ◇ $\mathsf{KeyGen}(\mathsf{param})$: Picks $(\mathbf{x}, \mathbf{y}) \xleftarrow{\$} \mathcal{S}_d^{2n}(\mathbb{F}_{q^m})$. Sets $\mathbf{h} = \mathbf{x}^{-1}\mathbf{y} \mod P$ and returns $\mathsf{pk} = \mathbf{h}$ and $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$.
- ◇ $\mathsf{Encrypt}(\mu, \mathsf{pk})$: Picks $(\mathbf{e}_1, \mathbf{e}_2) \xleftarrow{\$} \mathcal{S}_r^{2n}(\mathbb{F}_{q^m})$, sets $E = \mathsf{Supp}(\mathbf{e}_1, \mathbf{e}_2)$, $\mathbf{c} = \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{h} \mod P$. Computes $c' = \mu \oplus \mathsf{Hash}(E)$ and returns the ciphertext $C = (\mathbf{c}, c')$.
- ◇ $\mathsf{Decrypt}(C, \mathsf{sk})$: Sets $\mathbf{s} = \mathbf{x} \cdot \mathbf{c} \mod P$, $F = \mathsf{Supp}(\mathbf{x}, \mathbf{y})$ and $E = \mathsf{RSR}(F, \mathbf{s}, r)$. Returns $\mu = c' \oplus \mathsf{Hash}(E)$.

Fig. 2: Description of ROLLO-II [AAB+20a]

- ◇ $\mathsf{Setup}(1^\lambda)$: generates and outputs the global parameters $\mathsf{param} = (n, k, \delta, w, w_1, w_2, P)$ where $P \in \mathbb{F}_q[X]$ is an irreducible polynomial of degree n.
- ◇ $\mathsf{KeyGen}(\mathsf{param})$: Samples $\mathbf{h} \xleftarrow{\$} \mathbb{F}_{q^m}^n$, $\mathbf{g} \xleftarrow{\$} \mathcal{S}_n^n(\mathbb{F}_{q^m})$, $(\mathbf{x}, \mathbf{y}) \xleftarrow{\$} \mathcal{S}_{1,w}^{2n}(\mathbb{F}_{q^m})$, computes the generator matrix $\mathbf{G} \in \mathbb{F}_{q^m}^{k \times n}$ of $\mathcal{G}_\mathbf{g}(n, k, m)$, sets $\mathsf{pk} = (\mathbf{g}, \mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y} \mod P)$ and $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$, returns $(\mathsf{pk}, \mathsf{sk})$.
- ◇ $\mathsf{Encrypt}(\mathsf{pk}, \mu, \theta)$: uses randomness θ to generate $(\mathbf{r}_1, \mathbf{e}, \mathbf{r}_2) \xleftarrow{\$} \mathcal{S}_{(w_1,w_2)}^{3n}(\mathbb{F}_{q^m})$, sets $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2 \mod P$ and $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e} \mod P$, returns $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.
- ◇ $\mathsf{Decrypt}(\mathsf{sk}, \mathbf{c})$: returns $\mathcal{G}_\mathbf{g}.\mathsf{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y} \mod P)$.

Fig. 3: Description of the PKE version of RQC [AAB+20b]

Appendix B RBC library performances

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{67}}$	60	32	448	1 175
Inversion over $\mathbb{F}_{2^{67}}$	2 670	2 327	4 099	4 347
Squaring over $\mathbb{F}_{2^{67}}$	60	32	406	208
Multiplication over $\mathbb{F}_{2^{67}}^{83}$	73 821	3 220 639	316 721	-
Inversion over $\mathbb{F}_{2^{67}}^{83}$	771 595	-	6 554 298	-

Table 2: Performances in CPU cycles for $\mathbb{F}_{2^{67}}^{83}$ (ROLLO-I-128 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{79}}$	31	32	218	1 161
Inversion over $\mathbb{F}_{2^{79}}$	3 147	2 529	5 010	5 019
Squaring over $\mathbb{F}_{2^{79}}$	31	32	159	167
Multiplication over $\mathbb{F}_{2^{79}}^{97}$	79 367	4 801 501	370 442	-
Inversion over $\mathbb{F}_{2^{79}}^{97}$	989 418	-	4 889 575	-

Table 3: Performances in CPU cycles for $\mathbb{F}_{2^{79}}^{97}$ (ROLLO-I-192 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{97}}$	32	32	225	1 094
Inversion over $\mathbb{F}_{2^{97}}$	4 036	3 081	5 891	6 004
Squaring over $\mathbb{F}_{2^{97}}$	32	32	187	166
Multiplication over $\mathbb{F}_{2^{97}}^{113}$	87 471	7 607 949	353 243	-
Inversion over $\mathbb{F}_{2^{97}}^{113}$	1 403 285	-	6 232 926	-

Table 4: Performances in CPU cycles for $\mathbb{F}_{2^{97}}^{113}$ (ROLLO-I-256 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{83}}$	57	32	238	1 115
Inversion over $\mathbb{F}_{2^{83}}$	3 384	2 650	5 072	5 303
Squaring over $\mathbb{F}_{2^{83}}$	32	32	192	208
Multiplication over $\mathbb{F}_{2^{83}}^{189}$	235 746	20 555 390	621 525	-
Inversion over $\mathbb{F}_{2^{83}}^{189}$	3 287 743	-	12 547 730	-

Table 5: Performances in CPU for $\mathbb{F}_{2^{83}}^{189}$ (ROLLO-II-128 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{97}}$	32	32	225	1 094
Inversion over $\mathbb{F}_{2^{97}}$	4 036	3 081	5 891	6 004
Squaring over $\mathbb{F}_{2^{97}}$	32	32	187	166
Multiplication over $\mathbb{F}_{2^{97}}^{193}$	238 539	22 797 360	642 396	-
Inversion over $\mathbb{F}_{2^{97}}^{193}$	3 458 307	-	15 756 672	-

Table 6: Performances in CPU cycles for $\mathbb{F}_{2^{97}}^{193}$ (ROLLO-II-192 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{97}}$	32	32	225	1 094
Inversion over $\mathbb{F}_{2^{97}}$	4 036	3 081	5 891	6 004
Squaring over $\mathbb{F}_{2^{97}}$	32	32	187	166
Multiplication over $\mathbb{F}_{2^{97}}^{211}$	256 874	27 312 415	764 347	-
Inversion over $\mathbb{F}_{2^{97}}^{211}$	4 042 388	-	14 191 588	-

Table 7: Performances in CPU cycles for $\mathbb{F}_{2^{97}}^{211}$ (ROLLO-II-256 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{127}}$	32	32	223	1 118
Inversion over $\mathbb{F}_{2^{127}}$	5 320	3 924	7 296	7 822
Squaring over $\mathbb{F}_{2^{127}}$	32	90	161	166
Multiplication over $\mathbb{F}_{2^{127}}^{113}$	88 221	8 868 521	453 234	-
Inversion over $\mathbb{F}_{2^{127}}^{113}$	1 548 059	-	5 604 693	-

Table 8: Performances in CPU cycles for $\mathbb{F}_{2^{127}}^{113}$ (RQC-128 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{151}}$	63	215	231	1 351
Inversion over $\mathbb{F}_{2^{151}}$	7 581	6 488	9 878	10 384
Squaring over $\mathbb{F}_{2^{151}}$	65	100	214	185
Multiplication over $\mathbb{F}_{2^{151}}^{149}$	235 771	28 515 864	871 936	-
Inversion over $\mathbb{F}_{2^{151}}^{149}$	3 552 081	-	10 433 894	-

Table 9: Performances in CPU cycles for $\mathbb{F}_{2^{151}}^{149}$ (RQC-192 parameters)

Operation	RBC	$\texttt{mp}\mathbb{F}_q$	NTL	RELIC
Multiplication over $\mathbb{F}_{2^{181}}$	74	435	285	1 408
Inversion over $\mathbb{F}_{2^{181}}$	9 284	7 961	11 743	12 311
Squaring over $\mathbb{F}_{2^{181}}$	76	114	230	237
Multiplication over $\mathbb{F}_{2^{181}}^{179}$	382 830	52 895 485	1 332 610	-
Inversion over $\mathbb{F}_{2^{181}}^{179}$	5 734 491	-	16 249 680	-

Table 10: Performances in CPU cycles for $\mathbb{F}_{2^{181}}^{179}$ (RQC-256 parameters)

Appendix C ROLLO and RQC performances

Scheme	Keygen	Encaps	Decaps
ROLLO-I-128	869 509	112 651	736 912
ROLLO-I-192	1 075 191	124 980	834 851
ROLLO-I-256	1 514 003	150 117	1 280 401

Table 11: Performances of ROLLO-I on intel Skylake-X in CPU cycles

Scheme	Keygen	Encaps	Decaps
ROLLO-II-128	3 619 812	332 877	1 144 540
ROLLO-II-192	3 766 107	338 967	1 256 774
ROLLO-II-256	4 394 490	354 564	1 621 820

Table 12: Performances of ROLLO-II on Intel Skylake-X in CPU cycles

Scheme	Keygen	Encaps	Decaps
RQC-128	366 445	530 762	2 581 487
RQC-192	798 057	1 200 596	5 739 349
RQC-256	1 165 492	1 713 963	9 466 386

Table 13: Performances of RQC on Intel Skylake-X in CPU cycles

Scheme	Keygen	Encaps	Decaps
ROLLO-I-128	16 927 603	1 926 332	7 009 943
ROLLO-I-192	22 466 486	2 271 969	7 839 572
ROLLO-I-256	45 424 004	3 769 338	15 039 516

Table 14: Performances of ROLLO-I on ARM Cortex-M4 in cycles

Scheme	Keygen	Encaps	Decaps
ROLLO-II-128	85 063 257	6 844 408	17 321 266
ROLLO-II-192	128 155 854	9 687 469	24 668 141
ROLLO-II-256	152 145 827	10 867 964	29 573 929

Table 15: Performances of ROLLO-II on ARM Cortex-M4 in cycles

Scheme	Keygen	Encaps	Decaps
RQC-128	5 756 747	11 340 541	71 551 978
RQC-192	12 324 464	24 632 358	150 108 887

Table 16: Performances of RQC on ARM Cortex-M4 in cycles

Scheme	Keygen	Encaps	Decaps
ROLLO-I	16 927 603	1 926 332	7 009 943
ROLLO-II	85 063 257	6 844 408	17 321 266
RQC	5 756 747	11 340 541	71 551 978
frodokem640shake	91 940 068	109 310 982	109 009 172
kyber512	653 616	883 740	981 642
newhope512cca	715 680	1 128 510	1 186 054
ntruhps2048509	106 694 544	2 838 551	7 766 558
ntrulpr653	56 520 202	112 440 360	168 157 956
sikep434	672 303 199	1 100 796 989	1 174 307 957
sntrup653	599 438 684	56 563 524	170 044 505

Table 17: Performances of several KEM on ARM Cortex-M4 in cycles. These implementations are in plain C and target 128 bits security.

Scheme	Keygen	Encaps	Decaps
frodokem640aes	48 350 369	47 135 457	46 604 758
kyber512	470 998	596 970	555 224
newhope512cca	582 009	870 621	825 352
ntruhps2048509	77 457 221	606 804	555 866
sikep434	48 264 153	78 912 215	84 277 568

Table 18: Performances of several KEM on ARM Cortex-M4 in cycles. These implementations features Cortex-M4 specific optimizations and target 128 bits security.