# Constant-time algorithms for ROLLO

Carlos Aguilar-Melchor[2], Emanuele Bellini[1], Florian Caullery[1], Rusydi H. Makarim[1], Marc Manzano[1], Chiara Marcolla[1], and Victor Mateu[1]

[1] Darkmatter LLC, Abu Dhabi, UAE
[2] ISAE-SUPAERO, Université de Toulouse, France

# Table of Contents

**Abstract.** In this work, we propose a standalone and constant-time optimized implementation of ROLLO. More precisely, we combine original and known results and describe how to perform generation of vectors of given rank, multiplication with lazy reduction and inversion of polynomials in a composite Galois field, and Gaussian reduction of binary matrices. We also do a performance analysis to show the impact of these improvements on ROLLO-I-128. Through the SUPERCOP framework, we compare it with other 128-bit secure KEMs in the NIST competition.

**Keywords:** code-based cryptography · KEM · post-quantum cryptography · rank metric

## 1 Introduction

The development of Error Correcting Codes (ECC) is due to Richard W. Hamming in 1947. A description of Hamming's code appeared in Claude Shannon's A Mathematical Theory of Communication [29] and was quickly generalized by Marcel J. E. Golay [13].

Cryptography based on ECC traces back to McEliece's proposal in 1978 [22]. At the time, the RSA cryptosystem [27] was preferred over McEliece's for a simple reason: McEliece's public-key and ciphertext were too large to be practical and allow a widespread deployment.

The situation changes if one considers a potential quantum computer. Indeed, RSA and other current standard cryptographic primitives can be broken in polynomial time with a quantum-computer using Shor's algorithm [30], whereas the best quantum attacks against McEliece's cryptosystem are still exponential in the length of the used ECC [20].

A quantum computer that is efficient and large enough to break realistic cryptographic systems through Shor's algorithm does not exist yet. But the evolution pace of quantum computing [6] is a strong motivation to replace classical by quantum-resistant cryptosystems. As a consequence, in 2017, the National Institute of Standards and Technology (NIST) published a call for proposals to define new standards for Public-Key Encryption (PKE), digital signatures and Key-Encapsulation Mechanism (KEM) schemes [24]. This call has increased the momentum of the scientific community on Post-Quantum Cryptography (PQC) in general, and cryptography based on Error Correction Codes (ECC) in particular, due to the fact that ECC represents the most conservative approach for PKE and KEM. Indeed, McEliece benefits from an impressive 40 year long unsuccessful cryptanalysis effort, increasing strongly the confidence in the scheme.

In order to reduce key and ciphertext sizes in McEliece's scheme, it has been proposed to replace the Goppa codes used in the McEliece's original cryptosystem. Nonetheless, such attempts have often been broken by cryptanalysis efforts: as a recent example one may cite the QC-MDPC scheme [23] and the reaction attack proposed in [16].

Another direction of research considered the use of ECC based on *rank metric* instead of the classical Hamming distance. The notion of error correcting codes in rank metric was introduced by Gabidulin in [9] and used for the first time in cryptography by the Gabidulin, Paramonov, Tretjakov (GPT) cryptosystem [10]. Given that the complexity of decoding a random code in the rank metric is higher than decoding a random code using Hamming distance, it is possible to design cryptosystems with smaller keys and ciphertexts. However, the GPT scheme and its successors were broken by the cryptanalysis framework introduced by Overbeck [26] and [25] (see also [8] and the structural attack proposed in [12]).

The lessons learned in the designs and attacks of schemes based on rank metric made cryptographers confident enough to submit new code-based cryptography schemes to the NIST PQC standardization process. These schemes provide appealing performances and key and ciphertext sizes which allowed some of them to pass through the first round of the NIST PQC standardization process. Indeed, LAKE, LOCKER, Rank-Ouroboros which have recently been merged into ROLLO [1] and RQC [2] are still running in the competition.

Some practical questions are open for the package submitted to NIST for ROLLO. Indeed, this package only proposes a non constant-time reference implementation, that relies on an external dependency. It is not clear how complex would result the final code if the dependency is removed, and the performance of a constant-time optimized implementation is hard to guess, as constant-time will increase computing cycles whereas an optimized implementation will reduce them.

*Our contribution* In this work, we propose a standalone and constant-time optimized implementation of ROLLO. More precisely, we combine original and known results and describe how to perform generation of vectors of given rank, multiplication with lazy reduction and inversion of polynomials in a composite Galois field, and Gaussian reduction of binary matrices. All of these through reasonably optimized constant-time algorithms. We give explicit indication of how to apply Zassenhaus algorithm in the Rank Support Recovery algorithm described in the NIST submission of ROLLO [1]. We also describe in detail how to implement the underlying finite field arithmetic with constant time operations, with and without the use of vectorization techniques. We finally do a performance analysis to show the impact of these improvements on ROLLO-I-128 when compared with its reference implementation. We expect this work to shed light on the attainable performance for constant-time implementations of ROLLO and to help practitioners to make educated choices when implementing it, or other constant time code-based cryptographic algorithms.

*Structure of the paper* Section 2 introduces the basic concepts needed to understand the scheme and the subsequent algorithms. Section 3 describes ROLLO-I key exchange. Section 4 provides all the details regarding the binary field, vector space, and composite Galois field arithmetic, as well as the description of the Rank Support Recovery algorithm used in the decapsulation phase. Section 5 compares the performances of various KEM submissions to the NIST post-quantum competition. Section 6 draws the conclusions.

## 2 Preliminaries

In the following we let $q$ be a prime power and $\mathbb{F}_{q^m}$ be the finite field with $q^m$ elements where $m$ is a positive integer. We define a $[n,k]_{q^m}$ code $C$ over $\mathbb{F}_{q^m}$ as a vector subspace of $(\mathbb{F}_{q^m})^n$ of dimension $k$, where $n$ is called the length and $k$ is the dimension of the code. A *generator matrix* for an $[n,k]_{q^m}$ code $C$ is thus any $k \times n$ matrix $G$ whose rows form a basis for $C$. Note that the generator matrix of a code is not unique.

We now give the definition of rank metric over $(\mathbb{F}_{q^m})^n$.

**Definition 1 (rank metric over $(\mathbb{F}_{q^m})^n$).** *Let* $\mathbf{e} \in (\mathbb{F}_{q^m})^n$ *be written as* $(e_1, \dots, e_n)$*. Denote by* $e_{i,j}$ *the* $j$*-th component of* $e_i$ *seen as a vector in* $\mathbb{F}_{q^m}$*.*

*Then the rank weight of e, denoted by* $\mathrm{w_R}(\mathbf{e})$*, is defined as*

$$\mathrm{w_R}(\mathbf{e}) = \mathit{rank} \begin{pmatrix} e_{1,1} & \dots & e_{n,1} \\ \vdots & & \vdots \\ e_{1,m} & \dots & e_{n,m} \end{pmatrix}$$

*The rank distance between two vectors* $\mathbf{e}, \mathbf{f} \in (\mathbb{F}_{q^m})^n$ *is defined by* $\mathrm{w_R}(\mathbf{e} - \mathbf{f}) = ||\mathbf{e} - \mathbf{f}||$.

Let $\mathbf{x} = (x_1, \dots, x_n) \in C$ be a *codeword*. Then the *support* $E$ of $\mathbf{x}$, denoted $\mathsf{supp}(\mathbf{x})$, is the $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ generated by the coordinates of $\mathbf{x}$:

$$E = \langle x_1, \dots, x_n \rangle_{\mathbb{F}_q}.$$

In other words, $\mathsf{dim}(E) = \mathrm{w_R}(\mathbf{x})$ and any $\mathbf{e} \in E$ can be written as $\mathbf{e} = \sum_{i=1}^{n} \lambda_i x_i$ where $\lambda_i \in \mathbb{F}_q$.
Because a linear code is a vectorial subspace, it is the kernel of some linear transformation. In particular, there is an $(n-k) \times n$ matrix $H$, called a *parity check matrix* for the $[n,k]_{q^m}$ code $C$, that verifies $C = \{x \in (\mathbb{F}_{q^m})^n | Hx^T = 0\}$. As for the generator matrix, the parity check matrix of a code $C$ is not unique.

## 2.1 Ideal codes

In this section we present the definition of the *ideal Low Rank Parity Check (ideal LRPC) codes*, codes on which all the variants of ROLLO are based. Moreover we introduce the problem underlying the security of the schemes. We first recall the definition of *ideal codes* and *LRPC codes*.

**Definition 2 (Ideal Codes).** *Let* $P(X) \in \mathbb{F}_q[X]$ *be a polynomial of degree* $n$ *and* $g_1, g_2 \in \mathbb{F}_{q^m}^n$. *Let* $G_1(X) = \sum_{i=0}^{n-1} g_{1,i} X^i$ *and* $G_2(X) = \sum_{i=0}^{n-1} g_{2,i} X^i$ *the polynomials associated to* $g_1$ *and* $g_2$.

*Using interchangeably vector and polynomial representations for elements in* $\mathbb{F}_{q^m}^n$, *we define the* $[2n,n]_{q^m}$ *ideal code* $C$ *of generator* $(g_1, g_2)$ *as the code with generator matrix*

$$\left(\begin{array}{c|c} G_1(X) \mod P & G_2(X) \mod P \\ XG_1(X) \mod P & XG_2(X) \mod P \\ \vdots & \vdots \\ X^{n-1}G_1(X) \mod P & X^{n-1}G_2(X) \mod P \end{array}\right).$$

*If* $g_1$ *is invertible,* $C$ *can be written with generator* $(1, g_1^{-1} g_2)$.

Let $M_k(R)$ be the set of $k \times k$ matrices over the ring R.

**Definition 3 (LRPC codes).**
*Let* $H \in M_{(n-k)\times n}(\mathbb{F}_{q^m})$ *be a full rank matrix such that its coefficients generate an* $\mathbb{F}_q$*-subspace* $F$ *of small dimension* $d$.

$$F =< h_{i,j} >_{\mathbb{F}_q}.$$

*The code* $C$ *of parity check matrix* $H$ *is called an LRPC code of weight* $d$.

**Definition 4 (Ideal LRPC codes).**

*Let $F$ be a $\mathbb{F}_q$-subspace of dimension $d$ of $\mathbb{F}_{q^m}$, $(h_1, h_2)$ two vectors of $\mathbb{F}_{q^m}^n$ with support in $F$ and $P \in \mathbb{F}_q[X]$ a polynomial of degree $n$. Let $H_1$ and $H_2$ be two matrices defined by*

$$H_1 = \begin{pmatrix} h_1 \\ Xh_1 \mod P \\ \vdots \\ X^{n-1}h_1 \mod P \end{pmatrix}, H_2 = \begin{pmatrix} h_2 \\ Xh_2 \mod P \\ \vdots \\ X^{n-1}h_2 \mod P \end{pmatrix}.$$

*The code $C$ with parity check matrix $(H_1|H_2)$ is called an ideal LRPC code of type $[2n, n]_{q^m}$.*

The problems on which all the variants of ROLLO are based is a particular instance of Rank Syndrome Decoding problem (RSD) [1]. Specifically:

*Problem 1 (r-Ideal Rank Support Recovery).* Given a polynomial $P \in \mathbb{F}_q[X]$ of degree $n$, a vectors $\mathbf{h}_1, \ldots, \mathbf{h}_r \in (\mathbb{F}_{q^m})^n$, a syndrome $\mathbf{s}$ and a weight $w$, it is hard to find a support $E = \langle \mathbf{e}_0, \ldots, \mathbf{e}_{r-1} \rangle$ of dimension lower than $w$ such that

$$\mathbf{e}_0 + \mathbf{e}_1\mathbf{h}_1 + \ldots + \mathbf{e}_{r-1}\mathbf{h}_{r-1} = \mathbf{s} \mod P.$$

Moreover, we also need:

*Problem 2 (Ideal LRPC codes indistinguishability).* Given a polynomial $P \in \mathbb{F}_q[X]$ of degree $n$ and a vector $\mathbf{h} \in (\mathbb{F}_{q^m})^n$, it is hard to distinguish whether the ideal code $C$ with parity-check matrix generated by $\mathbf{h}$ and $P$ is a random ideal code or if it is an ideal LRPC code of weight $d$.

In other words, it is hard to distinguish if $\mathbf{h}$ was sampled uniformly at random or as $\mathbf{x}^{-1}\mathbf{y} \mod P$ where the vectors $\mathbf{x}$ and $\mathbf{y}$ have the same support of small dimension $d$.

## 3 Description of the scheme

ROLLO (**R**ank-**O**uroboros, **L**AKE and **LO**CKER) is the merge of three initial propositions to the NIST-PQC competition: Rank-Ouroboros (formerly known as Ouroboros-R), LAKE and LOCKER.

For clarification reasons, the authors chose to uniformize the name of their protocols: LAKE is renamed ROLLO-I, LOCKER is renamed ROLLO-II and Rank-Ouroboros is renamed ROLLO-III. Each of these schemes is presented in three security level, 128, 192, and 256.

As stated by the submission documentation all ROLLO variants follow the approach inaugurated by *the public key encryption protocol NTRU in 1998 [17]. The main idea behind the protocol is that the secret key consists in the knowledge of a small Euclidean weight vector, which is used to derive a double circulant matrix. This matrix is then seen as a dual matrix of an associated lattice and* *a specific decoding algorithm based on the knowledge of this small weight dual matrix is used for decryption.*

*This idea of having as a trapdoor a small weight dual matrix (with a specific associated decoding algorithm) can naturally be generalized to other metrics. It was done in 2013 with MDPC [23] for Hamming metric and also in 2013 for Rank metric with LRPC codes [11]. These three protocols derive from the same basic main idea, adapted for different metrics, which have different properties in terms of efficiency, size of parameters and security reduction.* [1]

As pointed out in the previous section, ROLLO is a variation of the LRPC rank metric approach and its security if proved under the Ideal LRPC indistinguishability and the 2-Ideal Rank Support Recovery [1, Theorem 4.2]. The schemes have a failure probability, but this probability is well understood and made very low (for more detail see Section 1.4.2 of [1]).

We now describe ROLLO-I in detail. The description of ROLLO-II and ROLLO-III can be found respectively in Appendix A.1 and A.2. Note that they all rely on the same set of parameters: positive integers $q, m, n, r, d$ and an irreducible polynomial $P(X) \in \mathbb{F}_{q^m}[X]$ of degree $n$.

The ROLLO-I Key-Encapsulation scheme (KEM) $=$ (KeyGen; Encaps; Decaps) is a triple of probabilistic algorithms together with a key space $\mathcal{K}$.

- **Key generation** generates a pair of public and secret key (pk; sk).
  - Private key:
    1. Randomly select a vectorial subspace $F$ of $\mathbb{F}_{q^m}$ of dimension $d$ and samples a couple of vectors $(\mathbf{x}, \mathbf{y}) \in F^n \times F^n$ such that $\mathbf{x}$ is invertible $\mod P$ (which is equivalent to $\mathbf{x}$ being a non-zero vector) and $rk(\mathbf{x}) = rk(\mathbf{y}) = d$.
    2. Set the secret key as $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$.
  - Public key:
    1. Compute $\mathbf{h} = \mathbf{x}^{-1}\mathbf{y} \mod P$.
    2. Set the public key as $\mathsf{pk} = \mathbf{h}$.
- **Encapsulation** uses the public key pk to produce an encapsulation $\mathbf{c}$, and a key $K \in \mathcal{K}$.
  1. Randomly select a vectorial subspace $E$ of $\mathbb{F}_{q^m}$ of dimension $r$ and samples uniformly a couple of vectors $(\mathbf{e}_1, \mathbf{e}_2) \in E^n \times E^n$ such that $rk(\mathbf{e}_1) = rk(\mathbf{e}_2) = r$.
  2. Compute $\mathbf{c} = \mathbf{e}_1 + \mathbf{e}_2\mathbf{h} \mod P$.
  3. Compute $K = G(E)$ where $G(E)$ is a hash function.
  4. Send $\mathbf{c}$.
- **Decapsulation** using the secret key sk and a ciphertext $\mathbf{c}$, recovers the key $K \in \mathcal{K}$ or fails and return $\perp$.
  1. Compute $\mathbf{s} = \mathbf{xc} = \mathbf{xe}_1 + \mathbf{ye}_2 \mod P$
  2. Use Rank Support Recovery (RSR) algorithm (Algorithm 12) to recover $E$. The RSR algorithm takes as input $F, \mathbf{s}$ and $r$ (see Section 4.4 for more detail).

3. get $K = G(E)$.

We refer to Table 1 for the actual set of ROLLO-I parameters. Note that the private key can be obtained from a seed, and in ROLLO official NIST submission the seed expander is initialized with 40 bytes long seeds.

| Instance | $q$ | $m$ | $n$ | $d$ | $r$ | $P$ | sk size | pk size | $c$ size | Security | failure rate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ROLLO-I-128 | 2 | 79 | 47 | 6 | 5 | $X^{47} + X^5 + 1$ | 40B | 465B | 465B | 128b | $2^{-30}$ |
| ROLLO-I-192 | 2 | 89 | 53 | 7 | 6 | $X^{53} + X^6 + X^2 + X + 1$ | 40B | 590B | 590B | 192b | $2^{-32}$ |
| ROLLO-I-256 | 2 | 113 | 67 | 8 | 7 | $X^{67} + X^5 + X^2 + X + 1$ | 40B | 947B | 947B | 256b | $2^{-42}$ |

**Table 1.** ROLLO-I parameters

## 4 Proposed algorithms

We redefined ROLLO starting from the following building blocks:

- the binary field arithmetic corresponding to operations in $\mathbb{F}_{q^m}$;
- the vector space arithmetic including:
  - the Gauss reduction algorithm for binary matrices;
  - the Zassenhaus algorithm for binary matrices;
  - the generation of elements of $\mathbb{F}_{q^m}[X]/P(X)$ of a given rank;
- the arithmetic in the composite Galois field $\mathbb{F}_{q^m}[X]/P(X)$ where $P(X)$ is the irreducible polynomial given in the parameters;
- the Rank Support Recovery algorithm (RSR) used in the decapsulation phase.

The key generation, encapsulation and decapsulation (or encryption and decryption) of all the variants of ROLLO are based only on these five blocks. Hence, we focused on optimizing every operations included in those layers as well as insuring the fact that they are constant time.

Given $\mathbf{x}, \mathbf{y}$ two binary vectors, in what follows, we denote with $\mathbf{x} \oplus \mathbf{y}$ the bit-wise XOR of $\mathbf{x}$ and $\mathbf{y}$, and with $\mathbf{x} \otimes \mathbf{y}$ the bit-wise AND of $\mathbf{x}$ and $\mathbf{y}$. With $\mathbf{x} \ll h$ and $\mathbf{x} \gg h$ with indicate respectively, the left and right shift of $\mathbf{x}$ by $h$ positions.

### 4.1 Binary field arithmetic

Here we focus on the field used by ROLLO-I-128 for illustrative purpose. We implemented finite field arithmetic for the binary field $\mathbb{F}_{2^m}$, with $m = 79$, representing elements as binary polynomials of degree $m - 1$ modulo an irreducible polynomial of degree $m$. We used the irreducible trinomial $f(x) = x^{79} + x^9 + 1$ provided by the Allan Steel database incorporated in Magma software [5] and also suggested by the authors of ROLLO. This trinomial has also lowest possible intermediate degree, allowing the shortest shift during the reduction operations.

To represent an element of the field we use 128-bit unsigned integer, using the type `__uint128_t`, and sometimes casting it to `__m128i`, with unused bits set to zero.

Addition and subtraction of two elements are a simple bit-wise XOR operation.

The multiplication of two field elements is performed in two steps: a carryless multiplication of the two elements seen as polynomials and a polynomial reduction modulo $f$.

All operations in the binary field layer execute in constant time.

### Carryless multiplication

*Plain C implementation* The carryless multiplication has been implemented by using recursive Karatsuba multiplication [21]. More specifically, we borrowed the constant time carryless multiplication of two 64 bit register using only bit manipulation from NTL and then built a Karatsuba method over this multiplication. The full multiplication is described in Algorithm 1 assuming that the 64 bit carryless multiplication is available[3].

**Algorithm 1:** $\mathsf{clmul}(a, b)$: carryless multiplication in $\mathbb{F}_{2^{79}}$

```
input  : a,b of type __uint128, represent two binary polynomials of degree
         78.
output : c of type __uint128[2], represents a binary polynomial of degree 156.
1 __uint128 aLbL, aHbH, tmp;
2 aLbL = clmulepi64(b & 0xFFFFFFFFFFFFFFFF, a & 0xFFFFFFFFFFFFFFFF);
3 aLbL = clmulepi64((b >> 64) & 0xFFFFFFFFFFFFFFFF, (a >> 64) &
   0xFFFFFFFFFFFFFFFF);
4 tmp = clmulepi64(((b >> 64) & 0xFFFFFFFFFFFFFFFF) ⊕ (b &
   0xFFFFFFFFFFFFFFFF), ((a >> 64) & 0xFFFFFFFFFFFFFFFF) ⊕ (a &
   0xFFFFFFFFFFFFFFFF);
5 tmp = tmp ⊕aLbL ⊕aHbH;
6 c[0] = aLbL ⊕ (tmp << 64);
7 c[1] = aHbH ⊕ (tmp >> 64);
8 return c
```

For squaring, which will be used in the inversion algorithm, we can use the fact that this operation actually consists in interleaving zeros to the current representation of the polynomial. Indeed, for $a \in \mathbb{F}_{2^m}$, $a^2 = \left(\sum_{i=0}^{m-1} a_i x^i\right)^2 =$

[3] The code is available in the file *mach_desc.h* of the library NTL [31], under the method **NTL_BB_MUL_CODE0**

$\sum_{i=0}^{m-1} a_i x^{2i}$. For example, if the current representation of $a$ was 11100101, then $clmul(a, a)$ will be 1010100000100010. To perform this operation, we decided to use a small modification of the method *Interleave bits with 64-bit multiply* given by Sean Eron Anderson on his web page *Bit Twiddling Hacks* [7] which is given in Algorithm 2.

**Algorithm 2:** interleave_zeros($a$): interleave zeros after each bit of a byte.

```
   input  : a of type unsigned char
   output : b of type unsigned short
1  b = ((x * 0x0101010101010101 & 0x8040201008040201) *
     0x0102040810204081 >> 49) & 0x5555
2  return c
```

The squaring method is straightforward from there and is given in Algorithm 3.

**Algorithm 3:** clmul($a, a$): vectorized carryless squaring in $\mathbb{F}_{2^{79}}$

```
   input  : a of type __uint128, represent two binary polynomials of degree 78.
   output : c of type __uint128[2], represents $a^2$ in a binary polynomial of
            degree 156.
1  __uint128 high = 0, low = 0;
2  for $i = 0 \dots 7$ do
3      low ⊕= (((__uint128) interleave_zeros((a >> (8 * i)) & 0xFF)) <<
         (16 * i));
4  for $i = 8 \dots 9$ do
5      high ⊕= (((__uint128) interleave_zeros((a >> (8 * i)) & 0xFF)) <<
         (16 * (i - 8)));
6  c[0] = low;
7  c[1] = high;
8  return c
```

*AVX2 optimization* When possible, the carryless multiplication step has been performed using Intel Advanced Vector Extensions 2 instructions (AVX2) [18]. In particular, the core of this function uses the _mm_clmulepi64_si128 instruction (see also [15]) to perform 64 times 64 bit binary polynomial multiplication.

The multiplication of two 79 bit binary polynomials is performed in a schoolbook fashion, by dividing the input in two 64 bit registers (one containing only 15 bits) and then applying four times the function _mm_clmulepi64_si128, which acts on 64 bits registers. The results is stored in a __m256i type (4 registers), but only the $2m - 2$ least significant bits are used, while the remaining ones are set to zero. We refer to this algorithm as the $\mathsf{clmul}(A, B)$ algorithm, and we present our C implementation in Algorithm 4. Let us remark that using Karatsuba multiplication [21] in Algorithm 4 instead of steps 3-6 would not give any advantage, as the cost of multiplication and addition with AVX2 instruction is very close.

**Algorithm 4:** $\mathsf{clmul}(a, b)$: carryless multiplication in $\mathbb{F}_{2^{79}}$

```
   input   : a,b of type __m128i, represent two binary polynomials of degree 78.
   output  : c of type __m256i, represents a binary polynomial of degree 156.
 1 __m128i aLbL, aLbH, aHbL, aHbH, aLbH_xor_aHbL;
 2 __m256i aLbL256, aLbH_xor_aHbL256, aHbH256;
 3 aLbL = _mm_clmulepi64_si128(b, a, 0x00);
 4 aLbH = _mm_clmulepi64_si128(b, a, 0x01);
 5 aHbL = _mm_clmulepi64_si128(b, a, 0x10);
 6 aHbH = _mm_clmulepi64_si128(b, a, 0x11);
 7 aLbH_xor_aHbL = _mm_xor_si128(aLbH, aHbL);
 8 __m128i zero = _mm_setzero_si128();
 9 aLbL256 = _mm256_set_m128i(zero, aLbL);
10 aLbH_xor_aHbL256 = _mm256_set_m128i(zero, aLbH_xor_aHbL);
11 aLbH_xor_aHbL256 = _mm256_permute4x64_epi64(aLbH_xor_aHbL256, 0xD2);
12 aHbH256 = _mm256_set_m128i(aHbH, zero);
13 c = _mm256_xor_si256(aLbL256, aLbH_xor_aHbL256);
14 c = _mm256_xor_si256(c, aHbH256);
15 return c
```

Note that Algorithm 4 is also suitable for fields of size up to $2^{128}$, which include all the fields used by the variants of ROLLO except ROLLO-III-256.

**Reduction** The $2m - 2$ bits result provided by the carryless multiplication is reduced back modulo $f$ to a $m$ bit field element, using standard techniques. The algorithm for reduction is presented in Algorithm 5, where the symbols $\ll, \gg$ denote field multiplication and division by $x$ respectively (left and right shift operators), $\oplus$ is the field addition (bit-wise XOR operator), $\otimes$ the bit-wise AND operator.

As for Algorithm 4, Algorithm 5 is suitable for fields of size up to $2^{128}$ up to the modification of the values of the masks, the amount of shifts and their width.

**Inversion** The inversion of an element $x \in \mathbb{F}_{2^{79}}$ has been implemented using Fermat's little Theorem stating that $x^{2^m-2} = x^{-1}$. The fixed exponentiation is achieved by the strategy presented in [**?**][Section 6.2] using an addition chain of length 9 in the case of $m = 79$:

$$1 \to 2 \to 3 \to 6 \to 9 \to 18 \to 36 \to 39 \to 78 \to 79$$

**Algorithm 5:** $\mathsf{red}_{\mathbb{F}_{2^{79}}}(a)$: reduction in $\mathbb{F}_{2^{79}}$

```
   input  : $a = (\alpha_{156}, \dots, \alpha_0) \in \mathbb{F}_2^{157}$
   output : $c = (a \mod f(x) = x^{79} + x^9 + 1) \in \mathbb{F}_2^{79}$
1  $a_L = (\alpha_{127}, \dots, \alpha_0) \in \mathbb{F}_2^{128}$
2  $a_H = (0, \dots, 0, \alpha_{156}, \dots, \alpha_{128}) \in \mathbb{F}_2^{128}$
3  $a_L = a_L \oplus (a_H \ll 58) \oplus (a_H \ll 49)$
4  $a_H = (a_L \otimes \texttt{0xFFFFFFFFFFFF8000}) \ll 64$
5  $a_H = a_H \gg 79$
6  $a_L = a_L \oplus a_H \oplus (a_H \ll 9)$
7  $c = a_L \otimes \texttt{0x7FFFFFFFFFFFFFFFFFFF}$
8  return $c$
```

which results in Algorithm 13. This algorithm being quite straightforward it is presented in appendix **??**.

## 4.2 Binary vector space arithmetic

In this section we describe the main algorithms used in the vector space, i.e. Gauss reduction, Zassenhaus algorithm, and the generation of vectors of given rank.

In our implementation, a binary matrix $M$, usually be indicated with uppercase letters, of size $m \times l$ is an array of `__uint128_t` of length $l$, where each element of the array is a matrix row $m_i$. Similarly, a vector space, or the support of a set of vectors is represented with uppercase letters and stored in arrays of `__uint128_t`.

**Gauss elimination algorithm** We follow [4] to implement Gaussian elimination in constant time. At the expense of a small constant-factor overhead, the authors of [4] eliminate timing leaks in the standard algorithm of Gaussian reduction. They suggest to add $1 - b$ times the second row to the first row, where $b$ is the first entry in the first row; and then similarly (with updated $b$) for the third row etc. Then, they add $b$ times the first row to the second row, where $b$ is the first entry in the second row; and then similarly for the third row etc. Then, they continue similarly through the other columns. The algorithm returns a binary matrix in systematic form The pseudocode of the algorithm can be found in Algorithm 6, where $M$ represents a binary matrix with `r` rows and `c` columns, $m_i$ is the binary vector representing the $i$-th row of the matrix $M$, and $m_{i,j}$ is the bit entry of the matrix $M$ at position $i, j$.

In our C implementation we store one line of the binary matrix in a variable of type `__uint128_t`. If we call this element `m[i]`, we can perform Steps 6-7 of Algorithm 6 in a constant number of operations as follows:

```
mask = -(((m[i] ^ m[k]) >> j) & 1);
m[i] = m[i] ^ m[k] & mask;
```

Similarly, also Steps 10-11 can be executed in constant-time.

**Algorithm 6:** gauss($M$): Gaussian elimination algorithm

```
input  : $M$: matrix with r rows and c columns
output : $M$: matrix with r rows and c columns reduced in systematic form
1  for $i = 0, \dots, \lfloor(r + (c - 1))/c\rfloor - 1$ do
2      for $j = 0, \dots, c - 1$ do
3          if $i \geq r$ then
4              stop
5          for $k = i + 1, \dots, r - 1$ do
6              if $m_{i,j} \oplus m_{k,j} \neq 0$ then
7                  $m_i = m_i \oplus m_k$
8          for $k = 0, \dots, r - 1$ do
9              if $k \neq i$ then
10                 if $m_{k,j} \neq 0$ then
11                     $m_k = m_k \oplus m_i$
12 return $M$
```

**Zassenhaus algorithm** The Zassenhaus algorithm is a method to compute a basis for the intersection and sum of two vectorial subspaces $U, V$ of a vector space $W$ of length $d$. Let us consider the two sets of generators of $U$ and $V$, i.e., $U = \langle u_0, \dots, u_{l_1}\rangle$ and $V = \langle v_0, \dots, v_{l_2}\rangle$. The algorithm creates the block matrix (1) of size $(l_1 + l_2) \times 2d$.

$$\begin{bmatrix} u_{0,0} & \dots & u_{0,d-1} & u_0 & \dots & u_{0,d-1} \\ \vdots & & & & & \vdots \\ u_{l_1,0} & \dots & u_{l_1,d-1} & u_{l_1,0} & \dots & u_{l_1,d-1} \\ v_{0,0} & \dots & v_{0,d-1} & 0 & \dots & 0 \\ \vdots & & & & & \vdots \\ v_{l_2,0} & \dots & v_{l_1,d-1} & 0 & 0 & \end{bmatrix} \tag{1}$$

After application of the Gauss elimination, the matrix has the form (2), reduced in row echelon form.

$$\begin{bmatrix} a_{0,0} & \dots & a_{0,d-1} & \star & \dots & \star \\ \vdots & & & \vdots & & \vdots \\ a_{l_3,0} & \dots & a_{l_3,d-1} & \star & \dots & \star \\ 0 & \dots & 0 & b_{0,0} & \dots & b_{0,d-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & b_{l_4,0} & \dots & b_{l_4,d-1} \\ 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix} \tag{2}$$

In (2), $\star$ stands for arbitrary numbers, $(a_0, \ldots, a_{l_3})$ is a basis of $V + U$and $(b_0, \ldots, b_{l_4})$ is a basis of $V \cap U$. The pseudocode can be found in Algorithm 7.

**Algorithm 7:** zassenhaus$(U, V)$: Zassenhaus algorithm.

```
input  : $U = (u_0, \ldots, u_{l_1})^T \in (\mathbb{F}_{2^m})^{l_1}, V = (v_0, \ldots, v_{l_2})^T \in (\mathbb{F}_{2^m})^{l_2}$
output : $A = (a_0, \ldots, a_{l_3})^T \in (\mathbb{F}_{2^m})^{l_3}, B = (b_0, \ldots, b_{l_4})^T \in (\mathbb{F}_{2^m})^{l_4}$
1 $\begin{bmatrix} A & \star \\ \mathbf{0} & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathsf{gauss}(\begin{bmatrix} U & U \\ V & \mathbf{0} \end{bmatrix})$
2 return $A, B$
```

**Generation of vectors of given rank** One of the non-trivial part of the encapsulation and key generation is to generate a vector $\mathbf{e} \in \mathbb{F}_{q^m}^n$ of a given rank, say $r$. One can adopt the strategy from [28][Section 5.2] which consists in generating $r$ random elements of $\mathbb{F}_{q^m}$, check if they are linearly independent and then generate a random linear combination of those vectors.

The approach we have chose is different and takes advantage of the fact that $r$ is usually small (maximum 8 for ROLLO). We start by initializing a list with the zero vector and a random vector. We then generate a second random vector, check if it is already in the list. If so, we discard it and generate another one, else we add its addition with all the previous vectors already in the list to the list. We end up generating a vectorial subspace $F$ of $\mathbb{F}_{q^m}$ of dimension $r$. We then draw randomly the coordinates of $\mathbf{e}$ from this list. The only caveat of this method is that the vector $\mathbf{e}$ can be of rank less than $r$ as its coordinates could be in a vectorial subspace of $F$. We therefore have to check the rank of $\mathbf{e}$ before outputting the result. The procedure is displayed in Algorithm 8. When compared to the method from [28][Section 5.2], this yields an average improvement of 30% in performance.

### 4.3 Composite Galois field arithmetic

An element in the composite Galois field $\mathbb{F}_{(2^m)^n}$ can be represented as a polynomial $\mathbf{a}(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ in $\mathbb{F}_{2^m}[x]/P(x)$, with $P(x) \in \mathbb{F}_2[x]$ irreducible of degree $n$, or, equivalently, as an array $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ of length $n$ of elements in $\mathbb{F}_{2^m}$. In our implementation, an element of $\mathbb{F}_{(2^m)^n}$ is an array of `__uint128_t` of length $n$, and we usually refer to it in the pseudocode with bold lowercase letters.

**Matrix multiplication with lazy reduction** The multiplication $\mathbf{a} \times \mathbf{b}$ [4] in $\mathbb{F}_{(2^m)^n}$, Algorithm 9, is performed as the following vector by matrix multiplica-

[4] When it is clear from the context, with abuse of notation we indicate $\mathbf{a} \times \mathbf{b}$ as $\mathbf{a} \cdot \mathbf{b}$ or $\mathbf{ab}$.

**Algorithm 8:** rank_vec_gen(r): generation of vector of a given rank

```
   input   : $r \in \mathbb{N}^\star$
   output  : $\mathbf{e} \in \mathbb{F}_{q^m}^n$ such that $rk(e) \leq r$
 1 $F := \{0\}$
 2 dim := 0
 3 while dim $< r$ do
 4     $v \leftarrow\$ \mathbb{F}_{q^m}$
 5     if $v \notin F$ then
 6         for $u \in F$ do
 7             $F := F \cup \{v + u\}$
 8         dim = dim +1
 9 while $rk(\mathbf{e}) < r$ do
10     for $i = 0, \ldots, r-1$ do
11         $\mathbf{e}_i \leftarrow\$ F$
12 return e
```

tion

$$(a_0, a_1, \ldots, a_{n-1}) \times \begin{bmatrix} \hat{b}_{0,0} & \cdots & \hat{b}_{0,n-1} \\ \hat{b}_{1,0} & \cdots & \hat{b}_{1,n-1} \\ \vdots & \ddots & \vdots \\ \hat{b}_{n-1,0} & \cdots & \hat{b}_{n-1,n-1} \end{bmatrix},$$

where $(\hat{b}_{i,0}, \cdots, \hat{b}_{i,n-1})$ are the coefficients of $\mathbf{b}(x) \cdot x^i \mod P(x)$. In ROLLO-I, $(b_{i,0} + b_{i,1}x + b_{i,2}x^2 + b_{i,3}x^3 + b_{i,4}x^4 + b_{i,5}x^5 + \cdots + b_{i,n-1}x^{n-1}) \cdot x \mod P(x) = b_{i,n-1} + b_{i,0}x + b_{i,1}x^2 + b_{i,2}x^3 + b_{i,3}x^4 + (b_{i,4} + b_{i,n-1})x^5 \cdots, b_{i,n-2}x^{n-1}$, since $x^{47} = x^5 + 1$.

This allows us to reduce the number of reduction in $\mathbb{F}_{2^m}$, since when we compute the field element $(a_0, a_1, \ldots, a_{n-1}) \times (\hat{b}_{i,0}, \cdots, \hat{b}_{i,n-1}) = (a_0\hat{b}_{i,0} + \ldots + a_{n-1}\hat{b}_{i,n-1}) = \sum_{j=0}^{n-1} a_j\hat{b}_{i,j}$, each $a_j\hat{b}_{i,j}$ can be computed using the carryless multiplication algorithm clmul, and the reduction $\mathsf{red}_{\mathbb{F}_{2^{79}}}$ is applied only at the end of the summation. The pseudo-code of the algorithm is presented in Algorithm 9.

**Polynomial inversion** For the inversion in the composite Galois field $\mathbb{F}_{(2^m)^n} \cong \mathbb{F}_{2^m}[x]/P(x)$, we use the technique presented in [14] in 1998, which improves the Itoh-Tsujii algorithm with pre-computed powers [19]. The idea is to compute $\mathbf{a}^{-1} = (\mathbf{a}^r)^{-1}\mathbf{a}^{r-1}, \mathbf{a} \in \mathbb{F}_{(2^m)^n}, \mathbf{a} \neq 0$, where $r = (2^{mn} - 1)/(2^m - 1)$. It is easy to prove that $\mathbf{a}^r \in \mathbb{F}_{2^m}$ as $(\mathbf{a}^r)^{2^m} = (\mathbf{a}^{1+2^m+2^{2m}+\ldots+2^{(n-1)m}})^{2^m} = \mathbf{a}^{1+2^m+2^{2m}+\ldots+2^{(n-1)m}} = \mathbf{a}^r$. This reduces inversion in the Galois field $\mathbb{F}_{(2^m)^n}$ to one inversion in the ground field $\mathbb{F}_{2^m}$, the computation of $\mathbf{a}^{r-1}$ and $n$ multiplications in $\mathbb{F}_{2^m}$.

To compute $\mathbf{a}^{2^m}$, one can notice that $\mathbf{a}^{2^m} = \left(\sum_{i=0} a_i x^i\right)^{2^m} \mod P = \sum_{i=0}^{n} a_i x^{i2^m} \mod P$ as $a_i \in \mathbb{F}_{q^m} \forall i = 0, \ldots, n-1$. It is then sufficient to pre-

**Algorithm 9:** poly_mul($\mathbf{a}, \mathbf{b}$): polynomial multiplication in $\mathbb{F}_{2^m}[x]/P(x)$

```
input  : $\mathbf{a} = (a_{n-1}, \dots, a_0), \mathbf{b} = (b_{n-1}, \dots, b_0) \in \mathbb{F}_{2^m}[x]/P(x)$
output : $\mathbf{c} = (\mathbf{a} \times \mathbf{b} \mod P(x)) \in \mathbb{F}_{2^m}[x]/P(x)$
1  $\mathbf{c} = 0$
2  for $j$=0, ..., $n$-2 do
3      for $i$=0, ..., $n$-1 do
4          $t = \mathsf{clmul}(a_j, b_i)$
5          $c_i = c_i \oplus t$
6      $\mathbf{b} = \mathbf{b} \cdot x \mod P(x)$
7  for $i$=0, ..., $n$-1 do
8      $t = \mathsf{clmul}(a_{n-1}, b_i)$
9      $c_i = c_i \oplus t$
10     $c_i = \mathsf{red}_{\mathbb{F}_{2^{79}}}(c_i)$
11 return $\mathbf{c}$
```

compute the values of $s_i = x^{i2^m} \mod P, \forall i = 0, \dots, n-1$. Therefore, the computation of $\mathbf{a}^{2^m}$ can be seen as a matrix multiplication as follow:

$$S.\mathbf{a}^T = \begin{pmatrix} 1 & s_{1,0} & s_{2,0} & \dots & s_{n-1,0} \\ 0 & s_{1,1} & s_{2,1} & \dots & s_{n-1,1} \\ \vdots & \dots & \dots & \vdots & \\ 0 & s_{1,n-1} & s_{2,n-1} & \dots & s_{n-1,n-1} \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

In addition, if $P$ has only binary coefficients (which is the case for all variants of ROLLO), the pre-computed values also have binary coefficients meaning that the previous matrix multiplication can be performed using only XORs. The last step is to remark that $\mathbf{a}^{2^{km}} = S^k.\mathbf{a}^T$ and we end up with an algorithm performing $n$ polynomial multiplications and binary matrix multiplications, one inversion in $\mathbb{F}_{2^m}$ followed by $n$ multiplications in $\mathbb{F}_{2^m}$.

Algorithm 11 summarizes how the inversion is performed. It uses Algorithm 10 to compute $\mathbf{a}^{2^{km}}$. The matrix $S$ in Algorithm 10 is a pre-computed matrix depending only on $P$ and $n$.

Notice that both Algorithm 10 and 11 can be coded such that they execute a constant number of operations. In particular, Steps 4-5 of Algorithm 10, can be performed in a constant time fashion by using a mask, as follows: compute $\texttt{mask} = 0 - S_{i,j}$, so that `mask` is 0 if $S_{i,j} = 0$ or a binary vector of 1's otherwise; then compute $t = a_j \otimes \texttt{mask}$ and finally $b_i = b_i + t$.

It is also possible to pre-compute all the matrices $S, S^2, S^3, \dots, S^{n-1}$ to avoid the steps 2 and 3 of Algorithm 10. This, for example, results in 28.6KB of precomputed matrices for ROLLO-I-128.

**Algorithm 10:** poly_pow$_m$($\mathbf{a}$): polynomial exponentiation $\mathbf{a}^{2^{km}}$ in $\mathbb{F}_{2^m}[x]/P(x)$

```
   input   : $\mathbf{a} = (a_0, \dots, a_{n-1}) \in \mathbb{F}_{2^m}[x]/P(x)$, the pre-computed matrix $S$ and
             $k \in \{1, \dots, n-1\}$
   output  : $\mathbf{b} = (b_0, \dots, b_{n-1}) = \mathbf{a}^{2^{km}}$
 1 $\hat{S} = S$
 2 for $i$ = 1, . . . , k-1 do
 3     $\hat{S} = \hat{S} \cdot S$
 4 for $i$=0, . . . , n-1 do
 5     $b_0 = 0$
 6     for $j$=0, . . . , n-1 do
 7         if $\hat{S}_{i,j} \neq 0$ then
 8             $b_i = b_i + a_j$
 9 return b
```

**Algorithm 11:** poly_inv($\mathbf{a}$): polynomial inversion in $\mathbb{F}_{2^m}[x]/P(x)$

```
   input   : $\mathbf{a} = (a_0, \dots, a_{n-1}) \in \mathbb{F}_{2^m}[x]/P(x)$
   output  : $\mathbf{b} = \mathbf{a}^{-1}$
 1 $\mathbf{c}$ = poly_pow$_1(\mathbf{a})$
 2 for $i$=2, . . . , n-1 do
 3     $\mathbf{t_1}$ = poly_pow$_i(\mathbf{a})$
 4     $\mathbf{c}$ = poly_mul$(\mathbf{t_1}, \mathbf{c})$
 5 $\mathbf{s}$ = poly_mul$(\mathbf{a}, \mathbf{c})$ ;                 // $\mathbf{a}^r = \mathbf{a} \cdot \mathbf{a}^{r-1}$
 6 $s_0 = s_0^{-1}$ ;                                    // $s_0 \in \mathbb{F}_{2^m}$
 7 for $i$=0, . . . , n-1 do
 8     $b_i = s_0 \cdot c_i$
 9 return b
```

## 4.4 Rank Syndrome Recovery algorithm and Decapsulation

In this section we describe the core of the decapsulation phase: the Rank Support Recovery (RSR) algorithm which was introduced in [11] and made constant time in [**?**].

Let $E, F$ be two $\mathbb{F}_q$-subspaces of $\mathbb{F}_{q^m}$ and let $(e_1, \dots, e_r)$ be a basis of $E$ and $(f_1, \dots, f_d)$ be a basis of $F$. So $\mathsf{dim}(E) = r$ and $\mathsf{dim}(F) = d$. We denote by $EF$ the subspace generated by the product of the elements of $E$ and $F$:

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

Note that $(e_i f_j)_{1 \leq i \leq r, 1 \leq j \leq d}$ is a generator family of $EF$. Thus, $\mathsf{dim}(EF) \leq rd$ and the equality holds with an overwhelming probability [1]. For that reason, we assume that $\mathsf{dim}(EF) = rd$.

Let $C$ be a LRPC code with parity check matrix $H \in (\mathbb{F}_{q^m})^{2n \times n}$ and let $\mathbf{s} = (s_1, \ldots, s_n)$ be a syndrome of the error vector $\mathbf{e} = (e_1, \ldots, e_{2n})$, that is, $H\mathbf{e}^T = \mathbf{s}^T$. Let $E$ be the support of $\mathbf{e}$ and $S$ be the support of $\mathbf{s}$. Since $S$ is a subspace of $EF$, its dimension is at most $rd$. Finally we denote by $B_i = f_i^{-1}S$.

The RSR algorithm (Algorithm 12) takes as input the base of the vector space $F$, the syndrome $\mathbf{s}$ and the dimension of $E$ i.e. $r$; and its output is (probably) $E$, i.e. the support of the error $\mathbf{e}$. The algorithm is divided in two parts: the first part *tries* to compute $EF$ and it is needed to reduce the decoding failure rate; the second part recovers the vector space $E$ (see [3] for more details).

Let us start with the second part of algorithm: recover the support $E$ of the error vector $\mathbf{e}$. Since the coordinates of the syndrome can be seen as elements in $EF$, the idea is to compute the support of the error as

$$E = B_1 \cap B_2 \cap \ldots \cap B_d, \text{ where } B_i = f_i^{-1}S.$$

In fact,

$$B_i = \{f_i^{-1}f_1e_1, f_i^{-1}f_2e_1, \ldots, f_i^{-1}f_de_r\} = \{e_1, \ldots e_r, f_i^{-1}f_je_t\}_{1\le j\le d, i\ne j, 1\le t\le r}$$

Note that this method fails to recover $E$ when the syndrome space $S$ is different from $EF$ and when the intersection contains others elements besides the $e_j$'s [1]. To minimize the probability that $\mathsf{dim}(S)$ is smaller than $rd$ we have to compute the vector space $EF$ as $S + F \cdot \sum_{i\ne j}(B_i \cap B_j)$ [3]. In particular in Section 1.4.2 of [1] the authors prove that the probability of not recovering $EF$ during the first part of the algorithm is very low if for each iteration one computes

$$S + F \cdot ((B_i \cap B_{i+1}) + (B_{i+1} \cap B_{i+2}) + (B_i \cap B_{i+2})). \tag{3}$$

In fact, if $\mathbf{x} \in F \cdot ((B_i \cap B_{i+1}) + (B_{i+1} \cap B_{i+2}) + (B_i \cap B_{i+2}))$ but $\mathbf{x} \notin S$, then $S + \mathbf{x} = EF$ and we can decode successfully. Thus, in Part I of Algorithm 12, we compute $EF$ as (3).

In Algorithm 12:

– we use capital letter both for the output of Zassenhaus algorithm (Section 4.2 p. 13) and the matrices with elements in $\mathbb{F}_{q^m}$. In this last case we denote by $J^{\{i\}}$ the $i$-th row of the matrix $J$;
– We indicate by $T, \_ = \mathsf{zassenhaus}(B_i, B_j)$ the first element of the Zassenhaus algorithm output, i.e. $B_i + B_j$ and with $\_, T = \mathsf{zassenhaus}(B_i, B_j)$ the second element of the output, that is, $B_i \cap B_j$.
– $T$ is a temporary value. The $i$-th element of $T$ is denoted by $t_i$
– $m_{kr+l}$ in Step 13 is the $(kr + l)$-th element of a temporary vector $\mathbf{m}$.

## 5 Performance

We benchmark our implementation of ROLLO-I-128 on Mac OSX 10.14 (Mojave) equipped with 2.6GHz Intel Core i7. We use SUPERCOP to compare our

**Algorithm 12:** RSR: Rank Support Recover (RSR) algorithm

```
input   : $F = \langle f_1, \dots, f_d \rangle$, $\mathbf{s} = (s_0, \dots, s_{n-1}) \in (\mathbb{F}_{q^m})^n$, $r$
output  : $E = \langle e_1, \dots, e_r \rangle$
// Part I: compute vector space $EF$.
// Step 1: pre-compute $B_0, \dots, B_{d-1}$ where $B_i = (b_{i,0}, \dots, b_{i,n-1})$.
for $i = 0, \dots, d-1$ do
    for $j = 0, \dots, n-1$ do
        $b_{i,j} = f_i^{-1} s_j$
    $B_i^T$ = gauss($B_i^T$)
// Step 2: pre-compute $B_i \cap B_{i+1}$ for $i = 0, \dots, d-2$.
for $i = 0, \dots, d-1$ do
    $\_, J^{\{i\}}$ = zassenhaus($B_i, B_{i+1}$)
// Step 3: compute $F \cdot ((B_i \cap B_{i+2}) + (B_i \cap B_{i+1}) + (B_{i+1} \cap B_{i+2}))$
for $i = 0, \dots, d-2$ do
    $\_, T$ = zassenhaus($B_i, B_{i+2}$)
    $T, \_$ = zassenhaus($T, J^{\{i\}}$)
    $T, \_$ = zassenhaus($T, J^{\{i+1\}}$)
    for $k = 0, \dots, d-1$ do
        for $l = 0, \dots, r-1$ do
            $m_{kr+l} = f_k \cdot t_l$
    // Step 4: compute $S + F \cdot ((B_i \cap B_{i+2}) + (B_i \cap B_{i+1}) + (B_{i+1} \cap B_{i+2}))$
    $T, \_$ = zassenhaus($\mathbf{s}, M$)
    if dim($T$) $\leq rd$ then
        $\mathbf{s} = T$
// Part II: recover the vector space $E$.
// Step 5: compute a base of $E$ as $\bigcap_{i=0}^{d-1} f_i^{-1} S$
for $j = 0, \dots, n-1$ do
    $t_j = f_0^{-1} \cdot s_j$
for $i = 1, \dots, d-1$ do
    for $j = 0, \dots, n-1$ do
        $d_{i,j} = f_i^{-1} \cdot s_j$
    $\_, T \leftarrow$ zassenhaus($T, D_i$)
// Step 6: expand the basis of $E$ to a full vector space $E = \{e_0, \dots, e_{2^r-1}\}$
$e_0 = 0$
for $i = 0, \dots, r-1$ do
    $e_{2^i} = t_i$
    for $j = 1, \dots, 2^i - 1$ do
        $e_{j+2^i} = e_j + t_i$
// Step 7: order $E$ error list to make it unique for the hash function
for $i = 0, \dots, 2^r - 2$ do
    for $j = 1, \dots, 2^r - i - 2$ do
        if $e_j > e_{j+1}$ then
            swap $(e_j, e_{j+1})$
return $E$
```

implementation with other existing KEMs. In the key generation function and the encryption function we use the random-number generator `randombytes()` provided by SUPERCOP. Note that our implementation uses a stand-alone implementation of SHA512, but for a fair comparison, we have switched to OpenSSL's SHA512 implementation, which is also used in the implementation of LAKE I (ROLLO-I-128 predecessor). All primitives are compiled using `clang` with parameters `-march=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -Wl,-no_pie`. For non-vectorized implementation, we disable the flag `-march=native`.

Table 2 describes the performance comparison of vectorized ROLLO-I-128 with other code-based KEMs available in SUPERCOP. Our implementation of ROLLO-I-128 outperform LAKE I for the encapsulation and decapsulation functions by factor of 12.9 and 2.9 respectively. The efficiency of encapsulation and decapsulation are also observed in the non-vectorized version, both by factor of 2.7 and 3. On the other hand, the key generation function of LAKE I is more efficient due to the constant-time algorithm that we implemented to compute the multiplicative inverse of nonzero elements in $\mathbb{F}_{2^m}[x]/P(x)$. The performance loss of the constant-time implementation is by a factor of 1.12 for the AVX2 implementation and 3.1 for the plain C implementation.

| Algorithm | Key Generation | Encapsulation | Decapsulation |
|---|---|---|---|
| ROLLO-I-128 AVX2 | $1,745,137$ | $30,305$ | $576,298$ |
| `lake1` | $1,554,265$ | $391,310$ | $1,711,494$ |
| `bike1l1/ref_ntl` | $140,078$ | $158,279$ | $1,859,941$ |
| `bike1l1nc/ref_ossl` | $243,858$ | $291,699$ | $8,578,806$ |
| `bike1l1sc/ref_ossl` | $489,785$ | $436,901$ | $3,775,149$ |
| `bike2l1/ref_ntl` | $2,535,915$ | $88,834$ | $1,737,511$ |
| `bike2l1nc/ref_ossl` | $12,873,353$ | $137,172$ | $7,987,542$ |
| `bike2l1sc/ref_ossl` | $12,944,118$ | $337,291$ | $2,705,820$ |
| `bike3l1/ref_ntl` | $117,140$ | $173,860$ | $2,536,004$ |
| `bike3l1nc/ref_ossl` | $152,356$ | $286,161$ | $9,132,028$ |
| `bike3l1sc/ref_ossl` | $495,591$ | $615,626$ | $4,406,908$ |
| `ledakem12` | $78,578,100$ | $2,705,404$ | $39,688,782$ |
| `ledakem13` | $33,972,819$ | $2,660,053$ | $43,427,083$ |
| `ledakem14` | $32,242,778$ | $3,478,499$ | $51,346,894$ |
| `ntskem1264` | $30,093,200$ | $64,482$ | $186,965$ |

**Table 2.** Comparison on the number of cycles to perform key generation, encapsulation, and decapsulation among code-based KEMs available in SUPERCOP.

## 6 Conclusion

In this work, we have presented several algorithms which make shed some light on the potential performance of a fully optimized constant-time implementation

| Algorithm | Key Generation | Encapsulation | Decapsulation |
|---|---|---|---|
| ROLLO-I-128 | $10,453,664$ | $214,753$ | $807,338$ |
| `lake1` | $3,361,545$ | $590,345$ | $2,415,689$ |

**Table 3.** Comparison on the number of cycles to perform key generation, encapsulation, and decapsulation for non-vectorized implementation of ROLLO-I-128 and LAKE I.

of ROLLO. It highlights that this proposal can be quite interesting from a computational point of view both with AVX2 and without. Future work will consist in porting these algorithms to other variants of ROLLO as well as some parts of RQC which might benefit from those improvements.

## References

1. Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Hauteville, A., Ruatta, O., Tillich, J.P., et al.: Rollo-rank-ouroboros, lake & locker (2018)
2. Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Rank quasi-cyclic (rqc) (2017), https://pqc-rqc.org/doc/rqc-specification_2017-11-30.pdf
3. Aragon, N., Gaborit, P., Hauteville, A., Ruatta, O., Zémor, G.: Low rank parity check codes: New decoding algorithms and applications to cryptography. arXiv preprint arXiv:1904.00357 (2019)
4. Bernstein, D.J., Chou, T., Schwabe, P.: McBits: fast constant-time code-based cryptography. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 250–272. Springer (2013)
5. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symbolic Comput. **24**(3-4), 235–265 (1997). https://doi.org/10.1006/jsco.1996.0125, http://dx.doi.org/10.1006/jsco.1996.0125, computational algebra and number theory (London, 1993)
6. Devoret, M.H., Schoelkopf, R.J.: Superconducting circuits for quantum information: An outlook. Science **339**(6124), 1169–1174 (2013)
7. Eron Anderson, S.: Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html, accessed: 2019-05-30
8. Faure, C., Loidreau, P.: A new public-key cryptosystem based on the problem of reconstructing $p$–polynomials. In: International Workshop on Coding and Cryptography. vol. 3969, pp. 304–315. Springer (2005). https://doi.org/10.1007/11779360_24
9. Gabidulin, E.M.: Theory of codes with maximum rank distance. Problemy Peredachi Informatsii **21**(1), 3–16 (1985)
10. Gabidulin, E.M., Paramonov, A., Tretjakov, O.: Ideals over a non-commutative ring and their application in cryptology. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 482–489. Springer (1991)
11. Gaborit, P., Murat, G., Ruatta, O., Zémor, G.: Low rank parity check codes and their application to cryptography. In: Proceedings of the Workshop on Coding and Cryptography WCC-2013, Bergen, Norway (04 2013)
12. Gaborit, P., Otmani, A., Kalachi, H.T.: Polynomial-time key recovery attack on the faure–loidreau scheme based on gabidulin codes. Designs, Codes and Cryptography **86**(7), 1391–1403 (Jul 2018)

13. Golay, M.: Notes on digital coding. Proc.I.R.E., IEEE (1949)
14. Guajardo, J., Paar, C.: Fast inversion in composite galois fields gf $((2^n)^m)$. In: IEEE international symposium on information theory. pp. 295–295. Citeseer (1998)
15. Gueron, S., Kounavis, M.E.: Intel® carry-less multiplication instruction and its usage for computing the gcm mode. White Paper (2010)
16. Guo, Q., Johansson, T., Wagner, P.: A key recovery reaction attack on qc-mdpc. IEEE Transactions on Information Theory (10 2018)
17. Hoffstein, J., Pipher, J., Silverman, J.H.: Ntru: A ring-based public key cryptosystem. In: Lecture Notes in Computer Science. pp. 267–288. Springer-Verlag (1998)
18. Intel® C++ Compiler 19.0 Developer Guide and Reference. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-overview-intrinsics-for-intel-advanced-vector-extensions-2-intel-avx2-instructions, accessed: 2019-05-27
19. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases. Information and computation **78**(3), 171–177 (1988)
20. Kachigar, G., Tillich, J.P.: Quantum information set decoding algorithms. In: Lange, T., Takagi, T. (eds.) Post-Quantum Cryptography. pp. 69–89. Springer International Publishing, Cham (2017)
21. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akademii Nauk SSSR, Translation in Physics-Doklady 7, 595-596, 1963 **145**(2), 293–294 (1962)
22. McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (1978)
23. Misoczki, R., Tillich, J.P., Sendrier, N., Barreto, P.S.L.M.: Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In: 2013 IEEE International Symposium on Information Theory. pp. 2069–2073 (July 2013)
24. NIST: Pqc call for proposals (2018), available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals
25. Overbeck, R.: Structural attacks for public-key cryptosystems based on gabidulin codes. Journal of Cryptology **21**(2), 280–301 (2008)
26. Overbeck, R.: A new structural attack for gpt and variants. In: International Conference on Cryptology in Malaysia. pp. 50–63. Springer (2005)
27. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (Feb 1978)
28. Salem Al Abdouli, A., Al Ali, M., Bellini, E., Caullery, F., Hasikos, A., Manzano, M., Mateu, V.: Drankula: A mceliece-like rank metric based cryptosystem implementation. pp. 230–241 (01 2018). https://doi.org/10.5220/0006838102300241
29. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal (1948)
30. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997)
31. Victor Shoup: Ntl: A library for doing number theory (2019), available at https://shoup.net/ntl/

## A ROLLO-II and ROLLO-III

### A.1 ROLLO-II

ROLLO-II is a Public Key Encryption (PKE) scheme which is defined by a triple of probabilistic algorithms = ($\mathsf{KeyGen}$; **Encryption**; **Decryption**) together with a key space $\mathcal{K}$. It is almost identical to ROLLO-I and can be seen as the PKE version of ROLLO-I. In details, we have:

– **Key generation** generates a pair of public and secret key ($\mathsf{pk}$; $\mathsf{sk}$).
  - Private key:
    1. Randomly select a vectorial subspace $F$ of $\mathbb{F}_{q^m}$ of dimension $d$ and samples a couple of vectors $(\mathbf{x}, \mathbf{y}) \in F^n \times F^n$ such that $\mathbf{x}$ is invertible mod $P$ (which is equivalent to $\mathbf{x}$ being a non-zero vector).
    2. Set the secret key as $\mathsf{sk} = (x, y)$.
  - Public key:
    1. Compute $h = \mathbf{x}^{-1}\mathbf{y} \mod P$.
    2. Set the public key as $\mathsf{pk} = `h$.

– **Encryption** uses the public key $\mathsf{pk}$ and a message $M$ to produce a ciphertext $C = (\mathbf{c}, \mathit{cipher})$.
  1. Randomly select a vectorial subspace $E$ of $\mathbb{F}_{q^m}$ of dimension $r$ and samples uniformly a couple of vectors $(\mathbf{e}_1, \mathbf{e}_2) \in E^n \times E^n$.
  2. Compute $\mathbf{c} = \mathbf{e}_1 + \mathbf{e}_2\mathbf{h} \mod P$.
  3. Compute $\mathit{cipher} = G(E) \oplus M$ where $G(E)$ is a hash function.
  4. Send $C = (\mathbf{c}, \mathit{cipher})$.

– **Decryption** using the secret key $\mathsf{sk}$ and a ciphertext $C$, recovers the message $M$ or fails and return $\perp$.
  1. Compute $\mathbf{s} = \mathbf{xc} = \mathbf{xe}_1 + \mathbf{ye}_2 \mod P$
  2. Use Rank Support Recovery (RSR) algorithm to recover $E$. The RSR algorithm takes as input $F, \mathbf{s}$ and $r$ (see Section 4.4 for more detail).
  3. get $M = G(E) \oplus \mathit{cipher}$.

We refer to Table 4 for the size of ROLLO-II parameters. Note that the private key can be obtained from a seed, and in ROLLO official NIST submission the seed expander is initialized with 40 bytes long seeds.

| Instance | $q$ | $m$ | $n$ | $d$ | $r$ | $P$ | $\mathsf{sk}$ size | $\mathsf{pk}$ size | $c$ size | Security | failure rate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ROLLO-II-128 | 2 | 83 | 149 | 8 | 5 | $X^{47} + X^5 + 1$ | 40B | 1546B | 1674B | 128b | $2^{-128}$ |
| ROLLO-II-192 | 2 | 107 | 151 | 8 | 6 | $X^{53} + X^6 + X^2 + X + 1$ | 40B | 2020B | 2148B | 192b | $2^{-128}$ |
| ROLLO-II-256 | 2 | 127 | 157 | 8 | 7 | $X^{67} + X^5 + X^2 + X + 1$ | 40B | 2493B | 2621B | 256b | $2^{-132}$ |

**Table 4.** ROLLO-II parameters

### A.2 ROLLO-III

ROLLO-III KEM $= (\mathsf{KeyGen}; \mathsf{Encaps}; \mathsf{Decaps})$ is a triple of probabilistic algorithms together with a key space $\mathcal{K}$.

- **Key generation** generates a pair of public and secret key $(\mathsf{pk}; \mathsf{sk})$.
  - Private key:
    1. Randomly select a vectorial subspace $F$ of $\mathbb{F}_{q^m}$ of dimension $d$ and samples a couple of vectors $(\mathbf{x}, \mathbf{y}) \in F^n \times F^n$ and a random vector $\mathbf{h} \in \mathbb{F}_{q^m}^n$.
    2. Set the secret key as $\mathsf{sk} = (x, y)$.
  - Public key:
    1. Compute $\mathbf{s} = \mathbf{x} + \mathbf{hy} \mod P$.
    2. Set the public key as $\mathsf{pk} = (\mathbf{h}, \mathbf{s})$.
- **Encapsulation** uses the public key $\mathsf{pk}$ to produce an encapsulation $\mathbf{c}$, and a key $K \in \mathcal{K}$.
  1. Randomly select a vectorial subspace $E$ of $\mathbb{F}_{q^m}$ of dimension $r$ and samples uniformly a triple of vectors $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{e_r}) \in E^{3n}$.
  2. Compute $\mathbf{s_r} = \mathbf{r}_1 + \mathbf{hr}_2 \mod P$ and $\mathbf{s_e} = \mathbf{sr}_2 + \mathbf{e_r} \mod P$
  3. Compute $K = G(E)$ where $G(E)$ is a hash function.
  4. Send $\mathbf{c} = (\mathbf{s_e}, \mathbf{s_r})$.
- **Decapsulation** using the secret key $\mathsf{sk}$ and a ciphertext $c$, recovers the key $K \in \mathcal{K}$ or fails and return $\perp$.
  1. Compute $\mathbf{e_c} = \mathbf{s_e} - \mathbf{ys_r} \mod P$
  2. Use Rank Support Recovery (RSR) algorithm to recover $E$. The RSR algorithm takes as input $F, \mathbf{e_c}$ and $r$ (see Section 4.4 for more detail).
  3. get $K = G(E)$.

We refer to Table 5 for the actual set of ROLLO-III parameters. Note that the private key can be obtained from a seed, and in ROLLO official NIST submission the seed expander is initialized with 40 bytes long seeds.

| Instance | $q$ | $m$ | $n$ | $d$ | $r$ | $P$ | $\mathsf{sk}$ size | $\mathsf{pk}$ size | $c$ size | Security | failure rate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ROLLO-III-128 | 2 | 101 | 47 | 6 | 5 | $X^{47} + X^5 + 1$ | 40B | 634B | 1188B | 128b | $2^{-30}$ |
| ROLLO-III-192 | 2 | 107 | 59 | 8 | 6 | $X^{59} + X^7 + X^4 + X^2 + 1$ | 40B | 830B | 1580B | 192b | $2^{-36}$ |
| ROLLO-III-256 | 2 | 131 | 67 | 8 | 7 | $X^{67} + X^5 + X^2 + X + 1$ | 40B | 1138B | 2196B | 256b | $2^{-42}$ |

**Table 5.** ROLLO-III parameters

**Algorithm 13:** $\mathsf{inv}_{\mathbb{F}_{2^{79}}}(a)$: inversion in $\mathbb{F}_{2^{79}}$

```
   input   : $a \in \mathbb{F}_2^{79}$
   output  : $c = a^{-1} \in \mathbb{F}_2^{79}$
 1 $r_0 = a$
 2 $r_1 = r_0^2$ ;                          // $r_1 = a^2$
 3 $r_0 = r_1 \cdot r_0$;                   // $r_0 = a^{2^2-1}$
 4 $r_1 = r_0^2$ ;                          // $r_1 = a^6$
 5 $r_0 = r_1 \cdot a$;                     // $r_0 = a^{2^3-1}$
 6 $r_2 = r_0$;                             // $r_2 = a^{2^3-1}$
 7 $r_1 = r_0^{2^3}$ ;                      // $r_1 = a^{2^6-2^3}$
 8 $r_0 = r_1 \cdot r_0$;                   // $r_0 = a^{2^6-1}$
 9 $r_1 = r_0^{2^3}$ ;                      // $r_1 = a^{2^9-2^3}$
10 $r_0 = r_1 \cdot r_2$;                   // $r_0 = a^{2^9-1}$
11 $r_1 = r_0^{2^9}$ ;                      // $r_1 = a^{2^{18}-2^9}$
12 $r_0 = r_1 \cdot r_0$;                   // $r_0 = a^{2^{18}-1}$
13 $r_1 = r_0^{2^{18}}$ ;                   // $r_1 = a^{2^{36}-2^{18}}$
14 $r_0 = r_1 \cdot r_0$;                   // $r_0 = a^{2^{36}-1}$
15 $r_1 = r_0^{2^3}$ ;                      // $r_1 = a^{2^{39}-2^3}$
16 $r_0 = r_1 \cdot r_2$;                   // $r_0 = a^{2^{39}-1}$
17 $r_1 = r_0^{2^{39}}$ ;                   // $r_1 = a^{2^{78}-2^{3}9}$
18 $r_0 = r_1 \cdot r_0$;                   // $r_0 = a^{2^{78}-1}$
19 $c = r_0^2$;                             // $c = a^{2^{79}-2}$
20 return $c$
```