Constant time algorithms for ROLLO-I-128

Carlos Aguilar-Melchor[1], Nicolas Aragon[2†], Emanuele Bellini[3], Florian Caullery[3], Rusydi H. Makarim[3] and Chiara Marcolla[3]

[1] ISAE-SUPAERO, Université de Toulouse, Toulouse, France
carlos.aguilar-melchor@isae-supaero.fr
[2] Université de Limoges, Limoges Cedex, France
nicolas.aragon@unilim.fr
[3] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
name.lastname@tii.ae

Abstract. In this work, we propose different techniques that can be used to implement the ROLLO, and partially RQC, family of algorithms in a standalone, efficient and constant time library. For simplicity, we focus our attention on one specific instance of this family, ROLLO-I-128. For each of these techniques, we present explicit code (with intrinsics when required), or pseudo-code and performance measures to show their impact. More precisely, we use a combination of original and known results and describe procedures for Gaussian reduction of binary matrices, generation of vectors of given rank, multiplication with lazy reduction and inversion of polynomials in a composite Galois field. We also carry out a global performance analysis to show the impact of these improvements on ROLLO-I-128. Through the SUPERCOP framework, we compare it to other 128-bit secure KEMs in the NIST competition. To our knowledge, this is the first optimized full constant time implementation of ROLLO-I-128.

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

[†] This work was partially funded by French DGA

Contents

1 Introduction

Through the NIST Post-Quantum Cryptography (PQC) Standardization Process [NIS18], the cryptographic community is evaluating candidate KEM, PKE, and signature schemes secure against both quantum and classical attacks. One of the requirements for these schemes to be secure is having a constant time implementation to avoid leakage of secret information through timing attacks. Furthermore, it is important to understand how efficient this constant time implementations are when running on real devices.

From an implementation perspective, while the cryptographic community has mostly focused its efforts on improving some categories of cryptosystems such as those based on lattices and codes on the Hamming metric, the same cannot be claimed for rank-based cryptosystems. Compared to lattice and code-based cryptography, rank-based cryptography is a relatively new and less explored field. Although the first rank based cryptosystem, the Gabidulin-Paramonov-Tretjakov (GPT) public key encryption scheme [GPT91], was introduced in 1991, and many analysis were presented in the subsequent years (such as [Ove05], [Ove08], [FL05], [GOK18]), only recently new schemes have been proposed, such as [Loi17, AAAAB+18, ASBB+19, BCH+18, BCG+19], some of which have also been submitted to the NIST PQC standardization process. Up until the algebraic attack recently presented in [BBC+20], these schemes seemed to provide appealing performance levels and key and ciphertext sizes, which enabled ROLLO [AMAB+18] (merge of LAKE, LOCKER, and Rank-Ouroboros) and RQC [AMAB+17] to pass the first round of the NIST PQC standardization process. Though, in its recent status report on the second round [AASA+20], NIST did not select ROLLO and RQC to advance on with the motivation that their security analysis needs more time to mature. On the other hand, NIST encouraged the cryptographic community to continue studying rank-based cryptosystems, as they offer a nice alternative to traditional Hamming metric codes with comparable bandwidth.

There were still some open questions of practicality pertaining to the most recently submitted NIST package for ROLLO (dated 2020/04/21 and available at `pqc-rollo.org`).

In June 2020, a note was released [DGK20], pointing out potential issues arising from part of the currently submitted ROLLO and RQC code not being constant time.

In this work, we present the first constant time implementation of the 128-bit secure rank-based KEM called ROLLO-I-128. This work complements and improves the preliminary results posted on the NIST website [AMBC$^+$] on the implementation of an earlier version of ROLLO-I-256. Other non-constant time independent implementations of ROLLO on other platforms can be found for example in [BCM$^+$19, ABC$^+$19], or [LMB$^+$19], where a software implementation on a Cortex M0 of the encapsulation routine, and a hardware implementation on a microcontroller with a crypto coprocessor, are presented, respectively.

1.1 Our contribution

In this work, we propose different techniques that can be used to implement ROLLO and part of the RQC family of algorithms in a standalone, efficient and constant time library. Recall that ROLLO-I-128, ROLLO-I-192, and ROLLO-I-256 have decryption failure probability of 2^{-28}, 2^{-34}, and 2^{-33}, which, cryptographically, are not considered small. We present, for each of the proposed techniques, explicit code (with intrinsics when required), or pseudo-code and performance measures to show their impact.

As a theoretical contribution, we describe a new constant time variant of Gaussian elimination that reduces any matrix to its (not necessarily reduced) row echelon form. The only previous constant time variant we are aware of [BCS13], only worked for full rank matrices, and returned a systematic form of such matrices, terminating the algorithm if this was not possible. Furthermore, after analyzing current non-constant time algorithms to generate a list of vectors with a given rank, we describe a novel a constant time probabilistic version of one of these algorithms, and we present a procedure for reducing the probability of failing to a desired value. We also present a variation of this method which returns the entire support of the vector list. This potentially allows trade-offs between the public key size and the performance of the encapsulation step.

From an implementation perspective, we describe in detail the process of implementing the underlying finite field arithmetic with constant time operations, with and without the use of vectorization techniques. We provide an explicit description of the application of the Zassenhaus algorithm in the Rank Support Recovery algorithm described in the NIST submission of ROLLO [AMAB$^+$18]. We show how efficient polynomial arithmetic can be conducted by applying multiplication with lazy reduction and inversion of polynomials in a composite Galois field defined by a pentanomial. All of these are implemented using reasonably optimized constant time algorithms. Finally, we carry out a performance analysis to show the impact of these improvements on our implementation of ROLLO-I-128 [1], when compared with its reference and optimized implementations. 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 rank-based cryptographic algorithms.

1.2 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 encapsulation method. 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 performance of our implementation of ROLLO-I-128 with the one of various KEM submissions to the NIST PQC standardization process. Section 6 presents the conclusions drawn from this study.

[1] The full code of our implementation and of the tests we run for the comparisons is available at https://github.com/peacker/constant_time_rollo.

2 Preliminaries

In this section, we first present the rings, fields and vector spaces we will work with as well as an associated metric, namely the rank metric, and then we will define error-correcting codes associated with this metric.

2.1 Structures and representations

In the following we let q be a prime power and m, n two positive integers. We will work with the finite fields of order q, q^m and q^{mn}: $\mathbb{F}_q, \mathbb{F}_{q^m}, \mathbb{F}_{q^{mn}}$. Of course there are multiple isomorphic fields of a given order, with multiple representations and leading to different calculation algorithms.

$\mathbb{F}_{\boldsymbol{q}}$**.** In this paper, as in the ROLLO specification, q will always be 2 and therefore elements and computations in $\mathbb{F}_q$ are associated to elements and computations in the modular ring $\mathbb{Z}/2\mathbb{Z}$.

$\mathbb{F}_{\boldsymbol{q^m}}$**.** As usual, elements in extensions of the base field $\mathbb{F}_q$ will be represented using quotients over the polynomial ring $\mathbb{F}_q[X]$. Thus elements and computations in $\mathbb{F}_{q^m}$ are associated to polynomial representations and computations over $\mathbb{F}_q[X]/\langle P_0 \rangle$ for an irreducible polynomial P_0 of degree m.

$\mathbb{F}_{\boldsymbol{q^{mn}}}$**.** Elements and computations in $\mathbb{F}_{q^{mn}}$ are similarly associated to polynomial representations and computations over $\mathbb{F}_{q^m}[X]/\langle P \rangle$ for an irreducible polynomial $P \in \mathbb{F}_q[X]$ of degree n. Note that these polynomials have coefficients in $\mathbb{F}_{q^m}$, so elements in $\mathbb{F}_{q^{mn}}$ are seen as polynomials (that live in $\mathbb{F}_{q^m}[X]/\langle P \rangle$) with polynomial coefficients (that live in $\mathbb{F}_q[X]/\langle P_0 \rangle$).

It is also quite practical to use vectors and matrices to represent, and operate on, polynomials. For a field F, $\mathcal{M}_{n,m}(F)$ represents the set of matrices with n rows and m columns of elements in F. When n equals m this set, together with classical matrix sum and product, forms a ring that we denote $\mathcal{M}_n(F)$. Of course we can map polynomials to vectors (of coefficients) and inversely so we often consider an element of $\mathbb{F}_{q^m}$ as an element of the vector space $\mathbb{F}_q^m$, and an element of $\mathbb{F}_{q^{mn}}$ as an element of the vector space $\mathbb{F}_{q^m}^n$. For a vector $\mathbf{v}$ we note the associated polynomial $\mathbf{v}(X)$, and for a polynomial p we note the associated vector $\text{vec}(p)$. When using a polynomial in a setting in which it is clear we have to use the vector representation (e.g. a matrix line, or a matrix/vector multiplication) we will not make the vec transformation explicit.

Vector additions are naturally defined in $\mathbb{F}_q^m$ or $\mathbb{F}_{q^m}^n$ and correspond to polynomial additions over $\mathbb{F}_{q^m}$ and $\mathbb{F}_{q^{mn}}$. We define the product of two vectors $\mathbf{u}, \mathbf{v}$ by $\mathbf{uv} = \text{vec}(\mathbf{u}(X)\mathbf{v}(X))$, and the inverse as $\mathbf{u}^{-1} = \text{vec}(\mathbf{u}^{-1}(X))$.

It is also possible to define vector multiplication directly over vector/matrices. In order to do this we will first define ideal matrices. As we will only describe explicitly multiplications in $\mathbb{F}_{q^{mn}}$ we will focus our definition on this specific setting.

Definition 1 (Ideal Matrices)**.** Let $P \in \mathbb{F}_q[X]$ be a polynomial of degree n and $\mathbf{v} \in \mathbb{F}_{q^m}^n$ the vector representation of an element of $\mathbb{F}_{q^{mn}}$. The ideal matrix generated by $\mathbf{v}$ modulo P is the matrix denoted $\mathcal{IM}_P(\mathbf{v}) \in \mathcal{M}_n(\mathbb{F}_{q^m})$ with n rows of the form $X^i\mathbf{v}(X) \bmod P$, with $i = 0, \ldots, n-1$.

The multiplication of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}_{q^{mn}}$ can be then computed with $\mathbf{uv} = \mathbf{u}\mathcal{IM}_P(\mathbf{v}) = (\mathcal{IM}_P(\mathbf{u})^T v^T)^T = \mathbf{vu}$. Note that this definition is compatible with the previous one as we have $\mathbf{u}\mathcal{IM}_P(\mathbf{v}) = \text{vec}(u(X)v(X))$.

2.2 Metric and support

Let $\mathbf{e} = (e_1, \ldots, e_n)$ be an element of $\mathbb{F}_{q^m}^n$. Denote by $e_{i,j}$ the j-th component of e_i, e_i being seen as an element of $\mathbb{F}_q^m$. Then the *rank weight* of e, denoted by $\mathsf{w_R}(\mathbf{e})$, is defined as $\mathsf{w_R}(\mathbf{e}) = \mathsf{rank}\left([e_{i,j}]_{i=1,\ldots,n,j=1,\ldots,m}\right)$ The rank distance between two vectors $\mathbf{e}, \mathbf{f} \in \mathbb{F}_{q^m}^n$ is defined by $\mathsf{w_R}(\mathbf{e} - \mathbf{f}) = ||\mathbf{e} - \mathbf{f}||$.

For $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$, 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 $\mathbf{x}$: $E = \langle x_1, \ldots, x_n \rangle_{\mathbb{F}_q}$. Note that $\mathsf{dim}(E) = \mathsf{w_R}(\mathbf{x})$ and that any $\mathbf{e} \in E$ can be written as $\mathbf{e} = \sum_{i=1}^n \lambda_i x_i$ where $\lambda_i \in \mathbb{F}_q$.

2.3 Codes

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. An element of a code C is called a *codeword*. 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.

As a linear code is a vector 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.

We present now 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 underlying problem on which relies the security of the schemes. We first recall the definition of *ideal codes* and *LRPC codes.*

Definition 2 (Ideal Codes)**.** Let $P \in \mathbb{F}_q[X]$ be a polynomial of degree n and $\mathbf{h_1}, \mathbf{h_2} \in \mathbb{F}_{q^m}^n$. We define the $[2n, n]_{q^m}$ ideal code C defined by $(\mathbf{h_1}, \mathbf{h_2})$ modulo P as the code with parity check matrix $\left(\mathcal{IM}_P(\mathbf{h_1})^T \;\middle|\; \mathcal{IM}_P(\mathbf{h_2})^T \right)$.

If $\mathbf{h_1}(X) = 1$ (and thus $\mathcal{IM}_P(\mathbf{h_1}) = I_n$) we say C is defined by $\mathbf{h_2}$ modulo P. If $\mathbf{h_1}(X)$ is invertible in $\mathbb{F}_{q^{mn}}$, the code C defined by $(\mathbf{h_1}, \mathbf{h_2})$ modulo P is the same as the code defined by $\mathbf{h_1}^{-1}\mathbf{h_2}$ modulo P.

Definition 3 (LRPC codes)**.** Let $H \in M_{n-k,n}(\mathbb{F}_{q^m})$ be a full rank matrix such that its coefficients generate an $\mathbb{F}_q$-subspace $F = \langle h_{i,j} \rangle_{\mathbb{F}_q}$ of small dimension d. The $[n, k]_{q^m}$ code C of parity check matrix H is called an LRPC code of weight d.

A $[2n, n]_{q^m}$ Ideal Code defined by $(\mathbf{h_1}, \mathbf{h_2})$ modulo a polynomial P can also be an LRPC code. Indeed, if $\mathbf{h_1}, \mathbf{h_2}$ are vectors in an $\mathbb{F}_q$-subspace of small dimension and P has its coefficients in $\mathbb{F}_q$, this will be the case. Such a code is called an Ideal LRPC code.

Definition 4 (Ideal LRPC codes)**.** Let F be a $\mathbb{F}_q$-subspace of dimension d of $\mathbb{F}_{q^m}$, $\mathbf{h_1}, \mathbf{h_2}$ two vectors of $\mathbb{F}_{q^m}^n$ with support in F and $P \in \mathbb{F}_q[X]$ a polynomial of degree n. The code C with parity check matrix $(\mathcal{IM}(\mathbf{h_1})^T | \mathcal{IM}(\mathbf{h_2})^T)$ is called an $[2n, n]_{q^m}$ ideal LRPC code.

The variant of ROLLO we focus on this paper, ROLLO-I, has a security proof based on two problems. The first is a support recovery problem, which is proven equivalent to the rank-metric version of the Syndrome Decoding problem (RSD) in [AMAB+18]. The second is an indistinguishability problem.

Problem 1 (r-Ideal Rank Support Recovery)**.** *Given a polynomial $P \in \mathbb{F}_q[X]$ of degree n, 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.*

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

As stated by the submission documentation all ROLLO variants follow the approach inaugurated by the public key encryption protocol NTRU in 1998 [HPS98]. As pointed out in the previous section, ROLLO is a variation of the LRPC rank metric approach and its security is proven assuming that the Ideal LRPC indistinguishability and the 2-Ideal Rank Support Recovery [AMAB+18, Theorem 4.2] problems are hard.

We now describe ROLLO-I in detail. The ROLLO-I Key-Encapsulation Mechanism (KEM) is a triple of probabilistic algorithms ($\mathsf{KeyGen}; \mathsf{Encaps}; \mathsf{Decaps}$). KeyGen: randomly sample $(\mathbf{x}, \mathbf{y})$ from a vector subspace F of $\mathbb{F}_{q^m}$ of dimension d, such that $\mathsf{w_R}(\mathbf{x}) = \mathsf{w_R}(\mathbf{y}) = d$. Set $\mathsf{pk} = \mathbf{h} = \mathbf{x}^{-1}\mathbf{y} \mod P$ and $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$. Encaps: randomly sample $(\mathbf{e}_1, \mathbf{e}_2)$ from a vector subspace E of $\mathbb{F}_{q^m}$ of dimension r, such that $\mathsf{w_R}(\mathbf{e}_1) = \mathsf{w_R}(\mathbf{e}_2) = r$. Compute $\mathbf{c} = \mathbf{e}_1 + \mathbf{e}_2\mathbf{h} \mod P$. Compute $K = G(E)$ where G is a hash function. Output $(\mathbf{c}, K)$. Decaps: Compute $\mathbf{s} = \mathbf{x}\mathbf{c} = \mathbf{x}\mathbf{e}_1 + \mathbf{y}\mathbf{e}_2 \mod P$. Use the Rank Support Recovery (RSR) algorithm (alg. 13) to recover E. The RSR algorithm takes as input $F = \mathsf{Supp}(\mathbf{x}, \mathbf{y})$ and $\mathbf{s}$ (see section 4.4 for more detail). If the RSR algorithm succeeds return $K = G(E)$, else return $\perp$.

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 the official NIST submission the seed expander was initialized with 40 bytes long seeds.

Table 1: ROLLO-I parameters

Instance	q	m	n	d	r	P	sk size	pk size	c size	Security	failure rate
ROLLO-I-128	2	67	83	8	7	$X^{83} + X^7X^4 + X^2 + 1$	40B	696B	696B	128b	2^{-28}
ROLLO-I-192	2	79	97	8	8	$X^{97} + X^6 + 1$	40B	958B	958B	192b	2^{-34}
ROLLO-I-256	2	97	113	9	9	$X^{113} + X^9 + X^2 + X + 1$	40B	1371B	1371B	256b	2^{-33}

As the last column of the table shows, the decapsulation algorithm has a non-zero failure probability. This probability is however well understood and made low enough to fit the NIST call for proposals (for more detail see Section 1.4.2 of [AMAB+18]).

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 Gaussian reduction algorithm for binary matrices, the Zassenhaus algorithm for binary matrices, and 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 the above blocks. Hence, we focused on optimizing every operations included in those layers as well as insuring the fact that they are constant time.

Target. We target processors with 64-bit carryless multiplications (2010 and onward for Intel) and provided a faster alternative if they also have AVX2 instructions (2013 and onward for Intel). The code examples assume GCC's `__uint128_t` type is available and uses GCC X86 intrinsics.

Notation. 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.

Table 2: Cycles per plain C carryless multiplication of polynomials of degree $m = 67$ (averaged over 4 seconds of execution on a Macbook Pro 2017 with an 2.9 GHz Quad-Core Intel Core i7). NTL-ROLLO is the NTL function `mul` defined in *GF2E.h* used initially by ROLLO, and NoNTL-ROLLO is the Karatsuba implementation in the NTL-free version of ROLLO. Polynomials are not reduced (output is of degree $2m - 2 = 132$).

Algorithm	NTL-ROLLO	NoNTL-ROLLO	alg. 14
Poly. Multiplication	187 cycles/op	243 cycles/op	157 cycles/op

4.1 Binary field arithmetic

In this section, we present the constant time vectorized operations we propose for $\mathbb{F}_{q^m}$. As shown in table 1, all variants of ROLLO-I have $q = 2$ and different values for m. Our algorithms work for all the values of m submitted to the NIST competition, but have to be slightly adapted for each value. In order to avoid repetitions, we will focus on the field used by ROLLO-I-128, and note what changes need to be done to adapt the algorithms for other values of m.

We implemented finite field arithmetic for the binary field $\mathbb{F}_{2^m}$, with $m = 67$, representing elements as binary polynomials of degree $m - 1$ modulo an irreducible polynomial of degree m. We used the irreducible pentanomial $P_0(X) = X^{67} + X^5 + X^2 + X + 1$ provided by the Allan Steel database incorporated in Magma software [BCP97] and also suggested by the authors of ROLLO. This pentanomial has also lowest possible intermediate degree, allowing the shortest shift during the reduction operations. No trinomial exists for $m = 67$.

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 (section 4.1 and section 4.1, or a carryless squaring of a single element in section 4.1) and a polynomial reduction (section 4.1). Inversion is performed using an addition chain (see section 4.1). As noted before, all operations in the binary field layer are executed in constant time, assuming the underlying primitives (and in particular carry-less multiplications) are constant time.

Carryless multiplication: plain C implementation The carryless multiplication has been implemented by using recursive Karatsuba multiplication [KO62]. More specifically, we borrowed from NTL[2] an implementation of a constant time carryless Karatsuba multiplication of two 64 bit register (which we call `ntlclmul64` in alg. 14) using only bit manipulation, and then added an extra level of Karatsuba method over this function. The full carryless multiplication $\mathsf{clmul}_K(a, b)$ is described in appendix A, alg. 14.

Table 2 compares this implementation with ROLLO's polynomial multiplications. The initial NTL-based ROLLO (submission date 2019/04/10) used NTL's generic carryless multiplication function[3]. As it is generic, this function goes through a set of tests and function calls before calling exactly the same code we used for `ntlclmul64`. The overhead (5 function calls, 6 if statements with two boolean tests for most of them, and a switch/case) is significant w.r.t. the final code of `ntlclmul64` (78 instructions). As a result, specializing the code by removing calls, conditional branchs, and extracting only the instructions needed for ROLLO we get a 15% speedup on polynomial multiplication with respect to NTL-ROLLO which called the generic function. The Karatsuba function implemented in the NTL-free version of ROLLO (submission date 2019/08/24), called NoNTL-ROLLO

[2] The code is available in the file *mach_desc.h* of the library NTL [Vic19], under the method `NTL_ALT1_BB_MUL_CODE0`.

[3] The `mul` function in *GF2E.h*.

in the table, is 30% slower than NTL's generic function. It seems thus that, in general, implementations of Karatsuba using NTL may obtain a nice performance upgrade just by importing/adapting the specialised code of NTL for this operation, as we did. We also notice that the latest ROLLO implementation dated 2020/04/21, is not NTL dependent anymore.

Carryless multiplication: AVX2 optimization When possible, the carryless multiplication step has been performed using Intel Advanced Vector Extensions 2 instructions (AVX2) [Int]. In particular, the core of this function uses the `_mm_clmulepi64_si128` instruction (see also [GK10]) to perform 64 times 64 bit binary polynomial multiplication.

The multiplication of two m bit binary polynomials is performed in a schoolbook fashion, by dividing the input in two 64 bit registers (one containing only $m - 64$ 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 $\mathsf{clmul}_S(a, b)$ algorithm, and we present our C implementation in appendix A, alg. 15. When irrelevant in the context, we will indicate with $\mathsf{clmul}(a, b)$ (with no subscript) the algorithm performing carryless multiplication, either using Karatsuba method in plain C or with schoolbook method and AVX instructions.

Let us remark that using Karatsuba multiplication [KO62] in this case would not give any advantage, as the cost of multiplication and addition with AVX2 instruction is very close. In practice, we show it even performs worse, due to alignment problems.

Table 3 shows, when comparing figures for NTL-ROLLO and others, that specializing code for ROLLO's setting has an even greater impact on performance when using AVX2, with no surprise. It also shows that alignment issues in Karatsuba have a very noticeable impact on performance and highlights the fact that ROLLO developers did the right choice opting for schoolbook multiplication in the NTL-free version of ROLLO. Our implementation has a little advantage on performance.

Table 3: Average cycles per AVX2 carryless multiplication of polynomials of degree $m = 66$ (averaged over 4 seconds of execution on a MacBook Pro 2017 with a 2.9 GHz Quad-Core Intel Core i7). Karatsuba `clmulepi64` and Schoolbook `clmulepi64` are the AVX2 implementations discussed in this section, NTL-ROLLO is the NTL function mul defined in *GF2E.h* used initially by ROLLO with AVX2 improvements, and NoNTL-ROLLO is the AVX2 Schoolbook implementation in the NTL-free version of ROLLO. Polynomials are not reduced (output is of degree $2m - 2 = 132$).

Algorithm	clmulepi64 Schoolbook	clmulepi64 Karatsuba	NTL-ROLLO	NoNTL-ROLLO
Poly. Multiplication	5.53 cycles/op	7.04 cycles/op	28 cycles/op	6.73 cycles/op

This difference is explained by the fact that the permutation done in our algorithm with `_mm256_permute4x64_epi64` allows us to avoid the cost of the load and store instructions, which are present at the beginning and end of each recursive call in the NIST submitted code.

Carryless squaring 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 $a \in \mathbb{F}_{2^m}$, $a^2 = \left(\sum_{i=0}^{m-1} a_i x^i\right)^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. For example, if the current representation of a was 11100101, then $\mathsf{clsqr}(a)$ 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* [EA]. The pseudocode is given in appendix A, alg. 16.

The squaring method is straightforward from there and its pseudocode is given in in appendix A, alg. 17. For the AVX2 version, a look-up table based on the instruction `_mm_shuffle_epi8` is implemented both in the submission and our work. The AVX2 performance are reported in Table 4.

Table 4: Average cycles per carryless squaring of polynomials of degree $m = 66$ (averaged over 4 seconds of execution on a MacBook Pro 2017 with a 2.9 GHz Quad-Core Intel Core i7). NoNTL-ROLLO (reference impl.) is the lookup table in the reference NTL-free version of ROLLO. Polynomials are not reduced (output is of degree $2m - 2 = 132$). All implementation are AVX2.

Algorithm	This work	NoNTL-ROLLO reference impl.	NoNTL-ROLLO optimized impl.
Poly. Squaring	5.38 cycles/op	16.35 cycles/op	5.80 cycles/op

Reduction The $2m - 2$ bits result provided by the carryless multiplication is reduced back modulo P_0 to a m bit field element, using standard techniques. The pseudocode of the algorithm for reduction is presented in appendix A, alg. 18. The AVX2 performances of the reduction are reported in table 7.

Inversion The inversion of an element $x \in \mathbb{F}_{2^m}$, described in appendix A, alg. 19, has been derived using Fermat's little Theorem stating that $x^{2^m-2} = x^{-1}$. The fixed exponentiation is achieved by the strategy presented in [PCJM18, Section 6.2] using the following addition chain of length 10:

$$1 \to 2 \to 3 \to 6 \to 12 \to 15 \to 30 \to 33 \to 66 \to 67\,.$$

The AVX2 performances of the binary field inversion are reported in table 7.

4.2 Binary vector space arithmetic

In this section, we describe the main algorithms used to manipulate vector spaces, i.e. Gaussian reduction, Zassenhaus algorithm, and the generation of vectors of given rank.

In our implementation, a binary matrix M, usually 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`.

4.2.1 Gaussian elimination algorithm

We introduce an original algorithm to perform a constant time Gaussian elimination to convert any binary matrix to a (not necessarily reduced) row echelon form and its extension to convert it to reduced row echelon form. This algorithm is somehow a generalization of the one presented in [BCS13], where Gaussian elimination was used to convert the binary matrix to a systematic form. In [BCS13], if the matrix is not systematic, the algorithm breaks. Otherwise, for each column, the algorithm first sets to 1 the bits of the diagonal, by scanning the rows of the matrix from below the current pivot to the bottom of the matrix, then sets to 0 the bits in the current column, except the diagonal, by scanning the full set of rows again. This is done in a constant time manner, due to the fact that, being the matrix systematic, the number of rows under the pivot are always the same for each column step. Though, in [BCS13], it is not defined how one could force the algorithm to continue when it is not possible to fix a 1 in the diagonal, i.e. when the matrix is not systematic. We solve the problem by always scanning all rows for each column, and by keeping track of the current pivot position, not necessarily in the diagonal. Let $\tilde{r}$ be the current pivot row position, i is the current scanned row and j the current scanned column. Then we perform

$$\begin{aligned} m_{\tilde{r}} &= m_{\tilde{r}} \oplus \mathsf{mask1} \cdot \mathsf{mask2} \cdot \mathsf{mask3} \cdot m_i \\ m_i &= \mathsf{mask1} \cdot \mathsf{mask3} \cdot m_{\tilde{r}} \oplus m_i \end{aligned}$$

where $\mathsf{mask1}$ is set to 1 if the current row is above the pivot $(i > r)$, $\mathsf{mask2}$ is set to 1 if the the bit $m_{i,j}$ is 0, and $\mathsf{mask3}$ is set to 1 if the bit $m_{\tilde{r},j}$ in the intersection of the current scanned row and column is 1. The steps above have the effect to leave the rows unchanged either when the current row is above the pivot row $m_{\tilde{r}}$ or, otherwise, when the bit $m_{i,j}$ is 0. On the other hand, when $m_{i,j}$ is 1, if the pivot bit $m_{\tilde{r},j}$ is 0, then the current row is swapped with the pivot row, and if the pivot bit $m_{\tilde{r},j}$ is 1, then the 1 in position (i, j) is flipped. Notice that, at the end of the algorithm, the pivot position is also the rank of the matrix. Compared to [BCS13], for each scan of the full set of rows, we perform less XOR operations, but we need to compute more masks. Though we have to scan all columns, while for the method from [BCS13] it is sufficient to scan the minimum between the number of rows and the number of columns. This makes the method of [BCS13] much faster for matrices with a small number of rows. We stress again that the method of [BCS13] only computes the systematic form of a matrix, and for this reason is, in general, faster.

Our method can be easily extended to compute the *reduced* row echelon form, by storing the pivot positions and then scanning all the rows r times, where r is the number of rows, to remove the 1's above the pivots.

The differences between our method and the one in [BCS13] are summarized in table 5 and table 6.

Table 5: Comparison of our proposed Gaussian elimination algorithm and the one from Bernstein et al. [BCS13], for a matrix with r rows and c columns.

Algorithm	#loops	#XOR	#masks	Output form	Input matrix requirement
[BCS13]	$\mathsf{r}(\mathsf{r} + (\mathsf{r} - 1)/2)$	$\mathsf{r}(\mathsf{r} + (\mathsf{r} - 1)/2)$	$\mathsf{r}(\mathsf{r} + (\mathsf{r} - 1)/2)$	systematic	systematic
This work [ref]	rc	rc	$3\mathsf{rc}$	row echelon form	any rank
This work [rref]	$\mathsf{rc} + \mathsf{r}^2$	$\mathsf{rc} + \mathsf{r}^2$	$3\mathsf{rc} + 2\mathsf{r}^2$	reduced row echelon form	any rank

Table 6: Clock cycles comparison of our proposed Gaussian elimination algorithm and the one from Bernstein et al. [BCS13], for a matrix with $\mathsf{r} = 10, 20, 30, 100$ rows and $\mathsf{c} = 67$ columns.

Algorithm	10 rows	20 rows	30 rows	100 rows
[BCS13]	2,241.31	9,547.64	21,030.64	230,444.41
This work [ref]	33,358.83	61,325.96	90,379.98	296,578.26
This work [rref]	35,669.20	74,649.81	117,940.18	590,984.32

The pseudocode of the two algorithms can be found in alg. 1 ([BCS13]), alg. 2, and alg. 3, 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 `m[i]` of the binary matrix in a variable of type `__uint128_t`. We can perform Steps 3-4 of alg. 1 in a constant number of operations as follows:

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

Similarly, also the other *if* statements of both algorithms can be easily executed in constant time.

Algorithm 1: to_syst(M)

```
input  : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$
output : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$ in systematic form
for $j = 0, \dots, \min(\mathsf{r}, \mathsf{c}) - 1$ do
    /* Fix 1 in diagonal */
    for $i = j + 1, \dots, \mathsf{r} - 1$ do
        if $m_{j,j} \oplus m_{i,j} = 1$ then
            $\mathsf{mask} = 1$
        $m_j = m_j \oplus \mathsf{mask} \cdot m_i$
    if $m_{j,j} = 0$ then
        stop
    /* Fix 0s in col $j$ */
    for $i = 0, \dots, \mathsf{r} - 1$ do
        if $i \neq j$ then
            if $m_{i,j} = 1$ then
                $\mathsf{mask} = 1$
            $m_i = m_i \oplus \mathsf{mask} \cdot m_j$
return $M$
```

Algorithm 2: to_ref(M)

```
input  : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$
output : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$ in row echelon form
$\tilde{r} = 0$
for $j = 0, \dots, \mathsf{c} - 1$ do
    for $i = 0, \dots, \mathsf{r} - 1$ do
        if $m_{\tilde{r},j} = 0$ then
            $\mathsf{mask1} = 1$
        if $m_{i,j} = 1$ then
            $\mathsf{mask2} = 1$
        if $m_{\tilde{r},j} = 1$ then
            $\mathsf{mask3} = 1$
        $m_{\tilde{r}} = m_{\tilde{r}} \oplus \mathsf{mask1} \cdot \mathsf{mask2} \cdot \mathsf{mask3} \cdot m_i$,
        $m_i = \mathsf{mask1} \cdot \mathsf{mask3} \cdot m_{\tilde{r}} \oplus m_i$
    if $m_{\tilde{r},j} = 1$ and $\tilde{r} < \mathsf{r}$ then
        $\tilde{r} = \tilde{r} + 1$
return $M, \tilde{r}$
```

Algorithm 3: to_rref(M)

```
input  : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$
output : $M \in \mathcal{M}_{\mathsf{r},\mathsf{c}}(\mathbb{F}_2)$ in reduced row echelon form
$\tilde{r} = 0,\ R = \{0, \dots, 0\},\ C = \{0, \dots, 0\}$
for $j = 0, \dots, \mathsf{c} - 1$ do
    for $i = 0, \dots, \mathsf{r} - 1$ do
        if $m_{\tilde{r},j} = 0$ then
            $\mathsf{mask1} = 1$
        if $m_{i,j} = 1$ then
            $\mathsf{mask2} = 1$
        if $m_{\tilde{r},j} = 1$ then
            $\mathsf{mask3} = 1$
        $m_{\tilde{r}} = m_{\tilde{r}} \oplus \mathsf{mask1} \cdot \mathsf{mask2} \cdot \mathsf{mask3} \cdot m_i$
        $m_i = \mathsf{mask1} \cdot \mathsf{mask3} \cdot m_{\tilde{r}} \oplus m_i$
    $C_{\tilde{r}} = j \cdot \mathsf{mask3}$
    $R_{\tilde{r}} = \tilde{r}$
    if $m_{\tilde{r},j} = 1$ and $\tilde{r} < \mathsf{r}$ then
        $\tilde{r} = \tilde{r} + 1$
    for $j = 0, \dots, \mathsf{r} - 1$ do
        for $i = 0, \dots, \mathsf{r} - 1$ do
            if $m_{i,C_j} = 1$ then
                $\mathsf{mask1} = 1$
            if $i < R_j$ then
                $\mathsf{mask2} = 1$
            $m_i = m_i \oplus \mathsf{mask1} \cdot \mathsf{mask2} \cdot m_{R_j}$
return $M, \tilde{r}$
```

4.2.2 Zassenhaus algorithm

The Zassenhaus algorithm is a method to compute a basis for the intersection and sum of two vector subspaces U, V of a vector space W of length m. 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 $(l_1 + l_2) \times 2m$.

$$\begin{bmatrix} u_{0,0} & \cdots & u_{0,m-1} & u_{0} & \cdots & u_{0,m-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ u_{l_1,0} & \cdots & u_{l_1,m-1} & u_{l_1,0} & \cdots & u_{l_1,m-1} \\ v_{0,0} & \cdots & v_{0,m-1} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ v_{l_2,0} & \cdots & v_{l_1,m-1} & 0 & \cdots & 0 \end{bmatrix} \quad (1) \qquad \begin{bmatrix} a_{0,0} & \cdots & a_{0,m-1} & \star & \cdots & \star \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{l_3,0} & \cdots & a_{l_3,m-1} & \star & \cdots & \star \\ 0 & \cdots & 0 & b_{0,0} & \cdots & b_{0,m-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & b_{l_4,0} & \cdots & b_{l_4,m-1} \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix} \quad (2)$$

After application of the Gauss elimination, the matrix has the form (2), reduced in row echelon form. 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 alg. 4.

Algorithm 4: $\mathsf{zassenhaus}(U, V)$: 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 : $A = (a_0, \ldots, a_{l_3})^T \in (\mathbb{F}_{2^m})^{l_3}, B = (b_0, \ldots, b_{l_4})^T \in (\mathbb{F}_{2^m})^{l_4}$
$\begin{bmatrix} A & \star \\ \mathbf{0} & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathsf{gauss}\left(\begin{bmatrix} U & U \\ V & \mathbf{0} \end{bmatrix}\right)$
return $A, B$
```

4.2.3 Generation of vectors of given rank

The generation of a vector $\mathbf{e} \in \mathbb{F}_{q^m}^n$ of a given rank, say r is probably the most delicate part of the key generation and encapsulation routines. We are not aware of any constant time algorithm performing this task. In this section, we analyze the two non-constant time strategies adopted in the current NIST submission (dated 2020/04/21) and in [AAAAB$^+$18, Section 5.2]. Then we derive a constant time version of the latter, with a probability of failure that can be set as small as desired, at the cost of increasing the complexity of the algorithm. Lastly, we also propose an alternative method that, while generating a vector of a given rank, also construct the full support of the vector. This last method could turn to be useful in the case a user could store a larger public key in memory, so to have the advantage of not reconstructing the support from its basis during the encapsulation phase.

The strategy from [AAAAB$^+$18] or from the NIST submission are based on the same idea: generating a basis of r random elements of $\mathbb{F}_{q^m}$ until they are linearly independent and then generate a random linear combination of those vectors. In the NIST submission, the r elements are randomly inserted in the error components, thus guaranteeing that the error will have rank r. The remaining $n - r$ positions are filled with random linear combinations of the basis elements. This algorithm is detailed in alg. 5. On the other hand, in [AAAAB$^+$18], the components of the error are all filled with random linear combinations of the basis elements, until the error has rank r. This algorithm is detailed in alg. 6. It is clear that both strategies are non-constant time. Notice also that, in [DGK20], the authors describe how the NIST submission implementation leaks the memory access pattern.

Both approaches can be turned to be constant time by removing the repeat and while loops, and iterating the algorithm enough times so that the probability of generating a vector list of the wrong rank becomes negligible. Our proposed constant time solution is based on this idea. Precisely, we first sample r independed elements of $\mathbb{F}_{q^m}$ randomly. In proposition 1, we derive the probability for those vectors to be linearly independent over $\mathbb{F}_2$. Second, we generate the components of the vector using masked linear combinations of the basis. Note that this algorithm also has the advantage to hide the memory access pattern. We show also that it is sufficient to repeat the full procedure one to reach a probability of failing equal to 2^{-60}, which is already way smaller than the ROLLO Decryption Failure Rate. However, if this is still a concern (for example when adapting this work to ROLLO-II), repeating the procedure twice leads to a probability of failing of

2^{-120}, and so on. The full algorithm is described in alg. 7.

Proposition 1. *The probability that r randomly sampled elements in $\mathbb{F}_2^m$ have rank r is $p = (1 - q^{-m}) \cdot (1 - q^{1-m}) \cdots (1 - q^{r-1-m})$.*

Proof. The first element $e_1 \in \mathbb{F}_{q^m}$ is independent if and only if it is different from zero. Since it is a vector in $\mathbb{F}_{q^m}$, we have $\Pr(e_1 = 0) = 1/q^m$. Then e_1, e_2 are linearly dependent if and only if $e_2 = ke_1$, where $k \in \mathbb{F}_q$. So $\Pr(e_2 = ke_1) = q/q^m$. We can continue this way for all the vectors until the last one, where we have that e_r is a linear combination of the previous ones if and only if $e_r = \sum_{i=1}^{r-1} k_i e_i$ where $k_i \in \mathbb{F}_q$. So $\Pr(e_r = \sum_{i=1}^{r-1} k_i e_i) = q^{r-1}/q^m$. □

For ROLLO-I-128 parameters, where $r = 7$ and $m = 67$, the probability to have a linear combination between r random vectors is 2^{-60}. Computing the probability that a random support is of the required dimension is only the first step of the evaluation of the failure probability of our algorithm. Assuming that a random support F of dimension r is available, we now have to compute the probability for a vector $\mathbf{e} \in F^n$ to be of rank strictly less than r. Let $f_1, \dots, f_r$ be a basis of F. The components $e_1, \dots, e_n$ can be written with coordinates in $f_1, \dots, f_r$: $e_i = \sum_{j=0}^{r} (e_i)_j f_j$ where $(e_i)_j \in F_q$. Let M be the $r \times n$ matrix over F_q such that $M_{j,i} = (e_i)_j$. Then, the fact that $\mathsf{w}_\mathsf{R}(\mathbf{e}) < r$ is equivalent to the fact that the matrix M is of rank $< r$. Since the coordinates of M are sampled randomly, this probability can be approximated by $q^{-(1+n-r)}$.

For ROLLO-I-128 parameters, where $r = 7$ and $n = 83$, the probability to obtain a vector with rank less than r is 2^{-77}, hence the probability that this process generates an error of weight $r - 1$ is $2^{-60} + 2^{-77}$ which can be approximated by 2^{-60}.

Now, one might generate multiple samples and if the cycle is repeated h times, the probability to fail becomes 2^{-120} for $h = 2$, and 2^{-180} for $h = 3$ and so on. To make this approach constant time, one can repeat the sampling as many times needed to reach the desired probability, each time computing the rank of the vector with the constant time Gaussian elimination algorithm proposed in section 4.2.1 and store the sampled vector space when it has the desired rank.

Algorithm 5: Given-rank vector generation: NIST code 20200421

```
input  : $r \in \mathbb{N}^\star$
output : $\mathbf{e} \in \mathbb{F}_{q^m}^n$ such that Rank$(e) \le r$
repeat
    $(b_1, \dots, b_r) \xleftarrow{\$} (\mathbb{F}_{2^m})^r$
until Rank$(b_1, \dots, b_r) = r$;
$J = \{j_1, \dots, j_r\} \xleftarrow{\$} \{1, \dots, n\}^r$
for $j \in J$ do
    $\mathbf{e}_j = b_i$
for $j \in \{1, \dots, n\} \setminus J$ do
    $(c_1, \dots, c_r) \xleftarrow{\$} \{0,1\}^r$
    $\mathbf{e}_j = \sum_{i=1}^{r} c_i b_i$
return e
```

Algorithm 6: Given-rank vector generation: [AAAAB+18]

```
input  : $r \in \mathbb{N}^\star$
output : $\mathbf{e} \in \mathbb{F}_{q^m}^n$ such that Rank$(e) \le r$
repeat
    $(b_1, \dots, b_r) \xleftarrow{\$} (\mathbb{F}_{2^m})^r$
until Rank$(b_1, \dots, b_r) = r$;
while Rank$(e) \neq r$ do
    for $i = 1, \dots, n$ do
        $(c_1, \dots, c_r) \xleftarrow{\$} \{0,1\}^r$
        $\mathbf{e}_j = \sum_{i=1}^{r} c_i b_i$
return e
```

Algorithm 7: Given-rank vector generation: rank_vec_gen(r)

```
input  : $r \in \mathbb{N}^\star$, $p \in [0,1]$
output : $\mathbf{e} \in \mathbb{F}_{q^m}^n$ such that Rank$(e) \le r$
for $j = 1, \dots, \lceil \lambda/(-\log_2 p) \rceil$ do
    $(b_1, \dots, b_r) \xleftarrow{\$} (\mathbb{F}_{2^m})^r$
    for $i = 1, \dots, n$ do
        $(c_1, \dots, c_r) \xleftarrow{\$} \{0,1\}^r$
        $\mathbf{e}'_j = \sum_{i=1}^{r} c_i b_i$
    if Rank$(\mathbf{e}') == r$ then
        $\mathbf{e} = \mathbf{e}'$
return e
```

Now we describe how to generate the entire support of the vector rather than just the basis. This approach takes advantage of the fact that r is usually small (maximum 9 for ROLLO-I). 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 vector subspace F of $\mathbb{F}_{q^m}$ of dimension

r. One can 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 vector subspace of F. We therefore have to check the rank of $\mathbf{e}$ before outputting the result, or run the algorithm twice to reach a probability of failing of 2^{-120} (as proved above). We also notice that an implementation of such method needs to take care of hiding the memory access pattern when randomly drawing the elements from the vector space. The method is detailed in its non-constant time version in alg. 8, and in its constant time version in alg. 9. Note that the mask operation in line 9 of alg. 9 should be done using a AND mask rather than a multiplication.

Algorithm 8: Given-rank vector and support generation

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

Algorithm 9: constant time given-rank vector and support generation

```
   input   : $r \in \mathbb{N}^\star$, $p \in [0, 1]$
   output  : $\mathbf{e} \in \mathbb{F}_{q^m}^n$ such that $\mathsf{Rank}(e) \leq r$
1  for $j = 1, \dots, \lceil \lambda/(-\log_2 p) \rceil$ do
2      $v_0 = 0$
3      for $i = 0, \dots, r-1$ do
4          $v_{2^i} \leftarrow_\$ \mathbb{F}_2^m$
5          for $j = 1, \dots, 2^i - 1$ do
6              $v_{j+2^i} = v_j + v_{2^i}$
7      for $i = 1, \dots, n$ do
8          $\mathbf{e}'_j \leftarrow_\$ \{v_0, \dots, v_{2^r-1}\}$
9      if $\mathsf{Rank}(\mathbf{e}') == r$ then
10         $\mathbf{e} = \mathbf{e}'$
11 return $\mathbf{e}, \mathsf{supp}(e) = \{v_0, \dots, v_{2^r-1}\}$
```

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.

4.3.1 Matrix multiplication with lazy reduction

The multiplication $\mathbf{a} \times \mathbf{b}$ [4] in $\mathbb{F}_{(2^m)^n}$, alg. 10, is performed as the following vector by matrix multiplication

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

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-128 we have $n = 83$, so $(b_{i,0} + b_{i,1}x + \ldots + b_{i,82}x^{82}) \cdot x \mod P(x) = b_{i,82} + b_{i,0}x + (b_{i,1} + b_{i,82})x^2 + b_{i,2}x^3 + (b_{i,3} + b_{i,82})x^4 + b_{i,4}x^5 + b_{i,5}x^6 + (b_{i,6} + b_{i,82})x^7 + \ldots, b_{i,81}x^{82}$, since $x^{82} = X^7 + X^4 + X^2 + 1$.

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

[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 $\mathbf{ab}$, also for matrix multiplications.

reduction $\mathsf{red}_{\mathbb{F}_{2^{67}}}$ is applied only at the end of the summation. The pseudo-code of the algorithm is presented in alg. 10.

Algorithm 10: $\mathsf{poly\_mul}(\mathbf{a}, \mathbf{b})$: 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 : $\mathbf{c} = (\mathbf{a} \times \mathbf{b} \mod P(x)) \in \mathbb{F}_{2^m}[x]/P(x)$
$\mathbf{c} = 0$
for $j = 0, \dots, n-2$ do
    for $i = 0, \dots, n-1$ do
        $t = \mathsf{clmul}(a_j, b_i)$
        $c_i = c_i \oplus t$
    $\mathbf{b} = \mathbf{b} \cdot x \mod P(x)$
for $i = 0, \dots, n-1$ do
    $t = \mathsf{clmul}(a_{n-1}, b_i)$
    $c_i = c_i \oplus t$
    $c_i = \mathsf{red}_{\mathbb{F}_{2^m}}(c_i)$
return $\mathbf{c}$
```

The AVX2 performances of the polynomial multiplication are reported in table 7.

4.3.2 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 [GP98] in 1998, which improves the Itoh-Tsujii algorithm with pre-computed powers [IT88]. 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 $a_i \in \mathbb{F}_{q^m} \forall i = 0, \dots, n-1$. It is then sufficient to pre-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:

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

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 $\mathbf{a}^{2^{km}} = S^k \cdot \mathbf{a}^T$ 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 12 summarizes how the inversion is performed. It uses alg. 11 to compute $\mathbf{a}^{2^{km}}$. The matrix S in alg. 11 is a pre-computed matrix depending only on P and n.

Notice that both alg. 11 and alg. 12 can be coded such that they execute a constant number of operations. In particular, Steps 4-5 of alg. 11, can be performed in a constant time fashion by using a mask, as follows: compute `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 $t = a_j \otimes$ `mask` and finally $b_i = b_i + t$.

Algorithm 11: 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 12: 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 : $\mathbf{b} = \mathbf{a}^{-1}$
c = poly_pow$_1(\mathbf{a})$
for $i$=2, ..., n-1 do
    $\mathbf{t_1}$ = poly_pow$_i(\mathbf{a})$
    c = poly_mul$(\mathbf{t_1}, \mathbf{c})$
s = poly_mul$(\mathbf{a}, \mathbf{c})$ ;              // $\mathbf{a}^r = \mathbf{a} \cdot \mathbf{a}^{r-1}$
$s_0 = s_0^{-1}$ ;                                     // $s_0 \in \mathbb{F}_{2^m}$
for $i$=0, ..., n-1 do
    $b_i = s_0 \cdot c_i$
return b
```

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 alg. 11. This, for example, results in 70.6 KB of pre-computed matrices for ROLLO-I-128, and a speed improvement of about 17%.

As an alternative method to compute the inverse of a polynomial, one might consider a constant time variant of Euclid's algorithm, as the one proposed in [BY19]. Though, this type of algorithm is usually more efficient for generic moduli, where the modular reductions in Fermat's method is considerably more expensive. After a comparison in the favor of a Sagemath [S$^+$20] implementation of the method described above against the script `recipx` provided in [BY19], we decided to discard this option.

The AVX2 performances of the reduction are reported in table 7.

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 [GMRZ13] and made constant time in [AGH$^+$19].

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, i.e. $EF = \langle\{ef \mid e \in E \text{ and } f \in F\}\rangle$. 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 [AMAB$^+$18]. 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 $\mathbf{s} = (s_1, \dots, s_n)$ be a syndrome of the error vector $\mathbf{e} = (e_1, \dots, e_{2n})$, 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 (alg. 13) 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 goal of this algorithm is to recover the vector space E (see [AGH$^+$19] for more details).

Let us explain how the algorithm recovers 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 $E = B_1 \cap B_2 \cap \ldots \cap B_d$, where $B_i = f_i^{-1}S$. In fact, $B_i = \{f_i^{-1}f_1e_1, f_i^{-1}f_2e_1, \dots, f_i^{-1}f_de_r\} = \{e_1, \dots e_r, f_i^{-1}f_je_t\}_{1\leq j\leq d, i\neq j, 1\leq t\leq r}$. 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 [AMAB$^+$18].

In alg. 13, we use capital letter both for the output of Zassenhaus algorithm (section 4.2 p. 11) 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 also 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, $B_i \cap B_j$. With T we indicate a temporary value. The i-th element of T is denoted by t_i

There are three conditions that need to be fulfilled for this algorithm to run in constant time: (1) the size of the inputs to the Zassenhaus algorithm have to be constant. Here we always input a basis of length rd for both vector spaces; (2) for inputs of the same size, the Zassenhaus algorithm needs to run in constant time. This was taken care of in section 4.2; (3) operations involving elements of $\mathbb{F}_{q^m}$ (addition, multiplication, etc.) need to run in constant time. This was taken care in section 4.1.

Notice that Step 3 of the algorithm would work if, instead of the reduced row echelon form of the basis, one computes the entire vector space E and then sorts it with respect to any order. For this particular choice of parameters, this second option is slower. It could become more efficient for a much larger m and a smaller base.

Algorithm 13: 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 : $E = \langle e_1, \dots, e_r \rangle$
// Recover the vector space $E$.
// Step 1: compute a basis of $S$ of length $rd$
$S \leftarrow \mathsf{gauss}(s)$
// Step 2: compute a basis of $E$ as $\bigcap_{i=0}^{d-1} f_i^{-1} S$
for $j = 0, \dots, rd - 1$ do
    $t_j = f_0^{-1} \cdot S_j$
for $i = 1, \dots, d - 1$ do
    for $j = 0, \dots, rd - 1$ do
        $d_{i,j} = f_i^{-1} \cdot S_j$
    $\_, T \leftarrow \mathsf{zassenhaus}(T, D_i)$
// Step 3: return reduced row echelon form of the basis of $E$
$\langle e_1, \dots, e_r \rangle = \mathsf{to\_rref}(T)$
return $\langle e_1, \dots, e_r \rangle$
```

5 Performance

We benchmark our implementation of ROLLO-I-128 on a MacBook Pro equipped with 2.6GHz Intel Core i7-8850H. To measure the performance of the single operations presented in this work, we use our own testing platform, and the results are reported in table 7.

Table 7: Average cycles per operation for the main algorithms presented in this work. Measurements have been taken enabling AVX2 instructions and averaging over 4 seconds of execution on a MacBook Pro 2017 with a 2.9 GHz Quad-Core Intel Core i7.

Algorithm	clmul(a, b)	clsq(a)	red$_{\mathbb{F}_{2^{67}}}(a)$	inv$_{\mathbb{F}_{2^{67}}}(a)$	poly_mul(a, b)	poly_inv(a)
Clock cycles	6.36	5.29	15.26	1,656.30	79,288.56	11,472,218.86
Algorithm	RSR	Keygen	Encaps	Decaps		
Clock cycles	9,513,722.01	12,729,075.41	1,385,871.94	9,981,462.15		

We use SUPERCOP version 20200618 [BL10] to compare our implementation with other existing KEMs by disabling Intel Hyper-Threading and Turbo Boost. 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 SHA256, but for a fair comparison, we have switched to OpenSSL's SHA256 implementation, which is also used in the implementation of ROLLO-I. 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`.

According to our profiler: about 85% of the key generation is taken by the polynomial inversion; 5% of the encapsulation time is occupied by the polynomial multiplication, while 91% of the time is spent in generating a basis and two polynomials whose list of coefficients has given rank r. About 70% of this last step (63% of the full encapsulation time) is taken by computing the rank of the list, to make sure it has the proper rank, while about 15% is taken from the `randombyte()` calls. about 75% of the decapsulation is taken by the Gaussian elimination step in the Zassenhaus algorithm. In the official ROLLO specification [AMAB+18], the following number (in thousands) of clock cycles are reported for, respectively, key generation, encapsulation and decapsulation: $3537, 395, 1754$. Our loss in the key generation is explained by the fact that ROLLO's team used a not constant-time GCD algorithm for the polynomial inversion. Our loss in the encapsulation is explained by the fact that ROLLO's team used a not constant-time generation of vectors with given rank, and in particular they did not have to check the rank of $\mathbf{e}_1$ and $\mathbf{e}_2$ two times. The not constant-time implementation of Gaussian elimination also explains the difference in the decapsulation step.

Table 8 reports the performance results of our implementation of ROLLO-I-128 with one cycle in the generation of vectors of given rank (`CT_rollo_fast`), and with two cycles (`CT_rollo_secure`). We also report the performances of the other Category 1 KEMs available in SUPERCOP.

Table 8: The number of cycles to perform key generation, encapsulation, and decapsulation of other KEMs available in SUPERCOP with 128-bit security .

Algorithm	Key Generation	Encapsulation	Decapsulation
`CT_rollo_secure/avx2`	**11,034,623**	**984,432**	**9,775,241**
`CT_rollo_fast/avx2`	**11,204,649**	**320,835**	**9,744,693**
`bikel1/avx2`	800,814	137,295	2,227,101
`frodokem640/optimized`	1,254,121	1,972,512	2,050,790
`frodokem640aes/optimized`	1,872,924	2,301,509	2,291,485
`frodokem640shake/x64`	4,552,208	4,924,284	4,880,325
`hqc128/avx`	895,079	1,002,802	1,406,262
`hqcrmrs128/avx`	777,538	904,170	1,241,698
`kyber512/avx2`	31,812	52,151	40,953
`kyber90s512/avx2`	19,355	28,815	22,685
`ledakem1264/portableopt`	3,967,977	253,374	2,731,123
`ledakem12sl/portableopt`	6,814,222	312,572	2,610,232
`ledakem1364/portableopt`	3,486,752	268,002	2,015,586
`ledakem13sl/portableopt`	5,858,734	345,372	2,771,222
`ledakem1464/portableopt`	2,621,831	247,425	2,217,174
`ledakem14sl/portableopt`	5,209,246	377,816	2,841,172
`ledakemcpa12/portableopt`	1,052,889	137,843	843,255
`ledakemcpa13/portableopt`	828,052	107,064	783,340
`ledakemcpa14/portableopt`	711,261	107,995	927,117
`lightsaber2/avx2`	56,314	75,549	72,500
`lotus128/avx2`	10,697,399	136,846	193,228
`mceliece348864/vec`	348,055,578	93,702	613,623
`mceliece348864f/vec`	275,194,978	80,356	558,679
`newhope512cca/ref`	126,527	196,300	224,699
`ntruhrss701/ref`	15,772,963	836,280	2,492,160
`ntskem1264/avx2`	46,274,216	102,050	300,127
`rolloi128/avx` (not CT)	1,151,479	158,417	1,198,809
`rolloii128/avx` (not CT)	4,385,668	611,519	2,010,480
`rqc128/avx` (not CT)	1,277,266	1,531,786	4,582,519
`sikep503/opt`	87,103,004	141,395,561	151,837,798
`threebears624r2cca/vec`	54,681	77,370	137,336
`threebears624r2ccax/vec`	57,809	75,771	102,596
`threebears624r2cpa/vec`	54,033	78,162	33,299
`threebears624r2cpax/vec`	54,846	77,376	16,092
`titaniumccastd/avx2`	1,747,766	1,695,331	1,945,030

6 Conclusion

In this work, we have presented several algorithms which shed some light on the potential performance of a fully optimized constant time implementation of ROLLO-I-128. 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

[AAAAB+18] Ameera Salem Al Abdouli, Mohamed Al Ali, Emanuele Bellini, Florian Caullery, Alexandros Hasikos, Marc Manzano, and Victor Mateu. DRANKULA: A McEliece-like Rank Metric based Cryptosystem Implementation. In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018)*, volume 2, pages 230–241, 01 2018.

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the nist post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology, 2020.

[ABC+19] A 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 on ASIA Public-Key Cryptography Workshop*, pages 23–30, 2019.

[AGH+19] Nicolas Aragon, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, and Gilles Zémor. Low rank parity check codes: New decoding algorithms and applications to cryptography. *arXiv preprint arXiv:1904.00357*, 2019.

[AMAB+17] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Rank Quasi-Cyclic (RQC), 2017.

[AMAB+18] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, et al. ROLLO - Rank-Ouroboros, LAKE & LOCKER, 2018. Available at: https://pqc-rollo.org/doc/rollo-specification_2020-04-21.pdf.

[AMBC+] Carlos Aguilar-Melchor, Emanuele Bellini, Florian Caullery, Rusydi H Makarim, Marc Manzano, Chiara Marcolla, and Victor Mateu. Constant-time algorithms for ROLLO. Available at: https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/caullery-constant-time-rollo.pdf.

[ASBB+19] Hamad Al Shehhi, Emanuele Bellini, Filipe Borba, Florian Caullery, Marc Manzano, and Victor Mateu. An IND-CCA-secure code-based encryption scheme using rank metric. In *Progress in Cryptology–AFRICACRYPT 2019: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings*, volume 11627, page 79. Springer, 2019.

[BBC+20] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier Verbel. Algebraic attacks for solving the rank decoding and minrank problems without gr\" obner basis. *arXiv preprint arXiv:2002.08322*, 2020.

[BCG+19] Emanuele Bellini, Florian Caullery, Philippe Gaborit, Marc Manzano, and Victor Mateu. Improved veron identification and signature schemes in the rank metric. In *Information Theory (ISIT), 2019 IEEE International Symposium on*. IEEE, 2019.

[BCH+18] Emanuele Bellini, Florian Caullery, Alexandros Hasikos, Marcos Manzano, and Victor Mateu. Code-based signature schemes from identification protocols in the rank metric. In *International Conference on Cryptology and Network Security*, pages 277–298. Springer, 2018.

[BCM+19] Emanuele Bellini, Florian Caullery, Rusydi Makarim, Marc Manzano, Chiara Marcolla, and Victor Mateu. Advances and challenges of rank metric cryptography implementations. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 325–328. IEEE, 2019.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: fast constant-time code-based cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 250–272. Springer, 2013.

[BL10] Daniel J Bernstein and Tanja Lange. ebacs: Ecrypt benchmarking of cryptographic systems: Supercop, 2010. Last access 15/07/2020. Available at: https://bench.cr.yp.to/supercop.html.

[BY19] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 340–398, 2019.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. Constant-time implementations in some proposed kems: the case of rollo and rqc. http://math.haifa.ac.il/shay/Side_Channels_2020_06_23_V01.pdf, 2020.

[EA] Sean Eron Anderson. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html. Accessed: 2019-05-30.

[FL05] Cédric Faure and Pierre Loidreau. A new public-key cryptosystem based on the problem of reconstructing p–polynomials. In *International Workshop on Coding and Cryptography*, volume 3969, pages 304–315. Springer, 2005.

[GK10] Shay Gueron and Michael E. Kounavis. Intel® carry-less multiplication instruction and its usage for computing the GCM mode. *White Paper*, 2010.

[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, Bergen, Norway*, 04 2013.

[GOK18] Philippe Gaborit, Ayoub Otmani, and Hervé Talé Kalachi. Polynomial-time key recovery attack on the Faure–Loidreau scheme based on Gabidulin codes. *Designs, Codes and Cryptography*, 86(7):1391–1403, 2018.

[GP98] Jorge Guajardo and Christof Paar. Fast inversion in composite galois fields GF $((2^n)^M)$. In *IEEE international symposium on information theory*, pages 295–295. Citeseer, 1998.

[GPT91] Ernst M. Gabidulin, A.V. Paramonov, and O.V. Tretjakov. Ideals over a non-commutative ring and their application in cryptology. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 482–489. Springer, 1991.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.

[Int] Intel® C++ Compiler 19.1 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: 2020-01-01.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF (2m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[KO62] A Karatsuba and Yu Ofman. 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.

[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.

[Loi17] Pierre Loidreau. A new rank metric codes based encryption scheme. In *International Workshop on Post-Quantum Cryptography*, pages 3–17. Springer, 2017.

[NIS18] NIST. Post-Quantum Cryptography Call for Proposals. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals, 2018. Accessed: 2020-01-01.

[Ove05] Raphael Overbeck. A new structural attack for GPT and variants. In *International Conference on Cryptology in Malaysia*, pages 50–63. Springer, 2005.

[Ove08] Raphael Overbeck. Structural attacks for public key cryptosystems based on Gabidulin codes. *Journal of cryptology*, 21(2):280–301, 2008.

[PCJM18] Stjepan Picek, Carlos A. Coello Coello, Domagoj Jakobovic, and Nele Mentens. Finding short and implementation-friendly addition chains with evolutionary algorithms. *Journal of Heuristics*, 24(3):457–481, 2018.

[S+20] W. A. Stein et al. *Sage Mathematics Software (Version 9.0)*. The Sage Development Team, 2020. http://www.sagemath.org.

[Vic19] Victor Shoup. NTL: A Library for doing Number Theory. https://shoup.net/ntl/, 2019. Accessed: 2020-01-01.

A Pseudocode for the binary field arithmetic

The plain C carryless multiplication algorithm $\mathsf{clmul}_K(a,b)$ is described in alg. 14. Notice that alg. 14 works for $64 < m < 129$.

Algorithm 14: $\mathsf{clmul}_K(a,b)$: carryless multiplication in $\mathbb{F}_{2^m}$ (Karatsuba, plain C).

```
input  : a,b of type __uint128, represent two binary polynomials of degree m − 1.
output : c of type __uint128[2], represents a binary polynomial of degree 2m − 2.
__uint128 aLbL, aHbH, tmp;
aLbL = ntlclmul64(b & 0xFFFFFFFFFFFFFFFF, a & 0xFFFFFFFFFFFFFFFF);
aLbL = ntlclmul64((b ≫ 64) & 0xFFFFFFFFFFFFFFFF, (a ≫ 64) & 0xFFFFFFFFFFFFFFFF);
tmp = ntlclmul64(((b ≫ 64) & 0xFFFFFFFFFFFFFFFF) ⊕ (b & 0xFFFFFFFFFFFFFFFF), ((a ≫
  64) & 0xFFFFFFFFFFFFFFFF) ⊕ (a & 0xFFFFFFFFFFFFFFFF);
tmp = tmp ⊕aLbL ⊕aHbH;
c[0] = aLbL ⊕ (tmp ≪ 64);
c[1] = aHbH ⊕ (tmp ≫ 64);
return c
```

The AVX2 carryless multiplication algorithm $\mathsf{clmul}_S(a,b)$ is described in alg. 15. Note that, as alg. 14, alg. 15 is suitable for fields $\mathbb{F}_{2^m}$ with $64 < m < 129$, which include all ROLLO-I and ROLLO-II variants. Let us recall that using Karatsuba multiplication [KO62] in alg. 15 instead of steps 3-6 would not give any advantage, as the cost of multiplication and addition with AVX2 instruction is very close. In practice, as we will show, it even performs worse, due to alignment problems.

Algorithm 15: $\mathsf{clmul}_S(a,b)$: carryless multiplication in $\mathbb{F}_{2^m}$ (schoolbook, AVX2).

```
   input  : a,b of type __m128i, represent two binary polynomials of degree m − 1.
   output : c of type __m256i, represents a binary polynomial of degree 2m − 2.
 1 __m128i aLbL, aLbH, aHbL, aHbH, aLbH_xor_aHbL;
 2 __m256i aLbH_xor_aHbL256, aHbHaLbL;
 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 aLbH_xor_aHbL256 = _mm256_set_m128i(zero, aLbH_xor_aHbL);
10 aLbH_xor_aHbL256 = _mm256_permute4x64_epi64(aLbH_xor_aHbL256, 0xD2);
11 aHbHaLbL = _mm256_set_m128i(aHbH, aLbL);
12 c = _mm256_xor_si256(aLbH_xor_aHbL256, aHbHaLbL);
13 return c
```

The algorithm to inverleave zeros used for the squaring algorithm is a small modification of the method *Interleave bits with 64-bit multiply* given by Sean Eron Anderson on his web page *Bit Twiddling Hacks* [EA] which is given in alg. 16.

Algorithm 16: $\mathsf{interleave\_zeros}(a)$: interleave zeros after each bit of a byte.

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

The squaring method is given in alg. 17. For the AVX2 version, a look-up table based on the instruction `_mm_shuffle_epi8` is implemented both in the submission and our work.

Algorithm 17: clsqr(a): vectorized carryless squaring in $\mathbb{F}_{2^m}$.

```
input   : a of type __uint128, represents a binary polynomial of degree $m-1$.
output  : c of type __uint128[2], represents $a^2$ in a binary polynomial of degree $2m-2$.
__uint128 high = 0, low = 0;
for $i = 0 \ldots 7$ do
    low ⊕= (((__uint128) interleave_zeros((a ≫ (8 * i)) & 0xFF)) ≪ (16 * i));
for $i = 8 \ldots 15$ do
    high ⊕= (((__uint128) interleave_zeros((a ≫ (8 * i)) & 0xFF)) ≪ (16 * (i - 8)));
c[0] = low;
c[1] = high;
return c
```

The algorithm for reduction is presented in alg. 18, 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 alg. 15, alg. 18 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.

Algorithm 18: $\mathsf{red}_{\mathbb{F}_{2^{67}}}(a)$: reduction in $\mathbb{F}_{2^{67}}$

```
input   : $a = (\alpha_{132}, \ldots, \alpha_0) \in \mathbb{F}_2^{133}$
output  : $c = (a \mod P_0(X) = X^{67} + X^5 + X^2 + X + 1) \in \mathbb{F}_2^{67}$
$a_L = (\alpha_{127}, \ldots, \alpha_0) \in \mathbb{F}_2^{128}$
$a_H = (0, \ldots, 0, \alpha_{132}, \ldots, \alpha_{128}) \in \mathbb{F}_2^{128}$
$a_L = a_L \oplus (a_H \ll 61) \oplus (a_H \ll 62) \oplus (a_H \ll 63) \oplus (a_H \ll 66)$
$a_H = (a_L \otimes \texttt{0xFFFFFFFFFFFFFFF8}) \ll 64$
$a_H = a_H \gg 67$
$a_L = a_L \oplus a_H \oplus (a_H \ll 1) \oplus (a_H \ll 2) \oplus (a_H \ll 5)$
$c = a_L \otimes \texttt{0x7FFFFFFFFFFFFFFFF}$
return $c$
```

The inversion of an element $x \in \mathbb{F}_{2^m}$ is described in alg. 19. This has been derived using Fermat's little Theorem stating that $x^{2^m-2} = x^{-1}$. The fixed exponentiation is achieved by the strategy presented in [PCJM18, Section 6.2] using the following addition chain of length 10:

$$1 \to 2 \to 3 \to 6 \to 12 \to 15 \to 30 \to 33 \to 66 \to 67\,.$$

Algorithm 19: $\mathsf{inv}_{\mathbb{F}_{2^{67}}}(a)$: inversion in $\mathbb{F}_{2^{67}}$

```
input   : $a \in \mathbb{F}_2^{67}$
output  : $c = a^{-1} \in \mathbb{F}_2^{67}$
$r_1 = a^2$ // $a^2$
$r_0 = r_1 \cdot a$ // $a^{2^2-1}$
$r_1 = r_0^2$ // $a^{2^3-2}$
$r_0 = r_1 \cdot a$ // $a^{2^3-1}$
$r_1 = r_0^{2^3}$ // $a^{2^6-2^3}$
$r_0 = r_1 \cdot r_0$ // $a^{2^6-1}$
$r_1 = r_0^{2^6}$ // $a^{2^{12}-2^6}$
$r_0 = r_0 \cdot r_1$ // $a^{2^{12}-1}$
$r_1 = r_0^{2^3}$ // $a^{2^{15}-2^3}$
$r_0 = r_0 \cdot r_1$ // $a^{2^{15}-1}$
$r_1 = r_0^{2^{15}}$ // $a^{2^{30}-2^{15}}$
$r_0 = r_0 \cdot r_1$ // $a^{2^{30}-1}$
$r_1 = r_0^{2^3}$ // $a^{2^{33}-2^3}$
$r_0 = r_0 \cdot r_1$ // $a^{2^{33}-1}$
$r_1 = r_0^{2^{33}}$ // $a^{2^{66}-2^{33}}$
$r_0 = r_0 \cdot r_1$ // $a^{2^{66}-1}$
$c = r_0^2$ // $a^{2^{67}-2}$
return $c$
```