IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 1, pp. 152–174. DOI:10.46586/tches.v2022.i1.152-174

A Constant-time AVX2 Implementation of a Variant of ROLLO

Tung Chou and Jin-Han Liou

Academia Sinica, Taipei, Taiwan
blueprint@citi.sinica.edu.tw, hank93304@citi.sinica.edu.tw

Abstract. This paper introduces a key encapsulation mechanism ROLLO$^+$ and presents a constant-time AVX2 implementation of it. ROLLO$^+$ is a variant of ROLLO-I targeting IND-CPA security. The main difference between ROLLO$^+$ and ROLLO-I is that the decoding algorithm of ROLLO$^+$ is adapted from the decoding algorithm of ROLLO-I. Our implementation of ROLLO$^+$-I-128, one of the level-1 parameter sets of ROLLO$^+$, takes 851823 Skylake cycles for key generation, 30361 Skylake cycles for encapsulation, and 673666 Skylake cycles for decapsulation. Compared to the state-of-the-art implementation of ROLLO-I-128 by Aguilar-Melchor et al., which is claimed to be constant-time but actually is not, our implementation achieves a 12.9x speedup for key generation, a 10.6x speedup for encapsulation, and a 14.5x speedup for decapsulation. Compared to the state-of-the-art implementation of the level-1 parameter set of BIKE by Chen, Chou, and Krausz, our key generation time is 1.4x as slow, but our encapsulation time is 3.8x as fast, and our decapsulation time is 2.4x as fast.

Keywords: NIST PQC standardization · constant-time implementations · code-based cryptography

1 Introduction

ROLLO is a code-based key encapsulation mechanism that was involved up to the second round of the NIST post-quantum cryptography standardization process. It is the merger of three first-round candidates Rank-Ouroboros [AMAB+17], LAKE [ABD+17a] and LOCKER [ABD+17b]. While most of the code-based submissions are based on Hamming metric, ROLLO is based on rank metric and makes use of so-called "low-rank parity-check" (LRPC) codes. The construction of ROLLO, according to the latest version of the specification (version 2020/04/21, available at [ABD+20]), is similar to the third-round code-based candidate BIKE [ABB+20], in the sense that each public key is the quotient of two "low-weight" ring elements. In other words, both schemes are NTRU-like [HPS98].

Unfortunately, the underlying hard problem of ROLLO was not fully studied at the beginning of the second round of the standardization process. During the second round, two algebraic attacks against ROLLO were published [BBC+20, BBB+20]. These attacks show that the proposed parameter sets do not reach the claimed security levels. For example, the parameter set ROLLO-I-128, which was claimed to be of 128-bit (pre-quantum) security, was shown to have only 71-bit security in [BBC+20]. In response to these attacks, the ROLLO team announced in the latest specification new parameter sets that achieved desired security levels under the attacks of [BBC+20, BBB+20]. However, ROLLO still failed to enter the third round.

Even though ROLLO will not be standardized by the current standardization process, ROLLO does have some interesting features. First of all, ROLLO does have very small

Licensed under Creative Commons License CC-BY 4.0.
Received: 2021-07-15 Accepted: 2021-09-15 Published: 2021-11-19

keys. The public keys of the level-1, 3, 5 parameter sets ROLLO-I-128, ROLLO-I-192, and ROLLO-I-256 are 696 bytes, 954 bytes, and 1371 bytes only. The key sizes are 2.2 to 5.3 times as small as the key sizes of the corresponding parameter sets of BIKE. Second, ROLLO does have a CCA2-secure variant ROLLO-II, while BIKE is only claimed to be CPA-secure. Under the assumption that there will not be attacks that outperform [BBC+20, BBB+20], it seems that ROLLO could be a more interesting option for standardization than BIKE. In fact, NIST also commented that

> "Despite the development of algebraic attacks, NIST believes rank-based cryptography should continue to be researched. The rank metric cryptosystems offer a nice alternative to traditional hamming metric codes with comparable bandwidth."

in the status report for the second-round candidates [AASA+20].

Of course, it takes time to see whether the practical security of ROLLO has been thoroughly studied, but [BBC+20, BBB+20] certainly help the community to understand more about the security of ROLLO. On the other hand, there has been very little effort in improving the performance of ROLLO. In particular, how to build a fast constant-time implementation of ROLLO seems to be a problem that has never been well-studied, and the goal of this paper is to show a solution.

1.1 Previous Works

We are aware of only two previous papers about implementing ROLLO. The first paper is [LMB+19], which presents an implementation of the second-round version of ROLLO. The target platform of [LMB+19] is ARM SecureCore SC300. The parameters and the decoding algorithm used in the paper are based on the second-round specification instead of the latest one. The authors did not claim that their implementation is constant-time, and there is no discussion about how to make the implementation constant-time. For these reasons, we think it is not very meaningful to consider the speeds reported in the paper.

The second paper is [AMAB+21], which presents two implementations of ROLLO-I-128 and claims that they are constant-time. The target platform is Intel Coffee Lake. Unfortunately, we found that the algorithm `to_ref` the authors used to reduce matrices to row echelon form is vulnerable to cache-timing attacks.

The algorithm `to_ref` is shown in Algorithm 1 [1]. As one can see, the indices i and j do not depend on the entries of M, so it is fine to access $M_{i,j}$ and M_i directly using memory load/store instructions. However, the value of $\tilde{\mu}$ does depend on the entries of M, so $M_{\tilde{\mu},j}$ and $M_{\tilde{\mu}}$ should not be accessed directly. Of course, it can be that the authors' implementation actually fixed this issue with some extra operations, but we have checked the source code (available at `https://github.com/peacker/constant_time_rollo`) and found that it is not the case. [AMAB+21] uses a similar algorithm `to_rref` to reduce matrices into reduced row echelon form, and `to_rref` has the same problem. We have not checked if other parts of the source code are actually constant-time.

Finally, the source code included in the second-round submission package of ROLLO is not constant-time. This was first pointed out in [DGK20].

1.2 Our Contribution

This paper introduces a new key encapsulation mechanism ROLLO$^+$ and presents a constant-time AVX2 implementation of it. ROLLO$^+$ is a variant of ROLLO-I. As ROLLO-I, ROLLO$^+$ targets only CPA security. The main difference between ROLLO$^+$ and ROLLO-I

[1] [AMAB+21, Algorithm 2] is not consistent with the source code and appears to be erroneous. Algorithm 1 is written based on the source code.

Algorithm 1 `to_ref` [AMAB+21]

Input: $M \in \mathbb{F}_2^{\mu \times \nu}$
Output: $M \in \mathbb{F}_2^{\mu \times \nu}$ in row echelon form

```
1: μ̃ ← 0
2: for j = 0, ..., ν − 1 do
3:     for i = 0, ..., μ − 1 do
4:         mask1 ← 0
5:         mask2 ← 0
6:         mask3 ← 0
7:         if i > r̃ then
8:             mask1 ← 1
9:         end if
10:        if M_{i,j} = 1 then
11:            mask2 ← 1
12:        end if
13:        if M_{μ̃,j} = 0 then
14:            mask3 ← 1
15:        end if
16:        M_μ̃ ← M_μ̃ + mask1 · mask2 · mask3 · M_i
17:        M_i ← M_i + mask1 · mask2 · M_μ̃
18:    end for
19:    if M_{μ̃,j} = 1 and μ̃ < μ then
20:        μ̃ ← μ̃ + 1
21:    end if
22: end for
23: return M
```

is that the decoding algorithm of ROLLO$^+$ is adapted from the decoding algorithm of ROLLO-I. We adapted the decoding algorithm so that some invalid ciphertexts can be identified and rejected.

Our implementation of ROLLO$^+$-I-128, one of the level-1 parameter sets of ROLLO$^+$, takes 851823 Skylake cycles for key generation, 30361 Skylake cycles for encapsulation, and 673666 Skylake cycles for decapsulation. Compared to the state-of-the-art implementation of ROLLO-I-128 (which is not constant-time) by Aguilar-Melchor et al., our implementation achieves a 12.9x speedup for key generation, a 10.6x speedup for encapsulation, and a 14.5x speedup for decapsulation. Compared to the state-of-the-art implementation of the level-1 parameter set of BIKE by Chen, Chou, and Krausz, our key generation time is 1.4x as slow, but our encapsulation time is 3.8x as fast, and our decapsulation time is 2.4x as fast.

As ROLLO$^+$ and ROLLO-I are very similar, we expect that key generation, encapsulation and decapsulation times similar to those of ROLLO-I can be achieved with our implementation techniques (see below). We also expect that our implementation techniques can be used to accelerate ROLLO-II and other rank-metric cryptosystems such as RQC [AMAB+20].

1.3 Techniques

We consider field multiplication in $\mathbb{F}_{2^{mn}}$, field inversion in $\mathbb{F}_{2^{mn}}$, and Gaussian elimination as the main building blocks. The speed of our implementation is achieved by optimizing these building blocks. Here is a brief overview of how we optimize the building blocks.

- There are 3 multiplications in $\mathbb{F}_{2^{mn}}$ in key generation, encapsulation, and decapsu-

lation. Each multiplication in $\mathbb{F}_{2^{mn}}$ involves one low-weight operand. In order to exploit the structure of the low-weight operand, we make use of matrix transposition to perform the multiplication. As far as we can tell, we are the first ones to introduce this idea.

- The decoding algorithm computes the intersection of several vector spaces. One way to compute the intersection is to use the Zassenhaus algorithm. On input generating sets of two vector spaces, the Zassenhaus algorithm builds a matrix from the generating sets and applies a Gaussian elimination. Gaussian elimination is also useful for checking the weights of sampled elements in key generation and encapsulation. In order to perform Gaussian eliminations in constant time, we generalized the algorithm presented in [BCS13, Section 6]. The original algorithm is only able to compute the systematic form (if it exists) of the input matrix, while our generalized algorithm is able to compute row echelon form or reduced row echelon form. The generalized algorthm turns out to be very efficient even under the constraint of being constant-time. As far as we can tell, we are the first ones to introduce this generalized algorithm.

- Key generation involves one field inversion in $\mathbb{F}_{2^{mn}}$. We make use of the Itoh-Tsuji algorithm [IT88] to perform the field inversions in $\mathbb{F}_{2^{mn}}$. We found that this is much faster than raising field elements to the power $2^{mn} - 2$.

At the end of each call to the Zassenhaus algorithm, we need to derive a generating set of the intersection for the next call. We use a small loop that is carefully designed to protect the procedure against timing attacks. We also propose an efficient way to generate low-weight elements using our algortihm for Gaussian elimination. Both the loop and the algorithm for generating low-weight elements are presented in Section 5.

1.4 Availability of Source Code

We plan to submit our implementation to the eBACS project [Be] so that the source code can be included in SUPERCOP. Our source code will be in the public domain.

1.5 Organization

Section 2 reviews the specification of ROLLO and introduces ROLLO$^+$. Section 3 presents how we optimize field multiplications and field inversions. Section 4 presents how we optimize Gaussian eliminations. Section 5 presents how we use the techniques in Section 3 and 4 to carry out the decoding algorithm and to sample low-weight elements. Section 6 shows some experiment results.

2 ROLLO and ROLLO$^+$

This section reviews the specification of ROLLO (mainly ROLLO-I) and introduces the specification of ROLLO$^+$. In particular, we argue that ROLLO$^+$ is IND-CPA secure as long as ROLLO-I is IND-CPA secure in this section.

2.1 Parameter Sets

The parameter sets of ROLLO and ROLLO$^+$ are shown in Table 1. As one can see ROLLO$^+$ simply takes parameters from ROLLO. We note that ROLLO-II is claimed to be CCA secure, while ROLLO$^+$-II only targets CPA security.

Table 1: Parameter sets of ROLLO and ROLLO$^+$.

instance	n	m	r	d	level
ROLLO-I-128	83	67	7	8	1
ROLLO-I-192	97	79	8	8	3
ROLLO-I-256	113	97	9	9	5
ROLLO-II-128	189	83	7	8	1
ROLLO-II-192	193	97	8	8	3
ROLLO-II-256	211	97	9	9	5
ROLLO$^+$-I-128	83	67	7	8	1
ROLLO$^+$-I-192	97	79	8	8	3
ROLLO$^+$-I-256	113	97	9	9	5
ROLLO$^+$-II-128	189	83	7	8	1
ROLLO$^+$-II-192	193	97	8	8	3
ROLLO$^+$-II-256	211	97	9	9	5

2.2 Finite Fields

Each parameter set of ROLLO and ROLLO$^+$ uses two distinct primes m and n. Each m is associated with a degree-m irreducible polynomial $P_m \in \mathbb{F}_2[x]$, and similarly each n is associated with a degree-n irreducible polynomial $P_n \in \mathbb{F}_2[y]$. The list of P_m's and P_n's is shown in Table 2. P_m and P_n are used to construct $\mathbb{F}_{2^m}$ as $\mathbb{F}_2[x]/(P_m)$ and $\mathbb{F}_{2^{mn}}$ as $\mathbb{F}_{2^m}[y]/(P_n)$. A field element $a \in \mathbb{F}_{2^m}$ will always be represented as a vector $(a_0, ..., a_{m-1}) \in \mathbb{F}_2^m$ such that $a = \sum_i a_i x^i$. A field element $\alpha \in \mathbb{F}_{2^{mn}}$ will be represented as a vector $(\alpha_0, ..., \alpha_{n-1}) \in \mathbb{F}_{2^m}^n$ such that $\alpha = \sum_i \alpha_i y^i$, if α is a part of a key pair or a ciphertext. Our implementation represents low-weight $\mathbb{F}_{2^{mn}}$ elements in a different way, which will be explained in Section 3.

We note that P_n is called P in ROLLO's specification [ABD$^+$20]. We call the polynomial P_n because this explicitly shows that its degree is n. Also, the specification does not mention the field $\mathbb{F}_{2^{mn}}$, even though it is actually used implicitly: the specification defines operations of ROLLO as arithmetic between polynomials over $\mathbb{F}_{2^m}$ modulo P (i.e., modulo P_n). For the purpose of this paper, we think it is better to describe operations as arithmetic in $\mathbb{F}_{2^{mn}}$.

Table 2: The lists of P_m's and P_n's as polynomials in $\mathbb{F}_2$[X].

m or n	P_m or P_n
67	$X^{67} + X^5 + X^2 + X + 1$
79	$X^{79} + X^9 + 1$
83	$X^{83} + X^7 + X^4 + X^2 + 1$
97	$X^{97} + X^6 + 1$
113	$X^{113} + X^9 + 1$
189	$X^{189} + X^6 + X^5 + X^2 + 1$
193	$X^{193} + X^{15} + 1$
211	$X^{211} + X^{11} + X^{10} + X^8 + 1$

Algorithm 2 Rank Support Recover (RSR) algorithm

Parameters: $m, n, d, r \in \mathbb{Z}$.
Input: An $\mathbb{F}_2$-subspace F of dimension d in $\mathbb{F}_{2^m}$, represented as $(f_1, \ldots, f_d) \in \mathbb{F}_{2^m}^d$ such that $F = \langle f_1, \ldots, f_d \rangle$, and $s = \sum_{i=0}^{n-1} s_i y^i \in \mathbb{F}_{2^{mn}}$.
Output: An $\mathbb{F}_2$-subspace E of $\mathbb{F}_{2^m}$.

1: Compute $S = \langle s_0, \ldots, s_{n-1} \rangle$.
2: $E \leftarrow \bigcap_{i=1}^{d} f_i^{-1} S$
3: **return** E

Algorithm 3 RSR$^+$ algorithm

Parameters: $m, n, d, r \in \mathbb{Z}$.
Input: An $\mathbb{F}_2$-subspace F of dimension d in $\mathbb{F}_{2^m}$, represented as $(f_1, \ldots, f_d) \in \mathbb{F}_{2^m}^d$ such that $F = \langle f_1, \ldots, f_d \rangle$, and $s = \sum_{i=0}^{n-1} s_i y^i \in \mathbb{F}_{2^{mn}}$.
Output: An $\mathbb{F}_2$-subspace E of $\mathbb{F}_{2^m}$ or $\perp$.

1: Compute $S = \langle s_0, \ldots, s_{n-1} \rangle$.
2: $E \leftarrow \bigcap_{i=1}^{d} f_i^{-1} S$
3: **if** $\dim(S) \leqslant dr$ **then**
4:     **return** E
5: **else**
6:     **return** $\perp$
7: **end if**

2.3 Terminologies and Notations

Given $\alpha \in \mathbb{F}_{2^{mn}}$, we define $\mathrm{Mat}(\alpha)$ as

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,m-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n-1,0} & \alpha_{n,1} & \cdots & \alpha_{n-1,m-1} \end{pmatrix} \in \mathbb{F}_2^{n \times m}.$$

Similarly, given $\alpha, \beta \in \mathbb{F}_{2^{mn}}$, we define $\mathrm{Mat}(\alpha, \beta)$ as the vertical concatenation of $\mathrm{Mat}(\alpha)$ and $\mathrm{Mat}(\beta)$.

The *support* of $\alpha \in \mathbb{F}_{2^{mn}}$, denoted as $\mathrm{Supp}(\alpha)$, is defined as

$$\mathrm{Supp}(\alpha) = \mathrm{RowSpace}(\mathrm{Mat}(\alpha)).$$

$\mathrm{Supp}(\alpha)$ can be viewed as the $\mathbb{F}_2$-subspace of $\mathbb{F}_{2^m}$ generated by α_i's, i.e.,

$$\langle \alpha_0, \ldots, \alpha_{n-1} \rangle \coloneqq \Big\{ \sum_i b_i \alpha_i \mid (b_0, \ldots, b_{n-1}) \in \mathbb{F}_2^n \Big\}.$$

Similarly, given $\alpha, \beta \in \mathbb{F}_{2^{mn}}$, $\mathrm{Supp}(\alpha, \beta)$ is defined as

$$\mathrm{RowSpace}(\mathrm{Mat}(\alpha, \beta)) = \mathrm{Supp}(\alpha) + \mathrm{Supp}(\beta)$$

and can be viewed as $\langle \alpha_0, \ldots, \alpha_{n-1}, \beta_0, \ldots, \beta_{n-1} \rangle$.

The *rank weight* (or simply weight) of α, denoted as $||\alpha||$, is defined as the dimension of $\mathrm{Supp}(\alpha)$. Similarly, we define the rank weight of (α, β), i.e., $||(\alpha, \beta)||$, as the dimension of $\mathrm{Supp}(\alpha, \beta)$. The set $\mathcal{S}_w^{2n}(\mathbb{F}_{2^m})$ is defined as

$$\{(\alpha, \beta) \in \mathbb{F}_{2^{mn}}^2 \mid ||(\alpha, \beta)|| = w\}.$$

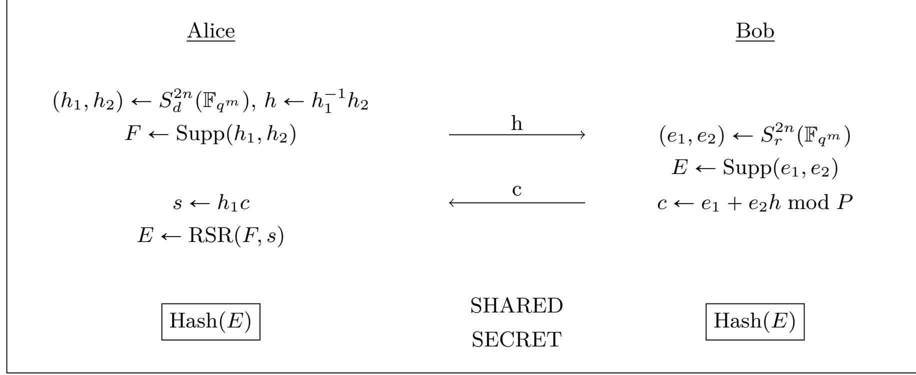

Figure 1: Informal description of ROLLO-I.

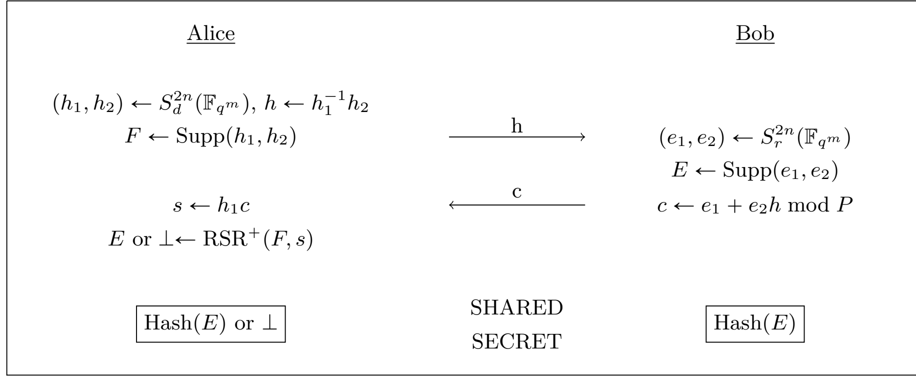

Figure 2: Informal description of ROLLO$^+$.

We note that the specification instead defines $\mathcal{S}_w^{2n}(\mathbb{F}_{2^m})$ as a set of vectors in $\mathbb{F}_{2^m}^{2n}$, so the definition there might look different from (but is actually equivalent to) our definition.

We use $\dim(S)$ to denote the dimension of a linear subspace S, and we use $\mathrm{rank}(M)$ to denote the rank of a matrix M.

2.4 Decoding Algorithms

ROLLO uses the Rank Support Recover (RSR) algorithm as the decoding algorithm. The algorithm is shown in Algorithm 2. On input an $\mathbb{F}_2$-subspace $F \subset \mathbb{F}_{2^m}$ of dimension d and $s \in \mathbb{F}_{2^{mn}}$ such that $\langle s_1, \ldots, s_n \rangle = S \subseteq EF := \langle \{ef \mid e \in E, f \in F\} \rangle$ for some $\mathbb{F}_2$-subspace $E \subset \mathbb{F}_{2^m}$ of dimension r, RSR recovers E with certain probability: RSR is a probabilistic algorithm, so it might fail to recover E. Note that F is always represented as a vector $(f_1, \ldots, f_d) \in \mathbb{F}_{2^m}^d$ such that $F = \langle f_1, \ldots, f_d \rangle$.

As we will show in the next subsection, RSR is used as a subroutine of decapsulation of ROLLO-I. The way decapsulation is defined ensures that $S \subseteq EF$ and thus $\dim(S) \leqslant dr$, as long as the ciphertext is valid. However, as shown in Algorithm 2, RSR does not check if $\dim(S) \leqslant dr$. Even when $\dim(S) > dr$, in which case the ciphertext must be invalid, RSR still computes and returns $\bigcap_{i=1}^d f_i^{-1}S$. As one might have expected, the larger $\dim(S)$ is, the more computation it takes to compute $\bigcap_{i=1}^d f_i^{-1}S$. A constant-time implementation of RSR thus has to take at least as much time as when $\dim(S) = \min(m, n)$.

In order to avoid spending extra time on processing these invalid ciphertexts, ROLLO$^+$

uses an adapted version of RSR, which we call the RSR$^+$ algorithm. The pseudocode of RSR$^+$ is shown in Algorithm 3. RSR$^+$ checks the dimension of S and only returns E when $\dim(S) \leqslant dr$. When $\dim(S) > dr$, the algorithm simply returns $\perp$. In this way, the running time of the algorithm can be bounded by the time for the case $\dim(S) = dr$.

In our implementation, each of S, $f_i^{-1}S$, and E is represented as an array of $\mathbb{F}_{2^m}$ elements that generate the subspace, which will be explained in more detail in Section 5.

2.5 Key generation, Encapsulation, Decapsulation

Key generation, encapsulation, and decapsulation of ROLLO-I are depicted in Figure 1. As shown in the figure, Alice starts with generating the public key h and secret key (h_1, h_2). Then, Alice sends the public key to Bob. On receiving the public key h, Bob runs the encapsulation algorithm to generate the ciphertext c and session key Hash(E), and he sends c to Alice. Here, Hash(E) means hashing the reduced row echelon form of the matrix where the rows are formed by the basis elements of E. ROLLO uses SHA256 as the hash function. With c and the secret key (h_1, h_2), Alice runs the decoding algorithm to obtain E and the session key Hash(E). We note that the specification denotes the secret key as (x, y), but we use (h_1, h_2) because we would like to save x and y for polynomials.

Key generation, encapsulation, and decapsulation of ROLLO$^+$ are depicted in Figure 2. The specification of ROLLO$^+$ is very similar to ROLLO-I. The differences between ROLLO-I and ROLLO$^+$ are listed below.

- ROLLO-I uses RSR as the decoding algorithm, while ROLLO$^+$ uses RSR$^+$.
- In ROLLO-I decapsulation always returns Hash(E), while in ROLLO$^+$ decapsulation returns $\perp$ if RSR$^+$ returns $\perp$.
- In ROLLO-I the secret key is defined as (h_1, h_2), while in ROLLO$^+$ the secret key is defined as (h_1, F). What decapsulation (of either ROLLO-I or ROLLO$^+$) needs are h_1 and F, so we think it is more natural to define the secret key in this way.

2.6 IND-CPA Security of ROLLO$^+$

We claim that ROLLO$^+$ is IND-CPA secure, as long as ROLLO-I is IND-CPA secure. To see this, let us recall the IND-CPA game as shown in, say, [Pei14]. In the game, the adversary is asked to distinguish between the following two experiments:

$$
\begin{array}{l|l}
pp \leftarrow \text{Setup}() & pp \leftarrow \text{Setup}() \\
(pk, sk) = \text{Gen}(pp) & (pk, sk) = \text{Gen}(pp) \\
(c, k) \leftarrow \text{Encaps}(pp, pk) & (c, k) \leftarrow \text{Encaps}(pp, pk) \\
 & k^* \leftarrow \mathcal{K} \\
\text{Output}(pp, pk, c, k) & \text{Output}(pp, pk, c, k^*)
\end{array}
$$

In the experiments, Setup is a function that outputs public parameters, Gen stands for the key generation algorithm, and Encaps stands for the encapsulation algorithm. As one can see the decapsulation algorithm is not used in the game, and sk is generated but not used. As ROLLO-I and ROLLO$^+$ only differ in the formats of secret keys and the decapsulation algorithms, clearly ROLLO$^+$ must be IND-CPA secure if ROLLO-I is IND-CPA secure.

In fact it is also easy to see that if (h_1, h_2) is the same in ROLLO-I and ROLLO$^+$, then on input the same valid ciphertext c, the two decapsulation algorithms will generate the same outputs. This implies that ROLLO$^+$ and ROLLO-I have the same failure rates for decapsulation.

2.7 Building Blocks

As mentioned in the introduction, our implementation of ROLLO$^+$ makes use of optimizations for the following building blocks.

- Field multiplication in $\mathbb{F}_{2^{mn}}$: we need to multiply h_1^{-1} and h_2 in key generation, multiply e_2 and h in encapsulation, and multiply h_1 and c in decapsulation. The reader should be aware that in each of the three multiplications, there is one operand of weight bounded by either d or r: the weights of h_2 and h_1 are bounded by d, and the weight of e_2 is bounded by r.
- Gaussian elimination: as previous implementations, our implementation uses the Zassenhaus algorithm to compute the intersection of two vector spaces, which requires to compute row echelon form of matrices. Our implementation also uses Gaussian elimination to check the weights of (h_1, h_2) and (e_1, e_2).
- Field inversion in $\mathbb{F}_{2^{mn}}$: we need to compute the inverse of h_1 in key generation.

We emphasize that ROLLO-I and ROLLO-II can be implemented using essentially the same building blocks.

3 Field Multiplications and Inversions

As described in Section 2.7, we need to multiply elements in $\mathbb{F}_{2^{mn}}$ in key generation, encapsulation, and decapsulation. This section shows how we build our multiplication functions for $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/(P_m)$ and $\mathbb{F}_{2^n} = \mathbb{F}_2[y]/(P_n)$ as subroutines of the multiplication function for $\mathbb{F}_{2^{mn}}$, and shows how we optimize our multiplication function for $\mathbb{F}_{2^{mn}}$ by exploiting the fact that one of the operands is of weight bounded by d or r. In key generation, it is necessary to compute the inverse of h_1. This section shows how we implement the field inversions in $\mathbb{F}_{2^{mn}}$.

3.1 Field multiplications in $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^n}$

As many cryptographic implementations that involve multiplications in binary fields, our implementation makes use of the `pclmulqdq` instruction. Given two 64-bit polynomials in $\mathbb{F}_2[x]$, i.e., binary polynomials of degree at most 63, the instruction computes the product of them.

Take our multiplication function for $\mathbb{F}_{2^{67}}$ for example, which has been shown in Figure 3. We consider the two operands a and b as polynomials $a = \sum_{i=0}^{66} a_i x^i \in \mathbb{F}_2[x]$ and $b = \sum_{i=0}^{66} b_i x^i \in \mathbb{F}_2[x]$. Let $c = \sum_{i=0}^{66} c_i x^i = ab \bmod P_{67}$. To carry out the field multiplication a is represented as $(a^{(0)}, a^{(1)})$, where $a^{(0)} = \sum_{i=0}^{63} a_i x^i$ and $a^{(1)} = \sum_{i=0}^{2} a_{i+64} x^i$. Similarly b is represented as $(b^{(0)}, b^{(1)})$. First, we obtain $a^{(0)}b^{(0)}$, $a^{(0)}b^{(1)}$, $a^{(1)}b^{(0)}$, and $a^{(1)}b^{(1)}$ by using `pclmulqdq` 4 times. Note that $a^{(1)}b^{(1)}$ consists of 5 bits only. Our goal is to compute

$$c = a^{(0)}b^{(0)} + (a^{(0)}b^{(1)} + a^{(1)}b^{(0)})x^{64} + a^{(1)}b^{(1)}x^{128} \bmod P_{67}.$$

To carry out the reduction modulo P_{67}, Let

$$c' = a^{(0)}b^{(0)} + (a^{(0)}b^{(1)} + a^{(1)}b^{(0)})x^{64} + a^{(1)}b^{(1)}x^{128} = c'^{(0)} + c'^{(1)}x^{64} + c'^{(2)}x^{128},$$

where $\deg(c'^{(i)}) < 64$ for all i. Observe that $x^{128} \equiv x^{61}(x^5 + x^2 + x + 1) \bmod P_{67}$, so $c = c^{(0)} + c^{(1)}x^{64} + c^{(2)}(x^5 + x^2 + x + 1)x^{61} \bmod P_{67}$. We thus compute $c'^{(2)}(x^5 + x^2 + x + 1)$ using one `pclmulqdq` to obtain

$$c'' = c'^{(0)} + c'^{(1)}x^{64} + c'^{(2)}(x^5 + x^2 + x + 1)x^{61} = c''^{(0)} + c''^{(1)}x^{64},$$

```
static inline void f2m_mul(f2m f1_times_f2, const f2m f1, const f2m f2)
{
    __m128i f1lof2hi, f1hif2lo, lo, middle, hi, tmp;
    __m128i f1_copy = _mm_set_epi64x(f1[1], f1[0]);
    __m128i f2_copy = _mm_set_epi64x(f2[1], f2[0]);
    __m128i poly;
    __m128i zero = _mm_setzero_si128();

    hi = _mm_clmulepi64_si128(f1_copy, f2_copy, 0x11);
    lo = _mm_clmulepi64_si128(f1_copy, f2_copy, 0x00);

    f1lof2hi = _mm_clmulepi64_si128(f1_copy, f2_copy, 0x10);
    f1hif2lo = _mm_clmulepi64_si128(f1_copy, f2_copy, 0x01);
    middle = _mm_xor_si128(f1lof2hi, f1hif2lo);
    middle = _mm_xor_si128(middle, _mm_unpacklo_epi64(zero, hi));

    poly = _mm_set_epi64x(0, 0x27);

    tmp = _mm_clmulepi64_si128(middle, poly, 0x01);
    lo = _mm_xor_si128(lo, _mm_unpacklo_epi64(zero, middle));
    lo = _mm_xor_si128(lo, _mm_slli_epi64(tmp, 61));
    lo = _mm_xor_si128(lo, _mm_unpacklo_epi64(zero, _mm_srli_epi64(tmp, 3)));
    lo = _mm_xor_si128(lo, _mm_clmulepi64_si128(_mm_srli_epi64(lo, 3), poly, 0x01));

    f1_times_f2[0] = _mm_extract_epi64(lo, 0);
    f1_times_f2[1] = 0x7 & _mm_extract_epi64(lo, 1);
}
```

Figure 3: Our multiplication function for $\mathbb{F}_{2^{67}}$.

where $\deg(c''^{(i)}) < 64$ for all i. Finally, observe that

$$c''^{(0)} + c''^{(1)}x^{64} = c''^{(0)} + (\sum_{i=0}^{2} c_i''^{(1)}x^i)x^{64} + (\sum_{i=3}^{63} c_i''^{(1)}x^i)x^{64}$$
$$= c''^{(0)} + (\sum_{i=0}^{2} c_i''^{(1)}x^i)x^{64} + (\sum_{i=0}^{60} c_{i+3}''^{(1)}x^i)x^{67}.$$

We thus compute $(\sum_{i=0}^{60} c_{i+3}''^{(1)}x^i)(x^5 + x^2 + x + 1)$ by using `pclmulqdq` to obtain

$$c = \sum_{i=0}^{66} c_i''x^i + (\sum_{i=0}^{60} c_{i+3}''^{(1)}x^i)(x^5 + x^2 + x + 1).$$

In addition to multiplications in $\mathbb{F}_{2^m}$, we also implemented multiplications in $\mathbb{F}_{2^n}$ using basically the same strategy. As described in the following subsection, our implementation uses both the multiplication functions for $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^n}$ to carry out multiplications in $\mathbb{F}_{2^{mn}}$. Note that this differs from previous implementations, as they do not use any multiplication function for $\mathbb{F}_{2^n}$.

3.2 Accelerating Multiplications in $\mathbb{F}_{2^{mn}}$ with Matrix Transposition

To multiply two elements in $\mathbb{F}_{2^{mn}}$, all previous implementations represent the operands as polynomials over $\mathbb{F}_{2^m}$ and carry out a generic polynomial multiplication using arithmetic in $\mathbb{F}_{2^m}$. However, each of the three multiplications mentioned in Section 2.7 involves one element of weight bounded by d or r, and we found that this can be exploited to make the multiplications much faster.

For each of the three multiplications, the task is to multiply $v, w \in \mathbb{F}_{2^{mn}}$, where (v, w) is either (h_2, h_1^{-1}), (h_1, c), or (e_2, h). Let $t = d$ if $v \in \{h_1, h_2\}$ or $t = r$ if $v = e_2$. Let $\beta_1, \ldots, \beta_t$ be a basis of $\mathrm{Supp}(h_1, h_2)$ if $v \in \{h_1, h_2\}$ or a basis of $\mathrm{Supp}(e_1, e_2)$ if $v = e_2$. We represent w as $(w_0, w_1, \cdots, w_{n-1})$ in $\mathbb{F}_{2^m}^n$ such that $w = w_0 + w_1 y + \cdots + w_{n-1} y^{n-1}$. In the meantime we represent v as two vectors $(\alpha_1, \ldots, \alpha_t) \in \mathbb{F}_{2^n}^t$ and $(\beta_1, \ldots, \beta_t) \in \mathbb{F}_{2^m}^t$, such that

$$v = \alpha_1 \beta_1 + \alpha_2 \beta_2 + \cdots + \alpha_t \beta_t.$$

To see why v can be represented in this way, let $v = v_0 + v_1 y + \ldots v_{n-1} y^{n-1}$ such that $v_i \in \mathbb{F}_{2^m}$ for all i. Suppose $v_i = \sum_j \alpha_{j,i} \beta_j$ where each $\alpha_{j,i} \in \mathbb{F}_2$, v can be rewritten as

$$(\sum_{j=1}^{t} \alpha_{j,0} \beta_j) + (\sum_{j=1}^{t} \alpha_{j,1} \beta_j) y + \cdots + (\sum_{j=1}^{t} \alpha_{j,n-1} \beta_j) y^{n-1}.$$

Now, let $\alpha_j = \sum_{i=0}^{n-1} \alpha_{j,i} y^i \in \mathbb{F}_{2^n}$, we have $v = \alpha_1 \beta_1 + \alpha_2 \beta_2 + \cdots + \alpha_t \beta_t$.

Following the discussion above, we have

$$wv = (w\beta_1)\alpha_1 + \cdots + (w\beta_t)\alpha_t.$$

Since w is represented as an array of n elements in $\mathbb{F}_{2^m}$, each $w\beta_i$ can be obtained by calling the multiplication function for $\mathbb{F}_{2^m}$ n times. Let

$$\gamma_i = w\beta_i = \gamma_{i,0} + \gamma_{i,1} y + \cdots + \gamma_{i,n-1} y^{n-1},$$

where each $\gamma_{i,j} \in \mathbb{F}_{2^m}$. Now the task is to compute $\gamma_i \alpha_i$ for each i. Mathematically, we can view $\mathbb{F}_{2^{mn}}$ as $\mathbb{F}_{2^n}[x]/(P_m)$, which means γ_i can be written as $\gamma'_{i,0} + \gamma'_{i,1} x + \cdots + \gamma'_{i,m-1} x^{m-1}$, where each $\gamma'_{i,k} \in \mathbb{F}_{2^n}$. Once we have $\gamma'_{i,0}, \ldots, \gamma'_{i,m-1}$, it would be easy to multiply each γ_i by α_i: we can simply call our multiplication function for $\mathbb{F}_{2^n}$ m times. But how can we derive $\gamma'_{i,k}$'s from $\gamma_{i,j}$'s?

This is actually not so hard to see. Let $\gamma_{i,j} = \sum_{k=0}^{m-1} \gamma_{i,j,k} x^k$, where $\gamma_{i,j,k} \in \mathbb{F}_2$. Then we have

$$\begin{aligned}
&\gamma_{i,0} + \gamma_{i,1} y + \cdots + \gamma_{i,n-1} y^{n-1} \\
=&(\sum_{k=0}^{m-1} \gamma_{i,0,k} x^k) + (\sum_{k=0}^{m-1} \gamma_{i,1,k} x^k) y + \cdots + (\sum_{k=0}^{m-1} \gamma_{i,n-1,k} x^k) y^{n-1} \\
=&(\sum_{j=0}^{n-1} \gamma_{i,j,0} y^j) + (\sum_{j=0}^{n-1} \gamma_{i,j,1} y^j) x + \cdots + (\sum_{j=0}^{n-1} \gamma_{i,j,m-1} y^j) x^{m-1}.
\end{aligned}$$

In other words, $\gamma'_{i,k} = \sum_{j=0}^{n-1} \gamma_{i,j,k} y^j$ for $k = 0, \ldots, m-1$.

To see how our implementation obtains $\gamma'_{i,k}$ for $k = 0, \ldots, m-1$, it is useful to consider the matrix

$$\mathrm{Mat}(\gamma_i) = \begin{pmatrix} \gamma_{i,0,0} & \gamma_{i,0,1} & \cdots & \gamma_{i,0,m-1} \\ \gamma_{i,1,0} & \gamma_{i,1,1} & \cdots & \gamma_{i,1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{i,n-1,0} & \gamma_{i,n-1,1} & \cdots & \gamma_{i,n-1,m-1} \end{pmatrix}.$$

γ_i is stored as an array of n $\mathbb{F}_{2^m}$ elements $\gamma_{i,0}, \ldots, \gamma_{i,m-1}$, meaning that the matrix is stored in a row-major fashion. Our implementation performs a matrix transposition to obtain an array of m $\mathbb{F}_{2^n}$ elements $\gamma'_{i,0}, \ldots, \gamma'_{i,m-1}$. We then multiply each $\gamma'_{i,k}$ by α_i using our multiplication function for $\mathbb{F}_{2^n}$ to obtain $\gamma_i \alpha_i$.

For ease of implementation, we actually augment the $n \times m$ matrix to obtain a 256×128 matrix. This is possible as $m < 128$ and $n < 256$. We then use an assembly function `transpose_64x256_sp_asm` to transpose the 256×128 matrix. What `transpose_64x256_sp_asm` does is essentially transposing 4 64×64 matrices in parallel. In total there are 8 64×64 submatrices in the 256×128 matrix, so to complete the matrix transposition we need to call `transpose_64x256_sp_asm` twice. The algorithm used by `transpose_64x256_sp_asm` to transpose each 64×64 matrix has been explained in [Cho17, Section 2], and the function simply vectorizes the algorithm so that 4 matrix transpositions can be carried out in parallel using logical instructions for YMM registers. We note that the assembly function is included in the source code of Classic McEliece [ABC+20]. The source code of Classic McEliece, including the function, is in the public domain.

We make use of lazy reduction to save operations. To multiply $\gamma'_{i,k}$'s by α_i, we do not actually use the complete multiplication function for $\mathbb{F}_{2^n}$. Instead, we only reduce the results so that they can fit into 256 bits. Then, we compute the array of m $\mathbb{F}_{2^n}$ elements $\sum_i \gamma'_{i,0}\alpha_i, \ldots, \sum_i \gamma'_{i,m-1}\alpha_i$ and reduce them fully. Finally, we transpose the matrix corresponding to this array to obtain an array of n $\mathbb{F}_{2^m}$ elements as the default representation of wv.

3.3 Field Inversions

We make use of the Bernstein-Yang constant-time GCD algorithm [BY19] to perform field inversions in $\mathbb{F}_{2^m}$. The basic version of the algorithm for modulo inversion is shown in the code segments in Figure 5.1 and 6.1 of [BY19], and in [BY19, Section 7.1] an adapted version for polynomials over $\mathbb{F}_3$ is shown. We consider the $\mathbb{F}_{2^m}$ element as a polynomial over $\mathbb{F}_2$ and compute its inverse modulo $P_m \in \mathbb{F}_2[x]$. The algorithm we use to compute inversions in $\mathbb{F}_{2^m}$ is adapted from the algorithm for polynomials over $\mathbb{F}_3$. The only modification we did is to replace the variable c in each step by g_0.

For field inversions in $\mathbb{F}_{2^{mn}}$, we use the Itoh-Tsuji algorithm [IT88]. Suppose the goal is to compute α^{-1} given a nonzero element $\alpha \in \mathbb{F}_{2^{mn}}$. Define $\gamma = 1 + 2^m + 2^{2m} + 2^{3m} + \cdots + 2^{(n-1)m}$. The idea is to compute α^{-1} as $\alpha^{\gamma-1}/\alpha^{\gamma}$. Define $\mathrm{frob}(\cdot)$ as the operation of raising the input to the power 2^m. In order to compute $\alpha^{\gamma-1}$, we let $t_0 = \alpha$ and compute $t_1 \leftarrow \mathrm{frob}(t_0)t_0$, $t_2 \leftarrow \mathrm{frob}^2(t_1)t_1$, $t_3 \leftarrow \mathrm{frob}^4(t_2)t_2$, and so on, until we have $t_{\lfloor \log_2(n-1) \rfloor}$. Suppose $n - 1 = (\sum_{i \in I} 2^i) + 2^{\lfloor \log_2(n-1) \rfloor}$ and let $t' = t_{\lfloor \log_2(n-1) \rfloor}$. We perform $t' \leftarrow \mathrm{frob}^{2^i}(t')t_i$ for each $i \in I$. Now $t' = \alpha^{1+2^m+2^{2m}+2^{3m}+\cdots+2^{(n-2)m}}$, so $\alpha^{\gamma-1}$ is computed as $\mathrm{frob}(t')$.

Once we have $\alpha^{\gamma-1}$, α^{γ} is computed as $\alpha \cdot \alpha^{\gamma-1}$. Observe that $2^{mn} - 1 = (2^m - 1)\gamma$ and thus $(\alpha^{\gamma})^{2^m-1} = \alpha^{2^{mn}-1} = 1$. This implies that $\alpha^{\gamma} \in \mathbb{F}_{2^m}$. Therefore, from $\alpha^{\gamma-1}$ and α^{γ}, we compute α^{-1} using 1 field inversion and n field multiplications in $\mathbb{F}_{2^m}$.

Note that each frob^{2^i} is a linear operation: suppose the input and output are viewed as vectors in $\mathbb{F}_{2^m}^n$, the operation simply multiplies the input by an $n \times n$ matrix over $\mathbb{F}_2$ to obtain the output. frob^{2^i} is thus very cheap for any i. In the Itoh-Tsuji algorithm, we have to perform a few generic multiplications in $\mathbb{F}_{2^{mn}}$. These multiplications are implemented with four layers of Karatsuba, which appears to be provide the best performance for the parameter sets.

4 Gaussian Elimination

As described in Section 2.7, we use Gaussian elimination to generate $(h_1, h_2) \in \mathcal{S}_d^{2n}(\mathbb{F}_{2^m})$ and $(e_1, e_2) \in \mathcal{S}_r^{2n}(\mathbb{F}_{2^m})$, and to perform the Zassenhaus algorithm. This section presents how we implement Gaussian elimination to reduce matrices into row echelon form or reduced row echelon form. We note that our implementation computes row echelon form instead of reduced row echelon form whenever possible to save computation.

4.1 A Constant-time Algorithm for Computing Systematic Form

Assume that matrix $A \in \mathbb{F}_2^{\mu\times\nu}$ has systematic form. In [BCS13, Section 6], the authors describe an algorithm for computing systematic form of A. We denote the ith row of A as A_i. The algorithm consists of the following steps.

1. Set $p = 1$.
2. For $i \in \{p+1, \ldots \mu\}$, $A_p \leftarrow A_p + A_i \cdot (1 - A_{p,p})$.
3. For $i \in \{1, \ldots, \mu\}\backslash\{p\}$, $A_i \leftarrow A_i + A_p \cdot A_{i,p}$.
4. If $p + 1 \leqslant \min(\mu, \nu)$, increase p by 1 and go back to Step 2.

In each iteration of the algorithm, column p of the matrix is scanned to search for the pth pivot. Step 2 sets $A_{p,p}$ to 1 by conditionally adding row i to row p for all $i > p$. Step 3 sets all $A_{i,p}$'s with $i \neq p$ to 0 by conditionally adding row p to each row i. The values of p and i at any point of the algorithm do not depend on the entries of the input matrix, so the algorithm is constant-time even if $A_i, A_p, A_{p,p}, A_{i,p}$ are accessed using memory load/store instructions.

Note that systematic form is the same as reduced row echelon form for A. If in Step 3 the set $\{1, \ldots, \mu\}\backslash\{p\}$ is replaced by $\{p+1, \ldots, \mu\}$, row-echelon form of A will be computed instead of reduced row echelon form. However, for matrices without systematic form, it is not guaranteed that the algorithm above will generate reduced row echelon form, and it is also not guaranteed that the corresponding algorithm with $\{p + 1, \ldots, \mu\}$ in Step 3 will generate row echelon form.

4.2 Computing Row Echelon Form and Reduced Row Echelon Form

We generalized the algorithm in Section 4.1 so that the reduced row echelon form of any A can be computed. The main idea is to search for the column index j of pivot p in each iteration. Once the column index is found, we can use steps that are similar to Step 2 and 3 in the algorithm in Section 4.1 to set $A_{p,j}$ to 1 and set all $A_{i,j}$ with $i \neq p$ to 0. Our algorithm consists of the following steps.

1. Set $p = 1$.
2. Set v to the logical OR of $A_p, \ldots, A_\mu$.
3. Find the index j of the first nonzero entry in v. If $v = 0$, set j to any value in $\{1, \ldots, \nu\}$.
4. For $i \in \{p+1, \ldots \mu\}$, $A_p \leftarrow A_p + A_i \cdot (1 - A_{p,j})$.
5. For $i \in \{1, \ldots, \mu\}\backslash\{p\}$, $A_i \leftarrow A_i + A_p \cdot A_{i,j}$.
6. If $p + 1 \leqslant \min(\mu, \nu)$, increase p by 1 and go back to Step 2.

In each iteration of the algorithm, we search for the pth pivot in the submatrix formed by $A_p, \ldots, A_\mu$ [2]. Step 2 and 3 find the column index j of the pivot. Step 4 ensures that $A_{p,j} = 1$ by conditionally adding A_i with $i > p$ to row A_p. Step 5 ensures that $A_{i,j} = 0$ for all $i \neq p$ by conditionally adding A_p to other rows.

We note that the algorithm works even if the input matrix is not full rank. Indeed, in this case, at the end of iteration $\text{rank}(A)$, the matrix must be in reduced row echelon form. In the last $\min(\mu, \nu) - \text{rank}(A)$ iterations, Step 4 and 5 will not change the matrix

[2] Actually the pth pivot must lie in column q with $q \geqslant p$, so we can make Step 2,3,4, and 5 only operate on columns of indices greater than or equal to p to save computation.

```
static inline
int vec256_find_first_one(__m256i v, __m256i *out)
{
    __m256i zero, mask;
    uint64_t index, mask_one;

    zero = _mm256_setzero_si256();
    mask = _mm256_cmpeq_epi64(v, zero);
    mask_one = ~_mm256_movemask_epi8(mask);
    index = (_tzcnt_u64(mask_one) & 24) >> 1;
    mask = _mm256_set1_epi64x(0x753100006420);
    mask = _mm256_srli_epi64(mask, index);
    a = _mm256_permutevar8x32_epi32(v, mask);
    index = _tzcnt_u64(_mm256_extract_epi64(v, 0));
    *out = mask;

    return index & 0x3F;
}
```

Figure 4: Our function for finding the index of the first nonzero bit in a 256-bit vector.

because $A_p, \ldots, A_\mu$ are always zero. We can modify the set in Step 5 to $\{p+1, \ldots, \mu\}$ so that the algorithm computes only row echelon form.

We have explained why the algorithm always computes reduced row echelon form and how it can be adapted to computed row echelon form. The remaining question is, how can we make the algorithm constant-time? In fact, it is not hard to see that the algorithm must be constant-time if finding the first nonzero entry of v (Step 3) and accesses of $A_{p,j}$ (Step 4) and $A_{i,j}$ (Step 5) are made constant-time. Below we show that the steps can be made constant-time by using AVX/AVX2 intrinsics.

4.3 Implementing Our Gaussian Elimination algorithm

Consider $128 < \nu \leqslant 256$. In this case, our implementation represents A as an array of μ 256-bit vectors of type `__m256i`. In each iteration of the algorithm in the previous subsection, the vector v is thus computed using $\mu - p$ ORs between the 256-bit vectors. We then use the function `vec256_find_first_one` in Figure 4 to find the index of the first one of v.

To understand how the function `vec256_find_first_one` works, let us first consider the case when v is not a zero vector. We first use the intrinsic `_mm256_cmpeq_epi64` to generate a 256-bit vector `mask` where each of the 4 64-bit blocks is set to `0xFF...F` if the corresponding block in v is 0, or `0x00...0` if the corresponding block in v is not 0. The 256-bit vector is then compressed into a 32-bit value `mask_one` using `_mm256_movemask_epi8`, such that each byte (of value either `0xFF` or `0x00`) in `mask` is reduced to the complement of the most significant bit of it. Then, we use the intrinsic `_tzcnt_u64` to compute the index of the first (i.e., the least significant) 1 in the 32-bit value. `_tzcnt_u64` is compiled into the `tzcnt` instruction. The `tzcnt` instruction returns the index of the least significant 1 of the operand, or 64 if the operand is 0. The return value of `_tzcnt_u64` thus lies in $\{0, 8, 16, 24\}$. The right shift ensures that the value of `index` must be 0, 4, 8, or 12 when the index of the first nonzero 64-bit block is 0, 1, 2, or 3, respectively. We then shift each 64-bit block of `_mm256_set1_epi64x(0x753100006420)` by `index` bits with `_mm256_srli_epi64`. The output `mask` of `_mm256_srli_epi64` can be used to broadcast the 64-bit block in a 256-bit vector that has the same index with the first nonzero block in v with `_mm256_permutevar8x32_epi32`. We then use `mask` to broadcast the first nonzero 64-bit block of v and use `_mm256_extract_epi64(v, 0)` to obtain the first nonzero block.

```
static inline
__m256i f2m_double_get_coeff_avx(__m256i ai, int index, __m256i mask)
{
    __m256i zero, one;

    zero = _mm256_setzero_si256();
    one = _mm256_set1_epi64x(1);

    ai = _mm256_srli_epi64(ai, index);
    ai = _mm256_and_si256(ai, one);
    ai = _mm256_sub_epi64(zero, ai);
    ai = _mm256_permutevar8x32_epi32(ai, mask);

    return ai;
}
```

Figure 5: Our function for extracting a specific bit in a 256-bit vector.

`_tzcnt_u64` is used to find the index of the first 1 in the 64-bit block. Finally, the function returns the result of `_tzcnt_u64`, which is the lower 6 bits of the index of the first 1 in v. The top 2 bits of the index of the first 1 in v are implicitly contained in `mask`. We set `*out` to `mask` so that the caller function gets the information of the top 2 bits.

Now consider the case when v is a zero vector. In this case, both the first and second lines involving the function `_tzcnt_u64` set `index` to 0. This means that Step 3 of the algorithm in Section 4.2 sets j to 1, which does not affect correctness of our algorithm.

We use the function in Figure 5 to obtain $A_{i,j}$ and $A_{p,j}$. Without loss of generality, let us consider the task of obtaining $A_{i,j}$. To use the function, the argument `ai` is set to A_i, and `index` and `mask` are set to the return value and `*out` generated by `vec256_find_first_one`. We first shift each 64-bit block in A_i to the right by `index` bits. This ensures that $A_{i,j}$ is moved to the least significant bit of the 64-bit block it lies in. The lines of `_mm256_and_si256` and `_mm256_sub_epi64` then set each bit of the 64-bit block to $A_{i,j}$. The 64-bit block is broadcasted using `_mm256_permutevar8x32_epi32(ai, mask)`, so that each bit of `ai` is set to $A_{i,j}$. Finally, `ai` is returned. We then compute an AND between this return value and A_p and XOR the result into A_i to carry out the operation $A_i \leftarrow A_i + A_p \cdot A_{i,j}$. We use essentially the same approach to carry out $A_p \leftarrow A_p + A_i \cdot (1 - A_{p,j})$.

5 RSR$^+$, RSR, and Sampling

This sections shows how we use the building blocks presented in the previous sections to implement RSR$^+$ and to sample random elements from $\mathcal{S}_d^{2n}$ and $\mathcal{S}_r^{2n}$. We also show how RSR can be implemented using essentially the same way as how RSR$^+$ is implemented.

5.1 Implementing RSR$^+$

Our implementation of RSR$^+$ consists of following steps.

1. Compute $(s'_1, \ldots, s'_{dr}) \in \mathbb{F}_{2^m}^{dr}$, the vector of $\mathbb{F}_{2^m}$ elements formed by the first dr rows of row echelon form of $\mathrm{Mat}(s)$.
2. $E \leftarrow f_1^{-1}\langle s'_1, \ldots, s'_{dr}\rangle$.
3. For $i = 2, \ldots, d$, $E \leftarrow E \bigcap f_i^{-1}\langle s'_1, \ldots, s'_{dr}\rangle$.
4. If $\mathrm{rank}(\mathrm{Mat}(s)) \leqslant dr$, return E. Otherwise, return $\perp$.

A minor optimization in our implementation is that we use Montgomery's trick to compute all f_i^{-1}'s: we compute $f_1, f_1 f_2, \ldots, f_1 \cdots f_n$, compute $(f_1 \cdots f_n)^{-1}$, and finally compute f_n^{-1} as $(f_1 \cdots f_{n-1}) \cdot (f_1 \cdots f_n)^{-1}$, compute f_{n-1}^{-1} as $(f_1 \cdots f_{n-2}) \cdot ((f_1 \cdots f_n)^{-1} \cdot f_n)$, and so on. Each subspace $f_i^{-1}\langle s'_1, \ldots, s'_{dr}\rangle$ is represented as an array containing

$$f_i^{-1} s'_1, f_i^{-1} s'_2, \ldots, f_i^{-1} s'_{dr}.$$

Following previous implementations of ROLLO, we compute the intersection of two subspaces using the Zassenhaus algorithm. Let U and V be two matrices over the same field and with the same number of columns. The Zassenhaus algorithm applies Gaussian elimination to the matrix

$$Z = \begin{pmatrix} U & U \\ V & 0 \end{pmatrix}$$

to obtain row echelon form

$$Z' = \begin{pmatrix} A & C \\ 0 & B \\ 0 & 0 \end{pmatrix},$$

where A and B do not have any zero rows. It is guaranteed that

$$\mathrm{RowSpace}(A) = \mathrm{RowSpace}(U) + \mathrm{RowSpace}(V)$$

and

$$\mathrm{RowSpace}(B) = \mathrm{RowSpace}(U) \bigcap \mathrm{RowSpace}(V).$$

Therefore, in the first iteration of Step 3, to compute

$$f_1^{-1}\langle s'_1, \ldots, s'_{dr}\rangle \bigcap f_2^{-1}\langle s'_1, \ldots, s'_{dr}\rangle,$$

we set row i of U to the vector formed by $f_1^{-1} s'_i$ for $i \in \{1, \ldots, dr\}$, set row i of V to the vector formed by $f_2^{-1} s'_i$ for $i \in \{1, \ldots, dr\}$, and apply the algorithm in Section 4.2 to compute its row echelon form Z'. Note that Z and Z' are $2dr \times 2m$ matrices.

After the Gaussian elimination, the rows of B form a basis of the intersection. We need to extract the basis or a generating set of the intersection so that we can continue with the next iteration of Step 3. This might seem to be a trivial task, but it is actually not. A naive implementation can easily leak secret information (e.g., the dimension of the intersection) though timing.

Let $\Delta = dr$. To obtain a generating set with Δ elements of the intersection in constant time, our implementation sets $Z^{(L)}$ to the left half of Z', sets $Z^{(R)}$ to the right half of Z', and carries out the following steps for $i = 1, \ldots, \Delta$.

(a) if $Z_i^{(L)} \neq 0$ and $Z_{i+\Delta}^{(L)} = 0$, set $Z_i^{(R)}$ to $Z_{i+\Delta}^{(R)}$.

(b) if $Z_i^{(L)} \neq 0$ and $Z_{i+\Delta}^{(L)} \neq 0$, set $Z_i^{(R)}$ to 0.

Each iteration can be easily converted into a sequence of logical operations, making the loop constant-time. We claim that after the last iteration, the first dr rows of $Z^{(R)}$ will form a generating set with dr elements of the intersection.

It might be easier to see why our claim is true by analysing how the loop changes $Z^{(R)}$ in different cases. Let μ and ν be the number of rows of A and B, respectively. Then Z' can be written as

$$\left(\begin{array}{ccc|ccc} a_{1,1} & \cdots & a_{1,m} & & & \\ \vdots & & \vdots & & C & \\ a_{\mu,1} & \cdots & a_{\mu,m} & & & \\ \hline & & & b_{1,1} & \cdots & b_{1,m} \\ & \mathbf{0} & & \vdots & & \vdots \\ & & & b_{\nu,1} & \cdots & b_{\nu,m} \\ \cline{4-6} & & & & 0 & \end{array}\right) = \left(Z^{(L)} \;\middle|\; Z^{(R)}\right).$$

C is a $\mu \times m$ matrix, and the 0 at the bottom right corner is a matrix of m columns and $2dr - \mu - \nu$ (which can be 0) rows. Note that we must have $\mu \geqslant \nu$ and $\nu \leqslant dr$.

We consider the following three cases: 1) $\mu \geqslant dr$, 2) $\mu < dr$ and $\mu + \nu \geqslant dr$, and 3) $\mu < dr$ and $\mu + \nu < dr$. In the first case, at the end of the loop, the first dr rows of $Z^{(R)}$ will become

$$\left(\begin{array}{ccc} \multicolumn{3}{c}{0} \\ \hline b_{1,1} & \cdots & b_{1,m} \\ \vdots & & \vdots \\ b_{\nu,1} & \cdots & b_{\nu,m} \\ \hline \multicolumn{3}{c}{0} \end{array}\right),$$

where the zero matrix at the bottom has $2dr - \mu - \nu$ rows, and the zero matrix on the top has $dr - \nu - (2dr - \mu - \nu) = \mu - dr$ rows. In the second case, the first dr rows of $Z^{(R)}$ will become

$$\left(\begin{array}{ccc} b_{dr-\mu+1,1} & \cdots & b_{dr-\mu+1,m} \\ \vdots & & \vdots \\ b_{\nu,1} & \cdots & b_{\nu,m} \\ \hline \multicolumn{3}{c}{0} \\ \hline b_{1,1} & \cdots & b_{1,m} \\ \vdots & & \vdots \\ b_{dr-\mu,1} & \cdots & b_{dr-\mu,m} \end{array}\right),$$

where the zero matrix in the middle has $dr - \nu$ rows. In the third case, the first dr rows of $Z^{(R)}$ will become

$$\left(\begin{array}{ccc} \multicolumn{3}{c}{0} \\ \hline b_{1,1} & \cdots & b_{1,m} \\ \vdots & & \vdots \\ b_{\nu,1} & \cdots & b_{\nu,m} \\ \hline \multicolumn{3}{c}{0} \end{array}\right),$$

where the zero matrix on the top has μ rows, and the zero matrix at the bottom has $dr - \mu - \nu$ rows. In other words, the first dr rows of $Z^{(R)}$ always form a generating set of the intersection after the loop is carried out, which shows the correctness of our claim.

At the end of Step 3, we obtain a $dr \times m$ matrix that forms a generating set of E. For decapsulation, we compute reduced row echelon form of this matrix to obtain a unique representation of E for hashing.

5.2 Implementing RSR

One can implement RSR using essentially the same steps as presented at the beginning of Section 5.1. One only needs to modify Step 1 so that it becomes

1. Compute $(s'_1, \ldots, s'_m) \in \mathbb{F}_{2^m}^{dr}$, the vector of $\mathbb{F}_{2^m}$ elements formed by the first m rows of row echelon form of $\mathrm{Mat}(s)$.

Indeed, since $m < n$, we have $S = \langle s'_1, \ldots, s'_m \rangle$. Modifying Step 1 in this way will change the dimension of the matrices Z and Z' in the Zassenhaus algorithm. Now the matrices are in $\mathbb{F}_2^{2m \times 2m}$ instead of $\mathbb{F}_2^{2dr \times 2m}$. As $dr < m$, a constant-time implementation for RSR is thus expected to be slower than a constant-time implementation for RSR$^+$. We note that the loop for extracting a generating set of the intersection also needs to be modified because the dimension of the matrices has changed. This can be done by setting Δ to m instead of dr.

5.3 Sampling from S_r^{2n} and S_d^{2n}

In the key generation algorithm, (h_1, h_2) is sampled from S_d^{2n}. In the encapsulation algorithm, (e_1, e_2) is sampled from S_r^{2n}. Without loss of generality, let us consider the task of generating (h_1, h_2). As explained in Section 3.2, h_1 and h_2 can be considered as

$$h_1 = \alpha_1\beta_1 + \alpha_2\beta_2 + \cdots + \alpha_d\beta_d,$$
$$h_2 = \gamma_1\beta_1 + \gamma_2\beta_2 + \cdots + \gamma_d\beta_d,$$

where $\alpha_i, \gamma_i \in \mathbb{F}_{2^n}$ and $\beta_i \in \mathbb{F}_{2^m}$ for all i. In fact, $\mathrm{Mat}(h_1, h_2)$ is simply

$$\begin{pmatrix} \alpha_1^T & \ldots & \alpha_d^T \\ \gamma_1^T & \ldots & \gamma_d^T \end{pmatrix} \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_d \end{pmatrix} \in \mathbb{F}_2^{2n\times m},$$

where α_i's, β_i's, and γ_i's are considered as row vectors over $\mathbb{F}_2$. By definition, $\mathrm{Supp}(h_1, h_2) = \mathrm{Supp}(h_1) + \mathrm{Supp}(h_2)$ is the row space of $\mathrm{Mat}(h_1, h_2)$. $\mathrm{Mat}(h_1, h_2)$ is of rank d if and only if both matrices being multiplied are of rank d (i.e., full rank).

Following the discussion above, we carry out the following steps to generate (h_1, h_2).

1. Generate a random $d \times m$ matrix and use the algorithm in Section 4.2 to check if it is full rank by reducing it to row echelon form. If the matrix is not full rank, repeat this step.
2. Generate a random $d \times 2n$ matrix and use use the algorithm in Section 4.2 to check if it is full rank by reducing it to row echelon form. If the matrix is not full rank, repeat this step.
3. Obtain β_i's from the $d \times m$ matrix. Obtain α_i's and γ_i's from the $d \times 2n$ matrix. Compute h_1 as $\sum_i \alpha_i\beta_i$ to obtain the polynomial representation.

We do not compute the polynomial representation of h_2 because we can use γ_i's and β_i's to carry out the multiplication between h_1^{-1} and h_2 using the technique introduced in Section 3.2.

6 Experiment Results and Discussions

This section presents some experiment results. All cycle counts for our implementation and the implementation of [CCK21] are measured on one core of an Intel Xeon E3-1220 v5 CPU (Skylake), with Turbo Boost and hyper-threading disabled. The cycle counts for [AMAB+21] are taken from the paper directly and are measured on an Intel Core i7-8850H CPU (Coffee Lake).

Table 3 shows the timings of some operations in our implementation. Each column correspond to a specific operation, as explained below.

Table 3: Cycle counts for several components in our implementation for ROLLO$^+$.

instance	mul F	mul E	inv	reduced	echelon
ROLLO$^+$-I-128	22649	20100	779880	23812	84401
ROLLO$^+$-I-192	25219	25223	915808	32543	110210
ROLLO$^+$-I-256	34527	34597	1382769	50670	183346
ROLLO$^+$-II-128	50651	44618	4502184	28496	92240
ROLLO$^+$-II-192	63852	63688	3795529	37535	109578
ROLLO$^+$-II-256	74067	66292	5528510	40243	138215

Table 4: Cycle counts for key generation, encapsulation, and decapsulation of the ROLLO-I implementations from [AMAB+21] (the paper did not implement ROLLO-II), our ROLLO$^+$ implementation, and the BIKE implementation from [CCK21].

instance	key gen.	encap.	decap	level	reference
ROLLO-I-128	11034623	984432	9775241	1	[AMAB+21]
	11204649	320835	9744693		
ROLLO$^+$-I-128	851823	30361	673666	1	this paper
ROLLO$^+$-I-192	980860	38748	878398	3	
ROLLO$^+$-I-256	1477519	55353	1635966	5	
ROLLO$^+$-II-128	4663096	70621	876533	1	this paper
ROLLO$^+$-II-192	4058419	94138	1060271	3	
ROLLO$^+$-II-256	4947630	90021	1497315	5	
`bikel1`	589625	114256	1643551	1	[CCK21]
`bikel3`	1668511	267644	5128078	3	

- "mul F" means a multiplication of two $\mathbb{F}_{2^{mn}}$ elements, where the support of one of the elements is a subset of F. The multiplications of $h_1^{-1} \cdot h_2$ and $h_1 \cdot c$ are of this type.
- "mul E" means a multiplication of two $\mathbb{F}_{2^{mn}}$ elements, where the support of one of the elements is a subset of E. The multiplication of $h \cdot e_2$ is of this type.
- "inv" means a field inversion in $\mathbb{F}_{2^{mn}}$, e.g., computation of h_1^{-1}.
- "echelon" means the process of reducing a $2dr \times 2m$ matrix over $\mathbb{F}_2$ into its row echelon form, which is used for the Zassenhaus algorithm.
- "reduced" means the process of reducing a $dr \times m$ matrix over $\mathbb{F}_2$ into its reduced row echelon form, which is used for deriving a unique representation of E for hashing.

Table 4 presents the cycle counts for key generation, encapsulation, and decapsulation of our ROLLO$^+$ implementation, the two ROLLO-I-128 implementations from [AMAB+21], and the BIKE implementation from [CCK21]. The numbers of our implementation and the implementation of [CCK21] are measured using the SUPERCOP benchmarking framework.

6.1 How about the Speed of ROLLO-I?

One might think that the reason why our implementation is an order of magnitude faster than the implementations of [AMAB+21] is because the specification of ROLLO$^+$ is different from ROLLO-I. This is not true.

First of all, the encapsulation algorithm of ROLLO$^+$ is identical to that of ROLLO-I, so our encapsulation time is exactly the encapsulation time one can achieve for ROLLO-I. Second, the key generation algorithm of ROLLO$^+$ is almost identical to that of ROLLO-I, so our key generation time is expected to be very close to the key generation time one can achieve for ROLLO-I. Finally, the decapsulation algorithm of ROLLO$^+$ is somewhat different from that of ROLLO-I, but it is easy to estimate the decapsulation time of ROLLO-I when our techniques are used. To use our techniques for ROLLO-I, the matrices for the Zassenhaus algorithm will be $2m \times 2m$ matrices instead of $2dr \times 2m$ matrices as discussed in Section 5.2. Assuming that the decapsulation time is dominated by the Zassenhaus algorithm, which is true according to our experiments, the decapsulation time of ROLLO-I-128 is expected to be $(m/dr)^2 = 1.43$ times our decapsulation time of ROLLO$^+$-I-128, which is still much smaller than 9744693.

6.2 How about Other Platforms?

One might wonder whether it is possible to port our implementations to non-x86 platforms. Although how ROLLO$^+$ should be implemented on other platforms is out of the scope of this paper, we explain below how this can be done.

First of all, as shown in Section 3, our implementation makes use of `pclmulqdq` for carryless multiplications. Many platforms do not support any instruction for carryless multiplications. However, one can still use instructions for integer multiplications to carry out carryless multiplications. For example, as shown in [CC21, Section 5.1.2], `umlal` can be used to carry out carryless multiplications on Cortex-M4. Similarly, one can easily implement a function that achieves the functionality of `pclmulqdq` on any reasonable platform using instructions for integer multiplications and logical instructions. With such a function, one can easily build multiplication functions for $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^n}$.

Our implementation also makes use of `tzcnt` for counting trailing zeros. Many platforms do not support any instruction for counting trailing zeros. However, as shown in [And05], there are many methods to count the number of trailing zeros without using any `tzcnt`-like instruction. Note that the methods are not constant-time but can all be easily made constant-time. On Cortex-M4, one can simply use `rbit` to reverse the bits in a word and `clz` to count leading zeros. Following the discussion above, one can easily implement the functionality of `tzcnt` on any reasonable platform.

With a function that implements the functionality of `tzcnt`, one can find the index j of the first nonzero entry of a vector: we need to do this for the vector v in Step 3 of the Gaussian elimination algorithm presented in Section 4.2. Assuming that v is represented as an array of b-bit words. It is easy to obtain the first nonzero word in the vector along with its index $j' = \lfloor j/b \rfloor$ in constant time using logical instructions. Then, using the function that implements the functionality of `tzcnt` and j', one can easily derive the index j. A function `find_first_one` for finding the index of the first nonzero entry in a 256-bit vector is shown in Figure 6. One can also easily obtain the jth entry of a vector in constant time using simple instructions. This operation is required in Step 4 and 5 of the Gaussian elimination algorithm. A function `get_coef` for obtaining the jth entry of a 256-bit vector is also shown in Figure 6.

The function `transpose_64x256_sp_asm` and the loop for obtaining the generating set of the intersection of two linear subspaces (Section 5.1) both consist of logical instructions, so it is easy to implement the same functionalities on any reasonable platform.

Acknowledgements

This work was funded by Taiwan Ministry of Science and Technology (MOST) Grant 109-2222-E-001-001-MY3. Special thanks to the Cybersecurity Center of Excellence Project at National Applied Research Labs, Taiwan.

References

[AASA$^+$20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process, 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf.

[ABB$^+$20] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay

Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. Bike – bit flipping key encapsulation, 2020. https://bikesuite.org/.

[ABC+20] Martin Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece, 2020. https://classic.mceliece.org/.

[ABD+17a] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LAKE, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/LAKE.zip.

[ABD+17b] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. LOCKER, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/LOCKER.zip.

[ABD+20] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, Gilles Zémor, Carlos Aguilar-Melchor, Slim Bettaieb, Loïc Bidoux, Magali Bardet, and Ayoub Otmani. ROLLO, 2020. https://pqc-rollo.org/index.html.

[AMAB+17] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Adrien Hauteville, and Gilles Zémor. Ouroboros-R, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/Ouroboros_R.zip.

[AMAB+20] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Gilles Zémor, Alain Couvreur, and Adrien Hauteville. RQC, 2020. https://pqc-rqc.org/.

[AMAB+21] Carlos Aguilar-Melchor, Nicolas Aragon, Emanuele Bellini, Florian Caullery, Rusydi H Makarim, and Chiara Marcolla. Constant time algorithms for ROLLO-I-128. *SN Computer Science*, 2(5):1–19, 2021.

[And05] Sean Eron Anderson. Bit twiddling hacks, 2005. https://graphics.stanford.edu/~seander/bithacks.html.

[BBB+20] Magali Bardet, Pierre Briaud, Maxime Bros, Philippe Gaborit, Vincent Neiger, Olivier Ruatta, and Jean-Pierre Tillich. An algebraic attack on rank metric code-based cryptosystems. In *Advances in Cryptology – EUROCRYPT 2020*, pages 64–93. Springer, 2020. https://arxiv.org/pdf/1910.00810.pdf.

[BBC+20] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier Verbel. Improvements of algebraic attacks for solving the rank decoding and minrank problems. In *Advances in Cryptology – ASIACRYPT 2020*, pages 507–536. Springer, 2020. https://arxiv.org/pdf/2002.08322.pdf.

[BCS13] Daniel J Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In *Cryptographic Hardware and Embedded Systems – CHES 2013*, pages 250–272. Springer, 2013. https://eprint.iacr.org/2015/610.pdf.

[Be] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems. Accessed Sep. 29, 2021. https://bench.cr.yp.to.

[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. https://eprint.iacr.org/2019/266.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 125–148, 2021. https://eprint.iacr.org/2021/492.pdf.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–124, 2021. https://eprint.iacr.org/2021/493.pdf.

[Cho17] Tung Chou. Mcbits revisited. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 213–231. Springer, 2017. https://eprint.iacr.org/2017/793.pdf.

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

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998. https://ntru.org/f/hps98.pdf.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and computation*, 78(3):171–177, 1988. https://core.ac.uk/download/pdf/82657793.pdf.

[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. 2019. https://eprint.iacr.org/2019/787.pdf.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In *International workshop on post-quantum cryptography*, pages 197–219. Springer, 2014. https://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf.

A The Functions find_first_one and get_coef.

```
uint32_t is_zero(uint32_t w)
{
        uint32_t t = w;

        t |= t >> 16;
        t &= 0xFFFF;
        t -= 1;
        t >>= 31;
        t = -t;

        return t;
}

int find_first_one(uint32_t v[8])
{
        uint32_t w = 0, z, found = 0;
        int i, j_prime = 0;

        for (i = 0; i < 8; i++)
        {
                z = is_zero(v[i]);
                w |= v[i] & (~found);
                found |= ~z;
                j_prime += (~found) & 1;
        }

        return ((tzcnt(w) & 31) + (j_prime << 5)) & 0xFF;
}

int get_coef(uint32_t v[8], int j)
{
        uint32_t w = 0;
        int i;

        for (i = 0; i < 8; i++)
                w |= is_zero(i ^ (j >> 5)) & v[i];

        return (w >> (j & 31)) & 1;
}
```

Figure 6: The functions find_first_one and get_coef.