# Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis

Agathe Cheriere[1], Lina Mortajine[2,3], Tania Richmond[1,4], and Nadia El Mrabet[3]

[1] CNRS, IRISA, Univ. Rennes, Inria, France
263 Avenue Général Leclerc, 35042 Rennes Cedex, France
agathe.cheriere@irisa.fr
[2] Wisekey Semiconductors, Arteparc de Bachasson, Bâtiment A, 13590 Meyreuil
[3] Mines Saint-Etienne, CEA-Tech, Departement SAS, F - 13541 Gardanne France
lina.mortajine@gmail.com
nadia.el-mrabet@emse.fr
[4] DGA - Maîtrise de l'Information, Bruz, France
BP7, 35998 Rennes Cedex 9, France
tania.richmond.nc@gmail.com

**Abstract.** ROLLO was a candidate to the second round of the NIST Post-Quantum Cryptography standardization process. In the last update in April 2020, there was a key encapsulation mechanism (ROLLO-I) and a public-key encryption scheme (ROLLO-II). In this paper, we propose an attack to recover the syndrome during the decapsulation process of ROLLO-I. From this syndrome, we explain how to recover the private key. We target two constant-time implementations: the C reference implementation and a C implementation available on *GitHub*. By getting power measurements during the execution of the Gaussian elimination function, we are able to extract on a single trace each element of the syndrome. This attack can also be applied to the decryption process of ROLLO-II.

**Keywords:** Side-Channel Attack, Power Consumption Analysis, ROLLO, Key-Recovery Attack, Rank Metric, LRPC Codes

## 1 Introduction

Nowadays number theory based cryptography, like RSA [11] or ECDSA [9], is efficient but weak against the Shor's quantum algorithm [13]. The existence of quantum algorithms pushed the National Institute of Standards and Technology (NIST) to started in 2016 the Post-Quantum Cryptography (PQC) standardization process to get signatures and, key-encapsulation mechanisms (KEM) or public-key encryption schemes (PKE), resisting to both classical and quantum attacks. In this work we focus on ROLLO, a code-based candidate in rank metric. Even if ROLLO has not been selected for the third round due to recent algebraic attacks [5, 6], NIST encouraged the community to study rank-metric

cryptosystems. They seem to be a good alternative to cryptosystems in Hamming metric, but were not studied enough at that point regarding side-channel analysis and embedded implementations. Indeed, public-key cryptosystems are commonly used in embedded systems. Thus it is essential to identify potential leakage to improve their resistance against side-channel attacks and ensure their security in practice. Kocher introduced side-channel attacks in 1996 [10]. An attacker can use information provided by a side-channel to extract secret data from a device executing a cryptographic primitive. The information leakage is exploited without having to tamper with the device. Two recent papers related to side-channel attacks on code-based cryptography in rank metric have been published [4, 12]. Both exploit timing leakage from the decoding failure rate of LRPC codes [8]. In this work, we focus on constant-time implementations of schemes using LRPC codes. We target two constant-time implementations of ROLLO, in particular the Gaussian elimination function for LRPC decoding. The first implementation is provided by the authors of ROLLO's proposal to NIST standardization process [2]. The second one only provides an implementation of ROLLO-I for 128 bits of security [1].

*Our contribution.* To the best of our knowledge, this is the first single-trace attack against different versions of the constant-time Gaussian elimination for error-correcting codes in rank metric. We show that the power consumption during the decapsulation/decryption process can provide enough information to make an efficient key-recovery attack on ROLLO schemes. Our attack allows us to recover various secret data such as:

- the private key in both ROLLO-I and -II via the syndrome recovery,
- the shared secret in ROLLO-I key-encapsulation mechanism, or the encrypted message in ROLLO-II public-key encryption.

We finally present two countermeasures to make the implementations resistant to the proposed attack.

*Organization.* In Section 2, we recall background on error-correcting codes in rank metric as well as ROLLO schemes. In Section 3, we explain our attack on the reference implementation using *rbc_ library*. We also provide a glimpse of countermeasures in Section 4. Finally we conclude in Section 5.

## 2 Background

ROLLO's submission is based on ideal Low-Rank Parity-Check (LRPC) codes. The latter were introduced in 2013 [8]. In this section, we briefly explain ideal LRPC codes, then recall the ROLLO proposal to NIST PQC standardization process.

### 2.1 Rank-metric codes

In the following, we denote by $q$ a power of a prime number, and let $m$, $n$, and, $k$ be positive integers such that $n > k$.

We also consider the isomorphism between the vector space $\mathbb{F}_{q^m}^n$ and the extension field $\mathbb{F}_{q^m}[Z]/(P_n)$ given by

$$\begin{aligned} \phi : \mathbb{F}_{q^m}^n &\to \mathbb{F}_{q^m}[Z]/(P_n) \\ (x_1, \ldots, x_n) &\mapsto \sum_{i=1}^{n} x_i Z^i \end{aligned}$$

with $P_n$ an irreducible polynomial of degree $n$ and $(P_n)$ the ideal of $\mathbb{F}_{q^m}[Z]$ generated by $P_n$. Note that the vector space $\mathbb{F}_{q^m}$ is isomorphic to $\mathbb{F}_q[z]/(P_m)$, with $P_m$ an irreducible polynomial of degree $m$ over $\mathbb{F}_q$.
Let us define a linear code $\mathcal{C}$ over $\mathbb{F}_{q^m}$ of length $n$ and dimension $k$ represented by its parity-check matrix $\mathbf{H} \in \mathbb{F}_{q^m}^{(n-k)\times n}$.
Since a codeword $\mathbf{x}$ is in $\mathbb{F}_{q^m}^n$, each of its coordinates $x_i$, for $1 \leq i \leq n$, can be associated to a vector $(x_{i,1}, \ldots, x_{i,m})$ in $\mathbb{F}_q^m$. Thus an element $\mathbf{x} \in \mathbb{F}_{q^m}^n$ can also be represented by a matrix as follows:

$$M(\mathbf{x}) = (x_{i,j})_{\substack{1\leq i\leq n \\ 1\leq j\leq m}} \in \mathbb{F}_q^{n\times m}.$$

For an element $\mathbf{x} \in \mathbb{F}_{q^m}^n$, the syndrome of $\mathbf{x}$ is defined as the vector $\mathbf{s} = \mathbf{H} \cdot \mathbf{x}^T$.
Considering the rank metric, the distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{F}_{q^m}^n$ is defined by

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

with $v = x - y$.
The support of a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$ is defined as the subset of $\mathbb{F}_{q^m}$ spanned over $\mathbb{F}_q$. Namely, the support of $\mathbf{x}$ is given by

$$\operatorname{Supp}(\mathbf{x}) = \langle x_1, \ldots, x_n \rangle_{\mathbb{F}_q}.$$

W.l.o.g., the support of $(\mathbf{x}, \mathbf{y})$ is $\operatorname{Supp}(\mathbf{x}, \mathbf{y}) = \langle x_1, \ldots, x_n, y_1, \ldots, y_n \rangle_{\mathbb{F}_q}$. The ideal LRPC codes base their structure on ideal codes.

Given a polynomial $P_n \in \mathbb{F}_q[Z]$ of degree $n$ and a vector $\mathbf{v} \in \mathbb{F}_{q^m}^n$, an ideal matrix generated by $\mathbf{v}$ is a $n \times n$ matrix defined by

$$\mathcal{IM}(\mathbf{v}) = \begin{pmatrix} \mathbf{v}(Z) & \bmod P_n \\ Z \cdot \mathbf{v}(Z) & \bmod P_n \\ \vdots & \\ Z^{n-1} \cdot \mathbf{v}(Z) & \bmod P_n \end{pmatrix}.$$

The authors of ROLLO restrained the definition of ideal LRPC (Low-Rank Parity Check) codes to $(2, 1)$-ideal LRPC codes for all variants of ROLLO [2].
Let $F$ be a $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ such that $dim(F) = d$. Let $(\mathbf{h}_1, \mathbf{h}_2)$ be a pair of two vectors in $\mathbb{F}_{q^m}^n$, such that $\operatorname{Supp}(\mathbf{h}_1, \mathbf{h}_2) = F$, and $P_n \in \mathbb{F}_q[Z]$ be a polynomial of degree $n$. A $[2n, n]_{q^m}$-code $\mathcal{C}$ is an ideal LRPC code if it has a parity-check matrix of the form

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

### 2.2 ROLLO

ROLLO is a second round submission to the post-quantum cryptography standardization process launched by NIST in 2016. Since the last update in April 2020, ROLLO is composed of two cryptosystems: ROLLO-I, a Key-Encapsulation Mechanism (KEM), and ROLLO-II, a Public-Key Encryption (PKE). Both are described in Figure 1. We use the following notations:

- $A \xleftarrow{\$} \mathbb{F}_{q^m}^k$ denotes the operation of selecting randomly $k$ vectors from the vector space $\mathbb{F}_{q^m}^k$, then $A \in \mathbb{F}_{q^m}^k$.
- $(\mathbf{u}, \mathbf{v}) \xleftarrow{\$}_l A$ denotes the operation of selecting randomly $2n$ linear combinations from the element $A$, then $\mathbf{u}, \mathbf{v} \in \mathbb{F}_{q^m}^n$ and $Supp(\mathbf{u}, \mathbf{v}) = A$.
- RSR denotes the Rank Support Recovery algorithm given in the specification of ROLLO to decode LRPC codes [2].

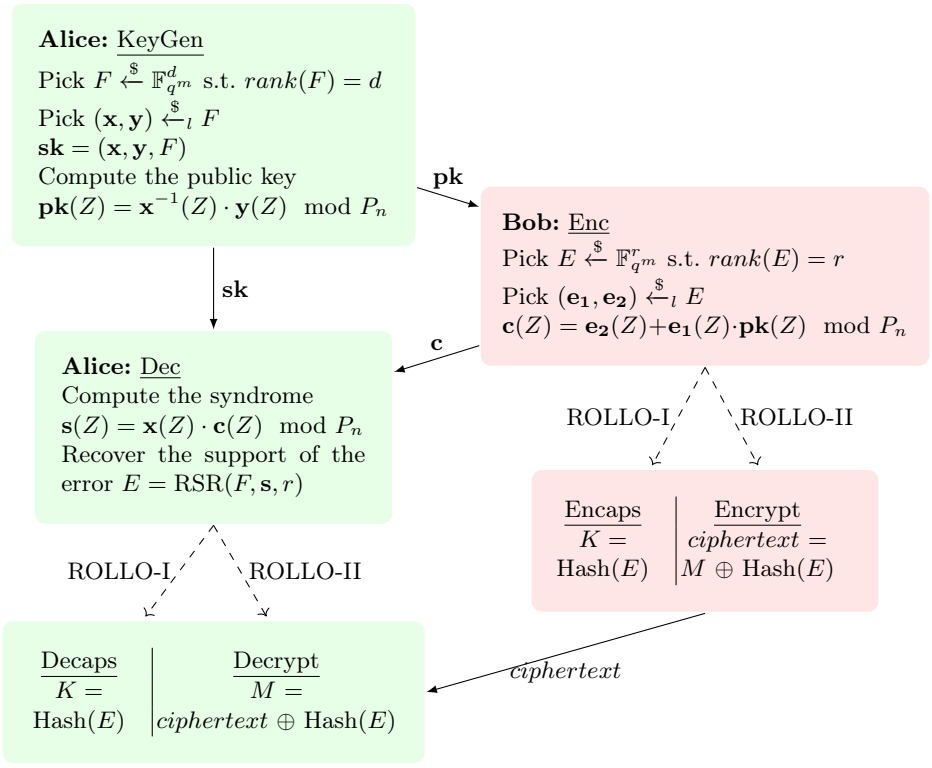


Fig. 1: ROLLO-I (KEM) and ROLLO-II (PKE) cryptosystems.

In the following, we focus on the vulnerabilities of implementations of Gaussian elimination process in the case of ROLLO.

## 3 Side-channel attack on Gaussian elimination in constant-time

In the RSR algorithm, we first compute the support of the syndrome [2]. The Gaussian elimination is applied to the syndrome matrix $\mathbf{S} = M(\mathbf{s}) \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ to compute its support. We know that the syndrome is first computed as:

$$\mathbf{s}(Z) = \mathbf{x}(Z) \cdot \mathbf{c}(Z) \mod P_n,$$

with $\mathbf{x}, \mathbf{c}, \mathbf{s} \in \mathbb{F}_{q^m}[Z]/(P_n)$. Therefore, with the knowledge of the syndrome $\mathbf{s}$ and the ciphertext $\mathbf{c}$, we can recover $\mathbf{x}$, a part of the private key as:

$$\mathbf{x}(Z) = \mathbf{s}(Z) \times \mathbf{c}(Z)^{-1} \mod P_n.$$

Knowing $\mathbf{x}$ can lead to a full recovery of the private key. First, we can get the second part of the private key $\mathbf{y}$ by computing

$$\mathbf{y}(Z) = \mathbf{pk}(Z) \times \mathbf{x}(Z) \mod P_n.$$

Then, the support of $\mathbf{y}$ and $\mathbf{x}$ gives the last part of the private key $F$.

The Gaussian elimination in constant-time requires to process each row in each column of the syndrome matrix. Thus, an attacker could be able to recover all values in this matrix. In case of a non constant-time Gaussian elimination, it is possible to treat only the rows under the pivot row. Therefore, the values in all rows above the pivot row remain unknown to the attacker. Consequently, constant-time provides an advantage to a side-channel attacker.
Secondly, the constant-time eases the detection of a pattern corresponding to the targeted operation inside the power trace. Once the attacker found the exact location of this pattern, it becomes straightforward to find the locations for each other iteration.
We analyzed two constant-time implementations of Gaussian elimination and discovered two possible leakages through power consumption. The first one has been provided as *Additionnal Implementations* in April 2020 for the second round of NIST PQC standardization process, and is available on the ROLLO candidate webpage [2]. We refer to it as the reference implementation. It uses the *rbc_ library*, which provides different functions to implement schemes using rank-metric codes [3]. The second implementation has been published on GitHub [1]. We refer to it as the *GitHub* implementation.
In this extended abstract, we only present the attack against the reference implementation.

*Notations.* We denote by $\otimes$ the multiplication between a scalar and a row of a matrix and by $\oplus$ the bitwise XOR between two bits or two rows of a matrix. The bitwise AND is represented by $\wedge$ and the bitwise NOT by $\neg$. The term *mask* does not refer to a boolean masking but to a variable giving the additions on rows according to values obtained from coefficients of the processed column.

### 3.1 Information leakage of the reference implementation

The reference implementation is based on Algorithm 1, which was first introduced in [7].

**Algorithm 1:** Gaussian elimination in constant time

**Input:** $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
**Output:** $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ in systematic form and $rank = min(dimension, n)$

```
1  dimension = 0
2  for j = 0, ..., m − 1 do
3      pivot_row = min(dimension, n − 1)
4      for i = 0, ..., n − 1 do
5          mask = s_{pivot_row,j} ⊕ s_{i,j}
6          tmp = mask ⊗ s_i
7          if i > pivot_row then
8              s_{pivot_row} = s_{pivot_row} ⊕ tmp
9          else
10             dummy = s_{pivot_row} ⊕ tmp
11     for i = 0, ..., n − 1 do
12         if i ≠ j then
13             mask = s_{i,j}
14             tmp = mask ⊗ s_{pivot_row}
15             if dimension < n then
16                 s_i = s_i ⊕ tmp
17             else
18                 dummy = s_i ⊕ tmp
19     dimension = dimension + s_{pivot_row,i}
```

The input matrix is composed of $n$ rows and $m$ columns. The algorithm outputs the matrix in systematic form and its rank. The first inner **for** loop (line 4) fixes the ones in the diagonal (corresponding to the pivots) and the second inner **for** loop (line 13) removes the ones in the pivot column. In both inner **for** loops in Algorithm 1, $mask \in \mathbb{F}_2$ is computed and multiplied with specific rows of the syndrome matrix. However, the multiplication of a 32-bit word $(u_0, \ldots, u_{31})_2$ with zero or one provides information leakage in the power traces. This allows us to recover all the $mask$ values computed during the process, then, the initial syndrome matrix.
Our attack consists in recovering the syndrome matrix

$$\mathbf{S} = \begin{array}{c} \xrightarrow{\qquad\qquad m \qquad\qquad} \\ \left\downarrow \vphantom{\begin{pmatrix} a \\ a \\ a \\ a \end{pmatrix}} \right. n \end{array} \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix}, \tag{1}$$

where $s_{i,j} \in \mathbb{F}_2$ for $(i,j) \in [\![0, n-1]\!] \times [\![0, m-1]\!]$. We denote by $\mathbf{S}_j$ the matrix obtained after the treatment of the $j$-th column of $\mathbf{S}$ and by $\mathbf{S}_j[k]$, the $k$-th

column of the matrix $\mathbf{S}_j$. The recovered *mask* values from the two inner **for** loops lead to a system of linear equations. This system is obtained from two steps described below.

*After the first inner **for** loop in Algorithm 1:* we recover the *mask* values $s_{pivot\_row,j} \oplus s_{i,j}$. If $mask = 0$, then the pivot row is unchanged. Otherwise, the $i$-th *row* is added to the pivot row. Then, the first loop provides the indices of rows XORed to the pivot row. We define

$$\sigma_j = (\sigma_{0,j}, \sigma_{1,j}, \ldots, \sigma_{n-1,j}), \quad \text{where } \sigma_{i,j} = \begin{cases} 0 & \text{if } mask = 0 \\ 1 & \text{if } mask = 1 \end{cases},$$

the vector containing all *mask* values recovered after the $j$-th iteration. We also define the matrix

$$J_k = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ \sigma_{0,k} & \cdots & 1 & \cdots & \sigma_{n-1,k} \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ \longleftarrow k\text{-th row} \\ \\ \\ \end{matrix},$$

$$\uparrow$$

$$k\text{-th column}$$

involved in the computation of the system of linear equations. For instance, let us consider the pivot row of index 0. After the first inner **for** loop, the syndrome matrix given in Equation 1 is under the following form

$$\begin{pmatrix} \sum_{i=0}^{n-1} \sigma_{i,0} s_{i,0} & \sum_{i=0}^{n-1} \sigma_{i,0} s_{i,1} & \cdots & \sum_{i=0}^{n-1} \sigma_{i,0} s_{i,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix}.$$

We notice in lines $7-8$ in Algorithm 1 that only rows with index greater than the pivot row index are added to the pivot row. Thus, after the treatment of the column $j$, we define $\sigma_{i,j} = 0$ for $i \leq pivot\_row$.

*After the second inner **for** loop in Algorithm 1:* the recovered *mask* values correspond to the coefficients $s_{i,j}$ of the matrix obtained after the first inner **for** loop. We denote by $\sigma'_j = (\sigma'_{0,j}, \ldots, \sigma'_{j-1,j}, *, \sigma'_{j+1,j}, \ldots, \sigma'_{n-1,j})$ the vector composed of *mask* values. The item $*$ represents the pivot that is not processed in the second loop. For the attack, $*$ is replaced by one.
On one hand, during the treatment of the $j$-th column, $\sigma'_j$ completes the system of linear equations. Assuming we want to recover the column 0, we use a linear solver on the system

$$J_0 \times \mathbf{S}[0] = (\sigma'_0)^t.$$

On the other hand, the vector $\sigma'_j$ allows us to recover all the operations performed on rows. These operations are taken into account in solving the system of linear equations of the $(j+1)$-th column. For this, we define the matrix

$$J'_k = \begin{pmatrix} 1 & \cdots & \sigma'_{0,k} & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'_{n-1,k} & \cdots & 1 \end{pmatrix} \begin{array}{l} \\ \\ \longleftarrow k\text{-th row} \\ \\ \\ \end{array} .$$

$$\uparrow$$

$$k\text{-th column}$$

Thus, during the treatment of the column $j$, for $j \geq 1$, we consider

$$\mathbf{S}_{j-1} = \left( \prod_{k=j-1,\ldots,0} J'_k \times J_k \right) \times \mathbf{S}.$$

In case there is no pivot in a column, all the *mask* values are equal to zero, thus

$$J'_k \times J_k = I_n.$$

Finally, to recover the column $j$, we solve the system of linear equations

$$J_{j-1} \times \left( \prod_{k=j-2,\ldots,0} J'_k \times J_k \right) \times \mathbf{S}[j-1] = (\sigma'_{j-1})^t.$$

*Experimental results.* An example of a power trace obtained after the execution of the Gaussian elimination on an ARM SecurCore SC300 32-bit processor (equivalent to Cortex-M3) is given in Figure 2. For a better understanding, we highlight each occurrence in the computation of the different masks. We can observe the difference of power consumption when 32-bit words are multiplied either by one or by zero. The difference of pattern leads us to recover the *mask* values of the two inner **for** loops. We also performed this experiment on a Cortex-M4 microcontroller.
The attack presented in this section has also been adapted to the *GitHub* implementation [1].

## 4 Countermeasures

We propose two solutions to protect the future implementations against our attack. In this section, we just detail one of them and give the general idea for the

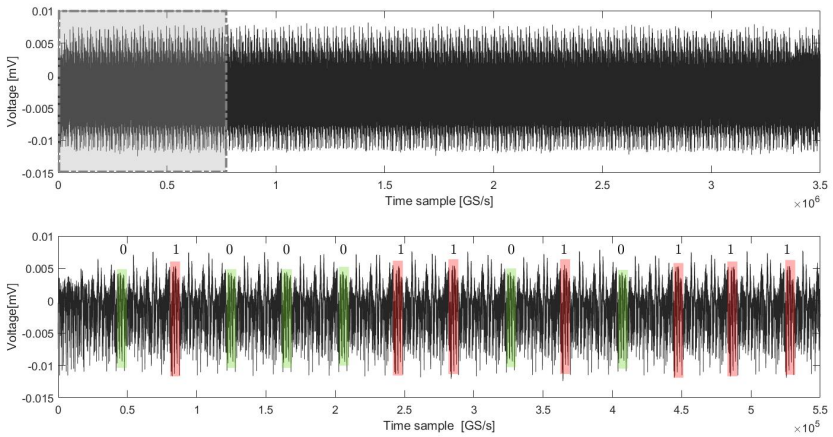


Fig. 2: Full trace and a zoom of the first inner loop

second one.
The first countermeasure consists in reducing the distinguisher between a multiplication of a word by zero or by one. Therefore we mask the processed coefficients. In the first inner **for** loop, we split the pivot row into two parts. Thus, for each iteration we compute

$$s_{pivot\_row} = s_{1pivot\_row} \oplus s_{2pivot\_row},$$

with $s_{1pivot\_row}, s_{2pivot\_row} \in \mathbb{F}_{2^m}$. The variable $tmp$ (line 6 - Algorithm 1) is then computed as

$$tmp = (mask\prime \wedge (s_i \oplus s_{2pivot\_row})) \vee (\neg mask\prime \wedge s_{2pivot\_row}),$$

with $mask\prime = \neg(mask - 1)$. Then, we can update the pivot row by computing

$$s_{pivot\_row} = s_{1pivot\_row} \oplus tmp.$$

If $i \leq pivot\_row$, we have

$$dummy = s_{1pivot\_row} \oplus tmp.$$

The same operations are performed in the second inner **for** loop by replacing the pivot row by the processed row $s_i$. With this countermeasure, whether the mask is zero or one, we always perform the same operations, namely two bitwise ANDs between non-zero and zero words. Thus, we are not able to distinguish different patterns when $mask$ equals 0 or 1. It is important to emphasize that the implementations with this countermeasure remain in constant-time.

The second countermeasure is based on shuffling. The treatment of each column is performed randomly by using an algorithm generating a random permutation of a finite set, such as the Fisher-Yates method. The choice is left to the

developer under condition of a proper implementation. With the randomization countermeasure, an attacker can distinguish patterns related to the masks values for both implementations, but not determine the order of elements. Moreover, a brute-force attack is not achievable. Indeed, an adversary has $n!$ possibilities for each column, which implies a total of $(n!)^m$ possibilities to recover the whole syndrome matrix. For instance, with ROLLO-I-128 parameters, the complexity is approximately $2^{27731}$. Thus, only the number of zeros and ones on the matrix would be known.

## 5 Conclusion and perspectives

We show in this paper that constant-time implementation of Gaussian elimination provided in [2] is sensitive to power consumption attacks. We exploit the weakness introduced by the variable *mask* to avoid previous timing attacks. This information leakage allows us to make the first attack by power consumption on the last implementation version given by the authors of ROLLO. We can also apply our side-channel attack on another implementation of ROLLO-I-128 [1]. These attacks can lead to a full key-recovery using one single trace. To secure the implementations, we propose two different countermeasures. The first one can be applied to [2] by hiding the values of *mask*. The second countermeasure can be applied to both implementations. The idea is to treat each row in a column of the matrix randomly. It adds randomness which makes our attack not exploitable in practice anymore. We base our work on traces got from Cortex-M3 and Cortex-M4 microcontrollers. The constant-time Gaussian elimination function is in the *rbc_library* library. This library is also used in the implementation of the RQC scheme. Even though the Gaussian elimination in constant time is not used in the RQC implementation, the entire library should be analyzed to find possible leakage. In particular, we want to analyze the Karatsuba function used in both ROLLO implementation and the polynomial multiplication for computation over ideal codes in RQC. Another perspective could be to analyze the various implementations of the Gaussian elimination in the third round candidates to the NIST PQC standardization process.

## References


[1] C. Aguilar-Melchor et al. *Constant time algorithms for ROLLO-I-128*. `https://eprint.iacr.org/2020/1066.pdf`. Source code available at `https://github.com/peacker/constant_time_rollo.git`. 2020.

[2] C. Aguilar-Melchor et al. *ROLLO-Rank-Ouroboros, LAKE & LOCKER*. 2019. URL: `https://pqc-rollo.org/`.

[3] N. Aragon and L. Bidoux. *rbc_library*. `https://rbc-lib.org/`. 2020.

[4] N. Aragon and P. Gaborit. "A key recovery attack against LRPC using decryption failures". In: *International Workshop on Coding and Cryptography (WCC), Saint-Jacut-de-la-Mer, France*. 2019.

[5] M. Bardet et al. "An Algebraic Attack on Rank Metric Code-Based Cryptosystems". In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by A. Canteaut and Y. Ishai. Cham: Springer International Publishing, 2020, pp. 64–93. ISBN: 978-3-030-45727-3.

[6] M. Bardet et al. "Improvements of Algebraic Attacks for Solving the Rank Decoding and MinRank Problems". In: *Advances in Cryptology – ASIACRYPT 2020*. Ed. by S. Moriai and H. Wang. Cham: Springer International Publishing, 2020, pp. 507–536. ISBN: 978-3-030-64837-4.

[7] D. J. Bernstein, T. Chou, and P. Schwabe. "McBits: Fast Constant-Time Code-Based Cryptography". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by G. Bertoni and J.-S. Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–272. ISBN: 978-3-642-40349-1.

[8] P. Gaborit et al. "Low Rank Parity Check codes and their application to cryptography". In: *International Workshop on Coding and Cryptography (WCC)*. Ed. by L. Budaghyan, T. Helleseth, and M. G. Parker. ISBN 978-82-308-2269-2. Bergen, Norway, 2013. URL: https://hal.archives-ouvertes.fr/hal-00913719.

[9] D. Johnson, A. Menezes, and S. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (2001), pp. 36–63. DOI: 10.1007/s102070100002.

[10] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology – CRYPTO*. Ed. by N. Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[11] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[12] S. Samardjiska et al. "A Reaction Attack Against Cryptosystems Based on LRPC Codes". In: *Progress in Cryptology – LATINCRYPT*. Springer International Publishing, 2019, pp. 197–216. DOI: 10.1007/978-3-030-30530-7_10.

[13] P. W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509.