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
263 Avenue Général Leclerc, 35042 Rennes Cedex, France
`agathe.cheriere@irisa.fr`
[2] Wisekey Semiconductors, Arteparc de Bachasson,
Bâtiment A, 13590 Meyreuil, France
[3] Mines Saint-Etienne, CEA-Tech, Departement SAS,
F - 13541 Gardanne France
`{lina.mortajine,nadia.el-mrabet}@emse.fr`
[4] DGA - Maîtrise de l'Information,
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 perform a private key-recovery. 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 [18] or ECDSA [11], is efficient but weak against the Shor's quantum algorithm [20]. The existence of quantum algorithms pushed the National Institute of Standards and Technology (NIST) to anticipate the time when an efficient quantum computer will be able to execute these algorithms and break commonly used public-key cryptography. In late 2016, NIST started 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. Among the historical schemes, as McEliece [14] or NTRU [10], there were recent proposals based on rank metric. Error-correcting codes in rank metric allow to reduce some drawbacks of Hamming metric, like the key-sizes. In the second round of this standardization process, there were two proposals in rank metric, namely ROLLO [2] and RQC [15]. Both were not selected for the third round due to some algebraic attacks [5, 6]. Nonetheless, NIST encouraged the community to study rank metric cryptosystems. "NIST believes rank-based cryptography should continue to be researched" in [16]. 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 [12]. 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. The first side-channel attack against a code-based cryptosystem was proposed in 2008 for McEliece in Hamming metric [21]. It was then followed by numerous others in more than a decade of research, with timing or power consumption attacks. More recently, there were two papers combining physical attacks with algebraic properties [8, 13]. We do not detail more those attacks since they are out of scope.

Related work. Two recent papers related to side-channel attacks on code-based cryptography in rank metric have been published [4, 19]. Both exploit timing leakage from the decoding failure rate of LRPC codes [9]. In this work, we focus on constant-time implementations of schemes using LRPC codes. We target two constant-time implementations of ROLLO, and in particular the Gaussian elimination function. The first one is provided by the authors of ROLLO's proposal to NIST [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 attack on ROLLO schemes. Our attack allow us to recover various secret data such as:

– the private key in both cryptosystems 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. Gaussian elimination is often used in coding theory to go from a dense parity-check matrix to a parity-check matrix in systematic form. With this work we want to point out that even recent implementations of this operation could be vulnerable to side-channel attacks. For instance, Gaussian elimination in constant-time is also used in Classic McEliece. Nonetheless the implementation differs from the one in ROLLO, and we have not investigated the leakage detection on Classic McEliece yet. But in case we are able to detect differences in power consumption for some values as presented in this paper, we could also apply our attack on this scheme to recover the parity-check matrix.

Organization of the paper. In Section 2, we recall elementary notions of error-correcting codes in rank metric as well as ROLLO schemes. In Section 3, we detail attacks on both implementations: the reference one using *rbc_library* and the proposal on *GitHub*. We also provide some experimental results for ROLLO-I-128. We discuss two different countermeasures in Section 4. Finally, we conclude this paper in Section 5.

2 Background

ROLLO's submission is based on ideal Low-Rank Parity-Check (LRPC) codes. The latter were introduced in 2013 [9]. In this section, we first give some details on ideal LRPC codes, then recall the ROLLO proposal to NIST PQC standardization process.

2.1 Rank metric codes

In the following sections, we denote by q a power of a prime number, and let m, n, and, k be positive integers such that $n > k$.
A linear code $\mathcal{C}$ over $\mathbb{F}_{q^m}$ of length n and dimension k is a subspace of $\mathbb{F}_{q^m}^n$. It is denoted by $[n,k]_{q^m}$, and can be represented by a parity-check matrix $\mathbf{H} \in \mathbb{F}_{q^m}^{(n-k)\times n}$ such that

$$\mathcal{C} = \{\mathbf{x} \in \mathbb{F}_{q^m}^n, \mathbf{H}.\mathbf{x}^T = 0\}.$$

An element $\mathbf{x} = (x_1, \ldots, x_n) \in \mathcal{C}$ is called a codeword. Since $\mathbf{x} \in \mathbb{F}_{q^m}^n$, each of its coordinate x_i, for $1 \leq i \leq n$, can be associated to a vector $(x_{i,1}, \ldots, x_{i,m})$. 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 $\mathbf{s} = \mathbf{H}.\mathbf{x}^T$. Considering the rank metric, the distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{F}_{q^m}^n$ is defined by

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

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 by the basis of $\mathbf{x}$. Namely, the support of $\mathbf{x}$ is given by

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

The ideal LRPC codes base their structure on ideal codes. Before defining them, it is important to note that there exists an 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. Let us also 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 $\mathbb{F}_q$.

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

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

An $[ns, nt]_{q^m}$-code $\mathcal{C}$, generated by the vectors $(\mathbf{g}_{i,j})_{\substack{i\in[1,\ldots,s-t]\\ j\in[1,\ldots,t]}} \in \mathbb{F}_{q^m}^n$, is an ideal code if a generator matrix in systematic form is of the form

$$\mathbf{G} = \begin{pmatrix} & \mathcal{IM}(\mathbf{g_{1,1}}) & \cdots & \mathcal{IM}(\mathbf{g_{1,s-t}}) \\ \mathbf{I}_{nt} & \vdots & \ddots & \vdots \\ & \mathcal{IM}(\mathbf{g_{t,1}}) & \cdots & \mathcal{IM}(\mathbf{g_{t,s-t}}) \end{pmatrix}.$$

In [2], the authors restrain the definition of ideal LRPC (Low-Rank Parity Check) codes to $(2,1)$-ideal LRPC codes that they used for all variants of ROLLO.
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 $\mathrm{Supp}(\mathbf{h}_1, \mathbf{h}_2) = F$, 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

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

To decode the LRPC codes, they use the Rank Support Recovery (RSR) algorithm described in ROLLO submission [2] and recalled in Appendix A.

2.2 ROLLO

ROLLO is a second round submission to the post-quantum standardization process launched by NIST in 2016. Since the last update in April 2020, it 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:

- $A \overset{\$}{\leftarrow} \mathbb{F}_{q^m}^k$ denotes the operation of selecting randomly k vectors from the vector space $\mathbb{F}_{q^m}^k$, then $A \in \mathbb{F}_{q^m}^k$.
- $(\mathbf{u}, \mathbf{v}) \overset{\$}{\leftarrow}_l A$ 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 Appendix A.

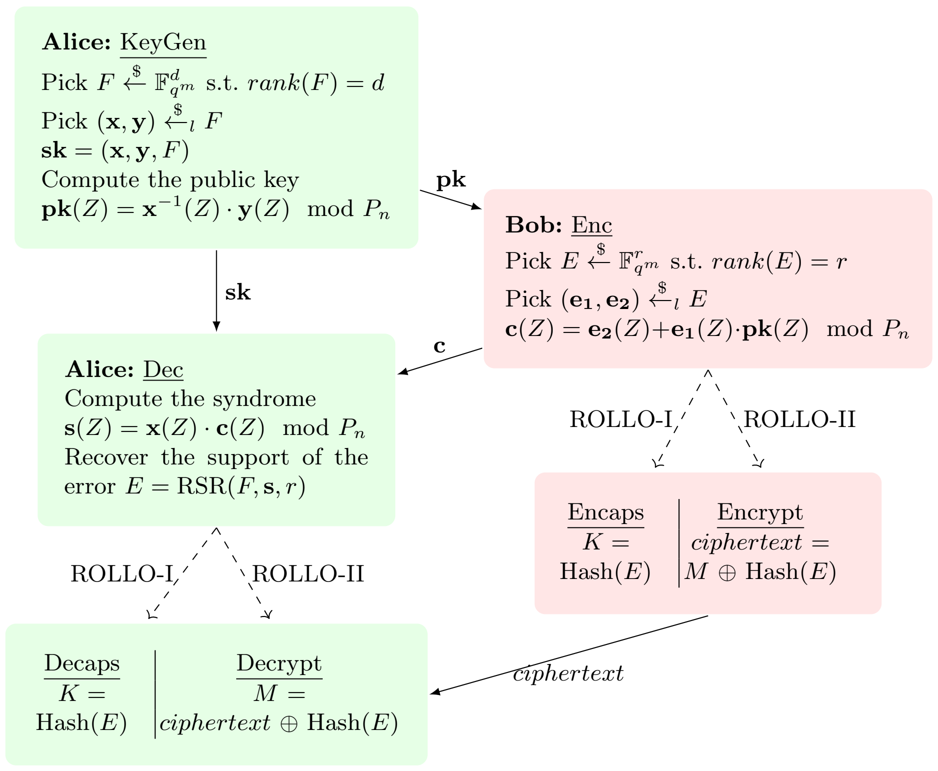

Alice: KeyGen
Pick $F \overset{\$}{\leftarrow} \mathbb{F}_{q^m}^d$ s.t. $rank(F) = d$
Pick $(\mathbf{x}, \mathbf{y}) \overset{\$}{\leftarrow}_l F$
$\mathbf{sk} = (\mathbf{x}, \mathbf{y}, F)$
Compute the public key
$\mathbf{pk}(Z) = \mathbf{x}^{-1}(Z) \cdot \mathbf{y}(Z) \mod P_n$
pk
sk
Bob: Enc
Pick $E \overset{\$}{\leftarrow} \mathbb{F}_{q^m}^r$ s.t. $rank(E) = r$
Pick $(\mathbf{e_1}, \mathbf{e_2}) \overset{\$}{\leftarrow}_l E$
$\mathbf{c}(Z) = \mathbf{e_2}(Z) + \mathbf{e_1}(Z) \cdot \mathbf{pk}(Z) \mod P_n$
c
Alice: Dec
Compute the syndrome
$\mathbf{s}(Z) = \mathbf{x}(Z) \cdot \mathbf{c}(Z) \mod P_n$
Recover the support of the error $E = \mathrm{RSR}(F, \mathbf{s}, r)$
ROLLO-I
ROLLO-II
Encaps
$K =$ Hash(E)
Encrypt
$ciphertext =$ $M \oplus$ Hash(E)
ROLLO-I
ROLLO-II
Decaps
$K =$ Hash(E)
Decrypt
$M =$ $ciphertext \oplus$ Hash(E)
$ciphertext$

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

Instance	d	r	P_n	P_m
ROLLO-I-128	8	7	$Z^{83} + Z^7 + Z^4 + Z^2 + 1$	$z^{67} + z^5 + z^2 + z + 1$
ROLLO-I-192	8	8	$Z^{97} + Z^6 + 1$	$z^{79} + z^9 + 1$
ROLLO-I-256	9	9	$Z^{113} + Z^9 + 1$	$z^{97} + z^6 + 1$
ROLLO-II-128	8	7	$Z^{189} + Z^6 + Z^5 + Z^2 + 1$	$z^{83} + z^7 + z^4 + z^2 + 1$
ROLLO-II-192	8	8	$Z^{193} + Z^{15} + 1$	$z^{97} + z^6 + 1$
ROLLO-II-256	9	8	$Z^{211} + Z^{11} + Z^2 10 + Z^8 + 1$	$z^{97} + z^6 + 1$

Table 1: ROLLO's parameters for each security level.

We unify tables of parameters from ROLLO's specification into Table 1. For the three security levels, $q = 2$. The name of each variant gives the targeted classical security level, e.g. ROLLO-I-128 is a classical 128-bit security level. The parameters d and r correspond respectively to the rank of the private key and the rank of the errors. The parameters n and m can respectively be obtained with the degrees of P_n and P_m.
In the following, we will focus on the vulnerabilities of the implementations of Gaussian elimination process. The latter is used several times in ROLLO cryptosystems, namely to compute:

- the support S of the syndrome $\mathbf{s}$
- the support of the error $(\mathbf{e}_1, \mathbf{e}_2)$ letting us recover the shared secret in the case of ROLLO-I or encrypt/decrypt a message in the case of ROLLO-II ;
- the intersections of two vector spaces during the decoding of the syndrome (Appendix A-RSR). These intersections determine the support E of the error:
$$E \leftarrow \bigcap_{1 \le i \le d} f_i^{-1} \cdot S,$$
with $F = \langle f_1, \dots, f_d \rangle$ the support of the private key.

Thus, the leakage coming from implementations of Gaussian elimination can allow a side-channel attacker to recover all the secret data. In the next section, we explain the attack on the syndrome. This analysis can be performed to recover the other mentioned data.

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

In the RSR algorithm (see Appendix A), we first compute the support of the syndrome. For that, the Gaussian elimination is applied to the syndrome matrix $\mathbf{S} = M(\mathbf{s}) \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ to calculate 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 compute $\mathbf{x}$, 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* [3], which provides different functions to implement schemes using rank metric codes. The second implementation has been published on GitHub [1]. We refer to it as the *GitHub* 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].
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

$$\mathbf{S} = \begin{array}{cc} & \xrightarrow{\qquad\qquad m \qquad\qquad} \\ n \Bigg\downarrow & \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix} \end{array}, \tag{1}$$

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 $j = 0, \ldots, m-1$ do
3       $pivot\_row = min(dimension, n-1)$
4       for $i = 0, \ldots, n-1$ do
5           $mask = s_{pivot\_row,j} \oplus s_{i,j}$
6           $tmp = mask \otimes s_i$
7           if $i > pivot\_row$ then
8               $s_{pivot\_row} = s_{pivot\_row} \oplus tmp$
9           else
10              $dummy = s_{pivot\_row} \oplus tmp$
11      for $i = 0, \ldots, n-1$ do
12          if $i \neq j$ then
13              $mask = s_{i,j}$
14              $tmp = mask \otimes s_{pivot\_row}$
15              if $dimension < n$ then
16                  $s_i = s_i \oplus tmp$
17              else
18                  $dummy = s_i \oplus tmp$
19      $dimension = dimension + s_{pivot\_row,i}$
```

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

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

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

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

In other words, we can compute it as

$$J_0 \times \mathbf{S} = \left(\begin{array}{c|c} 1 & \sigma_{1,0} \cdots \sigma_{n-1,0} \\ \hline \mathbf{0} & I_{n-1} \end{array}\right) \times \mathbf{S},$$

where I_{n-1} denotes the identity matrix of size $n-1$ and $\mathbf{0}$ a column of $n-1$ zeros.

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, σ'_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 σ'_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

$$J'_k = \begin{array}{cc} \left(\begin{array}{ccccc} 1 & \cdots & \sigma'_{0,k} & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'_{n-1,k} & \cdots & 1 \end{array}\right) & \begin{array}{l} \\ \\ \longleftarrow k\text{-th row} \\ \\ \\ \end{array} \\ \begin{array}{c} \uparrow \\ k\text{-th column} \end{array} & \end{array} .$$

For example, for the treatment of column 1 we consider the matrix

$$\mathbf{S}_0 = \underbrace{\left((\sigma'_0)^t \left| \begin{array}{c} \mathbf{0} \\ \hline I_{n-1} \end{array} \right. \right)}_{=J'_0} \times J_0 \times \mathbf{S}.$$

More generally, 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

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

3.2 Information leakage of the *GitHub* implementation

In this section, we denote by $\mathbf{1} = \underbrace{(11\ldots11)}_{m}$ and $\mathbf{0} = \underbrace{(00\ldots00)}_{m}$.

In [1], the authors introduced a row reduction in constant-time given in Algorithm 2, that can be seen as a generalization of the one presented in Algorithm 1. At the end of Algorithm 2, we obtain a matrix under the row echelon form. In order to ensure this, three masks are first computed according to coefficients and pivot processed. Each mask is equal to $\mathbf{1}$ or $\mathbf{0}$. The three masks influence the operations on rows (lines 5-6 in Algorithm 2) as presented in Figure 2. We notice that two paths (in red) lead to bitwise XOR on rows. First, when $mask1 = mask2 = mask3 = \mathbf{1}$, the pivot coefficient is fixed to one. This happens at most once per loop over j. Then, when $mask2 = mask3 = \mathbf{1}$ independently of $mask1$, the other ones in the processed column j are removed.

Algorithm 2: Row reduction 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 row echelon form and its $rank = pivot\_row$
1  $pivot\_row = 0$
2  for $j = 0, \dots, m-1$ do
3      for $i = 0, \dots, n-1$ do
4          if $s_{pivot\_row,j} == 0$ then
               $mask1 = \mathbf{1}$
           else
               $mask1 = \mathbf{0}$
           if $s_{i,j} == 1$ then
               $mask2 = \mathbf{1}$
           else
               $mask2 = \mathbf{0}$
           if $i \geq pivot\_row$ then
               $mask3 = \mathbf{1}$
           else
               $mask3 = \mathbf{0}$
5          $s_{pivot\_row} \leftarrow s_{pivot\_row} \oplus (s_i \wedge (mask1 \wedge (mask2 \wedge mask3)))$
6          $s_i \leftarrow s_i \oplus (s_{pivot\_row} \wedge (mask2 \wedge mask3))$
7      if $s_{pivot\_row,j} = 1$ and $pivot\_row < n$ then
8          $pivot\_row = pivot\_row + 1$
```

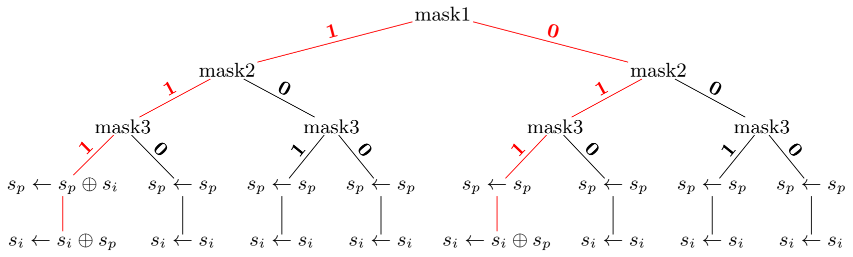

mask1
1
0
mask2
mask3
$s_p \leftarrow s_p \oplus s_i$
$s_p \leftarrow s_p$
$s_i \leftarrow s_i \oplus s_p$
$s_i \leftarrow s_i$

Fig. 2: Operations on matrix rows according to mask values. In red, paths leading to XOR on rows with s_p the pivot row and s_i the processed row.

In Algorithm 2, we observe two sources of leakage. The first one consists of the computation of $mask1$, $mask2$ and $mask3$. These masks are set in an equivalent way, algorithmically, to secret-dependent branches. However, they are determined before an iterative conditional branching, namely in a weighted sum in the *GitHub* implementation. We exploit a leakage in the computation of this weighted sum to recover theirs values. Figure 3 details the implementation of $mask2$ computation in the *GitHub* version.

```
1 int bf_compute_mask(bf_element_t *mask, bf_element_t *a, uint8_t
     bit_position)
2  {
3   // Determine the processed bit
4      uint8_t pos = bit_position / 64u;
5      // Determine the bit position
6      uint8_t bit =  ((uint64_t) (pos * (a->high >> (bit_position - 64
     u))) ^ (1u-pos)*(a->low >> bit_position)) & 0x1u;
7      mask->low = -((uint64_t) bit);
8      mask->high = (uint64_t) -((uint8_t) bit) & ROLLO_I_BF_MASK_HIGH;
9 }
```

Fig. 3: Function to compute $mask2$ introduced in Algorithm 2 and from *GitHub* implementation [1].

We notice that if the processed coefficient, defined as "bit" in line 6, is equal to one, all bits of $mask2$ are set to one, otherwise all bits are set to zero (lines 7-8). The same kind of operations are observed for $mask1$ and $mask3$. However, the leakage from flipping all the bits to 1 or to 0 differs. We deduce that it is possible to recover the masks values.
The second source of leakage comes from the bitwise AND and XOR applied on the syndrome matrix rows. Indeed, in lines 5-6 in Algorithm 2, the rows are XORed with either zero or non-zero row according to the masks values. The second source of leakage has not been exploited because it is equivalent to what we observe with the masks' recovery. However, it is always a good point of interest for side-channel attacks.

As in the previous attack, the masks values recovery allows us to obtain a system of linear equations. We define three vectors containing respectively the values of $mask1$, $mask2$ and $mask2 \wedge mask3$ after the iteration j:

$$\sigma_{mask1,j} = (\sigma_{0,j}, \dots, \sigma_{n-1,j}), \sigma_{mask2,j} = (\sigma'_{0,j}, \dots, \sigma'_{n-1,j}),$$

$$\sigma_{mask2 \wedge mask3,j} = (\sigma''_{0,j}, \dots, \sigma''_{n-1,j}),$$

with $\sigma_{i,j}$,$\sigma'_{i,j}$, $\sigma''_{i,j} = 0$ or 1 when $mask1, mask2, mask2 \wedge mask3 = \mathbf{0}$ or $\mathbf{1}$.
As we can see in Figure 2, when $mask1 = \mathbf{1}$, only one path leads to operations on rows. Moreover, once we have $mask1 = mask2 = mask3 = \mathbf{1}$, $mask1 = \mathbf{0}$ until the end of the column treatment. Then, we have three cases for the pivot:

– If the vector $\sigma_{mask1,j}$ contains only zeros, then the leading coefficient in the pivot row is already one.
– If the vector $\sigma_{mask1,j}$ contains only ones then either the pivot is on the last row, and we need to consider $mask2$ and $mask3$ or the column does not contain a pivot.
– If the vector $\sigma_{mask1,j}$ contains zeros and ones, the position of the last one is the index of the added row to the pivot row in the column j.

$$J_k = \begin{array}{c} \left(\begin{array}{cccccccc} 1 & \cdots & 0 & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \cdots & 1 \end{array}\right) \begin{array}{c} \\ \\ \leftarrow k \\ \\ \\ \end{array} \\ \begin{array}{cc} \uparrow & \uparrow \\ \text{pivot index} & \text{k} \end{array} \end{array} \quad J'_k = \begin{array}{c} \left(\begin{array}{ccccc} 1 & \cdots & \sigma''_{0,k} & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma''_{n-1,k} & \cdots & 1 \end{array}\right) \begin{array}{c} \\ \\ \leftarrow k \\ \\ \\ \end{array} \\ \begin{array}{c} \uparrow \\ k \end{array} \end{array} .$$

We determine the system of linear equations as previously with two matrices depending on respectively of $mask1$ and $mask2 \wedge mask3$:
The vector $\sigma_{mask2,j}$ depends on the coefficients processed in the column j. Therefore, $\sigma_{mask2,0}$ gives us the first column as there is no pre-processing on rows. After the first iteration, we have to consider XORs performed on rows of the matrix during the treatment of the column $j-1$.
For example, after the treatment of the column 0, the positions of the executed XORs are given in the resulting matrix $J'_0 \times J_0$. Thus, for the column 1, we use a linear solver on the system

$$J'_0 \times J_0 \times \mathbf{S}[0] = (\sigma_{mask2,1})^t.$$

More generally, to recover the column $j \geq 1$, we have to solve the system of linear equations

$$\left(\prod_{k=j-1,..,0} J'_k \times J_k \right) \times \mathbf{S}[j] = (\sigma_{mask2,j})^t.$$

3.3 Experimental results of our power consumption analysis

In this section, we demonstrate the practicability of the attack on an ARM SecurCore SC300 32-bit processor (equivalent to CORTEX-M3). We implemented ROLLO-I-128 in C. The first implementation corresponds to the reference one and the second to the *GitHub* version [1].
ROLLO-I-128 traces are captured with a Lecroy SDA 725Zi-A oscilloscope with a bandwidth of 2.5 GHz. We put a trigger right before the execution of the Gaussian elimination. The measurements for the reference implementation are given in Figure 4. The power trace of the first inner **for** loop (line 4 - Algorithm 1) is given in Figure 4a and the power trace of the second inner **for** loop (line 13-Algorithm 1) is given in Figure 4b. 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. For example, we observe in Figure 4 the beginning of the treatment of the first column:

$$\sigma_0 = (0,1,0,0,0,1,1,0,1,0,1,1,\ldots), \sigma'_0 = (*,0,1,1,1,0,0,1,0,1,0,0,0,\ldots).$$

The measurement for the *GitHub* implementation is given in Figure 5. We can first observe the difference of patterns for the three masks values at each inner iteration. For a better understanding, we highlight each time the computation of the different masks. Then, we can perform the attack from the recovered masks values:

$$\begin{aligned}&\sigma_{mask1,1} = (1,1,1,0,0,0,0,\ldots),\\ &\left.\begin{aligned}\sigma_{mask2,1} &= (1,0,1,0,0,1,0,\ldots)\\ \sigma_{mask3,1} &= (0,0,1,1,1,1,1,\ldots)\end{aligned}\right\} \quad \sigma_{mask2\wedge mask3,1} = (0,0,1,0,0,1,0,\ldots).\end{aligned}$$

The source codes of the attacks for the reference implementation is given in Appendix C and for the *GitHub* implementation is given in Appendix D.

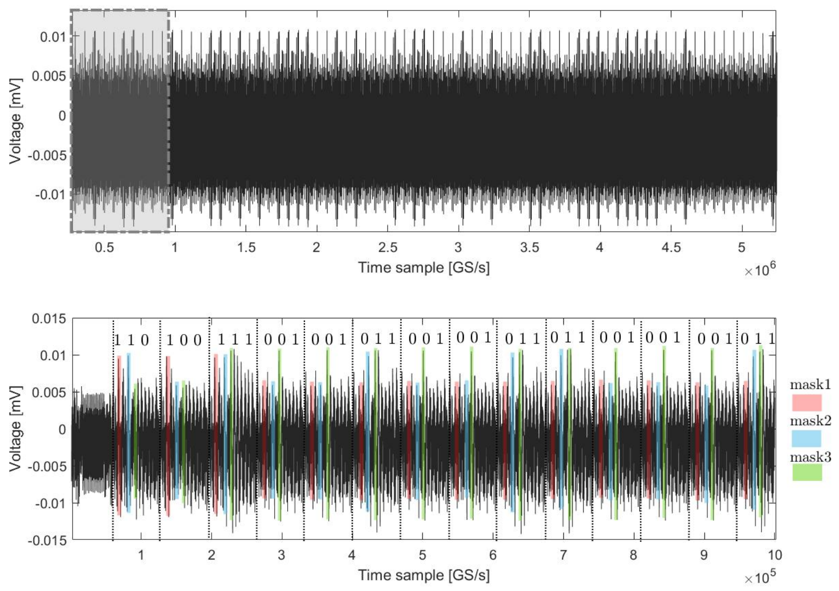
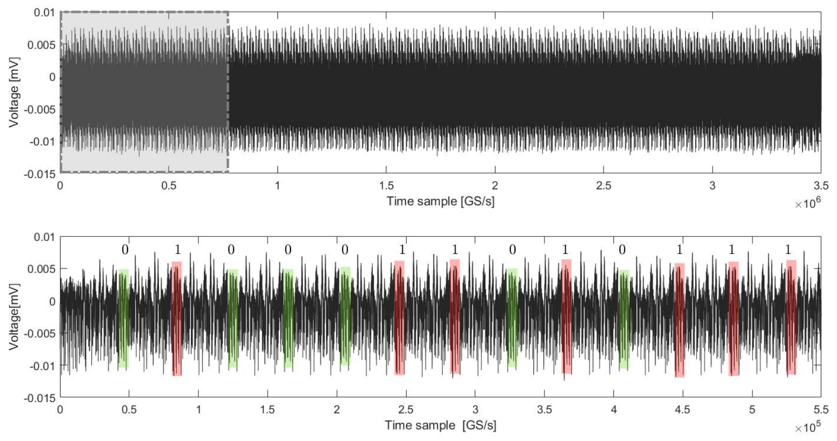


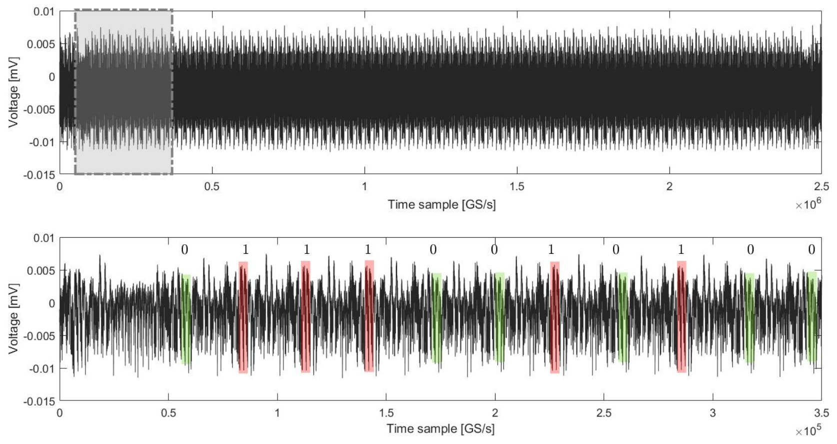
Fig. 5: Measurement for the *GitHub* implementation - trace of the treatment of one column in ROLLO-I-128.

3.4 Experiments with a Cortex-M4 and comparison

In this section, we show that the attack is also applicable on an ARM Cortex-M4. For the experiments we used the ROLLO-I-128 implementation provided in the *mupq* git [17] on a STM32F4 ChipWhisperer microcontroller. The traces are captured with a RTO2000 oscilloscope with bandwidth 3GHz. We put a trigger right before the execution of the Gaussian elimination.



(a) Full trace and a zoom of the first inner loop



(b) Full trace and a zoom of the second inner loop

Fig. 4: Measurements for the reference implementation - traces of the two inner loops in Gaussian elimination for the processing of one column in ROLLO-I-128: in green the treatment when the bit is 0 and in red when the bit is 1.

Figure 6 provides measurements obtained with a Cortex-M4. Similarly to Figure 4 with a Cortex-M3, the traces are annotated with rectangles and colors: green for a mask at 0 and red for a mask at 1.
We notice that in Figure 4a the difference between a mask at 0 and a mask at 1 is more pronounced than in Figure 6a. In fact, in the latter, the difference of power consumption between both masks is smaller and requires looking carefully at the end of the pattern to distinguish them. For Figure 4b and Figure 6b, the patterns for a mask at 0 and a mask at 1 are similar. However, we notice that the decreasing power in the pattern of a mask at 0 is more accentuated in Figure 6b.

4 Countermeasures

In this section, we propose two solutions to protect the future implementations against our attack. It is important to emphasize that the implementations with the countermeasures remain in constant-time.

First countermeasure for the reference implementation. The first countermeasure consists in reducing the differentiation between a multiplication of a word by zero or by one. For this, we mask the coefficients processed. 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

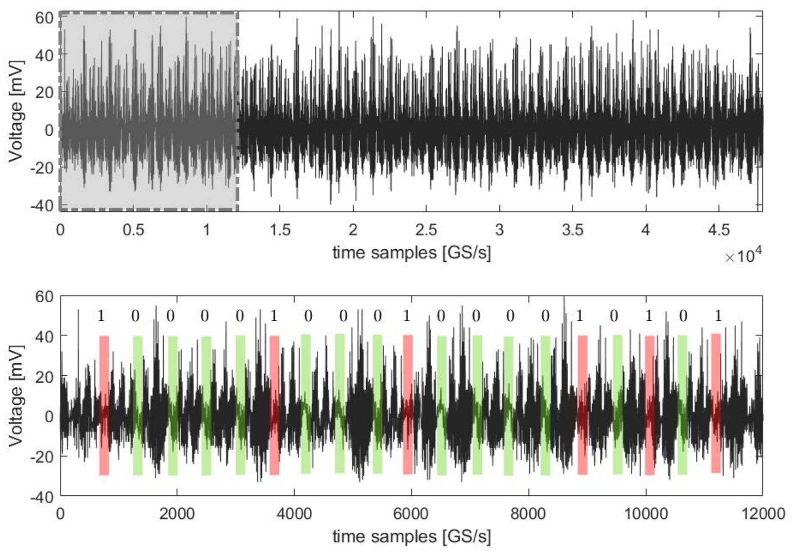
$$s_{pivot\_row} = s_{1pivot\_row} \oplus tmp.$$

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

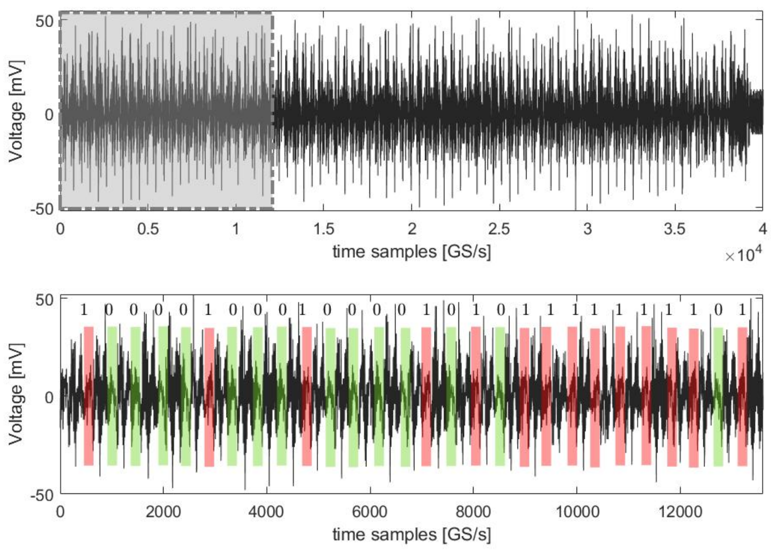
$$dummy = s_{1pivot\_row} \oplus tmp.$$

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. We applied the same set up as in Section 3.3 to illustrate this in Figure 7.

Second countermeasure for both implementations. 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 (given in Appendix F). The choice is left to the developer



(a) Full trace and a zoom of the first inner loop



(b) Full trace and a zoom of the second inner loop

Fig. 6: Measurement for the processing, on the Cortex M4, of one column in the Gaussian elimination from the reference implementation.

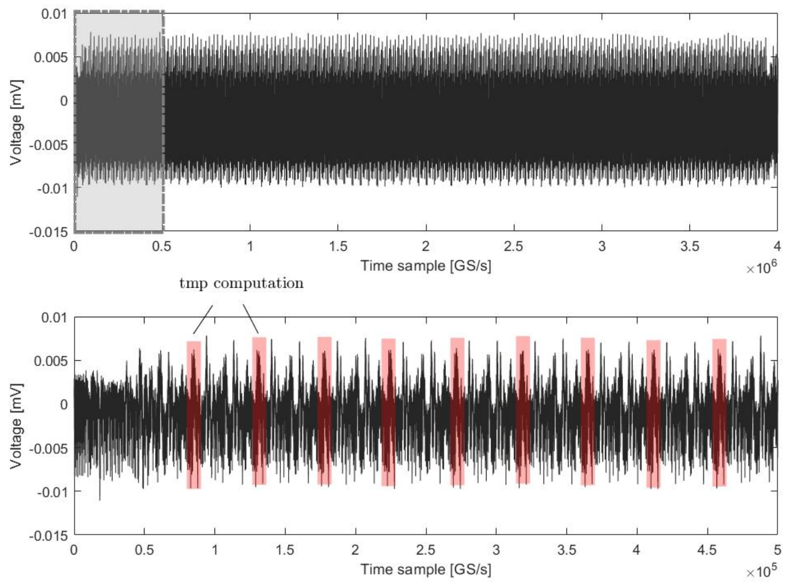


Fig. 7: First **for** loop trace of Gaussian elimination with masking countermeasure.

under condition of a good implementation.
Given a list of n elements to shuffle, the Fisher-Yates method starts with a random function that generates a random number j such that $1 \leq j \leq n-k$ where k is the number of elements already processed. Then, the elements are processed in decreasing order. Namely, in the first iteration, the last element of the list is swapped with the j-th element and goes on until the first element is reached.
For the reference implementation, a list containing the coefficients indexes is randomized before the two inner **for** loops. Then, at each iteration, the pivot row is chosen randomly and the randomized list gives the proceeding order for the other coefficients in the column. This countermeasure is presented in Appendix E (Algorithm 4). The indexes are shuffled before the two inner **for** loops, then there is no correlation between the masks of the first **for** loop (line 4 - Algorithm 1) and the masks of the second **for** loop (line 11 - Algorithm 1).
For the *GitHub* implementation, a similar countermeasure is performed (Algorithm 5 in Appendix E). In this case, for each column (in the main **for** loop) the pivot row is chosen randomly and the indexes are shuffled using Fisher-Yates method.
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 will be known.
We provide in Table 2 the performances' analysis for the SC300 processor of the impact of our countermeasures. This impact depends on the board and the used random number generator. We counted the cycles by using IAR Embedded Workbench IDE for ARM[5] compiler C/C++ with high-speed optimization level.

implementation	Reference			GitHub	
countermeasure	masking	randomization	without	randomization	without
# cycles ($\times 10^6$)	3,15	2,5	1,82	2,9	2,22

Table 2: Impact factor of Gaussian elimination with and without countermeasures for ARM securCore SC300 processor.

5 Conclusion and perspectives

We show in this paper that constant-time implementations of Gaussian elimination provided in [2] and [1] are both sensitive to power consumption attacks. The weakness, directly linked to the mask used to avoid timing attacks, allows us to make the first attack by side-channel on the last implementation version given by the authors of ROLLO. We can also apply our attack on another implementation of ROLLO-I-128. These attacks can lead to a full recovery of the private key using a single trace. To secure the implementations, we propose two different countermeasures. The first one can be applied to [2] by hiding the values of the 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. We show that the attacks are feasible in both cases even though there is some difference in the traces.
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.

[5] `https://www.iar.com/knowledge/learn/debugging/how-to-measure-execution-time-with-cyclecounter/`

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.-L. Cayrel et al. "Message-recovery Laser Fault Injection Attack on the Classic McEliece Cryptosystem". In: (2021). `https://eprint.iacr.org/2020/900`.

[9] 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`.

[10] J. Hoffstein, J. Pipher, and J. H. Silverman. "NTRU: A Ring-Based Public Key Cryptosystem". In: *Proceedings of the Third International Symposium on Algorithmic Number Theory* (1998), pp. 267–288.

[11] 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`.

[12] 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.

[13] N. Lahr et al. "Side Channel Information Set Decoding Using Iterative Chunking". In: *Advances in Cryptology – ASIACRYPT 2020*. Ed. by S. Moriai and H. Wang. Cham: Springer International Publishing, 2020, pp. 881–910. ISBN: 978-3-030-64837-4.

[14] R. J. McEliece. *A public-key cryptosystem based on algebraic coding theory*. Tech. rep. 44. Pasadena, CA: California Inst. Technol., 1978, pp. 114–116.

[15] C. A. Melchor et al. *Rank Quasi-Cyclic (RQC)*. `https://pqc-rqc.org/`. 2020.

[16] D. Moody et al. *Status report on the second round of the NIST post-quantum cryptography standardization process*. Tech. rep. 2020. DOI: `10.6028/nist.ir.8309`.

[17] *mupq git: Implementation second round NIST schemes for ARM Cortex-M4*. Source code available at `https://github.com/mupq/mupq/tree/Round2/crypto_kem`.

[18] 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.

[19] 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`.

[20] 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.

[21] F. Strenzke et al. “Side channels in the McEliece PKC”. In: *International Workshop on Post-Quantum Cryptography*. Springer. 2008, pp. 216–229.

A Rank Support Recovery Algorithm

We recall in Algorithm 3 the decoding algorithm used for LRPC codes.

Algorithm 3: Rank Support Recovery (RSR) algorithm

```
Input: An $\mathbb{F}_q$-subspace $F = \langle f_1, \ldots, f_d \rangle$, $\mathbf{s} = (s_1, \ldots, s_n)$ a syndrome of an
       error $\mathbf{e}$, $r$ the error's rank weight
Output: A candidate $E$ for the support of $\mathbf{e}$
1 // Part 1 : Compute the vector space $EF$
2 Compute $S = \langle s_1, \ldots, s_n \rangle$
3 // Part 2 : Recover the vector space $E$
4 Pre-compute every $S_i = f_i^{-1}.S$ for $i = 1$ to $d$
5 $E \leftarrow \bigcap_{1 \le i \le d} S_i$
6 return $E$
```

B Toy example for the attack for the reference implementation

Let us take a small example, with $q = 2$, $m = 5$ and $n = 7$, to illustrate the information leakage that we found.

Assume we want to recover the following matrix

$$\begin{pmatrix} 1&1&1&0&0 \\ 0&0&0&1&0 \\ 0&1&0&1&0 \\ 0&1&0&1&1 \\ 1&1&1&1&1 \\ 1&1&0&1&1 \\ 1&0&0&1&0 \end{pmatrix}$$

corresponding to the syndrome $\mathbf{s} \in \mathbb{F}_{2^5}^7$.

$\longrightarrow$

The searched matrix is defined as

$$\mathbf{S} = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} \end{pmatrix}$$

After the execution of the Gaussian elimination process, we guess from the power consumption analysis the masks in the first and second loops:

1. masks in the first loop for each column:

$$(*, 1, 1, 1, 0, 0, 0), (1, *, 1, 0, 1, 1, 0), (1, 0, *, 0, 1, 0, 1), (1, 1, 1, *, 0, 1, 1),$$
$$(1, 1, 1, 0, *, 1, 0)$$

2. masks of the second loop for each column:

$$(*, 0, 0, 0, 1, 1, 1), (1, *, 1, 1, 0, 0, 1), (0, 1, *, 1, 0, 1, 0), (1, 1, 1, *, 0, 1, 0),$$
$$(1, 1, 1, 0, *, 1, 1),$$

with $*$ the pivot. As explained in Section 3.1, the $*$ are replaced by one.

Let us focus on recovering the two first columns of the syndrome matrix. The recovered masks vector of the first loop $(1, 1, 1, 1, 0, 0, 0)$ provides the additions on the pivot row 0:

$$J_0 \times \mathbf{S} = \left(\begin{array}{c|c} 1 & 1\;1\;1\;0\;0\;0 \\ \hline \mathbf{0} & I_6 \end{array}\right) \times \mathbf{S}[0] = \begin{pmatrix} s_{0,0} + s_{1,0} + s_{2,0} + s_{3,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \\ s_{4,0} \\ s_{5,0} \\ s_{6,0} \end{pmatrix}.$$

The masks vector of the second loop $\sigma'_0 = (1, 0, 0, 0, 1, 1, 1)$ is the solution vector of the system of linear equations where $s_{i,j}$ are unknowns. Thus, by applying a *SageMath*[6] linear solver on the system

$$J_0 \times \mathbf{S}[0] = (1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1)^t,$$

we find the solution $(1, 0, 0, 0, 1, 1, 1)$, which corresponds to the first column of the syndrome matrix (see Appendix C for the source code). At the end of the process of the first column, we have the matrix

$$\mathbf{S}_0 = \left(\begin{array}{c|c} (\sigma'_0)^t & \begin{array}{c}\mathbf{0} \\ \hline I_6\end{array} \end{array}\right) \times J_0 \times \mathbf{S}.$$

For the second column, the recovered masks vector of the first loop is $(1, 0, 1, 0, 1, 1, 0)$. However, as explained in Section 3.1, only the rows for which the index row is greater than the index pivot row are added to the pivot row. Thus, in the recovered masks vector, we replace one by zero for $i < 1$. This gives us the vector $\sigma_1 = (0, 0, 1, 0, 1, 1, 0)$. In addition, the masks vector of the second loop is $\sigma'_1 = (1, 1, 1, 1, 0, 0, 1)$. We can then apply a *SageMath* linear solver on the system

$$\underbrace{\left(\begin{array}{c|c} 1 & 0\;0\;0\;0\;0\;0 \\ 0 & 1\;1\;0\;1\;1\;0 \\ \hline \mathbf{0} & I_5 \end{array}\right)}_{J_1} \times \mathbf{S}_0[1] = (1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1)^t.$$

The result of this system corresponds to the vector $(1, 0, 1, 1, 1, 1, 0)$.
At the end, we have the matrix

$$\mathbf{S}_1 = \left(\begin{array}{cc|c} \begin{array}{c} 1 \\ \mathbf{0} \end{array} & (\sigma'_1)^t & \begin{array}{c}\mathbf{0} \\ \hline I_5\end{array} \end{array}\right) \times J_1 \times \mathbf{S}_0.$$

[6] https://www.sagemath.org/

We perform the same for the three remaining columns.

C Source code for the attack on the reference implementation using *SageMath*

```
def matrix_equation(pivot_index, mask_firstloop,
    mask_secondloop, nbrow):

  # Initialization  of the  two  matrices  that will
    determine the system of equations
  Meq1 = identity_matrix(Zmod(2),nbrow)
  Meq2 = identity_matrix(Zmod(2),nbrow)

  # Placing coefficients at 1 for the additions on the pivot
    row, defined thanks to the 'masks' of the first loop
  for i in range(len(mask_firstloop)):
    if(mask_firstloop[i]):
      Meq1[pivot_index,i]=1

  # Placing coefficients at 1 for the additions on rows with
    the leading coefficient at 1, defined thanks to the masks
     of the second loop
  for i in range(len(mask_secondloop)):
    if(mask_secondloop[i]):
      Meq2[i,pivot_index]=1

  return Meq1,Meq2

def matrix_equation_columnindex(masks_firstloop,
    masks_secondloop,nbrow,columnindex):

  M =[]

  # Initialization of M with the matrices of equations
  for i in range(columnindex+1):
    M.append(matrix_equation(i, masks_firstloop[i],
    masks_secondloop[i], nbrow))

  return M

def equations_to_solve(masks_firstloop, masks_secondloop,
    nbcolumn,nbrow,columnindex):

  # Initiatialization of the matrices to define the system of
     equations
  Meq = matrix_equation_columnindex(masks_firstloop,
    masks_secondloop,nbrow,columnindex)

  # Multiplication of the matrices to determine all the
    equations for the column "columnindex"
  Stmp = Meq[0][0]
  for i in range(columnindex):
    Stmpbis = Meq[i][1]* Stmp
    Stmp =Meq[i+1][0] * Stmpbis

  #Solving the system of equations
  S = Stmp.solve_right(matrix(Zmod(2),masks_secondloop[
    columnindex]).transpose())

  return S
```

D Source code for the attack on the *GitHub* implementation using *SageMath*

```
# matrix_equation returns a matrix of all the additions made
    on rows to get the system of equation for given pivot
    index and masks.
def matrix_equation(index_column, pivot_index, mask1, mask2,
    mask3, nbrow):
  copy_pivot_index = pivot_index

    # If mask1 is full of ones then the column does not
    contain a pivot or the pivot is on the last row.
    # In the first case we return the identity matrix,
    # Else the position of the first zero on mask1 determines
     the pivot's position.
  if(mask1.count(0)==0):
    if(mask2[-1]&mask3[-1] == 1):
      pivot_position= nbrow-1
    else:
      return identity_matrix(Zmod(2),nbrow), copy_pivot_index
  else:
    if(mask1.index(0)==0):
      pivot_position= pivot_index
    else:
      pivot_position = mask1.index(0) -1

  # Initialization of the two matrices that will determine
    the system of equations
  Mpivot = identity_matrix(Zmod(2),nbrow)
  Mrows = identity_matrix(Zmod(2),nbrow)

  # Then the matrix Mpivot get an additionnal one that
    indicates which row has been added to pivot row
  Mpivot[pivot_index,pivot_position] = 1

    # The pivot row is added to the processed row when (mask2
    [i] and mask3[i])=1.
    # We use those mask to determine when operations on the rows have been performed except for the pivot row.
    # All the operations are represented by a one in the matrix Mrows at the position i (number of the processed row) and the pivot index.
  for i in range(nbrow):
    if((mask2[i]&mask3[i])==1):
      Mrows[i,pivot_index]=1

  # Meq is the matrix representation of all the addition to make to get the system of equation
  Meq = Mrows*Mpivot
  copy_pivot_index = pivot_index + 1
  return Meq, copy_pivot_index

def matrix_equation_columnindex(mask1s, mask2s,mask3s,nbrow,columnindex):
  M =[]
  pivot_index=0
  # Initialization of M with the matrices of equations
  for i in range(columnindex):
    R = matrix_equation(i, pivot_index, mask1s[i], mask2s[i], mask3s[i],nbrow)
    # In the case there is no pivot in a column, the index pivot is unchanged
    pivot_index = R[1]
    M.append(R[0])

  return M

def solution_vector(mask2s,columnindex):
  # Return the vector solution of the system of equation for the column "columnindex"
  return matrix(Zmod(2),mask2s[columnindex])

def equations_to_solve(mask1s, mask2s,mask3s,nbcolumn,nbrow,columnindex):
  # In the case we want to recover the column 0 of the matrix, the vector mask2 gives directly the solution
  if(columnindex==0):
    S = solution_vector(mask2s,columnindex)
    return S

  # Initiatialization of the matrices to define the system of equations
  Meq = matrix_equation_columnindex(mask1s,mask2s,mask3s,nbrow,columnindex)

  # Multiplication of the matrices to determine all the equations for the column "columnindex"
  Stmp = identity_matrix(Zmod(2),nbrow)
  for i in range(columnindex):
    Stmp = Meq[i]*Stmp

  #Solving the system of equations
  v_sol = solution_vector(mask2s,columnindex)
  S = Stmp.solve_right(v_sol.transpose()).transpose()

  return S
```

co

E Algorithms for countermeasures with randomization

Algorithm 4: Gaussian elimination in constant time with randomization

```
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
1  $mask = dimension = 0$
2  $L \leftarrow$ array containing indexes $0, \dots, n-1$
3  for $j = 0, \cdots, m-1$ do
4      $pivot\_row = min(dimension, n-1)$
5      $randpivot = \text{random}(pivot\_row + 1, n-1)$
6      exchange $s_{pivot\_row}$ and $s_{randpivot}$
7      $L = \text{FisherYatesShuffle}(L)$
8      for $i = 0, \cdots, n-1$ do
9          $randrow = L[i]$
10         $mask = s_{pivot\_row,j} \oplus s_{randrow,j}$
11         $tmp = mask \otimes s_{randrow}$
12         if $randrow > pivot\_row$ then
13             $s_{pivot\_row} = s_{pivot\_row} \oplus tmp$
14         else
15             $dummy = s_{pivot\_row} \oplus tmp$
16     $L = \text{FisherYatesShuffle}(L)$
17     for $i = 0, \cdots, n-1$ do
18         $randrow = L[i]$
19         if $randrow \neq j$ then
20             $mask = s_{randrow,j}$
21             $tmp = mask \otimes s_{pivot\_row}$
22             if $dimension < n$ then
23                 $s_{randrow} = s_{randrow} \oplus tmp$
24             else
25                 $dummy = s_{randrow} \oplus tmp$
26     $dimension = dimension + s_{pivot\_row,i}$
```

Algorithm 5: Row echelon form in constant time with randomization

```
   Input: $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
   Output: $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ in row echelon form and its $rank = pivot\_row$
 1 $pivot\_row = 0$
 2 $L \leftarrow$ array containing indexes $0, \ldots, n-1$
 3 for $j = 0, \cdots, m-1$ do
 4     $randpivot = \mathrm{random}(pivot\_row + 1, n-1)$
 5     exchange $s_{pivot\_row}$ and $s_{randpivot}$
 6     $L = \mathrm{FisherYatesShuffle}(L)$
 7     for $i = 0, \cdots, n-1$ do
 8         $randrow = L[i]$
 9         if $s_{pivot\_row,j} == 0$ then
10             $mask1 = \mathbf{1}$
11         else
12             $mask1 = \mathbf{0}$
13         if $s_{randrow,j} == 1$ then
14             $mask2 = \mathbf{1}$
15         else
16             $mask2 = \mathbf{0}$
17         if $randrow \geq pivot\_row$ then
18             $mask3 = \mathbf{1}$
19         else
20             $mask3 = \mathbf{0}$
21         $s_{pivot\_row} \leftarrow s_{pivot\_row} \oplus s_i \wedge (mask1 \wedge (mask2 \wedge mask3))$
22         $s_{randrow} \leftarrow s_{randrow} \oplus s_{pivot\_row} \wedge (mask2 \wedge mask3)$
23     if $s_{pivot\_row,j} = 1$ and $pivot\_row < n$ then
24         $pivot\_row = pivot\_row + 1$
```

F Fisher-Yates Algorithm

Algorithm 6: FisherYatesShuffle

```
   Input: $L$ list of $n$ elements
   Output: the list $L$ shuffled
 1 for $i = n-1, \cdots, 0$ do
 2     $j = \mathrm{random}() \mod i$
 3     exchange $L_i$ and $L_j$
```