Engineering Practical Rank-Code-Based Cryptographic Schemes on Embedded Hardware. A Case Study on ROLLO

Jingwei Hu, Wen Wang, Kris Gaj, Liping Wang, and Huaxiong Wang

***Abstract*—In this paper, we investigate the practical performance of rank-code based cryptography on FPGA platforms by presenting a case study on the quantum-safe KEM scheme based on LRPC codes called ROLLO, which was among NIST post-quantum cryptography standardization round-2 candidates. Specifically, we present an FPGA implementation of the encapsulation and decapsulation operations of the ROLLO KEM scheme with some variations to the original specification. The design is fully parameterized, using code-generation scripts to support a wide range of parameter choices for security levels specified in ROLLO. At the core of the ROLLO hardware, we presented a generic approach for hardware-based Gaussian elimination, which can process both non-singular and singular matrices. Previous works on hardware-based Gaussian elimination can only process non-singular ones. However, a plethora of cryptosystems, for instance, quantum-safe key encapsulation mechanisms based on rank-metric codes, ROLLO and RQC, which are among NIST post-quantum cryptography standardization round-2 candidates, require performing Gaussian elimination for random matrices regardless of the singularity. To the best of our knowledge, this work is the first hardware implementation for rank-code-based cryptographic schemes. The experimental results suggest rank-code-based schemes can be highly efficient.**

***Index Terms*—FPGA implementation, gaussian elimination, post-quantum cryptography, rank metric code.**

I. Introduction

POST-QUANTUM cryptography (PQC) refers to cryptographic public-key algorithms which exploit the hard problems thought to be outside bounded-error quantum polynomial time (BQP) class. As of 2021, most popular public-key algorithms rely on the BQP class problems and are theoretically insecure against an attack by a quantum computer. Even though it is not yet known to what extent a future quantum computer can be used to solve BQP problems successfully, many cryptographers are designing new algorithms to prepare for a time when quantum computing becomes a real threat. This line of work has gained greater attention from academics and industry, including a large European project PQCRYPTO, and a standardization competition initiated by NIST [1] in 2017. Currently, the PQC research is mostly focused on lattice-based, code-based, hash-based, isogeny-based, and multi-variate cryptography [2].

For code-based schemes, the first public-key encryption (PKE) was introduced in 1978 by McEliece [3]. The basic construction specifies a public random linear code from a secret known binary Goppa code. A ciphertext is a codeword of that public random linear code plus a few random errors. An adversary who attempts to decrypt this ciphertext without knowing the backdoor has to solve the problem of decoding a random linear code, which is proved to be NP-complete. The security of the McEliece system has remained stable despite dozens of attacks (mostly based on information set decoding) over 40 years. With some modifications including a more efficient dual PKE suggested by Niederreiter [4], a tight conversion to IND-CCA2 KEM, and implementation speedups, the McEliece system is synthesized to be a conservative, well-understood key encapsulation mechanism (KEM) as *Classic McEliece* [5] which is selected as a finalist of the NIST PQC standardization process.

An interesting variant of McEliece system was recently proposed using QC-MDPC codes [6]. This is a Niederreiter PKE that uses the so-called moderate density parity check (MDPC) codes in quasi-cyclic (QC) form. The QC form suffices to represent the parity check matrix by its first row, which leads to a shorter key. Due to this lightweight characteristic for compressing information, QC codes are popularly used in many follow-up work, among which BIKE [7], HQC [8] and LEDAcrypt [9] are the outstanding representatives. BIKE and HQC have proceeded to the final round of the NIST PQC standardization competition, and LEDAcrypt is a round-two candidate. On the other hand, the use of QC-MDPC codes introduces decoding failures which might be vulnerable to reaction attacks. For example, using decoding failures, the GJS attack [10] is a powerful key recovery attack against CCA-secure BIKE. They estimate the time complexity for attacking the 80-bit CPA (or CCA2)-secure version as $2^{43.6}$ (or $2^{55.3}$) operations. The key observation from the GJS attack is that it is possible to distinguish the dependence between

Manuscript received 7 December 2021; revised 11 November 2022; accepted 20 November 2022. Date of publication 2 December 2022; date of current version 9 June 2023. This work was supported in part by A*Star, Singapore under Grant SERC A19E3b0099, in part by the Singapore Ministry of Education under Grant MOE2019-T2-2-083, and in part by the National Natural Science Foundation of China under Grant 62002239. Recommended for acceptance by X. Cheng. *(Corresponding author: Jingwei Hu.)*

Jingwei Hu and Huaxiong Wang are with the School of Physical and Mathematical Sciences, Nanyang Technological University,, Singapore 639798 (e-mail: davidhu@ntu.edu.sg; hxwang@ntu.edu.sg).

Wen Wang is with the Deparment of Electronic Engineering, Yale University, New Haven, CT 06520 USA (e-mail: wen.wang.ww349@yale.edu).

Kris Gaj is with the Deparment of Electronic and Computer Engineering, George Mason Univeristy, Fairfax, VA 22030 USA (e-mail: kgaj@gmu.edu).

Liping Wang is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100045, China (e-mail: wangliping@iie.ac.cn).

Digital Object Identifier 10.1109/TC.2022.3225080

the secret key and the decoding failure probability. Therefore, it is necessary to keep the decoding failure probability negligibly small (e.g., 2^{-128}) to thwart this type of attack.

The code-based schemes described above are all constructed on the Hamming metric. In the 1990s, another type of code-based cryptography, known as GPT cryptosystem [11] emerged, whose security was based on an alternative metric, the so-called rank metric. The difference with the McEliece cryptosystem consists of the choice of the family of codes and the choice of the metric. At that time, Gabidulin code was the only family of rank-metric codes with an efficient algebraic polynomial time decoding algorithm and thus was selected for the GPT system. However, Overbeck in 2005 proposed a framework that could be adapted to all variants of Gabidulin codes-based encryption schemes [12]. Recent years have witnessed the revival of rank-metric code-based cryptography mainly attributed to two new variant schemes called RQC [13] and ROLLO [14]. Roughly speaking, the security of these new schemes is based on the rank syndrome decoding problem, which has been proven hard with a probabilistic reduction to the well-known syndrome decoding problem in Hamming metric. In particular, RQC uses Gabidulin code as the underlying coding system, which is publicly known. A nice property RQC has is that the scheme's security does not rely on the knowledge of the error-correcting code being used and naturally resists the Overbeck's attack. ROLLO is proposed by using the low rank parity check (LRPC) codes. Unlike Gabidulin codes which contain a huge vector space invariant under the Frobenius automorphism suffering from complete cryptanalysis, LRPC codes are weakly structured and behave like a random linear code in rank metric after proper scrambling. Moreover, RQC and ROLLO benefit from the ideal code structure to compress the key size and reduce the computational complexity. The decoding failure probability is well studied and can easily be chosen to meet security requirements. RQC and ROLLO are selected by NIST as the round-2 candidates but they do not survive to the final round due to recent attacking progress [15], [16]. This algebraic attack suggests the hardness of the rank syndrome decoding problem is underestimated, and a minor modification should be introduced to keep RQC and ROLLO secure. Despite the loss of the election, NIST encourages further investigation of rank-code-based schemes as the official status report [17] on the second round of candidates states "Rank-based cryptography should continue to be researched. The rank metric cryptosystems offer a nice alternative to the traditional Hamming metric codes with comparable bandwidth."

Inspired by the confidence in the code-based cryptography, many publications on hardware implementations have started to appear. All of the work only focuses on the McEliece/Niederreiter framework in the Hamming metric. There exists no cryptographic hardware for the McEliece/Niederreiter in the rank metric. For example, Heyse and Güneysu in [18] report that a Goppa code-based Niederreiter decryption operation consumes 58.78 μs on a Xilinx Virtex6-LX240T FPGA for $n = 2048$ and $t = 27$ for 80-bit symmetric security. Wen Wang et al. present an FPGA-based key generator for the Goppa code-based Niederreiter cryptosystem [19] and later a full implementation [20] which includes modules for encryption, decryption, and key generation. The hardware for the classic McEliece scheme has been presented in [21] recently. The first implementation of MDPC code-based McEliece cryptosystem on embedded devices was presented in [22] in 2013. A lightweight MDPC-McEliece has been implemented on FPGAs by sequentially manipulating cyclic rotations of the private key in block RAMs [23]. An area-time efficient MDPC-Niederreiter has been implemented on FPGAs, which exploits a resource-balanced MDPC decoding unit [24]. An FPGA-based key generator for BIKE is presented in [25] which outperforms the round-2 reference BIKE hardware implementation by nine times in terms of run-time. A full implementation of round-3 BIKE is studied in [26] which implements the Black-Gray-Flip decoder for the first time on hardware. The key encapsulation method of LEDAcrypt, also known as LEDAkem [27], is presented in Xilinx Artix-7 FPGAs and shows its supremacy over BIKE regarding area efficiency [28]. These works, as mentioned above, do not consider the new types of arithmetic used in rank-metric. Despite the existence of ROLLO software implementations [29], [30], [31], the hardware architecture and performance for rank-code-based schemes remain unexplored.

Contributions: This paper presents the first efficient and scalable FPGA-based cryptographic hardware for a post-quantum KEM/PKE using rank-metric codes (ROLLO). The main contributions include:

- A new, constant-time hardware design of Gaussian elimination for large-scaled (singular) matrices over binary field.
- A tunable, constant-time approach and design of polynomial multiplication over $\mathbb{F}_{2^m}[z]$.
- A more efficient, bounded decoding failure rate, and constant-time Rank Support Recovery (RSR) algorithm and its hardware implementation.
- A complete implementation of the ROLLO data encapsulation and decapsulation in support of various security parameters, which uses automatic code-generation scripts to generate vendor-neutral HDL codes.[1]

Last but not least, we would like to mention that the implementation of ROLLO is mostly used as a case study on rank code based schemes to further explore their real performance on embedded hardware. The proposed designs are versatile and applicable to implement other rank code based schemes. For example, RQC also uses the polynomial multiplication for computation over ideal code and the Gaussian elimination for low-rank vector space generation. The flexible and efficient architectures with different area and performance trade-offs presented in this work, which strictly follow the parameterization methodology, can be easily adapted to accelerate RQC and other (quantum-safe) schemes based on rank metric codes.

II. Preliminaries of ROLLO

ROLLO is a compilation of two cryptographic schemes, ROLLO-I and ROLLO-II, which are among 26 round-2 candidates to the NIST's process for post-quantum cryptography standardization. ROLLO is based on rank metric codes, and the difficult problem on which ROLLO relies is the syndrome decoding

[1]ROLLO hardware project can be found at https://github.com/davidhoo1988/rollo-hardware

TABLE I
Security (Bit-Level) of Different Parameter Sets Used in ROLLO, Taken From [14]

Instance	parameters (n, m, r, d)	pk size	sk size	ct size	security
ROLLO-I-128	$(83, 67, 7, 8)$	696 bytes	40 bytes	696 bytes	128
ROLLO-I-192	$(97, 79, 8, 8)$	958 bytes	40 bytes	958 bytes	192
ROLLO-I-256	$(113, 97, 9, 9)$	1371 bytes	40 bytes	1371 bytes	256
ROLLO-II-128	$(189, 83, 7, 8)$	1941 bytes	40 bytes	2089 bytes	128
ROLLO-II-192	$(193, 97, 8, 8)$	2341 bytes	40 bytes	2469 bytes	192
ROLLO-II-256	$(211, 97, 8, 9)$	2559 bytes	40 bytes	2687 bytes	256

problem in the rank metric. In this paper, we stay focused on the latest parameters (see Table I) in the April-21-2020 version [14]. It is also worth noting that ROLLO proposes 6 instances, each of which uses distinct system parameters. Therefore, it is challenging to realize the entire ROLLO, which spans such a wide range of parameter values, in hardware. To tackle this challenge, the parameterization methodology is considered in this work such that the modules are fully parameterized and quickly switched from one parameter set to another for re-synthesis. More importantly, the actual implementation of ROLLO introduces a new challenge for hardware-based Gaussian-elimination: the computation in ROLLO requires Gaussian-eliminating a matrix with an unknown rank, and it is most likely that the matrix under operation is singular. Valid manipulation for such a matrix goes beyond the applicability of the existing Gaussian elimination hardware.

In the following subsections, we use the same notations from the ROLLO NIST submission document. Let $S_w^n(\mathbb{F}_{q^m})$ stand for the set of vectors of length n and rank weight w over $\mathbb{F}_{q^m}$ and $S_{1,w}^n(\mathbb{F}_{q^m})$ stand for the set of vectors of length n and rank weight w, such that *its support contains* 1

$$S_w^n(\mathbb{F}_{q^m}) = \{x ∈ \mathbb{F}_{q^m}^n : \dim \mathrm{Supp}(x) = w\}$$

$$S_{1,w}^n(\mathbb{F}_{q^m}) = \{x ∈ \mathbb{F}_{q^m}^n : \dim \mathrm{Supp}(x) = w, 1 ∈ \mathrm{Supp}(x)\}.$$

A. ROLLO-I as a KEM

ROLLO-I, formerly known as LAKE, is a CPA-secure Key Encapsulation Mechanism (KEM) running in the category “post-quantum key exchange”. A Key-Encapsulation scheme KEM = (KeyGen, Encap, Decap) is a triple of probabilistic algorithms together with a key space K. The key generation algorithm KeyGen generates a pair of public and secret key (pk, sk). The encapsulation algorithm Encap uses the public key pk to produce an encapsulation c of a key $K ∈ K$. Finally Decap using the secret key sk and an encapsulation c, recovers the key $K ∈ K$ or fails and returns $\perp$.

ROLLO-I is formally described in Algorithm 1. The RSR algorithm is the rank support recover algorithm proposed in [32] to recover the rank support of the error vector from the secret syndrome. P is an irreducible polynomial of $\mathbb{F}_q[X]$ of degree n and constitutes a parameter of the cryptosystem.

It is worthwhile to mention that in the encapsulation/encryption step, two random polynomials of degree n over $\mathbb{F}_{2^m}$, i.e., e_1 and e_2 have rank support $\mathrm{Supp}(e_1, e_2) = r$. In other terms, $e_i (i = 1, 2)$ formulates a vector space represented by a $n \times m$ matrix with small rank r. This is where universal Gaussian elimination comes into play.

Algorithm 1: Formal Description of ROLLO-I.

```
1 KeyGen($1^\lambda$): Pick $(x, y) \xleftarrow{\$} S_d^{2n}(\mathbb{F}_{q^m})$. Set
  $h = x^{-1}y \bmod P$, and return $pk = h$, $sk = (x, y)$.
2 Encap(pk): Pick $(e_1, e_2) \xleftarrow{\$} S_r^{2n}(\mathbb{F}_{q^m})$, set
  $E = \mathrm{Supp}(e_1, e_2)$, $c = e_1 + e_2 h \bmod P$. Compute
  the shared secret key $K = \mathrm{Hash}(E)$ and return c.
3 Decap(c, sk): Set $s = xc \bmod P$, $F = \mathrm{Supp}(x, y)$
  and $E \leftarrow \mathrm{RSR}(F, s, r)$. Recover $K = \mathrm{Hash}(E)$.
```

B. ROLLO-II as a PKE

ROLLO-II, formerly known as LOCKER, is a CPA-secure Public Key Encryption (PKE) running in the category “post-quantum public-key encryption” and can be adapted for CCA2 security via the HHK framework for the Fujisaki-Okamoto transformation [33]. A PKE scheme is defined by three algorithms: the key generation algorithm KeyGen, which takes on input the security parameter λ and outputs a pair of public and private keys (pk, sk). The encryption algorithm Encrypt(M, pk), which outputs the ciphertext C corresponding to the message M and the decryption algorithm Decrypt(C, sk), which outputs the plaintext M.

A formal description of ROLLO-II is given in Algorithm 2. P is an irreducible polynomial in $\mathbb{F}_q[X]$ of degree n and constitutes a parameter of the cryptosystem. The symbol $\oplus$ denotes the bitwise XOR. The framework of ROLLO-II is almost identical to ROLLO-I except that ROLLO-II uses the randomly generated Hash(E) to mask the message M and then publishes it as a part of the ciphertext. ROLLO-II is a PKE scheme, and thus the decoding failure rate of the underlying coding system, i.e., LRPC codes must be constrained to an extremely low level (for example, 2^{-128}). Fulfillment of this requirement leads to a significantly larger key size (typically 2-3 times larger).

It is worth noting that at the core of the decapsulation/decryption step, the rank support recovery (RSR$(\cdot)$) algorithm requires computing the intersection of two vector spaces F and s, which is equivalent to Gauss-eliminating a large-sized matrix. This type of matrix is inevitably singular and very large such that the previous designs in the open literature can do nothing about it.

III. Design Parameters for ROLLO Underlying Arithmetic

This section describes ROLLO hardware at the bottom level. First, a flexible $\mathbb{F}_{2^m}$ arithmetic unit is presented. Based on

Algorithm 2: Formal Description of ROLLO-II.

```
1 KeyGen($1^\lambda$): Pick (x, y) $\xleftarrow{\$} S_d^{2n}(\mathbb{F}_{q^m})$. Set
   $h = x^{-1}y \bmod P$, and return $pk = h$, $sk = (x, y)$.
2 Encrypt(M,pk): Pick $(e_1, e_2) \xleftarrow{\$} S_r^{2n}(\mathbb{F}_{q^m})$, set
   $E = \mathrm{Supp}(e_1, e_2)$, $c = e_1 + e_2 h \bmod P$. Compute
   $cipher = M \oplus \mathrm{Hash}(E)$ and return the ciphertext
   $C = (c, cipher)$.
3 Decrypt(C,sk): Set $s = xc \bmod P$, $F = \mathrm{Supp}(x, y)$
   and $E \leftarrow \mathrm{RSR}(F, s, r)$. Return
   $M = cipher \oplus \mathrm{Hash}(E)$.
```

Algorithm 3: Digit-Level Interleaved Multiplication.

```
Input: polynomial $a(x) = \sum_{i=0}^{m-1} a_i x^i$,
       $b(x) = \sum_{i=0}^{m-1} b_i x^i$, irreducible polynomial
       $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$ with $a_i, b_i, f_i \in \mathbb{F}_2$
Output: $c(x) = a(x)b(x) = \sum_{i=0}^{m-1} c_i x^i \in \mathbb{F}_{2^m}$
1 $c(x) \leftarrow (\sum_{i=0}^{d-1} a_{m-d+i} x^i) \cdot b(x)$
2 for $i \leftarrow m/d - 2$ to 0      // scan $a(x)$ digit by
    digit
3 do
4     $c(x) \leftarrow c(x)x^d + (\sum_{\ell=0}^{d-1} a_{id+\ell} x^\ell) b(x)$
5 return $c(x)$
```

$\mathbb{F}_{2^m}$ arithmetic, an advanced $\mathbb{F}_{2^m}[z]$ polynomial multiplier supporting different area-time trade-offs is described. Further, a SHA3-based hash function and Keccak sponge-based random number generator specified for ROLLO are detailed. Finally, a constant-time and flexible Gaussian elimination module, the kernel of linear algebra-related computations, is presented.

A. $\mathbb{F}_{2^m}$ *Arithmetic*

In this subsection, we discuss our design choices for implementing $\mathbb{F}_{2^m}$ arithmetic with $m = 67, 79, 83, 97$ which covers the entire specified ROLLO parameter sets. In particular, these specified binay fields feature short irreducible polynomials as moduli, i.e., $f(x) = x^m + x^k + 1$ and $f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1 (k_3 > k_2 > k_1)$.

To balance the trade-off between time and area, we naturally extend the bit-level interleaved multiplication which interleaves standard multiplication and modulo reduction to the digit-level interleaved multiplication as described in Algorithm 3 based on Horner's method [34]: For each iteration of the algorithm, several bits (d-bits) of $a(x)$, i.e., $[a_{id+d-1}, \ldots, a_{id}]$ are read out to be multiplied by $b(x)$. Then the multiplication result is added by $c(x)x^d$ to accumulate $c(x)$.

Based on the description shown in Algorithm 3, we propose a generic architecture to perform the interleaved $\mathbb{F}_{2^m}$ multiplication as depicted in Fig. 1. The input polynomials $a(x)$ and $b(x)$ are loaded into two m-bit registers. `Mul` performs the multiplication of $a(x)$ and a digit fraction of $b(x)$ where the register holding $b(x)$ is cyclicly rotated to extract the target digit fraction of $b(x)$. The multiplication result from the component

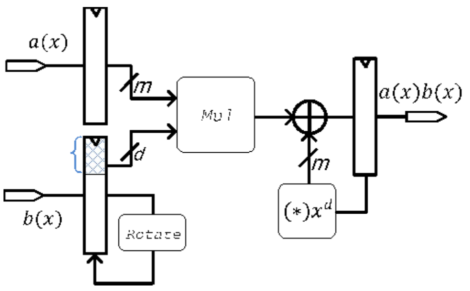


Fig. 1. Generic architecture for the $\mathbb{F}_{2^m}$ multiplier.

`Mul` is added by the output of the component $(*)x^d$, i.e., $c(x)x^d$, and finally updates the register $c(x)$. This process repeats $\lceil m/d \rceil$ cycles plus one extra cycle for the initial data loading, and thus the total delay of the multiplier is $\lceil m/d \rceil + 1$ cycles. In the actual implementations of ROLLO presented in this work, the multiplier is configured with bit-width of operand $d = 16$ and the cycle delay is denoted as Delay_{mul} which is a basic constant used in the next subsection.

B. $\mathbb{F}_{2^m}[z]/\langle P(z)\rangle$ *Arithmetic*

In this subsection, the arithmetic over $\mathbb{F}_{2^m}^n$, which is isomorphic to the specific polynomial ring $\mathbb{F}_{2^m}[z]/P(z)$ with extremely sparse polynomial $P(z) \in \mathbb{F}_2[z]$ of degree n, is discussed and optimized. Before detailing our algorithms on $\mathbb{F}_{2^m}^n$ multiplication, the representation of the element of $\mathbb{F}_{2^m}^n$ and its storage structure in the memory must be clarified. The element $A(z)$ of $\mathbb{F}_{2^m}^n$ is represented as $A(z) = \sum_{i=0}^{n-1} a_i z^i$ and thus, it is natural to store n coefficients of $A(z)$ into $\lceil n/d \rceil$ memory cells each of which contains d coefficients as shown in Fig. 2(a).

Basic Idea: The practical approach we consider is to multiply one coefficient a_i of $A(z)$ by a shifted $B(z)$ each time as follows:

$$A(z) \cdot B(z) = \sum_{i=0}^{n-1} a_i z^i \cdot B(z) = \sum_{i=0}^{n-1} a_i \cdot B(z) z^i.$$

Note that to implement the calculation of $a_i \cdot B(z)z^i$ on hardware, one cannot extract the entire $B(z)$ out of the memory to perform $B(z)z^i$ as what is done on $c(x)x^i$ in the $\mathbb{F}_{2^m}$ arithmetic. $B(z)$ is mn bits long, ranging between 5,561 and 20,467 bits, and therefore it is infeasible to be held by the registers within the FPGA fabric. A practical and also scalable solution is to perform $B^{(i)}(z) = B(z)z^i$ in the memory and later to perform the multiplication of a_i by every coefficient of $B^{(i)}(z)$ based on the proposed $\mathbb{F}_{2^m}$ multiplier. The memory-based rotation technique is first presented in [23] when tackling the multiplication over $\mathbb{F}_2[z]/z^r + 1$ used in BIKE. We extend their approach to more generic polynomial rings over $\mathbb{F}_{2^m}[z]/P(z)$. In a nutshell, $B^{(i)}(z) = B(z)z^i$ is a cheap recursive operation to be done in the memory as (assume $P(z)$ is a trinomial)

$$\begin{aligned} B^{(i+1)}(z) &= B^{(i)}(z) \cdot z \\ &= \mathrm{Cyclic}\left(B^{(i)}(z)\right) + b_{n-1}^{(i)} z^k, \end{aligned}$$

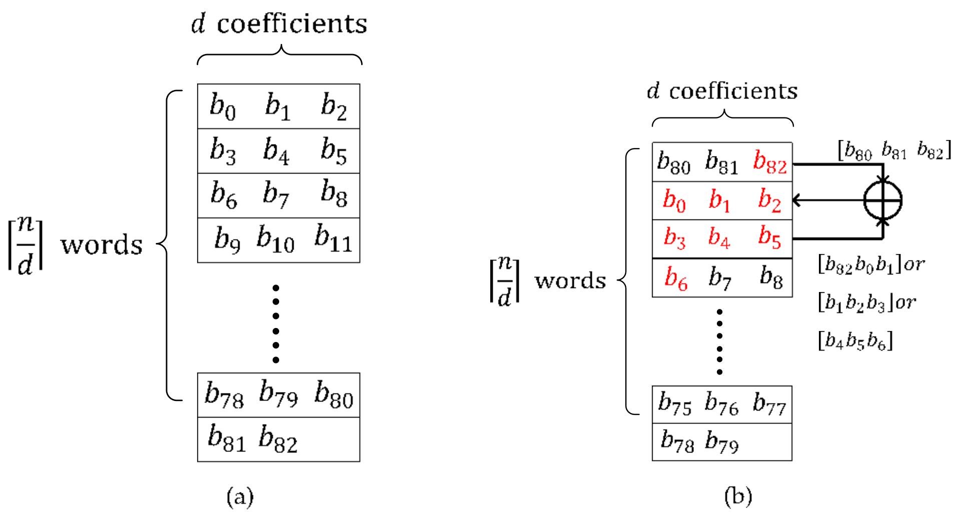


Fig. 2. Illustration for the memory storage format for polynomials over $\mathbb{F}_{2^m}$, taking ROLLO-I-128 for instance. (a) Polynomial $B(z) = \sum_{i=0}^{n-1} b_i z^i$ represented in memory, with $n = 83, d = 3$. (b) Operation to perform $B(z)z^d$ from $B(z)$ in memory, with $n = 83, d = 3$.

where $\text{Cyclic}(B^{(i)}(z)) = \sum_{j=0}^{n-1} b^{(i)}_{j-1} z^j$ denotes the cyclic shift of the polynomial $B^{(i)}(z)$ and thus the calculation of $B^{(i+1)}(z)$ is viewed as the cyclic $B^{(i)}(z)$ plus one (or three) term polynomial with coefficient b_{n-1}.

By use of this method, n $\mathbb{F}_{2^m}$ multiplications suffice to compute $a_i B(z) z^i$ and further n times iterative accumulation of $a_i B(z) z^i$ for $i = 0, 1, \ldots, n-1$ returns the desired result of $A(z)B(z)$. To sum up, the computational complexity of this approach is bounded by n^2 $\mathbb{F}_{2^m}$ multiplications yet the memory cost keeps unchanged.

Improvement: Our new technique for reducing the cycle counts of the polynomial ring multplication follows the suggestion of [25] to generalize their scheme used for the specific polynomial ring $\mathbb{F}_2[z]/z^r + 1$. Essentially, we exploit the hidden parallelism for computing $a_i B(z) z^i$ and $a_{i+1} B(z) z^{i+1}$ which aids the construction of $\mathbb{F}_{2^m}$ multiplier array.

Let $Delay_{mul}$ denote the cycle delay of one $\mathbb{F}_{2^m}$ multiplication. The total time delay of $\mathbb{F}_{2^m}^n$ multiplication is roughly estimated by $n^2 Delay_{mul}$ as we have discussed. This estimation can be reduced if multiple coefficients of $A(z)$ are simultaneously extracted, in other terms, we assume that the extraction of at most d coefficients from one memory cell is possible. Now the multiplication of $A(z)B(z)$ is rewritten as

$$A(z)\cdot B(z) = \sum_{i=0}^{\lceil n/d\rceil-1} A_i(z) z^{id}\cdot B(z) = \sum_{i=0}^{\lceil n/d\rceil-1} A_i(z)\cdot B(z) z^{id},$$

where $A_i(z) = a_{di} + a_{di+1}z + \cdots + a_{di+d-1}z^{d-1}$ is a polynomial of degree $d-1$ at most and essentially a digit fraction of the polynomial $A(z)$. By use of this format, $B^{(id)}(z) = B(z)z^{id}$ is efficiently performed in the memory based on the observation of the following recurrence (assume $P(z)$ is a trinomial):

$$\begin{aligned} B^{(i+1)d}(z) &= B^{(id)}(z)\cdot z^d \\ &= d\text{-Cyclic}\left(B^{(id)}(z)\right) + \sum_{i=0}^{d-1} b^{(id)}_{n-d+i} z^{k+i}, \end{aligned}$$

where $d\text{-Cyclic}(B^{(id)}(z)) = \sum_{j=0}^{n-1} b^{(id)}_{j-d} z^j$ denotes the cyclic shift of the polynomial $B^{(id)}(z)$ by d positions. Likewise, $B^{(id)}(z) = B(z)z^{id}$ can be rewritten as follows if $P(z)$ is a pentanomial:

$$\begin{aligned} B^{(i+1)d}(z) &= B^{(id)}(z)\cdot z^d \\ &= d\text{-Cyclic}\left(B^{(id)}(z)\right) + \sum_{k=k_1,k_2,k_3}\sum_{i=0}^{d-1} b^{(id)}_{n-d+i} z^{k+i}. \end{aligned}$$

Fig. 2(b) depicts the operation to perform this recurrence using the block memory with $n = 83, k_3 = 7, k_2 = 4, k_1 = 2, d = 3$ which is used in the ROLLO-I-128 instance. First, the cyclic rotation is performed such that the first entry of $B(z)$ moves forward to the second entry, the second entry moves to the third, and so forth. Later, the entry $[b_{80}b_{81}b_{82}]$ and the entry $[b_{82}b_0b_1]$ ($[b_1b_2b_3]$ and $[b_4b_5b_6]$ for k_2 and k_3, respectively) are extracted to sum up and then written back to complete the computation of $B(z)z^d$. Meanwhile, $[a_{id}, \ldots, a_{id+d-1}]$ are extracted from the memory to perform $A_i(z)\cdot B^{(id)}(z) = \sum_{j=0}^{d-1} a_{id+j}\cdot B^{(id)}(z)z^j$. Note this multiplication is split to d parts and each part, i.e., $a_{id+j}\cdot B^{(id)}(z)z^j$ can be parallelized to compute using multiple $\mathbb{F}_{2^m}$ multipliers (implemented as an array of $\mathbb{F}_{2^m}$ multipliers).

FPGA Architecture: The schematic architecture for the proposed multiplier is depicted in Fig. 3. The multiplier uses two block memories, i.e., `mem A` and `mem B` for input polynomials $A(z)$ and $B(z)$, and uses one block memory `mem C` for the result polynomial $C(z) = A(z)B(z) \bmod P(z)$. In particular, `mem B` is dual-ported to facilitate cyclic rotation within memory. The

TABLE II
PERFORMANCE OF THE CONFIGURABLE $\mathbb{F}_{2^m}[z]$ MULTIPLIERS FOR ROLLO-I ON XILINX ARTIX-7 FPGA

Instance	Quotient Ring	digit	freq	cycle	slice	memory	slice*cycle/freq
ROLLO-I-128	$\mathbb{F}_{2^{83}}[z]/(z^{83}+z^7+z^4+z^2+1)$	3	210	6192	2154	12	$6.4*10^4$
		4	200	3616	3757	15	$6.8*10^4$
		5	**190**	**2452**	**4285**	**18**	$5.5*10^4$
		6	180	1726	5454	21	$5.2*10^4$
		7	180	1312	6425	24	$4.7*10^4$
ROLLO-I-192	$\mathbb{F}_{2^{97}}[z]/(z^{97}+z^6+1)$	3	210	7924	2931	14	$1.1*10^5$
		4	200	4604	3587	17	$8.2*10^4$
		5	**190**	**2984**	**4584**	**20**	$7.2*10^4$
		6	180	2180	5201	24.5	$6.3*10^4$
		7	170	1502	6312	27.5	$5.6*10^4$
ROLLO-I-256	$\mathbb{F}_{2^{113}}[z]/(z^{113}+z^9+1)$	3	200	13418	2737	17.5	$1.8*10^5$
		4	190	7892	4586	21	$1.9*10^5$
		5	**180**	**5018**	**5350**	**25**	$1.5*10^5$
		6	150	3462	6670	29.5	$1.5*10^5$
		7	150	2792	7488	33	$1.4*10^5$

The bold suggests the optimal configuration.

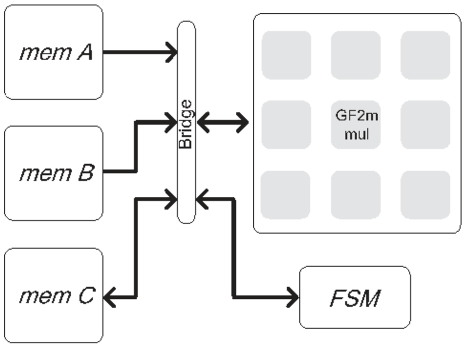


Fig. 3. Schematic of the proposed for $\mathbb{F}_{2^m}[z]$ multiplier using the computation matrix core of $\mathbb{F}_{2^m}$ multipliers.

kernel of the multiplier consists of the $d \times d$ $\mathbb{F}_{2^m}$ multiplier array under control of a finite state machine FSM. FSM initializes at state INIT and transits to state PRE for data pre-processing. Then it switches repeatedly between the state MUL for $\mathbb{F}_{2^m}$ multiplications and the state POST for data post-processing, and finally moves back to state INIT.

Performance: For the trinomial $P(z)$, INIT takes 1 clock cycle, PRE takes 3 clock cycles, MUL takes $(Delay_{mul} + 1)dn/de$ cycles, POST takes $Delay_{mul} + 3$ clock cycles. MUL and POST repeat dn/de times. In summary, the total cycle counts are calculated as: $4 + dn/de \cdot [(Delay_{mul} + 1)dn/de + Delay_{mul} + 3] \approx (Delay_{mul} + 1)\ \frac{n}{d}\ ^2$.

Likewise, for the pentanomial $P(z)$, INIT takes 1 clock cycle, PRE takes 3 clock cycles, MUL takes $(Delay + 1)dn/de$ cycles, POST takes $Delay + 19$ clock cycles. In summary, the total cycle counts are calculated as: $4 + dn/de \cdot [(Delay_{mul} + 1)dn/de + Delay_{mul} + 19] \approx (Delay_{mul} + 1)\ \frac{n}{d}\ ^2$. The concrete FPGA implementation results for $\mathbb{F}_{2^m}[z]$ used in ROLLO-I and ROLLO-II are reported in Table II and Table III, respectively.

C. Hash Function and Random Number Generator

We use SHA-3 as the hash engine for ROLLO encapsulation in our design. For easier migration from a stand-alone SHA3 implementation to our ROLLO coprocessor, we choose a well-documented open-sourced SHA3 project from OpenCores.org [35]. In ROLLO implementations, SHA3 is used to hash a very limited number of data, i.e., the low-rank (typically 7,8,9) vector space. Therefore, the low-throughput open-sourced core fits well, and we use it as the ROLLO hashing engine. The performance of the proposed SHA3 hash architecture is evaluated on Xilinx Artix-7 as shown in Table IV. The experiment focuses on the execution of the hash function ($K = \mathtt{Hash}(E)$ listed in the formal description of ROLLO, see Algorithm 1) used in the encapsulation and decapsulation of ROLLO-I. The results reaffirm the efficiency of our new design: The hashing process only takes 50 ⍰ 175 cycles, runs at 190 MHz, and consumes about 619 slices only.

We choose $SHAKE - 128$ which uses KECCAK-$[r = 1344, c = 256]$ as the underlying random number generator (RNG) to replace the relatively bulky AES-based RNG proposed in the ROLLO NIST submission package. This KECCAK-based RNG can be built in a very compact core that is suitable for devices with scarce resources as the state can be stored in 200 bytes and can produce 1,344 pseudo-random bits within 24 clock cycles. On the security aspects, KECCAK-$[r = 1344, c = 256]$ based RNG provides 128-bit post quantum security. The security level can scale up if the KECCAK capacity c increases. For example, KECCAK-$[r = 1088, c = 512]$ provides provides 256-bit post quantum security. It is worth noting that the Keccak-based RNG deviates from the ROLLO specification and that hence known-answer-test of the ROLLO submission does not apply here. However, our design is modular and can be replaced by the AES-based RNG if the users request a standard implementation that strictly follows the specification. We have synthesized our $SHAKE - 128$ based RNG in Xilinx Artix-7 FPGAs, and the post place-and-route reports show that the throughput of Keccak-PRNG is 11.2 Gbit/s, running at 200 Mhz and occupying 600 slices.

D. Gaussian Elimination on Systolic Array

Gaussian elimination is the most computing-intensive and distinguishing operation in rank metric cryptography, particularly in the context of ROLLO. The performance of Gaussian elimination on a large-sized matrix over a finite field is

TABLE III
PERFORMANCE OF THE CONFIGURABLE $\mathbb{F}_{2^m}[z]$ MULTIPLIERS FOR ROLLO-II ON XILINX ARTIX-7 FPGA

Instance	Quotient Ring	digit	freq	cycle	slice	memory	slice*cycle/freq
ROLLO-II-128	$\mathbb{F}_{2^{189}}[z]/(z^{189}+z^6+z^5+z^2+1)$	3	200	33394	2383	14	$4.0 * 10^5$
		4	190	19684	4149	18.5	$4.3 * 10^5$
		5	**180**	**12544**	**4933**	**22**	$3.4 * 10^5$
		6	170	9028	6416	25	$3.4 * 10^5$
		7	160	6538	7918	29	$3.2 * 10^5$
ROLLO-II-192	$\mathbb{F}_{2^{193}}[z]/(z^{193}+z^{15}+1)$	3	200	38744	2664	17.5	$5.2 * 10^5$
		4	190	22152	4561	21	$5.3 * 10^5$
		5	**180**	**14122**	**5457**	**25**	$4.3 * 10^5$
		6	170	10168	6900	29.5	$4.1 * 10^5$
		7	160	7368	7825	33	$3.6 * 10^5$
ROLLO-II-256	$\mathbb{F}_{2^{211}}[z]/(z^{211}+z^{11}+z^{10}+z^8+1)$	3	190	47290	2789	17.5	$6.9 * 10^5$
		4	170	26716	4655	21	$7.3 * 10^5$
		5	**160**	**17806**	**5773**	**25**	$6.4 * 10^5$
		6	150	12640	7785	29.5	$6.6 * 10^5$
		7	150	9490	8740	33	$5.5 * 10^5$

The bold suggests the optimal configuration.

TABLE IV
CONFIGURABLE SHA3 HASH ENGINE FOR ROLLO ON XILINX ARTIX-7 FPGA WHERE `digit` DENOTES THE NUMBER OF $\mathbb{F}_{2^m}$ ELEMENTS STORED IN ONE BLOCK MEMORY CELL

Instance	digit	freq	cycle	slice	memory	slice*cycle/freq
ROLLO-I-128	1	190	125	625	1.5	411
	2		75	654	2	258
	3		50	651	2	171
	4		50	644	2	169
ROLLO-I-192	1	190	150	625	2	493
	2		75	660	2	260
	3		50	644	2	169
	4		50	642	2	168
ROLLO-I-256	1	190	175	632	2	582
	2		100	648	2	341
	3		75	652	2	257

another bottleneck other than the multiplication over the ring $\mathbb{F}_{2^m}[z]/P(z)$. For example, ROLLO-II PKE.encrypt generates a random $r \times m$ matrix over $\mathbb{F}_2$ to represent the error vector space E, which requires performing Gaussian elimination to get its reduced-row-echelon form. ROLLO-II PKE.decrypt requests Gaussian elimination to intersect the secret syndrome spaces $S_i = f_i^{-1}S$ for finding the r linearly independent bases of the secret error vector space E. We are facing a new challenge in rank-code based cryptosystem, namely, triangularizing a singular matrix in ROLLO. In this subsection, we will detail our generalized approach, which not only solves this new problem but is also applicable to the Gaussian elimination cases used in the classic Niederreiter cryptosystem and multi-variate cryptosystem.

1) Previous Work: From a geometric point of view, the hardware architectures for Gaussian elimination over a finite field fall into two groups: triangular and linear, each of which is subdivided into three types: systolic array [36], systolic network [37], and systolic line [38]. In these Gaussian elimination designs, the data is fed into a processor array in a systolic fashion and eventually the result of Gaussian elimination is stored in the prcoeesor array and can be streamed out for post-processing if required.

a) Triangular shaped array: Triangular shaped array is a two dimensional array where all nodes in the array shape a triangle. This array is triangular because Gaussian elimination causes all nodes except the pivot node to be zero for each column of the matrix and these zeros are unnecessary to be saved. In 1989, Hochet et al. described for the first time the triangular systolic array for doing Gaussian elimination a matrix over $\mathbb{F}_q$ [36]. This work was further adapted for faster processing using systolic network [37] and systolic line [38]. For code based cryptography, this architecture for Gaussian elimination is applied to large matrices over $\mathbb{F}_2$ specified in the key-pair generation of the classic McEliece cryptosystems based on Goppa code. See [19], [39] for details. In general, triangular-shaped array sets the priority for time complexity while sacrificing space complexity. It typically completes one Gaussian elimination for a $k \times l$ matrix in $\Theta(k + l)$ of time and $\Theta(kl)$ of space.

b) Linear shaped array: Linear shaped array is one dimensional linear-shaped array where all nodes in the array shape a horizontal line as described in [38]. It only preserves the first line of the triangular array while all intermediate results are pushed to an array of shift registers waiting for the next round of processing. Linear systolic array is more area-efficient than the triangular-shaped array if the Gaussian elimination is performed on a matrix over an extended finite field $\mathbb{F}_{2^m}$. It typically completes one Gaussian elimination for a $k \times l$ matrix in $\Theta(kl)$ of time and $\Theta(l)$ of space.

These previous designs are capable of eliminating a binary matrix of the size $k \times l$, with $k = l$, which removes the shape limit existing in the Gaussian elimination hardware mentioned above. The pre-requisite for successful Gaussian elimination is that the input matrix must be full-rank. Gaussian elimination is the most computing-intensive and also a distinguishing operation in rank-metric-code-based cryptography like ROLLO and RQC. However, these rank-code-based schemes require performing Gaussian elimination on mediumsize and large-size matrices over a binary field, and more importantly, these matrices can be rank-deficient. In contrast, the current state-of-the-art designs cannot process such type.

2) Main Idea: Dynamical Dual-Mode Switch: In our work, we focus on Gaussian elimination for matrices over $\mathbb{F}_2$ which is the type of Gaussian elimination used in ROLLO. The process of such Gaussian elimination makes use of two types of elementary row operations which must be performed on the rows of a matrix:

- Swap the positions of two rows.
- Add one row to another row.

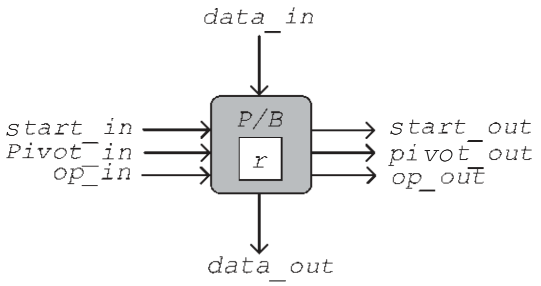
By using these row operations, a matrix can always be transformed into an upper triangular matrix (row-echelon form), and further simplified to reduced row-echelon form where the leftmost nonzero entry in each row is 1, and every column containing such leftmost nonzero entry has zeros elsewhere. In this work, we use the term 'triangularization' to denote the operation of putting the input matrix into its row-echelon form and 'systemization' to denote the operation of putting a row-echelon formed matrix into its reduced-row-echelon form.

The most challenging part of this work is that the position of pivot nodes in our Gaussian elimination architecture is flexible. In contrast, all previous work [36], [37], [38], [40] assumes the pivots are always found along the diagonal. This flexibility of pivot position renders these works inapplicable in ROLLO. Our new idea of implementing Gaussian elimination is as follows: Each node is configured to have dual functions for every iteration of Gaussian elimination: either the pivot node or the basic node depending on the data input from the above node and the control input from the left-hand-side node. The pivot node behaves actively as the pivot in that particular row and propagates the operational signal to its right-hand-side basic nodes. The basic node behaves passively according to the operation signal, i.e., PASS, ADD, or SWAP for performing elementary row operations, which are specified as follows:

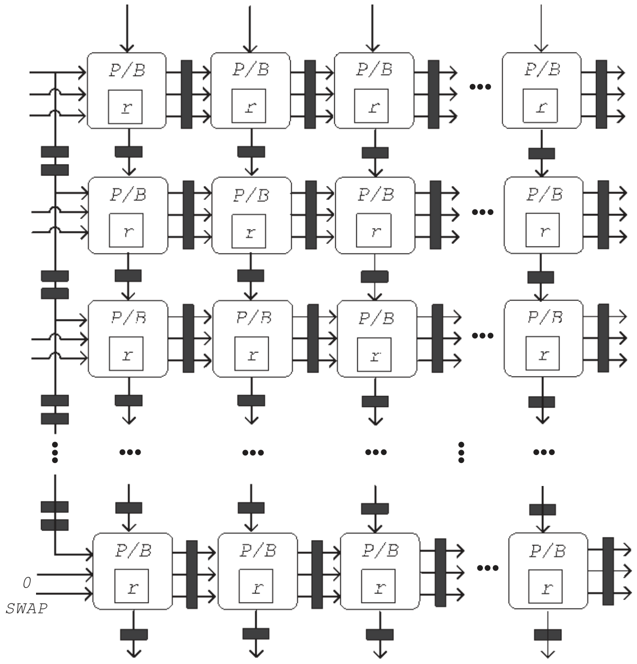
- PASS: The node passes the input data into the output port and retains the data stored in the internal register of the node.
- ADD: The node adds the input data and the internal register data, and then outputs sum. Meanwhile, the node retains the internal register data.
- SWAP: The node swaps the input data and the internal register data, i.e., the node outputs the internal register data and then updates the internal register with the input data.

A detailed description of the proposed node is shown in Fig. 4(a). It has 8 signals and 6 of them are identical to the ports of classic nodes presented in the literature including data_in, data_out, start_in, start_out, op_in and op_out. The difference is that a new pair of signals, pivot_in and pivot_out is used to determine whether the current node is pivot or not and broadcast this message to its right neighboring node. With such new mechanism, the node can dynamically switch between pivot and non-pivot for each input data update described in Fig. 5: *start_in* triggers the initial phase when the data flushes into the internal register r for the first time. Otherwise, the node resides in the normal phase when it acts as either the pivot node or the basic(non-pivot) node depending on the 2-bit signal {r, pivot_in}: if {r, pivot_in}= 2^0b10, the node acts as pivot. Otherwise, it acts as basic node.

3) Triangular Systolic Array Design: With the new node design for Gaussian elimination, we chose a triangular systolic array (TSA) as the basic architecture for implementing Gaussian elimination in this work. Fig. 4(b) depicts the overview of the Gaussian elimination systolic array for any matrices over $\mathbb{F}_2$ including singular and non-singular ones. The basic structure is arranged in a rectangular shape such that every signal of



(a)



(b)

Fig. 4. A versatile node proposed to solve non-fixed-pivot row exceptions in Gaussian elimination. (a) Signal declaration of the new design of the node. (b) Overview of the two dimensional array for Gaussian elimination, each node is pipelined to reduce the critical path delay.

the node is pipelined, allowing all data and control signals to be propagated in a systolic manner. An improvement of this architecture is that all r registers in the nodes below the diagonal of the systolic array are always zero, whatever the input matrix is after the Gaussian elimination is done. Therefore, these nodes (which are drawn in dotted lines) can be removed and finally result in a triangular-shaped array.

4) Gaussian Elimination for ROLLO Encryption: In the encryption part, the matrix has relatively small dimension of $r \times m$, e.g., $r = 7, 8, 9$ and $m = 67, 79, 97$ are used in ROLLO-I. In this case, it is natural to realize the entire Gaussian triangularization/systemization using a single systolic array. Note that systemization is necessary to acquire a unique representation of error vector space **E** such that the subsequent hash function always outputs a correct shared key. To better understand the mechanism of the proposed Gaussian elimination architecture, Fig. 5 describes the behavior of the node in triangularization and Fig. 6 illustrates a step-by-step procedure for a single systolic

```
//init phase
 if (start_in)
    r <= data_in  data_out=1’b0  op_out=‘SWAP’
    pivot_out = pivot_in ? 1’b1 :
                        data_in ? 1’b1 : 1’b0
//normal phase
 else
    //pivot node, work actively
    if ({r, pivot_in} == 2’b10)
        r <= r  pivot_out = 1’b1
        if (data_in)
            data_out = data_in ^ r
            op_out = ‘ADD’
        else
            data_out = data_in
            op_out = ‘PASS’
    //if pivot still not found, perform ‘SWAP’
    else if ({r, pivot_in} == 2’b00)
        r <= data_in  pivot_out = data_in
        data_out = r  op_out = ‘SWAP’
    //basic node, work passively
    else
        op_out = op_in  pivot_out = pivot_in
        if (op_in == ‘SWAP’)
            data_out = r  r <= data_in
        else if (op_in == ‘ADD’)
            data_out = data_in ^ r  r <= r
        else if (op_in == ‘PASS’)
            data_out = data_in  r <= r
```

Fig. 5. Behavior description of the node used in the proposed systolic array for matrix triangularization, written in Verilog-like pseudocode.

array to transform a 4×4 matrix $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$ to its row echelon form $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ within 10 clock cycles. The data colored in red indicates the value stored in the `r` register of the node. The data colored in blue indicates the buffered `data_out` signal in the pipeline register between two neighboring nodes as shown in Fig. 4(b). The circled value indicates it is the current pivot of that particular row. Note that the input matrix must be fed into the array in a skewed form for systolic processing as shown in step-(0), i.e., at the first clock cycle, the systolic array takes one bit ‘0’ as input. At the second clock cycle, the systolic array takes two bit ‘00’ as inputs, and so forth. In step-(1), ‘0’ has been updated to the node at the upper-left corner of the systolic array. Given the input signals `{r,pivot_in}==2’b00`, this particular node executes the `SWAP` operation, which updates `r` by ‘0’, outputs ‘0’ and also passes ‘`SWAP`’ signal and `pivot_out` = 0 to its right neighbor. In step-(2), consider the first row of the systolic array, the leftmost node executes `SWAP` again since `{r, pivot_in} == 2’b00` and the second node executes ‘`SWAP`’, which is passed from the leftmost node in step-(0). In step-(3), the second node in the first row acts as the pivot since `{r, pivot_in} ==2’b10`. This pivot node (circled in the figure) ignores the ‘`SWAP`’ signal passed from step-(2) but performs ‘`PASS`’ as a replacement. An analogous pattern can be found in step-(4), where the pivot node in the first row ignores ‘`SWAP`’ but executes ‘`ADD`’ since the input data is ‘1’. Eventually, when all input data are flushed into the internal registers of all nodes distributed at four distinct rows as shown in step-(10), the input matrix has been successfully eliminated in the desired row echelon form. In summary, the total delay for triangularizing a $k \times l$ matrix is $2k + l - 2$.

On the other hand, the Gaussian systemization is required right after the triangularization process to shape the matrix to the systematic form. Fig. 7 describes the behavior logic of the node. In our actual implementations, the functionalities for triangularization and systemization are merged to a single node.

Finally, we test the performance of our systolic array for Gaussian elimination in compliance with ROLLO-II.encrypt. The implementation results are collected in Table V.

TABLE V
GAUSSIAN ELIMINATION PERFORMANCE (TRIANGULARIZATION + SYSTEMIZATION) FOR A $k \times l$ MATRIX USED IN ROLLO-II.ENCRYPT ON XILINX ARTIX-7 FPGA

Instance	$k \times l$	freq	cycle	slice	slice*cycle/freq
ROLLO-II-128.encrypt	7×67	400	153	1149	439.5
ROLLO-II-192.encrypt	8×79	400	180	1575	708.8
ROLLO-II-256.encrypt	9×97	400	219	2131	1166.7

5) Gaussian Elimination for ROLLO Decryption: ROLLO decryption requires calculating the intersection of two vector spaces in the rank support recovery algorithm and later systemizing the intersected vector space to reconstruct the secret shared key K. Such intersection uses the Zassenhaus algorithm in which the Gaussian elimination for a large $2n \times 2m$ matrix over $\mathbb{F}_2$ is performed. In this case, it is infeasible to realize the large-scale elimination on systolic array directly since the resource utilization has exceeded the maximum capacity of Xilinx Artix-7 FPGAs. The solution proposed in this work is to divide the large matrix into several smaller blocks and to conquer each submatrix using a relatively small systolic array. We are particularly interested in a division where the smaller blocks share the same column width as the large matrix. In other terms, we use a $d \times 2m$ Gaussian elimination systolic array to process larger $2n \times 2m$ matrices with $m = 67, 79, 83, 97$, $n = 83, 97, 113, 189, 193, 211$ and $d = 8, 9$. This requirement removes the storage of intermediate operation codes (`op_in` signals from each Gaussian elimination node).

We detail our idea of using a smaller systolic array to triangularize a larger matrix here. The node behavior mode must be modified such that the node can correctly load data in or load data off to the external memory. Therefore, we add a new feature, called `swap_in`, to the input signal lists of the node as shown in Fig. 8. `swap_in` is triggered to output the data within the internal register and, menwhile, update the register by the input data at the specific timing when the systolic array requires to load the register data off to the memory. A simple example, i.e., to transform a 4×4 matrix $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$ to its row echelon form $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ on a 2×4 systolic array is

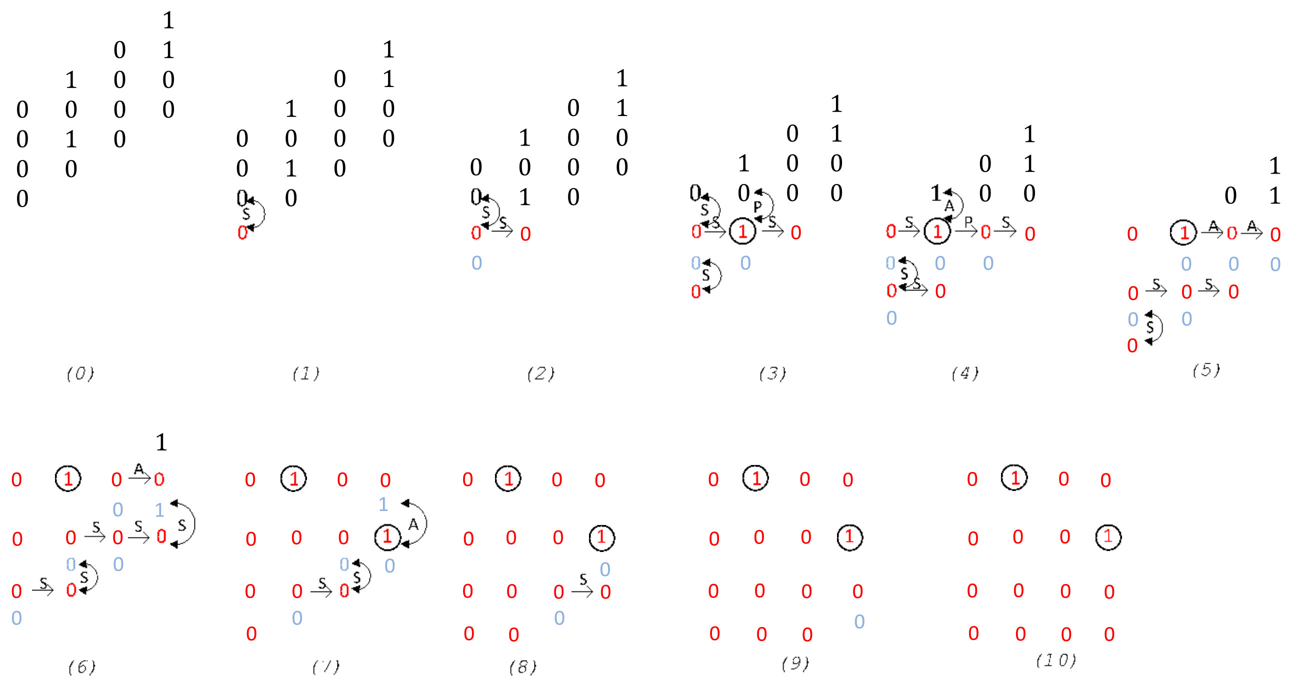


Fig. 6. A toy example for the proposed Gaussian elimination hardware by triangularizing a 4×4 matrix over $\mathbb{F}_2$. Triangularization means putting the input matrix into its row-echelon form. The signals ‘S’, ‘P’, and ‘A’ stand for ‘SWAP’, ‘PASS’ and ‘ADD’, respectively.

```
//init phase
if (op_in == 'SWAP')
    r <= r  data_out = r  op_out = 'SWAP'
    pivot_out = pivot_in ? 1'b1 : r ? 1'b1 : 1'b0
//normal phase
else
    //pivot node, work actively
    if ({r, pivot_in} == 2'b10)
        r <= r  pivot_out = 1'b1
        if (data_in)
            data_out = data_in ^ r
            op_out = 'ADD'
        else
            data_out = data_in
            op_out = 'PASS'
    //if pivot still not found, perform 'PASS'
    else if ({r, pivot_in} == 2'b00)
        r <= r  pivot_out = 1'b0
        data_out = data_in  op_out = 'PASS'
    //basic node, work passively
    else
        op_out = op_in  pivot_out = pivot_in
        if (op_in == 'ADD')
            data_out = data_in ^ r  r <= r
        else if (op_in == 'PASS')
            data_out = data_in  r <= r
```

Fig. 7. Behavior description of the node used in the proposed systolic array for matrix systemization, written in Verilog-like pseudocode.

```
//init phase
if (start_in)
    r<=data_in  data_out=1'b0  op_out='SWAP'
    pivot_out=pivot_in ? 1'b1 :
                   data_in ? 1'b1 : 1'b0
//new feature, used in load-off phase
else if (swap_in)
    r<=data_in  data_out=r  op_out='SWAP'
    pivot_out = pivot_in ? 1'b1 :
                data_in ? 1'b1 : 1'b0
//normal phase
else
    op_out = op_in  pivot_out = pivot_in
    if (op_in == 'SWAP')
        data_out = r  r <= data_in
    else if (op_in == 'ADD')
        data_out = data_in ^ r  r <= r
    else if (op_in == 'PASS')
        data_out = data_in  r <= r
```

Fig. 8. Behavior description of the node used in the proposed systolic array for matrix triangularization specified for ROLLO decryption, written in Verilog-like pseudocode.

depicted step-by-step in Fig. 9. The entire process requires two rounds of Gaussian eliminations: The first round costs 10 steps which manipulate the entire four rows of the input matirx and eliminate the first two rows, and finally load the four rows back to memory; The second round costs 8 steps which manipulate only the last two rows of the input matrix and then load back to memory.

In Round-1, initially at step-(0), the input matrix $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$ is prepared in skewed form and fed to the array; At step-(1), the upperleft node accepts ‘0’ to its internal register and `{r, pivot_in}==2'b00` triggers ‘`SWAP`’ signal; At step-(2), on the one hand, the upperleft node outputs ‘0’ to the buffer register due to the ‘`SWAP`’ signal from step-(1),

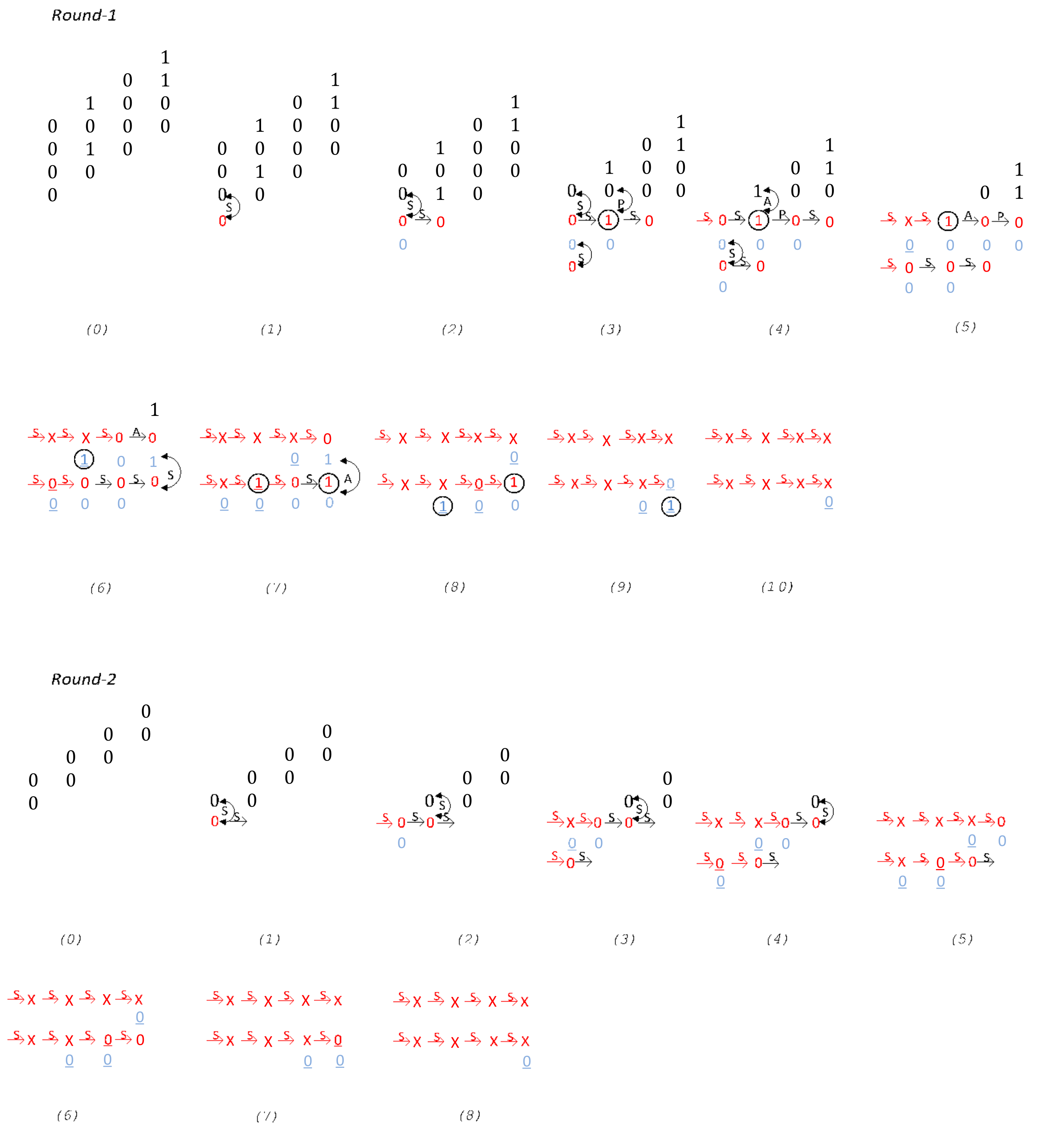


Fig. 9. A toy example to transform 4×4 matrix over F_2 by the proposed 2×4 systolic array. Triangularization refers to putting the input matrix into its row-echelon form.

and again performs ‘SWAP’ since `{r, pivot}==2'b00`. On the other hand, the second node in the first row of the array receives ‘SWAP’ passed by the leftmost node in the last step and therefore, executes ‘SWAP’ accordingly; At step-(3), the second node in the first row acts as a pivot since `{r, pivot_in}==2'b10`; At step-(4), the `swap_in` is externally triggered on the upperleft node for loading-off; Starting from step-(5), the `swap_in` signals of the two leading nodes of the respective rows of the systolic array keep enabled until the array finally loads all effective data out to the external memory at step-(10). It is worth mentioning that the systolic array outputs the result matrix $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ in reversed order, i.e., first, it outputs the last row, then second last one, and eventually the first one. It is easily seen that the first two rows of the result matrix has been eliminated correctly at the end of Round-1.

The Round-2 process mostly repeats what has been described for Round-1 except that the input matrix has two rows which are extracted from the last two rows of the result matrix mentioned in Round-1. In general, it costs D/d (assume $d \mid D$ for simplicity) rounds for triangularizing a $D \times l$ matrix with a single $d \times l (d < D)$ systolic array within about $\frac{(D+2l)\cdot D}{2d}$ cycles.

To sum up, the architectures presented in this subsection are two-fold: The Gaussian elimination module for ROLLO encryption is based on systolic array design and the new dual-mode switching node for processing a (singular) matrix; the Gaussian elimination module for ROLLO decryption reuses the former to process any large-sized matrices while preserving constant resource utilization. The proposed method for Gaussian elimination is constant-time and thus is secure against timing attacks. In addition, the proposed systolic array for Gaussian elimination is fully parameterized at compile time to support rapid configurations for different sets of parameters without the need to re-write the hardware code.

IV. Higher Level Description for ROLLO and Hardware Architecture

This section describes the ROLLO hardware at a higher level. It is worth mentioning that the CPA-secure ROLLO can be converted to a CCA2-secure KEM when the HHK [33] framework for the Fujisaki-Okamoto transformation is applied. Therefore, we focus on the CCA2-secure parameter sets and include the core functionalities, e.g., encryption and decryption in this work. First, the ROLLO encryption hardware, together with its key components, is presented, evaluated, and compared with related work. Then the ROLLO decryption hardware is presented, evaluated, and compared. In particular, the Rank Support Recovery algorithm which is the kernel of the decoding of LRPC codes is optimized and further adapted for efficient hardware implementation.

A. ROLLO Encapsulation/Encryption

ROLLO-I is a KEM = (KeyGen, Encap, Decap) but ROLLO-II is a PKE = (KeyGen, Encrypt, Decrypt). In this subsection, we bundle the implementation discussions on ROLLO-I. Encap and ROLLO-II.Encrypt as they share similar descriptions.

Error Vector Space E: The first step of encryption is to generate a small dimensional subspace of $\mathbb{F}^n_{q^m}$ This subspace is called the error vector space E and let its dimension be r. A random $r \times m$ matrix over $\mathbb{F}_q$ is generated from our `Keccak-PRNG` (as introduced in Section III-C) and fed into the Gaussian elimination systolic array for rank determination. This random matrix is accepted to represent E if and only if its rank is r.

Gaussian Elimination: Although the randomly generated **E** is invertible with overwhelming probability, it is necessary to fully implement the Gaussian elimination on **E** to obtain its reduced-row-echelon form **E_rref**, and thus the representation of the vector space **E** is uniquely determined for the correctness of the decryption. Gaussian elimination for **E** includes two phases: triangularization and systemization which costs $2k + l - 2$ and $k + l$ cycles, respectively. The total execution of Gaussian elimination (including necessary memory access) consumes $3k + 2l + 4$ cycles.

SHA3 Hashing: The standard SHA3 hashing (see Section III-C) costs $25\lceil r/digit\rceil$ where $digit$ is the number of $\mathbb{F}_{2^m}$ elements stored in one block memory cell and in our implementation $digit$ is set to 1. Moreover, the encryption engine exploits a centralized control unit for all computation components; thus, interaction among them takes slightly longer. The modified hashing core costs two more cycles to hash the content in one memory cell and thus uses $27r$ plus a few more cycles for state logic transition.

Random Errors from E: For all ROLLO instances, it is necessary to generate some random vectors with small rank weight from the error vector space **E**. To achieve this, all the basis vectors from **E** are first extracted and then inserted at a random address to the block memory representing that random error. Later, a random linear combination of all the basis vectors is performed and inserted at other addresses of the same block memory. Such random linear combination is essentially the vector-matrix product of $[a_1, a_2, \ldots, a_r] \cdot \begin{bmatrix} \tilde{e}_1 \\ \tilde{e}_2 \\ \vdots \\ \tilde{e}_r \end{bmatrix}$ where a_i is the random bit generated from the `Keccak-PRNG` and the basis vector $\tilde{e}_i$ comes from the error vector space **E**. Since a_i is binary and the actual implementation is cheap XORs among the selected $\tilde{e}_i$'s, which is done in a single clock cycle. The performance bottleneck is how fast the `Keccak-PRNG` generates the random bits. In our case, the `Keccak-PRNG` generates 1344 bits within 25 cycles and therefore, it takes about $25 \cdot \lceil dn/\lfloor\frac{1344}{r}\rfloor\rceil$ cycles to generate such a random error vector with n $\mathbb{F}_{2^m}$ entries.

Quotient Ring Multiplication: $\mathbb{F}_{2^m}[z]$ multiplication is the most time-consuming operation in the ROLLO encryption scheme. We use the proposed $\mathbb{F}_{2^m}[z]$ multiplier based on the $\mathbb{F}_{2^m}$ multiplication array shown in Section III-B. To facilitate the pipeline streaming for the error vector generation, we set the digit width of the $\mathbb{F}_{2^m}[z]$ multiplier to a small constant number (Precisely, we set in the actual implementations for each memory cell holding 5 $\mathbb{F}_{2^m}$ coefficients). Our parameterized multiplier allows tuning the digit width and adjusting the resource utilization as desired. Note the quotient ring multiplier is the densest core of the entire ROLLO encryption hardware, and the freedom for such modularity makes implementing all variants of ROLLO on Artix-7 FPGA possible.

Hardware Architecture: Fig. 10 depicts the overview architecture for the ROLLO encryption. In ROLLO-I, The `RNG` provides the necessary randomness to drive `Low-Rank Polynomial Generator` for generating the error space E and subsequently for the two 'small' error vectors e_1, e_2. `Gaussian Systemizer` transforms E to its reduced row echelon form E_{rref} and then checks its rank value. Finally, the cipher c is calculated via the polynomial multiplier and adder, and the other cipher K is calculated by hashing E_{rref}. Likewise,

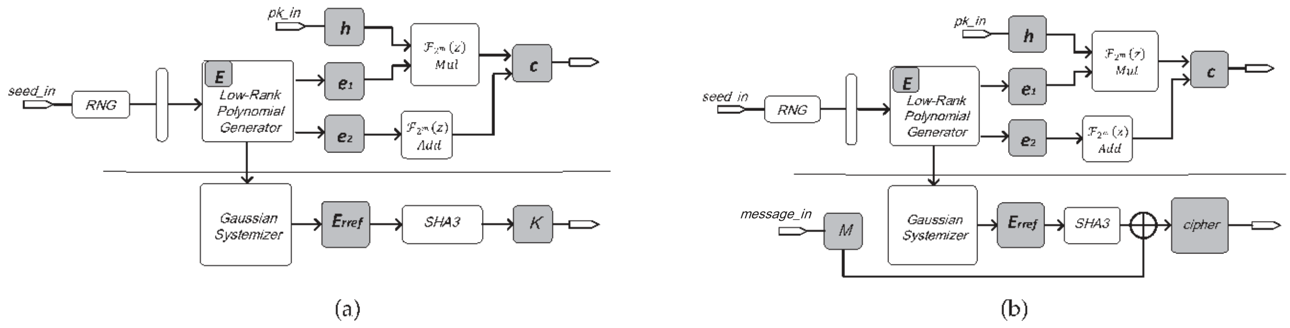

pk_in
h
seed_in
RNG
E
Low-Rank Polynomial Generator
e1
e2
$\mathcal{F}_{2^m}(z)$ Mul
$\mathcal{F}_{2^m}(z)$ Add
c
Gaussian Systemizer
Erref
SHA3
K
(a)
pk_in
h
seed_in
RNG
E
Low-Rank Polynomial Generator
e1
e2
$\mathcal{F}_{2^m}(z)$ Mul
$\mathcal{F}_{2^m}(z)$ Add
c
message_in
M
Gaussian Systemizer
Erref
SHA3
cipher
(b)

Fig. 10. ROLLO encryption hardware. (a) Hardware architecture for ROLLO-I key encapsulation. (b) Hardware architecture for ROLLO-II data encryption.

the architecture for ROLLO-II encryption is almost identical to that for ROLLO-I except for the way of manipulating the final ciphertext: ROLLO-I outputs the hash value K directly, whereas ROLLO-II encrypts the message m by XORing K. Moreover, the circuit size for ROLLO-II is generally larger since ROLLO-II requires an extremely low decoding failure rate to satisfy the security requirement. This reacts to increase the size of parameter sets.

B. ROLLO Decapsulation/Decryption

1) Rank Support Recovery Algorithm: ROLLO NIST submission package includes a constant-time algorithm for the Rank Support Recovery (RSR) subroutine used in the decryption algorithm as shown in Algorithm 4. The idea to recover the rank support from the syndrome space S is to compute $S_i = f_i^{-1}S$ for each i which contains the error support e_j $(j = 1, 2, \ldots r)$ of the error space E and then to retrieve the entire error space E as the intersections of all S_i's. However, the intersections may not return E correctly due to two factors. First, the syndrome space S may not contain all $f_i e_j$ for $i = 1, \ldots, d$ and $j = 1, \ldots, r$ as its rank support, then some S_j's only contain a (strict) subset of $e_1, e_2, \ldots, e_r$ and thus the intersections of these S_j's cause the failure of rank support recovery. Second, the intersection of S_i's may contain some shared vectors that are not the rank support, and thus the intersection returns an incorrect vector space larger than the expected E. Nevertheless, an appropriate selection of system parameters n, m can fix this problem and reduce the probability of decryption failure exponentially. Therefore, ROLLO is immune to potential reaction attacks [41], [42] which exploit the decryption failure.

2) Vector Space Intersection: Single Vector Space Intersection: At the heart of RSR algorithm, the intersection of two vectors space S_i and S_j dominates the performance, and worse still, such intersection is repeated for $d - 1$ times. Therefore, special care must be taken for efficient implementations. We first consider a single round of vector space intersection. The standard technique for such intersection is performed by Zassenhaus algorithm as follows: Let S_1 and S_2 be represented in an n-by-m binary matrix form as $[s_{\tilde{1},1}, s_{\tilde{1},2}, \ldots, s_{\tilde{1},n}]^T$ and $[s_{\tilde{2},1}, s_{\tilde{2},2}, \ldots, s_{\tilde{2},n}]^T$ with $s_{\tilde{1},i} \in \mathbb{F}_{2^m}$ and $s_{\tilde{2},j} \in \mathbb{F}_{2^m}$. In other terms, the row space of the associated matrix formulates the vector space of S_1 and S_2, respectively. Zassenhaus algorithm

---

Algorithm 4: Constant-Time Rank Support Recover (RSR) Algorithm.

```
Input: s = $(s_1, \ldots, s_n) \in \mathbb{F}_{q^m}^n$ a syndrome of an error
        e of weight $r$ and of support $E$
Output: A candidate for the vector space $E$
// Compute the vector space $EF$
1 Compute the syndrome vector space $S = \langle s_1, \ldots, s_n \rangle$;
// Recover the vector space $E$ from
  $S_i$'s
2 Compute every $S_i = f_i^{-1}S$ for $i = 1$ to $d$;
3 $E \leftarrow \cap_{i=1}^{d} S_i$;
4 return E;
```

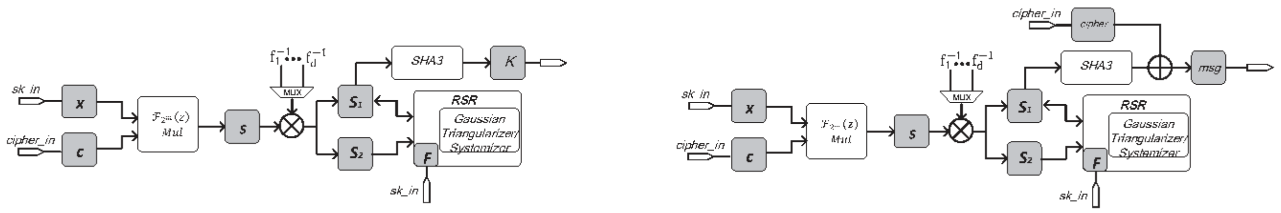
constructs a $2n$-by-$2m$ matrix using S_1 and S_2 and then reduces it into row-echelon form by use of Gaussian Elimination

$$\begin{array}{c|c} S_1 & S_1 \\ \hline S_2 & 0 \end{array} \longrightarrow \begin{pmatrix} \begin{array}{c|c} \hat{S_1} & \hat{S_2} \\ \hline 0 & E \\ \hline 0 & 0 \end{array} \end{pmatrix},$$

where E is the exact intersection of S_1 and S_2.

After the triangularization above is completed, we can extract the targeted intersection of two vector spaces as E. Then E is intersected with S_3 to update a smaller E, and so forth. Note that this intersected error vector space E has smaller and smaller dimensions and eventually reduces to $r = 7, 8, 9$ after all rounds of vector space intersection are completed, and we must systemize this small vector space to formulate the unique row-reduced-echelon form for hashing to produce the symmetric secret key (See the last step in the decapsulation algorithms). In this case, the $x \times 2m (r \leq x < m)$ systolic array proposed in Section III-D is reused for both triangularization and systemization tasks.

Multiple Consecutive $d - 1$ Intersections: It is worth noting that the RSR algorithm requires $d - 1$ times of single vec-tor space intersection to retrieve the error E. Indeed, we can significantly reduce the time complexity of these consecutive intersections by dynamically adjusting the row number of the input matrix: Each round of RSR algorithm reduces the dimension of E, and therefore, the row number of large matrix $\begin{array}{c|c} E & E \\ \hline S_i & 0 \end{array}$ gradually decreases as the round number increases.


sk_in
cipher_in
x
c
$\mathbb{F}_{2^m}(z)$ Mul
s
$f_1^{-1} \cdots f_d^{-1}$
MUX
S_1
S_2
SHA3
K
RSR
Gaussian Triangularizer/ Systemizer
F
sk_in
cipher_in
cipher
SHA3
msg

(a) Hardware architecture for ROLLO-I key decapsulation. (b) Hardware architecture for ROLLO-II data decryption.

Fig. 11. ROLLO decryption hardware. (a) Hardware architecture for ROLLO-I key decapsulation. (b) Hardware architecture for ROLLO-II data decryption.

This observation leads to faster matrix triangularization used in RSR. However, this fast implementation is not constant-time and may raise security concerns on timing attacks.

A better strategy is to further reduce the matrix size while keeping the constant-time computation. It is achievable if one notices that the dimension of E cannot exceed $r \cdot d$ as RSR iterates. For all ROLLO instances, $r \cdot d$ is always smaller than n; therefore, triangularizing a $(n + rd) \times 2m$ matrix is faster than triangularizing a $2n \times 2m$ matrix. In short, this new strategy triangularizes a $2n \times 2m$ matrix in the first iteration to intersect S_1 and S_2 for E. Starting from the second intersection, it triangularizes a smaller $(n + rd) \times 2m$ matrix to get $E \cap S_i$ where one operand E has rd rows while the other operand $S_i, i = 3, 4, \ldots$ has n rows. This asymmetry between operands can be manipulated to accelerate the triangularization. Such asymmetric computation repeats $d - 2$ times before the final E with dimension r is retrieved.

3) ROLLO Decryption Hardware: In this subsection, we discuss the implementation details on ROLLO-I.Decap and ROLLO-II.Decrypt.

Quotient Ring Multiplication: The multiplier is reused here in decryption hardware. The cycle count for the multiplication is estimated as $(Delay_{mul} + 1)d^n e\frac{2}{d}$.

Vector Space Reformulation (S_i Gen): When the syndrome space S is calculated from the quotient ring multiplier, the next step is to formulate the vector spaces S_1 and S_2, and then to construct a $2n \times 2m$ binary matrix as $\begin{array}{c|c} S_1 & S_1 \\ \hline S_2 & 0 \end{array}$ for memory storage. Note that the input S is arranged in a $dn/de \times md$ binary matrix and we need extra control logic to process m-bits by m-bits of S: One $\mathbb{F}_{2^m}$ multiplier is utilized to perform $S_i = f_i^{-1}S$ for every entry of S_i iteratively. In summary, let $Delay_{mul}$ and $Delay_{rd}$ denote the delay of $\mathbb{F}_{2^m}$ multiplier and the delay of memory-read, respectively, the cycle count for vector space reformulation is $d(Delay_{mul} + 2)n + d(Delay_{rd} + 2)$.

Rank Support Recovery (RSR): When S_1 and S_2 are ready in memory, RSR will perform Zassenhaus algorithm on $\begin{array}{c|c} S_1 & S_1 \\ \hline S_2 & 0 \end{array}$ to retrieve the intersection as the error vector space $E = S_1 \cap S_2$. Note that the dimension of E is upper bounded by $r \cdot d$ and therefore E is always written back to the first $r \cdot d$ rows of memory such that the first $dim(E)$ rows store E and the remaining $rd - dim(E)$ rows store null vector. Next, $S_i (i = 3, \ldots, d)$ is written to the following n rows of memory to formulate the large matrix as $\begin{array}{c|c} E & E \\ \hline S_i & 0 \end{array}$ for Zassenhaus algorithm to update a new and further reduced E. The Zaussenhaus algorithm is repeatedly performed $d - 1$ times to extract the correct E after which a final matrix systemization of E is required for hashing.

TABLE VI
Detailed Cycle Count Analysis on ROLLO Encryption/ Encapsulation for the Claimed 128-bit/192-bit/256-bit Security Level (SL), Respectively

Process		ROLLO-I.encap	ROLLO-II.encrypt
generate K	generate $\mathbf{E}$	175/200/225	175/200/200
	GE over $\mathbb{F}_2$	159/186/225	191/222/222
	SHA3	191/218/245	191/218/218
generate $\mathbf{c}$	generate $\mathbf{c}_1, \mathbf{e}_2$	50/50/50	50/100/100
	$\mathbb{F}_{2^m}[z]$ mul	2452/2984/5018	12544/14122/17806
	$\mathbb{F}_{2^m}[z]$ add	36/39/39	81/75/81
Total	—	2713/3273/5332	12850/14497/18187

TABLE VII
Detailed Cycle Count Analysis on ROLLO Decryption/Decapsulation for the Claimed 128-bit/192-bit/256-bit Security Level (SL), Respectively

Process		ROLLO-I.decap	ROLLO-II.decrypt
generate K	compute $\mathbf{s}$	2452/2984/5018	12544/14122/17806
	generate $\mathbf{S_i}$	6024/7032/11241	15168/17032/20943
	$\mathbf{E} \leftarrow RSR(\cdot)$	16231/22881/36174	41736/48717/59931
	SHA3	189/216/243	189/216/216
Total	—	24,896/33,113/52,676	69,637/80,087/98,896

Hardware Architecture: Figs. 11(a) and 11(b) depict the hardware architecture for the ROLLO decryption/decapsulation. The critical components include the $\mathbb{F}_{2^m}[z]$ multiplier and the Gaussian Elimination systolic array which occupy the majority of the hardware utilization. ROLLO-I and ROLLO-II share almost an identical architecture though ROLLO-II decryption is relatively larger due to the larger system parameter n. The only difference at the top level is that ROLLO-I outputs the hash value K directly whereas ROLLO-I decrypts the cryptogram by XORing K.

V. Concrete Results on ROLLO Hardware

Table VI lists the cycle count for ROLLO encryption/ encapsulation on Xilinx Artix-7 FPGAs. To compare among ROLLO encryption hardware, ROLLO-I is the fastest and the smallest: All three variants use about 3k5k cycles to do data encapsulation. ROLLO-II is much slower, using around

TABLE VIII
PERFORMANCE OF ROLLO HARDWARE AND COMPARISON WITH EXISTING WORK ON PQC HARDWARE, TARGETING NIST SECURITY LEVELS 1/3/5

Encryption part							
Scheme	SL [bits]	Platform	$f_{\max}$ [MHz]	Time/Op [μs]	Cycles [$\times 10^3$]	Slices/LUTs/FFs	BRAM
LRPC code:							
ROLLO-I-128	128	Artix-7	180	16	2.8	7524/24154/11735	24.5
ROLLO-II-128			170	76	12.9	9532/31360/16029	31.5
ROLLO-I-192	192	Artix-7	170	19	3.3	8140/25831/13621	27.5
ROLLO-II-192			170	85	14.5	9139/29695/16957	31.5
ROLLO-I-256	256	Artix-7	170	32	5.4	9855/32002/17381	34.5
ROLLO-II-256			170	107	18.2	9506/31164/17024	31.5
MDPC/LDPC code:							
pre-BIKE [22]	80	Virtex-6	351.3	14	4.8	2920/8856/14426	0
BIKE [26]	128	Artix-7	121.95	100	12.0	4540/14829/3471	10
BIKE [43][a]	128	Artix-7	113	132	15.0	7332/25549/5462	34
LEDAkem [28]	128	Artix-7	235	3000	712.0	33/104/53	1
Goppa code:							
Niederreiter [20]	256	Stratix-V	448	12	5.4	—/—/—	0
Classic McEliece [5]	128	Artix-7	105.6	26	2.7	—/81339/132190	236
Classic McEliece [5]	192	Virtex-7	130.8	26	3.4	—/109484/168939	446
Classic McEliece [5]	256	Virtex-7	136.6	37	5.0	—/122624/186194	589
Decryption part							
Scheme	SL [bits]	Platform	$f_{\max}$[MHz]	Time/Op[μs]	Cycles [$\times 10^3$]	Slices/LUTs/FFs	BRAM
LRPC code:							
ROLLO-I-128	128	Artix-7	180	138	24.9	11412/36772/21832	23.5
ROLLO-II-128			175	398	69.6	14160/45207/26598	29
ROLLO-I-192	192	Artix-7	180	184	33.1	12724/40465/25079	26.5
ROLLO-II-192			175	458	80.1	15357/49855/30831	33
ROLLO-I-256	256	Artix-7	180	293	52.7	15130/49419/30623	34.5
ROLLO-II-256			175	565	98.9	15620/51852/30641	34.5
MDPC/LDPC code:							
pre-BIKE [22]	80	Virtex-6	222.5	125	27.9	2920/8856/14426	0
BIKE [26]	128	Artix-7	100	1900	190.0	8862/30977/5092	29
BIKE [43][a]	128	Artix-7	113	1892	215.0	7332/25549/5462	34
LEDAkem [28]	128	Artix-7	140	18700	2620.0	870/2222/658	13
Goppa code:							
Niederreiter [20]	128	Virtex-6	267	40	10.2	6571/—/—	23
Classic McEliece [5]	128	Artix-7	105.6	95	10.0	—/81339/132190	236
Classic McEliece [5]	192	Virtex-7	130.8	112	14.6	—/109484/168939	446
Classic McEliece [5]	256	Virtex-7	136.6	181	24.7	—/122624/186194	589

[a]*This work implements the unified BIKE core including key generation, data encapsulation and data decapsulation and thus the SLICEs/LUTs/ FFs and BRAMs report the resource utilization of the entire BIKE core.*

13k□18k cycles, but the advantage for ROLLO-II is that it can do long-term data encryption.

The cycle count information for the proposed ROLLO decryption hardware on Artix-7 is collected in Table VII. The primary factor accountable for the cycle delay is the RSR algorithm, which relies on the Gaussian elimination systolic array. The secondary factor is the $\mathbb{F}_{2^m}[z]$ multiplication. Note that the performance of ROLLO-II decryption is about 2-3 times worse than that of ROLLO-I. This observation can be interpreted as follows: The primary and the second factors are more or less a quadratic function of n and m as $n^2 + m \cdot n$. The value of n used in ROLLO-II is about twice as large as ROLLO-I, giving rise to the overall 2-3 times performance degradation.

In terms of encryption performance, compared with other PQC hardware in the literature shown in Table VIII, our ROLLO-I encryption hardware is faster than the Classic McEliece encryption hardware [5]. Our ROLLO-II encryption hardware is about three times slower than [5] but uses much fewer slices and BRAMs. The fastest implementation on the Niederreiter cryptosystem is reported in [20], and our work is comparable with theirs regarding the cycle count. Moreover, our ROLLO-I hardware is also faster than the latest BIKE hardware implementations [26], [43]. It even outperforms an obsolete BIKE hardware with an 80-bit parameter set presented in [22], which also targets high-speed implementation. Our work is significantly faster but also uses more hardware resources than the area-efficient LEDAkem hardware shown in [28].

In terms of decryption performance, compared with the optimized software implementation using AVX2 instructions included in the ROLLO NIST submission, the cycle count is reduced by at least 20 and 16 times for ROLLO-I, and ROLLO-II, respectively. Compared with other PQC hardware implementations on FPGA listed in Table VIII, the decryption throughput of ROLLO is advantageous: It is apparently faster than the quasi-cyclic MDPC/LDPC code-based schemes, including BIKE [26], [43] and LEDAkem [28], and even approaches the state-of-art implementation of the Niederreiter cryptosystem featuring high-speed [20]. BIKE and LEDAkem appear to outperform ROLLO and Classic McEliece regarding slices and BRAM usage. On the other hand, ROLLO uses fewer slices and BRAMs but runs slower than the Classic McEliece [5].

VI. CONCLUSION

This paper presented a complete FPGA implementation of the ROLLO scheme using LRPC codes. It is the first hardware implementation of a rank-metric code-based cryptosystem that supports varying security parameters. The efficiency of our design is achieved by a novel Gaussian elimination structure, a simplified implementation strategy for the rank support recovery algorithm, and a fast interleaved polynomial multiplier, among others. The proposed parameterized architectures, such as the Gaussian elimination and the polynomial multiplication, are not limited to instances used in ROLLO but also fully support other rank-code-based schemes. For example, RQC applies the identical family of ideal codes to construct the public key and the ciphertext. Therefore, the $\mathsf{F}_{2^m}[z]$ arithmetic presented in this work can be directly reused as the underlying arithmetic. At the core of RQC key generation and data encryption algorithm, the generation of vectors over F_2 with prescribed rank weight is demanding, and the parameterized Gaussian elimination systolic array can be adapted effortlessly for this task.

REFERENCES

[1] L. Chen et al., "Report on post-quantum cryptography," *Nat. Inst. Standards Technol. Inter. Rep.*, vol. 8105, pp. 1–15, 2016.
[2] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-Quantum Cryptography*, Berlin, Germany: Springer, 2009, pp. 1–14.
[3] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *DSN Prog. Rep.*, vol. 42, no. 44, pp. 114–116, 1978.
[4] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems Control Inf. Theory-Problemy Upravleniya I Thorii Informatsii*, vol. 15, no. 2, pp. 159–166, 1986.
[5] T. Chou et al., "Classic McEliece: Conservative code-based cryptography," NIST submissions, 2017. [Online]. Available: https://classic.mceliece.org/
[6] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. Barreto, "MDPC-McEliece: New McEliece variants from moderate density parity-check codes," in *Proc. IEEE Int. Symp. Inf. Theory*, 2013, pp. 2069–2073.
[7] N. Aragon et al., *BIKE: Bit Flipping Key Encapsulation*, 2020. [Online]. Available: https://bikesuite.org/files/v4.1/BIKE_Spec.2020.10.22.1.pdf
[8] J.-C. D. Philippe Gaborit, *Hamming Quasi-Cyclic (HQC) April 2020*, 2020. [Online]. Available: https://pqc-hqc.org/doc/hqc-specification_2020--10-01.pdf
[9] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, *Low-dEnsity parity-check coDe-bAsed cryptographic systems (LEDAcrypt) April 2020*, 2020. [Online]. Available: https://www.ledacrypt.org/documents/LEDAcrypt_v3.pdf
[10] Q. Guo, T. Johansson, and P. Stankovski, "A key recovery attack on MDPC with CCA security using decoding errors," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2016, pp. 789–815.
[11] E. M. Gabidulin, A. Paramonov, and O. Tretjakov, "Ideals over a non-commutative ring and their application in cryptology," in *Proc. Workshop Theory Appl. Cryptographic Techn.*, 1991, pp. 482–489.
[12] R. Overbeck, "A new structural attack for GPT and variants," in *Proc. Int. Conf. Cryptol. Malaysia*, 2005, pp. 50–63.
[13] J.-C. D. Philippe Gaborit, *Rank Quasi-Cyclic (RQC) April 2020*, 2020. [Online]. Available: https://https://pqc-rqc.org/doc/rqc-specification_2020--04-21.pdf
[14] J.-C. D. E. A. Philippe Gaborit, *ROLLO - Rank-Ouroboros, LAKE, LOCKER, updated on April 21st, 2020*, 2020. [Online]. Available: https://pqc-rollo.org/doc/rollo-specification_2020--04-21.pdf
[15] M. Bardet et al., "An algebraic attack on rank metric code-based cryptosystems," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2020, pp. 64–93.
[16] M. Bardet et al., "Algebraic attacks for solving the rank decoding and minrank problems without gröbner basis," 2020, *arXiv: 2002.08322*.
[17] G. Alagic et al., "Status report on the second round of the NIST post-quantum cryptography standardization process," US Dept. of Commerce, NIST, vol. 2, pp. 16–17, 2020.
[18] S. Heyse and T. Güneysu, "Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2012, pp. 340–355.
[19] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes," in *Proc. Int. Conf. Cryptographic Hardware Embedded Syst.*, 2017, pp. 253–274.
[20] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based Niederreiter cryptosystem using binary Goppa codes," in *Proc. Int. Conf. Post-Quantum Cryptography*, 2018, pp. 77–98.
[21] P.-J. Chen et al., "Complete and improved FPGA implementation of classic McEliece," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2022, no. 3, pp. 71–113, Jun. 2022.
[22] S. Heyse, I. Von Maurich, and T. Güneysu, "Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices," in *Proc. Int. Conf. Cryptographic Hardware Embedded Syst.*, 2013, pp. 273–292.
[23] I. von Maurich and T. Güneysu, "Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices," in *Proc. Conf. Des. Autom. Test Europe*, 2014, Art. no. 38.
[24] J. Hu and R. C. Cheung, "Area-time efficient computation of Niederreiter encryption on QC-MDPC codes for embedded hardware," *IEEE Trans. Comput.*, vol. 66, no. 8, pp. 1313–1325, Aug. 2017.
[25] J. Hu, W. Wang, R. C. Cheung, and H. Wang, "Optimized polynomial multiplier over commutative rings on FPGAs: A case study on BIKE," in *Proc. Int. Conf. Field-Programmable Technol.*, 2019, pp. 231–234.
[26] J. Richter-Brockmann and T. Güneysu, "Folding BIKE: Scalable hardware implementation for reconfigurable devices," Tech. Rep., Cryptology ePrint Archive 2020/897, 2020.
[27] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAkem: A post-quantum key encapsulation mechanism based on QC-LDPC codes," in *Proc. Int. Conf. Post-Quantum Cryptography*, 2018, pp. 3–24.
[28] J. Hu, M. Baldi, P. Santini, N. Zeng, S. Ling, and H. Wang, "Lightweight key encapsulation using LDPC codes on FPGAs," *IEEE Trans. Comput.*, vol. 69, no. 3, pp. 327–341, Mar. 2020.
[29] J. Lablanche, L. Mortajine, O. Benchaalal, P.-L. Cayrel, and N. El Mrabet, "Optimized implementation of the NIST PQC submission ROLLO on microcontroller," *IACR Cryptol. ePrint Arch.*, vol. 2019, 2019, Art. no. 787.
[30] C. Aguilar-Melchor, N. Aragon, E. Bellini, F. Caullery, R. H. Makarim, and C. Marcolla, "Constant time algorithms for ROLLO-I-128," *SN Comput. Sci.*, vol. 2, no. 5, pp. 1–19, 2021.
[31] T. Chou and J.-H. Liou, "A constant-time AVX2 implementation of a variant of ROLLO," *IACR Trans. Cryptographic Hardware Embedded Syst.*, 2022, pp. 152–174.
[32] P. Gaborit, G. Murat, O. Ruatta, and G. Zémor, "Low rank parity check codes and their application to cryptography," in *Proc. Workshop Coding Cryptography*, 2013, pp. 1–13.
[33] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in *Proc. Theory Cryptography Conf.*, 2017, pp. 341–371.
[34] W. G. Horner, "XXI. A new method of solving numerical equations of all orders, by continuous approximation," *Philos. Trans. Roy. Soc. London*, vol. 109, pp. 308–335, 1819.
[35] H. Hsing, *SHA3 (KECCAK) OpenCores Project*, 2013. [Online]. Available: https://opencores.org/projects/sha3
[36] B. Hochet, P. Quinton, and Y. Robert, "Systolic Gaussian elimination over GF (p) with partial pivoting," *IEEE Trans. Comput.*, vol. 38, no. 9, pp. 1321–1324, Sep. 1989.
[37] C.-L. Wang and J.-L. Lin, "A systolic architecture for computing inverses and divisions in finite fields $GF(2^m)$," *IEEE Trans. Comput.*, vol. 42, no. 9, pp. 1141–1146, Sep. 1993.
[38] A. Rupp, T. Eisenbarth, A. Bogdanov, and O. Grieb, "Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalytic applications," *Integration*, vol. 44, no. 4, pp. 290–304, 2011.
[39] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, "A novel cryptoprocessor architecture for the McEliece public-key cryptosystem," *IEEE Trans. Comput.*, vol. 59, no. 11, pp. 1533–1546, Nov. 2010.
[40] W. Wang, J. Szefer, and R. Niederhagen, "Solving large systems of linear equations over GF(2) on FPGAs," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2016, pp. 1–7.
[41] N. Aragon and P. Gaborit, "A key recovery attack against LRPC using decryption failures," in *Proc. Int. Workshop Coding Cryptography*, 2019, pp. 1–10.
[42] S. Samardjiska, P. Santini, E. Persichetti, and G. Banegas, "A reaction attack against cryptosystems based on LRPC codes," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, 2019, pp. 197–216.
[43] J. Richter-Brockmann, M.-S. Chen, S. Ghosh, and T. Güneysu, "Racing BIKE: Improved polynomial multiplication and inversion in hardware," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2022, pp. 557–588, Nov. 2021.

Jingwei Hu received the BSc degree from Dalian Maritime University, and the MEng degree from Tianjin University, in 2014, and the PhD degree from the City University of Hong Kong. He is currently a postdoctoral research fellow with Nanyang Technological University. His research interest includes secure multi-party computations, fully homomorphic encryptions and embedded cryptographic system.

Wen Wang received the BS degree from the University of Science and Technology of China, and the MS and PhD degrees from Yale University, in 2015, 2017, and 2021, respectively. Currently, she works as a research scientist in Intel Labs. Her research interests include applied cryptography and privacy-enhancing technologies.

Kris Gaj is a professor with the ECE Department, George Mason University. He is a co-director of the Cryptographic Engineering Research Group (CERG). He has been involved in most previous and current cryptographic competitions, from AES to PQC.

Liping Wang received the BS and MS degrees in mathematics from the University of Information Engineering, Zhengzhou, China, in 1992 and 1995, respectively, and the PhD degree from the University of Science and Technology of China, Hefei, China, in 2003. She is a professor with the State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China. Her current research interests include post-quantum public-key cryptography and privacy computing.

Huaxiong Wang received the PhD degree in mathematics from the University of Haifa, Israel, in 1996, and the PhD degree in computer science from the University of Wollongong, Australia, in 2001. He is a professor with Nanyang Technological University, Singapore, where he also served as the head of division of mathematical sciences from 2013 to 2015. Prior to joining NTU, in 2006, he held faculty positions with Macquarie University and University of Wollongong, Australia, and visiting positions with ENS de Lyon in France, City University of Hong Kong, National University of Singapore and Kobe University in Japan. He has more than 20 years of experience in research on cryptography and information security. He received the inaugural Award of Best Research Contribution awarded by the Computer Science Association of Australasia, in 2004. He was the invited speaker of ASIACRYPT 2017. He served as the program co-chair of Asiacrypt 2020 and 2021.