Journal of Cryptographic Engineering (2024) 14:383–397
https://doi.org/10.1007/s13389-024-00355-3

RESEARCH ARTICLE

Universal Gaussian elimination hardware for cryptographic purposes

Jingwei Hu[1] · Wen Wang[2] · Kris Gaj[3] · Donglong Chen[4] · Huaxiong Wang[1]

Received: 8 November 2023 / Accepted: 22 April 2024 / Published online: 22 May 2024

Abstract
In this paper, we investigate the possibility of performing Gaussian elimination for arbitrary binary matrices on hardware. In particular, we presented a generic approach for hardware-based Gaussian elimination, which is able to 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. We accordingly implemented an optimized and parameterized Gaussian eliminator for (singular) matrices over binary fields, making the intense computation of linear algebra feasible and efficient on hardware. To the best of our knowledge, this work solves for the first time eliminating a singular matrix on reconfigurable hardware and also describes the a generic hardware architecture for rank-code based cryptographic schemes. The experimental results suggest hardware-based Gaussian elimination can be done in linear time regardless of the matrix type.

Keywords Post-quantum cryptography · Gauassian elimination · FPGA

1 Introduction

From computational efficiency point of view, Gaussian elimination on an $n \times n$ matrix requires $\mathcal{O}(n^3)$ divisions, $\mathcal{O}(n^3)$ multiplications, $\mathcal{O}(n^3)$ additions, and $\mathcal{O}(n^3)$ subtractions, for a total of $\mathcal{O}(n^3)$ arithmetic operations. There are numerous applications of Gaussian elimination in nearly any area of computer science. Cryptology is no exception, with matrix problems arising both in cryptanalysis and cryptography. In the introductory part, we briefly outline the areas where our hardware-based Gaussian elimination is of the most relevance.

Algebraic cryptanalysis of symmetric-key ciphers. Cryptanalysis of symmetric-key ciphers frequently involves systems of linear equations (SLEs), which can be efficiently solved using Gaussian elimination. This is because the majority of deterministic symmetric ciphers can be represented as finite state machines whose output can be described by a (sometimes rather complicated) boolean function of the initial internal state and input values (if any) giving rise to SLEs over $\mathbb{F}_2$. For instance, linearization methods [1, 2] have gained lots of attention during the last decade and are widely used nowadays. Here, the nonlinear system is first simplified, then linearized and solved as an SLE. To make cryptanalysis with linearization methods feasible, one is reliant upon efficient SLE solvers.

Implementing asymmetric cryptography. Gaussian elimination also plays a central role in some cryptographic applications. For example, the performance of digital signature schemes based on multivariate quadratic polynomials highly depends on the efficiency of solving small SLEs over finite extension fields. This class of digital signature schemes is of special interest due to its resistance to quantum computer attacks. For the generation of a signature using the Rainbow signature scheme [3] with recently recommended parameter

✉ Donglong Chen
donglongchen@uic.edu.cn

Jingwei Hu
davidhu@ntu.edu.sg

Wen Wang
wen.wang.ww349@yale.edu

Kris Gaj
kgaj@gmu.edu

Huaxiong Wang
hxwang@ntu.edu.sg

1 Nanyang Technological University, Singapore, Singapore

2 Yale University, New Haven, USA

3 George Mason University, Fairfax, USA

4 BNU-HKBU United International College, Zhuhai, China

sets, two SLEs of dimension 12×12 over $\mathbb{F}_{2^8}$ need to be solved. In [4], a generic hardware architecture for this kind of signature schemes is proposed, with an SLE solver being a major building block. Furthermore, Gaussian elimination is the most compute-intensive and also a distinguishing operation in rank metric cryptography and particularly in the context of ROLLO [5]. For example, ROLLO-II PKE.encrypt generates a random 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 also requires Gaussian elimination to intersect the secret syndrome spaces $S_i = f_i^{-1} S$ for finding the linearly independent bases of the secret error vector space E.

Contributions. Based on the parallel nature of hardware, we propose a new approach that could Gaussian-eliminate arbitrary matrices over a binary field in constant $\Theta(n)$ steps, which remains unsolved prior to this work. Our work differs from the previous work in two aspects:

- The new design presents a new mechanism called dual-mode switch to determine the pivot position on the fly and thus can Gaussian-eliminate a singular matrix, whereas the previous designs assume the input matrix is non-singular and cannot return a correct answer for a singular matrix.
- The new design is constant-time, fully parameterized and open-sourced .[1] The HDL codes for our hardware design are auto-generated by a Python script and thus can be easily adapted for different matrix sizes used in numerous cryptographic applications.

This paper is roughly structured as follows. We start with a brief discussion of previous work on hardware-based Gaussian elimination. Then, we provide a high-level review of the ROLLO specification, which is later used as a case study for our Gaussian elimination design. We then present our new algorithm, which guides the hardware implementation of a Gaussian elimination design. The novel algorithm and the new hardware design can Gaussian-eliminate both singular and non-singular matrices. After that, further applications of the hardware architecture are discussed, including how to Gaussian-eliminate a medium-sized binary matrix using the proposed design and reuse this new module for any large-sized matrices. We also showcase how to adapt our new Gaussian eliminatation module to ROLLO hardware design. Finally, we show our proof-of-concept implementations on contemporary low-cost FPGAs.

[1] The automation tools and reference implementations can be found at https://github.com/davidhoo1988/gaussian-elimination-hardware.

2 Related 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, systolic network, and systolic line.

2.1 Triangular-shaped array

A 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 of a matrix over $\mathbb{F}_q$ [6]. This work was further adapted for faster processing using triangular systolic network (TSN) [7] and triangular systolic line (TSL) [8]. In general, a 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.

2.1.1 Linear-shaped array

A linear-shaped array is a one-dimensional array, where all nodes in the array form a horizontal line as described in [8]. 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. A 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.

In addition to the systolic architectures, a different hardware approach realizing Gaussian elimination including backward-substitution over $\mathbb{F}_{2^k}$, called GSMITH, is presented in [8]. It consists of a rectangular array of simple cells exhibiting local as well as some global connections. The running time of GSMITH is non-constant-time, depending on the probability distribution of the matrix entries. The implementation results suggest the timing performance is close to that of TSL but uses more hardware resources.

For cryptographic purposes, Gaussian elimination hardware for $n \times n$ square matrices over $\mathbb{F}_{2^k}$, also known as linear system of equations (LSE) solver [4, 9, 10], is explored in multivariate cryptosystems. If the LSE solver encounters under-determined equations, i.e., the matrix associated with the linear system of equations is not full-rank, the solver throws out an exception and halts. Note that the LSE solver can identify the under-determined equations without fully

performing the Gaussian elimination. It suffices to make such a decision whenever one of the pivots along the diagonal of the matrix is zero. In other words, the LSE solver in the open literature cannot eliminate singular matrices.

On the other hand, the only two hardware implementations of Gaussian elimination used in Hamming-metric-code-based cryptography that are closely related to our work are presented in [11, 12]. They are used in the key generation of the classic Niederretier code-based scheme. These designs are capable of eliminating a binary matrix of the size $k \times l$, with $k \neq l$, which removes the shape limit existing in the LSE solver mentioned above. The pre-requisite for successful Gaussian elimination is that the input matrix must be full-rank. For rank-metric-code-based cryptography like ROLLO and RQC, Gaussian elimination is the most computing-intensive and also a distinguishing operation. However, these rank-code-based schemes require performing Gaussian elimination on medium-size and large-size matrices over a binary field, and most importantly, these matrices can be rank-deficient. Unfortunately, the current state-of-the-art designs cannot process such type of matrices.

3 Preliminaries of ROLLO

This section introduces the rank-metric code based cryptographic scheme—ROLLO [5] which heavily relies on a universal Gaussian elimination utility. The requirement for universal Gaussian elimination utility posts a new challenge for cryptographic hardware designers. 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. It is worth mentioning that 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. Effective manipulation for such a matrix goes beyond the applicability of the existing Gaussian elimination hardware.

Let $\mathcal{S}_w^n(\mathbb{F}_{q^m})$ stand for the set of vectors of length n and rank weight w over $\mathbb{F}_{q^m}$ and $\mathcal{S}_{1,w}^n(\mathbb{F}_{q^m})$ stand for the set of vectors of length n and rank weight w, such that *its support contains* 1:

$$\mathcal{S}_w^n(\mathbb{F}_{q^m}) = \{\mathbf{x} \in \mathbb{F}_{q^m}^n : \dim \mathrm{Supp}(\mathbf{x}) = w\}$$
$$\mathcal{S}_{1,w}^n(\mathbb{F}_{q^m}) = \{\mathbf{x} \in \mathbb{F}_{q^m}^n : \dim \mathrm{Supp}(\mathbf{x}) = w, 1 \in \mathrm{Supp}(\mathbf{x})\}$$

3.1 ROLLO-I

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 $\mathcal{K}$. 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 $K \in \mathcal{K}$. Finally `Decap` using the secret key `sk` and an encapsulation c, recovers the key $K \in \mathcal{K}$ or fails and returns $\perp$.

ROLLO-I is formally described in Algorithm 1. The RSR algorithm is the rank support recover algorithm proposed in [13] 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.

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

Algorithm 1: Formal Description of ROLLO-I

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 $\mathtt{Supp}(\mathbf{e_1}, \mathbf{e_2}) = r$. 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.

3.2 ROLLO-II

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 [14]. 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. 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 are inapplicable.

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

Algorithm 2: Formal Description of ROLLO-II

4 A new approach for hardware-based Gaussian elimination

This section describes a new approach for Gaussian elimination on a systolic array. Based on this method, we design a constant-time and flexible Gaussian elimination module to overcome the difficulty of implementing Guassian elimination for arbitrary matrices which are useful for many cryptographic schemes, e.g., ROLLO, in which intensive linear-algebra-related computations are required.

4.1 Gaussian elimination on a systolic array

In this work, we use the terms 'triangularization' to denote the operation of putting the input matrix into its row-echelon form, *aka* Gaussian elimination, and 'systemization' to denote the operation of putting a row-echelon formed matrix into its reduced-row-echelon form. The combination of triangularization and systemization is also referred to as Gauss-Jordan elimination. We are facing a new challenge in rank-code based cryptosystem, namely, triangularizing a singular matrix in ROLLO/RQC. 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.

4.1.1 Core idea—pivot/non-pivot mode switch

In our work, we focus on Gaussian elimination for matrices over $\mathbb{F}_2$. As mentioned before, the most challenging part for universal Gaussian elimination hardware is that the position of pivot nodes in our Gaussian elimination architecture is flexible. Our new idea of implementing Gaussian elimination is to assign pivot-node functionality or non-pivot-node functionality on the fly: Each node is configured to have dual functionalities for every iteration of Gaussian elimination. The node can be converted to either pivot node or basic(non-pivot) 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 operates passively based on the operation signal received, namely `PASS`, `ADD`, or `SWAP`, to execute elementary row operations. Gaussian elimination involves a sequence of elementary row operations, which can be broken down into row switching, row multiplication, and row addition. Therefore, organizing and executing row switchings, row multiplications, and row additions properly is sufficient for computing Gaussian elimination. Row switching is executed using the `SWAP` instruction, row addition using the `ADD` instruction, and row multiplication over $\mathbb{F}_2$ is achieved by simply retaining the row, thus requiring the `Pass` instruction.

A systematic exposition of the proposed node is shown in Fig. 1. The node uses an internal register `r` for storage. It also has 9 signals and 6 of them are identical to the ports of classic nodes presented in the literature [11, 12], including 3 input ports `data_in`, `start_in`, `op_in` and 3 output ports `data_out`, `start_out`, `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. Also, an additional input signal `mode_in` is augmented to switch between matrix triangularization process and matrix systemization process. All input signals drive a centralized control logic module `CTRL` which outputs selector signals including `r_sel`, `op_out_sel`, `pivot_out_sel`, and `data_out_sel`. Then, these selector signals accordingly select the output values of `r`, `op_out`, `pivot_out`, and `data_out`.

With this new mechanism of dual-mode switch, the node can dynamically switch between the pivot and non-pivot functionality for each input data update, as shown in Fig. 2, to perform matrix triangularization with the node mode signal `mode_in` set to 1'b0: The entire process can be split into two stages, initial phase and normal phase. In the initial phase, the input data flushes into the node internal register `r` for the first time by asserting the signal *start_in*; In the normal phase, the node acts as either the pivot node or the basic(non-pivot) node to update the value of `r` depending on the 2-bit signal {`r`, `pivot_in`}: if {`r`, `pivot_in`}$== 2'b10$, it means that the internal register *r* is for the first time updated to a nonzero value and thus the node acts as pivot. Otherwise, it means that the pivot has been found already (since `pivot_in`$== 1'b1$) and thus it acts as basic node by executing passively the

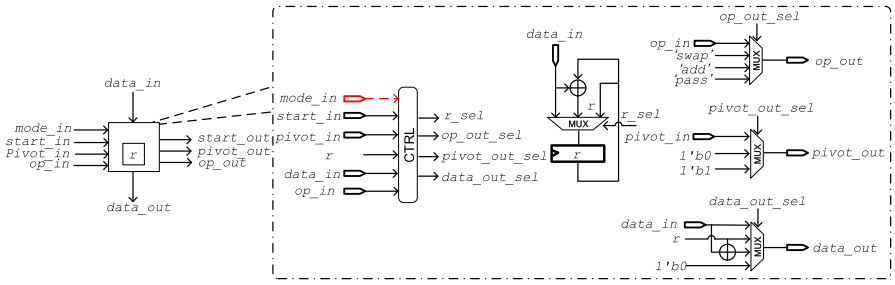


Fig. 1 A versatile node proposed to solve non-fixed-pivot row exceptions in Gaussian elimination. The node can operate three types of transformations including: 1. `PASS`: The node passes the input data `data_in` onto the output port `data_out` and retains the data stored in the internal register `r`; 2. `ADD`: The node adds the input data `data_in` and the internal register data `r`, and then outputs sum onto `data_out`. Meanwhile, the node retains the internal register data `r`; 3. `SWAP`: The node swaps the input data `data_in` and the internal register data `r`, i.e., the node outputs `r` onto `data_out` and then updates `r` with `data_in`

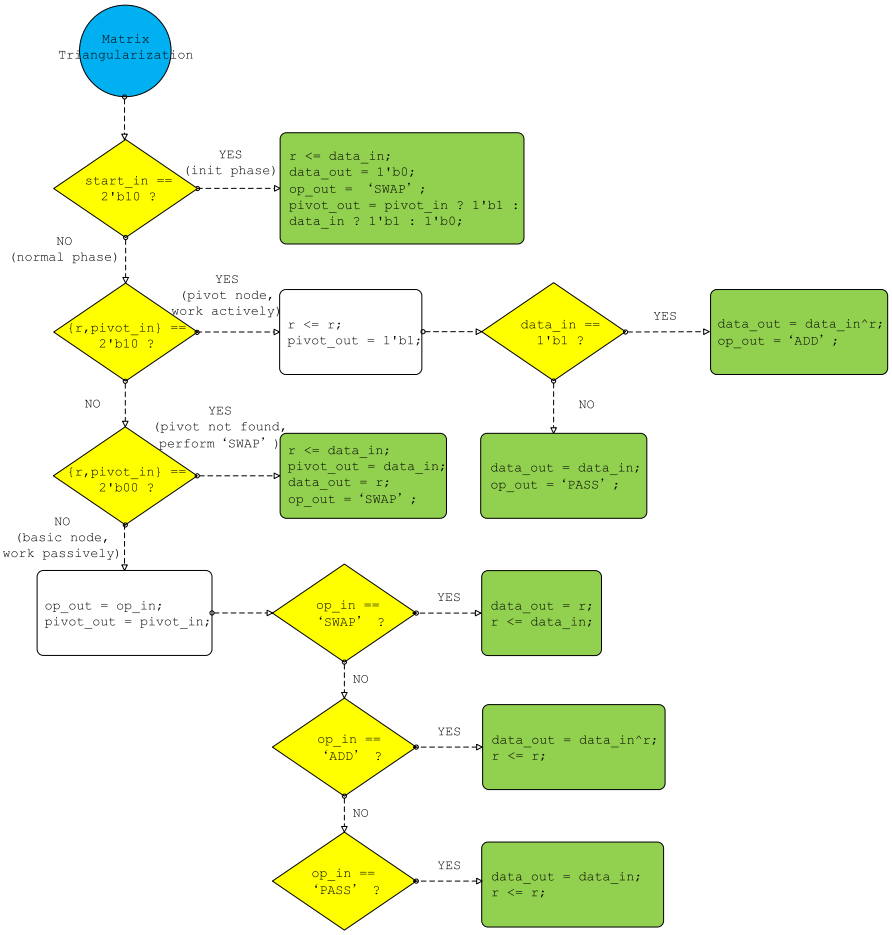


Fig. 2 Behavior description of the node used in the proposed systolic array for small/medium matrix triangularization (by setting the node mode signal `mode_in`=1'b0), written in Verilog-like pseudocode

instruction (`SWAP`, `ADD`, or `PASS`) passed from the signal `op_in`.

4.1.2 Triangular systolic array design

With the new design of node for Gaussian elimination, the next decision we need to make is the selection of systolic array. First, consider the triangular-shaped arrays. Triangular systolic line (TSL) and triangular systolic network (TSN) have lower cycle latency and lower resource usage but they are not suitable for our case: TSN only maintains its efficiency for small matrices as its critical path propagates throughout the whole network. The critical path latency of TSL is a good candidate for the previous Gaussian elimina-

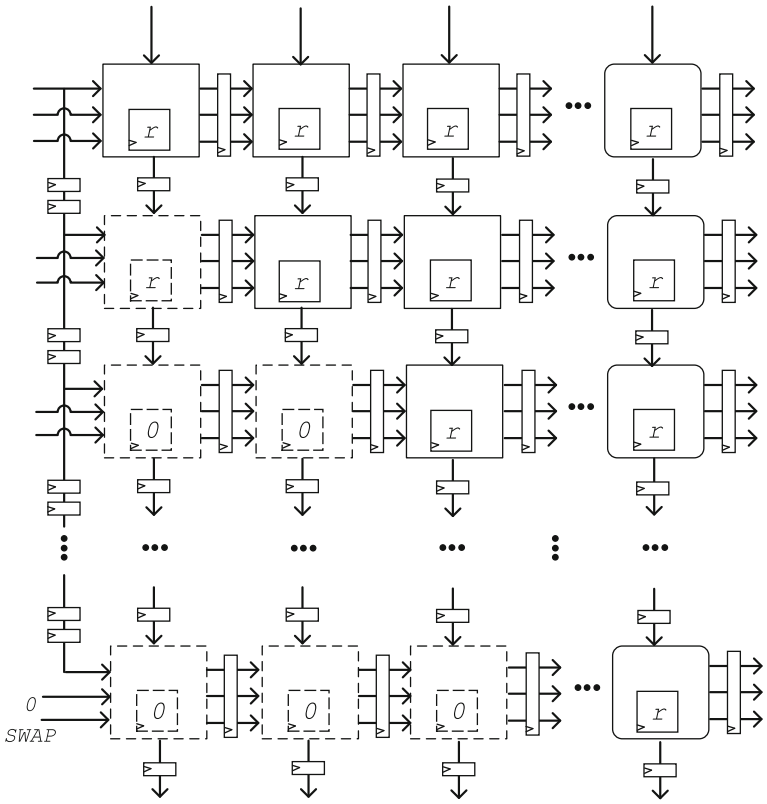


Fig. 3 Overview of the two dimensional array for Gaussian elimination. Each node is pipelined to reduce the critical path delay

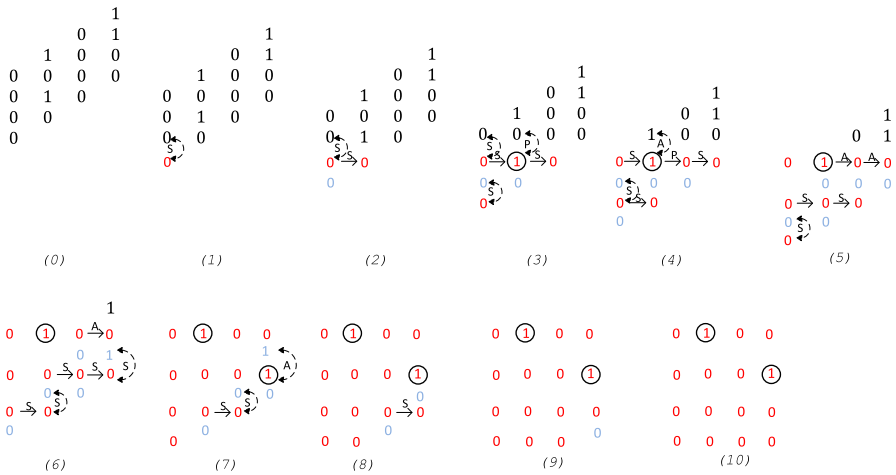


Fig. 4 A toy example for the proposed Gaussian elimination hardware by triangularizing a 4×4 matrix over $\mathbb{F}_2$ using the node logic shown in Fig. 2. Here triangularization means putting the input matrix into its row-echelon form. The signals 'S', 'P', and 'A' stand for 'SWAP', 'PASS' and 'ADD', respectively

tion work but not for ours. Specifically, the horizontal latency (i.e. the total delay along one row of the computation array) of previous work is affected merely by wiring. However, the horizontal latency of the new node in the array is much longer and involves propagating data from the leftmost node to the rightmost one. Therefore, TSL can only be useful for small matrices. On the other hand, the clock frequency of triangular systolic array (TSA) remains as high as 200 MHz despite the matrix dimension increasing from 20 to 90 [8]. Secondly, the linear-shaped arrays, linear systolic array (LSA), and linear systolic line (LSL) are efficient for matrix over $\mathbb{F}_{2^m}$ but not for matrix over $\mathbb{F}_2$ [8]. Based on the above discussion, TSA is chosen as the basic architecture for implementing Gaussian elimination in this work.

Figure 3 presents an 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 the node is pipelined, allowing all data and control signals to be propagated in a systolic manner. It is worthwhile to mention that in order to make this systolic array works correctly for matrix triangularization, in the initial phase as discussed in Fig. 2, the signal `pivot_in` of the first node in every row of the systolic array must be de-asserted (indicating that the pivot is

not yet found). Moreover, an improvement of this architecture is that all r registers in the nodes below the diagonal of the systolic array are always zero independently of the input matrix after the Gaussian elimination. Therefore, these nodes can be removed and are drawn with a dotted line.

4.2 Cryptographic applications for the proposed Gaussian elimination

This subsection describes how to leverage the proposed Gaussian elimination method for accelerating Gaussian elimination for any matrix size. The architectures presented in this subsection are two-fold: The Gaussian elimination module for medium-sized matrices is based on the systolic array design and the new dual-mode switching node for processing a (singular) matrix; the Gaussian elimination module for large-sized matrices 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. This characteristic is important for security concerns since the Gaussian elimination used in cryptographic context may directly operate on the secret sensitive information and any vulnerability exploited from the timing information might endanger the cryptosystem. 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. This is a great advantage for implementers to sketch Gaussian elimination modules for different sizes of matrices used in different cryptographic applications.

4.2.1 Gaussian elimination for medium-sized matrices

We first consider how to Gaussian-eliminate a matrix of relatively small size. For example, in the ROLLO 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 not only matrix triangularization but also matrix systemization is necessary to acquire a unique representation of error vector space $\mathbf{E}$ such that the subsequent hash function always outputs a correct shared key. To better understand the mechanism of the proposed Gaussian elimination architecture, Fig. 2 describes a flow chart of the behavior of the node in triangularization: The symbol '=' indicates blocking assignment, and the symbol '<=' indicates nonblocking assignment in Verilog, respectively. The blue circle shape denotes the start of the algorithm; the yellow diamond shape denotes branch condition, and the green rectangular shape denotes the end of the algorithm.

Figure 4 illustrates a step-by-step procedure for a single systolic 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 (i.e., the solid flat rectangular shape in Fig. 3) between two neighboring nodes. 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 via pipeline buffering 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-(0), the signal `start_in` attached to the upper-left node in the systolic array is assertive, and thus the internal register `r` will be updated to '0' in the next clock cycle (see the red-colored '0' in step-(1), Fig. 4);
- In step-(1), the update propagates to the node positioned at the upper-left corner of the systolic array. Given the input signals `r`, `pivot_in==2'b00`, this specific node performs the `SWAP` operation. Consequently, `r` is updated to '0', and the node outputs '0'. Moreover, it propagates the '`SWAP`' signal along with `pivot_out = 0` to its right neighbor in the subsequent cycle. Notably, as the node currently does not represent a pivot, it broadcasts the message 'pivot not found';
- In step-(2), consider the first row of the systolic array, the leftmost node has updated '0' to the buffer register (colored in blue) and executes `SWAP` again since {`r`, `pivot_in`} == `2'b00`, and the second node executes '`SWAP`', which is passed from the leftmost node in step-(1). Consider the second row of the systolic array, the `start_in` signal attached to the first node is assertive and thus ready to accept the value '0' stored in the (blue) buffer register in the next cycle;
- In step-(3), examination of the first row reveals that the second node acts as the pivot due to `r`, `pivot_in ==2'b10`. As a pivot, this node disregards the `SWAP`' signal from the previous step and instead executes `PASS`'. In contrast, considering the second row, the first node has been updated to '0' and also emits the `SWAP` signal due to `r`, `pivot_in` == `2'b00`;
- 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. Further, a careful calculation shows the total delay for triangularizing a $k \times l$ binary matrix is:

$$2k + l - 2$$

and the following proposition proves the correctness of the proposed method.

Proposition 1 *The proposed Gaussian elimination systolic array can triangularize any $k \times l$ matrix over $\mathbb{F}_2$ correctly.*

Proof We prove here the correctness of matrix triangularization, by induction on the rows of the systolic array. Let M represent the input $k \times l$ matrix. Initially, consider the first row of the systolic array when the `start_in` signal is active. At this time, the first row of M denoted as M_1 is loaded. In the subsequent clock cycles when `start_in` is inactive, the logic specified in the systolic array nodes will update M_1 by j-th row of the matrix M_j whenever the pivot element in M_j lays ahead of M_1. This logic guarantees that the first row of the systolic array will eventually find the matrix row with the most significant '1'.

Next, consider the i-th row of the systolic array. Suppose at some time, the i-th row finds the matrix row with i-th significant '1'. In this case, the 1-st significant, 2-nd, up to $i-1$-th significant matrix rows should reside in the above $i-1$ rows of the systolic array such that the data stored in the first i rows of the systolic array are in a triangular shape. This triangular shape filters every row of the target matrix M and let the $i+1$-th, $i+2$-th, up to k-th significant row pass (note that the k-th significant row might be a null vector if M is not full-rank). Therefore, the logic specified in the $i+1$ row of the systolic array will eventually find the matrix row with the $i+1$-th significant '1', which is equivalent to finding the matrix row with the most significant '1' in the remaining $k-i$ unsorted rows of M. Following this induction, the systolic array always re-arranges the matrix M in a triangular shape. □

On the other hand, the Gaussian systemization is required immediately after the triangularization process to shape the matrix to the systematic form. Figure 5 describes the behavior logic of the node for matrix systemization, and Fig. 6 illustrates a toy example of how to systemize a 4×4 binary matrix $\begin{bmatrix} 1&1&0&0 \\ 0&1&0&1 \\ 0&0&1&0 \\ 0&0&0&1 \end{bmatrix}$ within 7 clock cycles. Compared with the triangularization process shown in Fig. 4, the node in the systemization process behaves in two different ways: First, at the initial stage (step-(0)), the leftmost node in each of the four rows is triggered by the signals `op_in='SWAP'` and `pivot_in=1'b0` where '`SWAP`' here slightly differs from the previous one by retaining the value in the internal register, meanwhile, outputing this value to `data_out`. Second, whenever the node cannot determine the pivot position, i.e., `{r,pivot_in}==2'b00`, it always performs '`PASS`'. During step-(1) and step-(7), the systolic array gradually outputs the result matrix $\begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{bmatrix}$ in reversed order and skewed format: it first outputs the last row [0001], then [0010] and [0100], and finally the first row [1000].

The correctness of systemization can be proven using the similar arguments shown in Proposition 1 and we skip the details in this paper. In summary, the total delay for $k \times l$ matrix systemization is a linear function of the matrix size as:

$$k + l$$

4.2.2 Gaussian elimination for large-sized matrices

Some cryptographic applications require eliminating large matrices. Processing large matrices is essential and performance-critical in these applications. For example, the ROLLO decryption requires to calculate the intersection of two vector spaces in the rank support recovery algorithm and later to systemize the intersected vector space to reconstruct the secret shared key K, which dominates the performance of ROLLO decryption. 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 a single systolic array by the method we proposed for medium-size matrices since the resource utilization has exceeded the maximum capacity of most Xilinx FPGAs. The new solution proposed for large matrices in this work is to divide the large matrix into several smaller blocks and to conquer each submatrix using a relatively small systolic array. There exists a tradeoff between hardware utilizaiton and processing time: we use a smaller Gaussian elimination hardware which can be tolerated on most FPGA platforms to Gauss-eliminate a large matrix but the price we pay is the increase of processing time of Gaussian elimination. In this work, we are particularly interested in such a tradeoff (or division) where our Gaussian elimination hardware can directly process a smaller matrix which shares the same column width as the large matrix does. This requirement removes the storage of intermediate operation codes (`op_in` signals from each Gaussian elimination node).

Again, we describe our idea with the same example used in Fig. 4 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 $\begin{bmatrix} 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}$. An additional function we add to the node is the signal `swap_in`, which permits to load the sorted rows out of the systolic array to external memory at desired timing. We assume here the 4×4 matrix is too 'large' to process and the 2×4 systolic array is exploited to do this task. The triangularization is done within two rounds of Gaussian

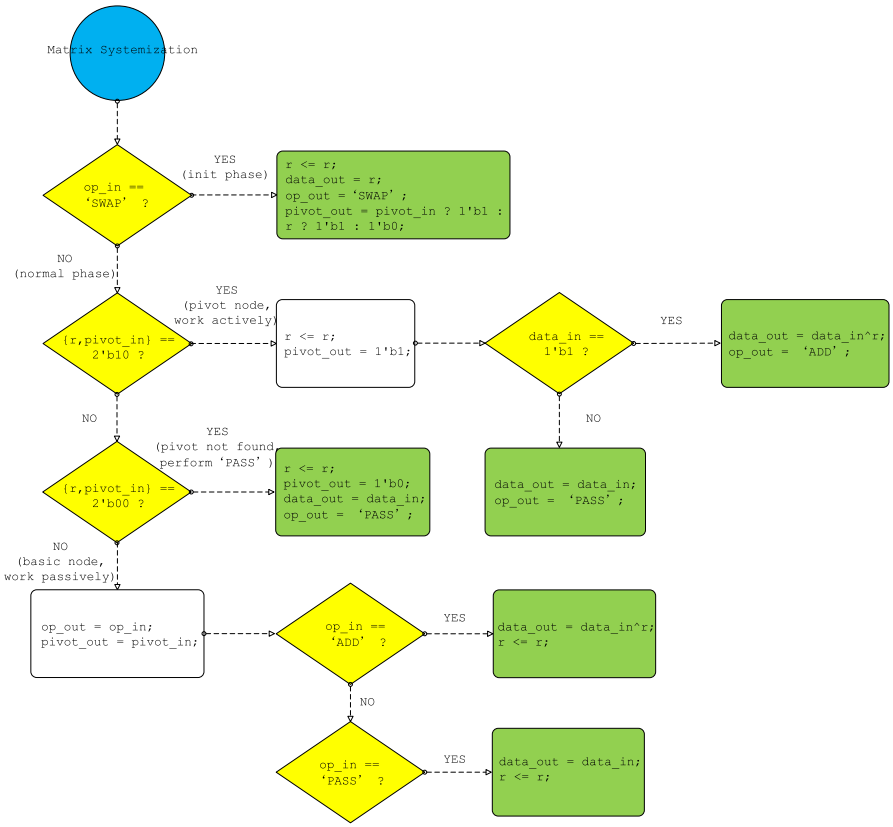


Fig. 5 Behavior description of the node used in the proposed systolic array for matrix systemization (by setting the node mode signal `mode_in`=1'b1), written in Verilog-like pseudocode

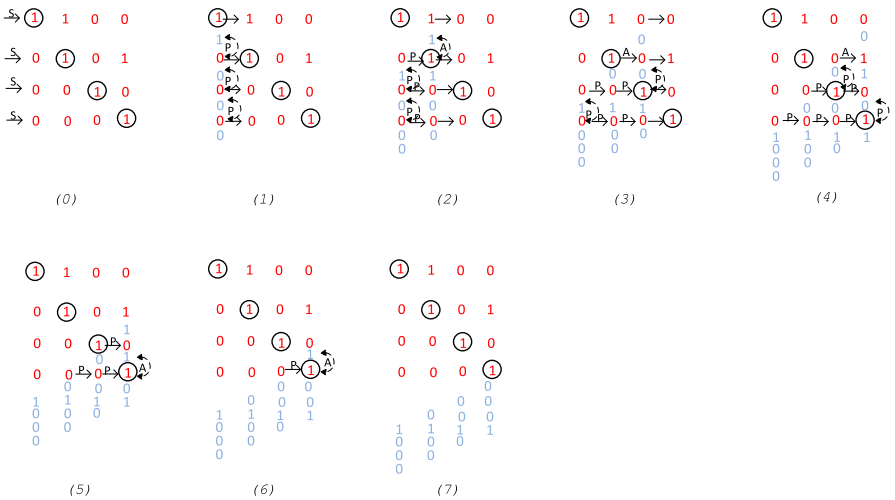
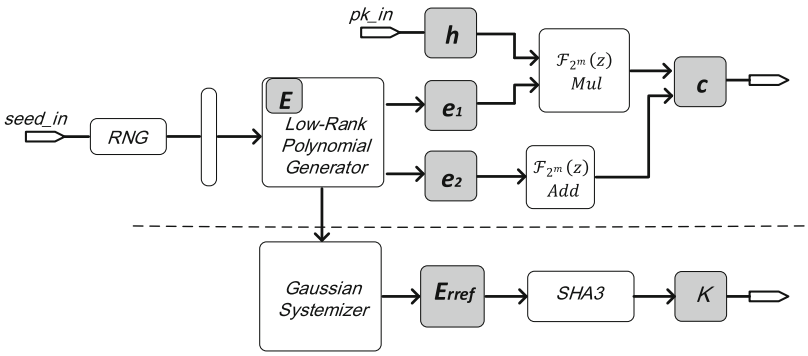


Fig. 6 A toy example for the proposed Gaussian elimination hardware by transforming a 4×4 matrix over $\mathbb{F}_2$ using the node logic shown in Fig. 5. Systemization refers to transforming a row-echelon matrix into its reduced-row-echelon form

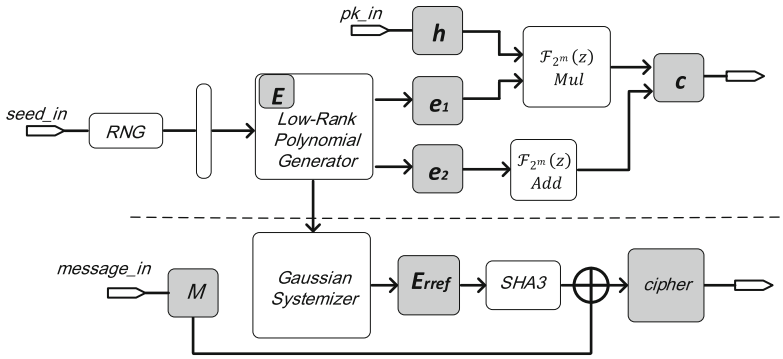
elimination. In Round-1, the systolic array sorts (Gaussian-eliminates) the first two rows of the matrix into $\left[\begin{smallmatrix}0&1&0&0\\0&0&0&1\end{smallmatrix}\right]$ and then loads them out. In Round-2, the systolic array sorts the remaining unsorted two lines of the matrix and then loads them out.

In more details, 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 the red colored texts in Fig. 9. `swap_in` is triggered to output the data within the internal register `r` and, meanwhile, 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 $\left[\begin{smallmatrix}0&0&0&0\\0&1&0&0\\0&0&0&1\\0&1&0&1\end{smallmatrix}\right]$ to its row echelon form $\left[\begin{smallmatrix}0&1&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{smallmatrix}\right]$ on a 2×4 systolic array is depicted step-by-step in Fig. 10. 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

Fig. 7 ROLLO encryption hardware

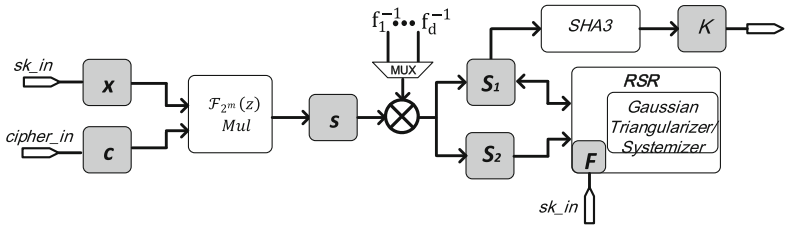


(a) Hardware architecture for ROLLO-I key encapsulation

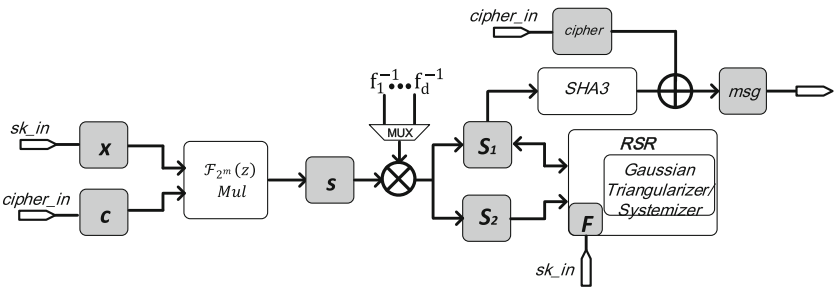


(b) Hardware architecture for ROLLO-II data encryption

Fig. 8 ROLLO decryption hardware



(a) Hardware architecture for ROLLO-I key decapsulation



(b) Hardware architecture for ROLLO-II data decryption

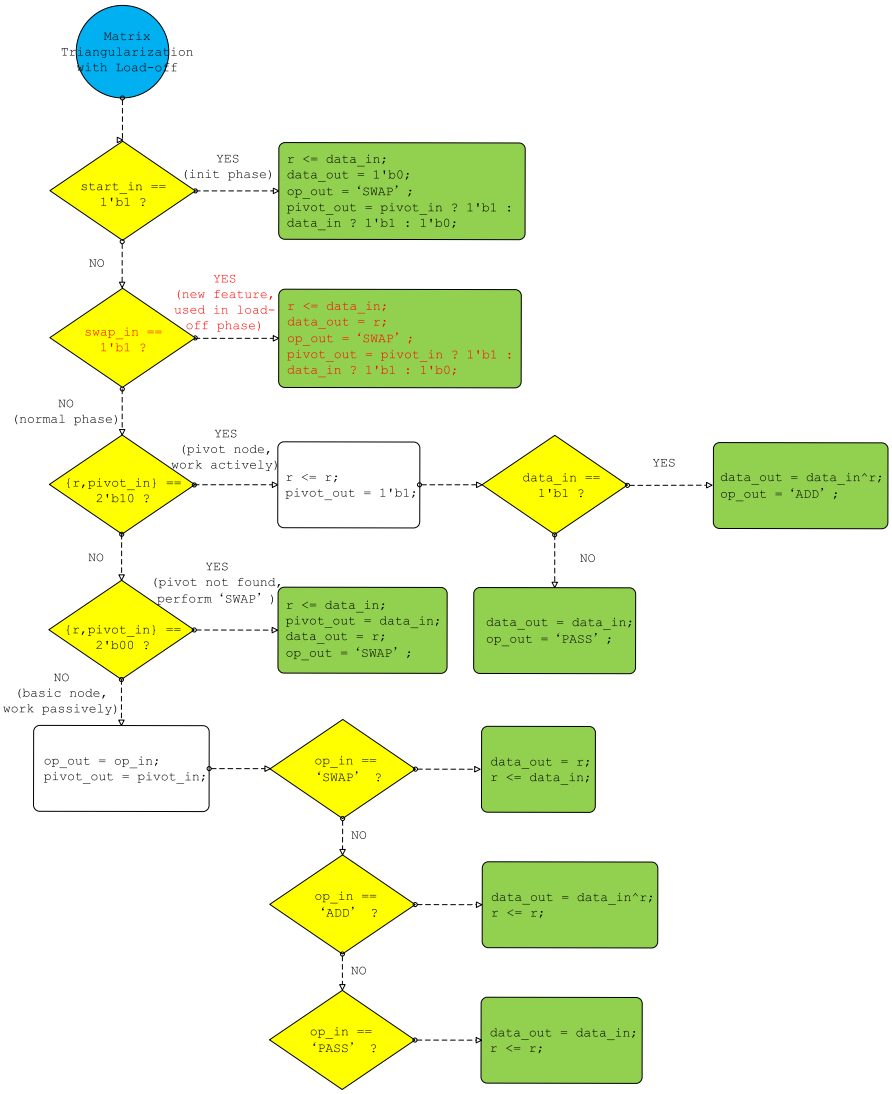


Fig. 9 Behavior description of the node used in the proposed systolic array for large matrix triangularization (by setting the node mode signal `mode_in`=1'b0), written in Verilog-like pseudocode

which manipulate only the last two rows of the input matrix and then load back to memory.

Specifically, in Round-1, initially at step-(0), the input matrix $\begin{bmatrix} 0&0&0&0\\ 0&1&0&0\\ 0&0&0&1\\ 0&1&0&1 \end{bmatrix}$ 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), 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 previous 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 upper-left node for loading-off to external memory storage; Starting from step-(5), the `swap_in` signals of the two leading nodes of the respective rows of the systolic array keep assertive 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,* firstly, 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 sorted 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. The exact results are stated and proved in Proposition 2.

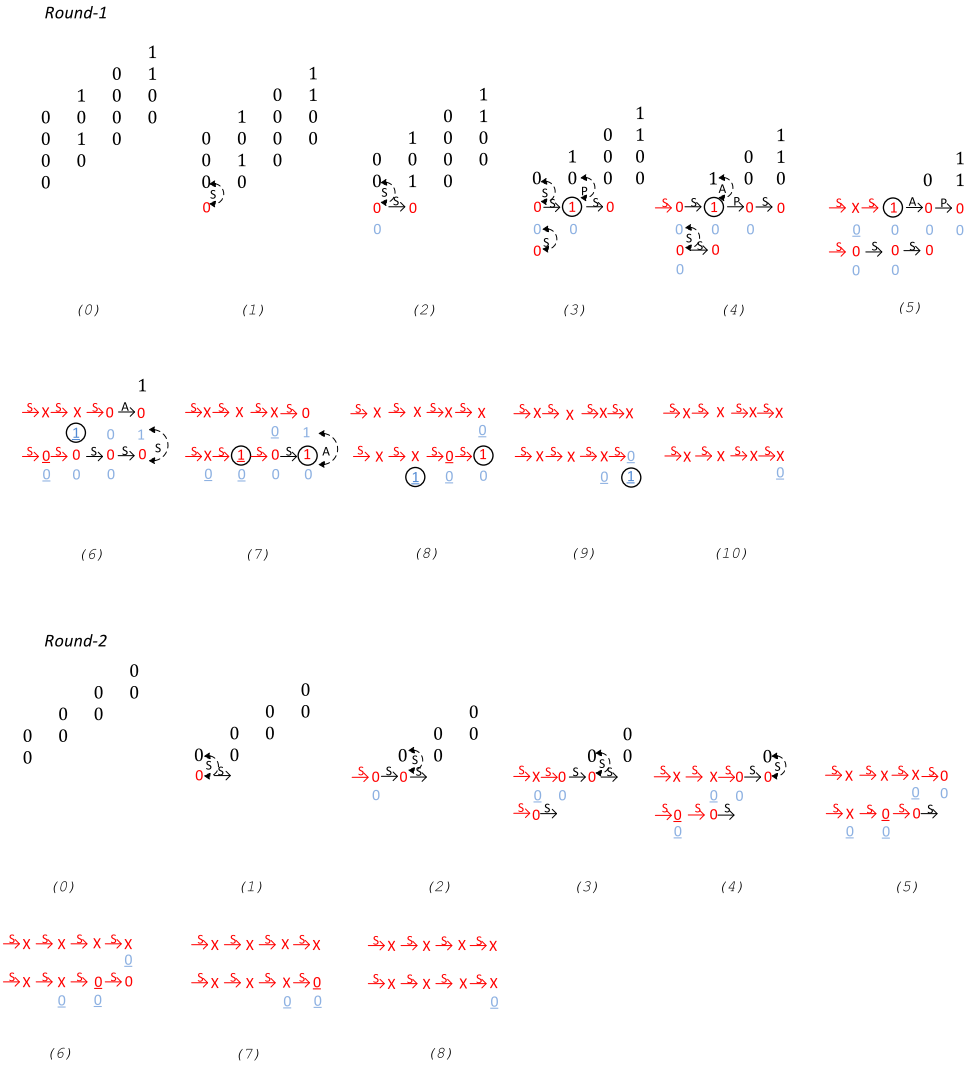


Fig. 10 A toy example to transform 4×4 matrix over $\mathbb{F}_2$ by the proposed 2×4 systolic array using the node logic shown in Fig. 9. Triangularization refers to putting the input matrix into its row-echelon form

Proposition 2 *The total number of rounds for one $d \times l (d < D)$ systolic array to triangularize a $D \times l$ matrix is D/d; A particular i-th round costs $D + l - 2 + (3 - i)d$ cycles to complete. The total cycle count for one $d \times l (d < D)$ systolic array to triangularize a $D \times l$ matrix is $\frac{D}{d}(\frac{D}{2} + l + \frac{5}{2}d - 2)$.*

Proof Each round sorts d rows of the D-rows matrix and thus the round complexity is D/d. At Round-$i (i = 1, \ldots, D/d)$, the first $(i-1)d$ rows have been sorted already and the systolic array needs to process the remaining $D-(i-1)d$ rows. It takes $2d-1$ cycles for the first output of the systolic array to appear since the first column of the systolic array consists of $2d$ registers; Note that the first output of the systolic array belongs to the unsorted $D-id$ rows and it takes $l-1+D-id$ cycles to output the entire $D-id$ unsorted rows; Finally, it takes d cycles to load out the sorted d rows stored in the nodes of the systolic array. These three parts contribute to the total cycle count of Round-i: $(2d-1)+(l-1+D-id)+d$. Therefore, the accumulation of all rounds gives the final cycle delay as $\sum_i D+l-2+(3-i)d = \frac{D}{d}(D+l-2+3d-\frac{d+D}{2}) \approx \frac{D}{d}(D+l-\frac{D}{2}) = \frac{(D+2l)D}{2d}$. □

4.3 High level description for implementing ROLLO

This subsection describes the adaptation of the proposed Gaussian elimination module for the complete 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 [14] 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.

4.3.1 ROLLO encapsulation/encryption

In encryption part, Gaussian elimination is essential to generate a unique symmetric key K: The error space $\mathbf{E}$, which is reprensented as a $k \times l$ binary matrix, must be converted

to reduced row-echelon form $\mathbf{E_{rref}}$, and thus hashing $\mathbf{E_{rref}}$ eventually returns the key K. Gaussian elimination on $\mathbf{E}$ includes two phases: first triangularize and then systemize which costs $2k + l - 2$ and $k + l$ cycles, respectively.

Hardware architecture

Figure 7 depicts the top-level 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 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 ciphertext c is calculated via the polynomial multiplier ($\mathbb{F}_{2^m}[z]$ `multiplier`) and adder, and K is calculated by hashing E_{rref} through the SHA3 module. Likewise, 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 for satisfying the security requirement and this results in increasing parameter values.

4.3.2 ROLLO decapsulation/decryption

In this subsection, we discuss the implementation details of ROLLO-I.Decap and ROLLO-II.Decrypt. The most critical component in the decryption part, is called `Rank Support Recovery (RSR)`. When the two syndrome spaces, i.e., S_1 and S_2, are ready in memory, the module RSR will perform the RSR algorithm (see Algorithm 3) to retrieve the intersection as the error vector space $E = S_1 \cap S_2$. The primary operation in RSR algorithm is the Zaussenhaus algorithm which returns the intersection of two vector spaces (see Algorithm 4). The most computational-intensive task of Zaussenhaus algorithm is essentially trianguarization on a relatively large block matrix $\left[\begin{array}{c|c} S_1 & S_1 \\ \hline S_2 & 0 \end{array}\right]$. 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, $S_i (i = 3, \cdots, d)$ is written to the following n rows of memory to formulate the large matrix as $\left[\begin{array}{c|c} E & E \\ \hline S_i & 0 \end{array}\right]$ 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.

Hardware architecture

Figures 8a and 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 contributes the majority of the hardware utilization. ROLLO-I and ROLLO-II share almost an identical architecture though the 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-II decrypts the cryptogram by XORing K.

Input: $\mathbf{s} = (s_1, \ldots, s_n) \in \mathbb{F}_{q^m}^n$ a syndrome of an error $\mathbf{e}$ 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 Si's`
2 Compute every $S_i = f_i^{-1} S$ for $i = 1$ to d
3 $E \leftarrow \cap_{i=1}^{d} S_i$ `// Repeat d times Zaussenhaus algorithm`
4 **return** E

Algorithm 3: Constant-Time Rank Support Recover (RSR) algorithm

input : Vector space $S_1 = (\vec{s_{1,1}}, \cdots, \vec{s_{1,n}})^T$ and Vector space $S_2 = (\vec{s_{2,1}}, \cdots, \vec{s_{2,k}})^T$
output: Intersection of vector space S_1 and S_2 as $S_1 \cap S_2$
1 Create a block matrix as $\left[\begin{array}{c|c} S_1 & S_1 \\ \hline S_2 & 0 \end{array}\right]$.
2 Perform Gaussian elimination (triangularization) on the block matrix above and obtain an updated block matrix as
$\left[\begin{array}{c|c} S_1 \cup S_2 & \ddots \\ \hline 0 & S1 \cap S_2 \\ \hline 0 & 0 \end{array}\right]$.
3 Return $S_1 \cap S_2$.

Algorithm 4: Zaussenhaus algorithm

4.4 Performance and comparisons

We show in Table 1 the scalability of our approach by implementing Gaussian eliminator for three different matrix sizes, 20×20, 50×50, and 90×90, on Xilinx Virtex-5 FPGA. This FPGA family is selected to enable fair comparison with previous work presented in [8]. We choose [8] as the primary comparison target since this work implements various systolic architectures for Gaussian elimination on the same device. We also include the experimental data on Xilinx Spartan-3 FPGA, reported in [15], which is the most recent hardware-based Gaussian elimination implementation that we are aware of.

Compared with the triangular architectures, including TSA, TSL, and TSN, our design mostly retains as high frequency as theirs due to the full pipeline structure. It also uses almost the same number of clock cycles. The significant increase of slice utilization is primarily due to the dual

Table 1 Gaussian elimination performance for square $k \times k$ binary matrix on Xilinx Virtex-5 FPGA, compared with the TSA, TSL, TSN, LSA, LSL presented in [8, 15]

Instance	$k \times k$	Device	Freq (MHz)	Cycle	Slice	Slice*Cycle/Freq
This work	20×20	Virtex-5	600	60	1228	122
	50×50		500	150	6499	1949
	90×90		500	270	21,954	11,855
TSA [8]	20×20	Virtex-5	600	80	363	58
	50×50			200	1727	691
	90×90			360	5804	4179
TSL [8]	20×20	Virtex-5	500	60	161	19
	50×50			150	912	274
	90×90			270	3082	1664
TSN [8]	20×20	Virtex-5	102	40	160	63
	50×50			100	715	701
	90×90			180	2045	17, 529
LSA [8]	20×20	Virtex-5	550	400	55	40
	50×50			2500	171	777
	90×90			8100	291	4286
LSL [8]	20×20	Virtex-5	550	400	33	24
	50×50			2500	78	355
	90×90			8100	116	1708
TSL [15]	50×50	Spartan-3	178	150	3129	2636

Table 2 Performance of ROLLO hardware and comparison with other code-based PQC hardware, targeting NIST security level 1 (128-bit pre-quantum security)

Instance	Freq (MHz)	Time (ms)	Cycle ($\times 10^3$)	LUTs/FFs	BRAM
ROLLO-I.encap	180	0.016	2.8	24,154/11,735	24.5
ROLLO-I.decap	148	0.138	24.9	36,772/21,832	23.5
ROLLO-II.encap	170	0.076	12.9	31,360/16,029	31.5
ROLLO-II.decap	175	0.398	69.6	45,207/26,598	29
McEliece.encap [16]	113	0.3	30	40,018/61,881	177.5
McEliece.decap [16]		0.9	100		
HQC.encap [17]	148	0.6	90	20,000/16,000	12.5
HQC.decap [17]		1.3	190		

All the designs are synthesized on an Artix-7 FPGA

switching mode used in the node: The conventional triangular architectures for $k \times k$ binary matrices consist of k pivot nodes along the diagonal of the systolic array and $k(k-1)/2$ non-pivot nodes at the remaining positions. Therefore, the area complexity is determined by the number of non-pivot nodes, which is quadratic as a function of k. On the contrary, our design consists of $k(k+1)/2$ dual-functional nodes, and the number of these nodes determines the slice count. The dual-functional node presented in this work can be roughly interpreted as a combination of the pivot and non-pivot nodes, and the pivot utilization outweighs the non-pivot.

In terms of speed, compared with the linear architectures, our design is significantly faster, since it runs in $\Theta(k)$ steps, whereas linear architectures run in $\Theta(k^2)$ steps. The linear architectures are advantageous in lightweight applications since the resource utilization increases linearly with the dimension k. On top of that, when previous designs are used as SLE solvers, they cannot return valid solutions for unsolvable under-determined equations, which are equivalent to Gaussian-eliminating singular matrices. Our new design, however, overcomes this difficulty (Fig. 10).

Finally, we evaluate the performance of the ROLLO encapsulation and decapsulation hardware by integrating the proposed Gaussian elimination module on Xilinx Artix-7 FPGA. The implementation results are collected and compared with state-of-the-art code-based scheme implementations of classic McEliece [16] and HQC [17], as shown in Table 2. Both [16] and [17] offer lightweight and high-speed implementations on a unified architecture for key generation, data encapsulation, and data decapsulation. Since our work is more oriented towards high performance, we only list the high-speed implementation results of [16] and [17].

Compared with McEliece and HQC, ROLLO demonstrates faster runtime and consumes fewer clock cycles. However, the unified architecture design of McEliece and HQC offers advantages in resource utilization, despite McEliece requiring a significant amount of block memory due to the large public key.

5 Conclusions

This paper explored the possibility of realizing Gaussian elimination for arbitrary binary matrices on hardware. The idea stems from the proposed dynamical dual switching mode, which allows the hardware to determine the position of pivot elements in each row of the matrix on the fly. The correctness of the universal Gaussian elimination using this new type of switching mode is strictly proved. We showcased the usefulness of hardware-based Gaussian elimination for medium-sized and large-sized binary matrices. It is the first available hardware architecture for Gaussian elimination that supports quantum-resisting rank-code-based cryptography with varying security parameters.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China under Grant 62002023, in part by Guangdong Provincial Key Laboratory IRADS under Grant 2022B1212010006 and Grant R0400001-22, in part by Guangdong Province General Universities Key Field Project (New Generation Information Technology) under Grant 2023ZDZX1033, in part by UIC Research under Grant UICR04202401-21, and in part by A*Star, Singapore under research grant SERC A19E3b0099.

References

1. Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 392–407. Springer (2000)
2. Yang, B.-Y., Chen, J.-M., Courtois, N.T.: On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In: International Conference on Information and Communications Security, pp. 401–413. Springer (2004)
3. Ding, J., Schmidt, D.: Rainbow, a new multivariable polynomial signature scheme. In: International Conference on Applied Cryptography and Network Security, pp. 164–175. Springer (2005)
4. Bogdanov, A., Eisenbarth, T., Rupp, A., Wolf, C.: Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 45–61. Springer (2008)
5. Philippe Gaborit, J.-C.D.e.a.: ROLLO—Rank-Ouroboros, LAKE, LOCKER, Updated on April 21st. https://pqc-rollo.org/doc/rollo-specification_2020-04-21.pdf (2020)
6. Hochet, B., Quinton, P., Robert, Y.: Systolic Gaussian elimination over GF (p) with partial pivoting. IEEE Trans. Comput. **38**(9), 1321–1324 (1989)
7. Wang, C.-L., Lin, J.-L.: A systolic architecture for computing inverses and divisions in finite fields $GF(2^m)$. IEEE Trans. Comput. **42**(9), 1141–1146 (1993)
8. Rupp, A., Eisenbarth, T., Bogdanov, A., Grieb, O.: Hardware SLE solvers: efficient building blocks for cryptographic and cryptanalytic applications. Integration **44**(4), 290–304 (2011)
9. Tang, S., Yi, H., Ding, J., Chen, H., Chen, G.: High-speed hardware implementation of rainbow signature on FPGAS. In: International Workshop on Post-Quantum Cryptography, pp. 228–243. Springer (2011)
10. Balasubramanian, S., Carter, H.W., Bogdanov, A., Rupp, A., Ding, J.: Fast multivariate signature generation in hardware: the case of rainbow. In: 2008 International Conference on Application-Specific Systems, Architectures and Processors, pp. 25–30. IEEE (2008)
11. Shoufan, A., Wink, T., Molter, H.G., Huss, S.A., Kohnert, E.: A novel cryptoprocessor architecture for the McEliece public-key cryptosystem. IEEE Trans. Comput. **59**(11), 1533–1546 (2010)
12. Wang, W., Szefer, J., Niederhagen, R.: FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes. In: International Conference on Cryptographic Hardware and Embedded Systems, pp. 253–274. Springer (2017)
13. Gaborit, P., Murat, G., Ruatta, O., Zémor, G.: Low rank parity check codes and their application to cryptography. In: Proceedings of the Workshop on Coding and Cryptography WCC, vol. 2013 (2013)
14. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki–Okamoto transformation. In: Theory of Cryptography Conference, pp. 341–371. Springer (2017)
15. Wang, W., Szefer, J., Niederhagen, R.: Solving large systems of linear equations over gf (2) on FPGAS. In: 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–7. IEEE (2016)
16. Chen, P.-J., Chou, T., Deshpande, S., Lahr, N., Niederhagen, R., Szefer, J., Wang, W.: Complete and improved FPGA implementation of classic McEliece. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2022). https://doi.org/10.46586/tches.v2022.i3.71-113
17. Philippe Gaborit, J.-C.D.: Hamming Quasi-Cyclic (HQC) April 2020. https://pqc-hqc.org/doc/hqc-specification_2020-10-01.pdf (2020)