Complete and Improved FPGA Implementation of Classic McEliece*

Po-Jen Chen[1,2], Tung Chou[2], Sanjay Deshpande[3], Norman Lahr[4], Ruben Niederhagen[5], Jakub Szefer[3] and Wen Wang[3]

[1] GIEE, National Taiwan University, Taipei, Taiwan, mooseedsheeran@gmail.com
[2] CITI, Academia Sinica, Taipei, Taiwan, blueprint@crypto.tw
[3] CASLAB, Deptartment of Electrical Engineering, Yale University, New Haven, US,
sanjay.deshpande@yale.edu,jakub.szefer@yale.edu,wen.wang.ww349@yale.edu
[4] ACE, Fraunhofer SIT, Darmstadt, Germany, norman@lahr.email
[5] IMADA, University of Southern Denmark, Odense, Denmark, ruben@polycephaly.org

Abstract. We present the first specification-compliant constant-time FPGA implementation of the Classic McEliece cryptosystem from the third-round of NIST's Post-Quantum Cryptography standardization process. In particular, we present the first complete implementation including encapsulation and decapsulation modules as well as key generation with seed expansion. All the hardware modules are parametrizable, at compile time, with security level and performance parameters. As the most time consuming operation of Classic McEliece is the systemization of the public key matrix during key generation, we present and evaluate three new algorithms that can be used for systemization while complying with the specification: hybrid early-abort systemizer (HEA), single-pass early-abort systemizer (SPEA), and dual-pass early-abort systemizer (DPEA). All of the designs outperform the prior systemizer designs for Classic McEliece by 2.2× to 2.6× in average runtime and by 1.7× to 2.4× in time-area efficiency. We show that our complete Classic McEliece design for example can perform key generation in 5.2 ms to 20 ms, encapsulation in 0.1 ms to 0.5 ms, and decapsulation in 0.7 ms to 1.5 ms for all security levels on an Xlilinx Artix 7 FPGA. The performance can be increased even further at the cost of resources by increasing the level of parallelization using the performance parameters of our design.

Keywords: Classic McEliece · Key Encapsulation Mechanism · Code-Based Cryptography · PQC · FPGA · Hardware Implementation

1 Introduction

In 2016 NIST started a standardization process[1] with the goal to standardize cryptographic primitives that are secure against attacks aided by quantum computers. There are several families of post-quantum cryptography: hash-based, code-based, lattice-based, multivariate, and isogeny-based cryptography. One of the "finalists" in the third round of the standardization process is the code-based key encapsulation mechanism (KEM) Classic McEliece [ABC+20]. Classic McEliece is generally considered a conservative choice: Its security properties are relatively well understood, but its public key size ranges from 0.25 to 1.3 megabytes. Despite its name honoring Robert J. McEliece as the founder of

*September 5, 2022. This version of the paper provides fixes to errata in the version of TCHES 2022.3. A list of all errata (marked in blue in the text) is provided at the end of this document.

[1] https://csrc.nist.gov/projects/post-quantum-cryptography

code-based cryptography, Classic McEliece uses the syndrome-based dual variant of the McEliece cryptosystem [McE78] introduced by Harald Niederreiter [Nie86].

An important aspect of the NIST standardization process is the performance of the submissions both in software and in hardware, and there have been many publications providing software and hardware optimizations. Optimized software implementations of Classic McEliece for x86 systems are described, e.g., in [BCS13, Cho17] and an implementation for a Cortex M4 system in [CC21]. There have been several hardware implementations of McEliece and Niederreiter cryptosystems. For example, Eisenbarth et. al. [EGHP09] describe a hardware design for the McEliece cryptosystem including encryption and decryption; the design by Shoufan et. al. [SWM$^+$10] includes key generation, encryption, and decryption. Gosh et. al. [GDUV12] as well as Massolino et. al. [MBR15] target decryption only. A hardware implementation of encryption and decryption for the Niederreiter variant is provided by Heyse et. al. in [HG13]. López-García et. al. [LGCN20] describe a hardware-software co-design for the McEliece cryptosystem. These hardware publications do not target the exact parameter sets and algorithmic specifications of the Classic McEliece submission since they either pre-date the Classic McEliece specification or implement different variants of the original cryptosystems of McEliece and Niederreiter.

The hardware implementation accompanying the specification of Classic McEliece is described in [WSN17, WSN18]. However, this hardware implementation covers only the core functionalities of key generation, encryption, and decryption, but it does not cover encapsulation and decapsulation as well as generation of the keys from a seed.

Motivation. Each attempt of public-key generation, when the systematic variants of key generation are used, fails with a high probability: It fails whenever the input parity-check matrix can not be reduced to systematic form (for convenience, in the remainder of the paper we say that a matrix is “systemizable” if it can be reduced to systematic form). Software implementations released by the Classic McEliece team, along with the implementation in [CC21], thus make use of early abort so that matrices that are not systemizable are quickly detected. The early abort does not violate constant-time requirements, since it triggers an entirely new secret and public key generation. The early abort approach is faster because it only needs to operate on the leftmost $(n-k) \times (n-k)$ submatrix in order to detect whether the whole $(n-k) \times n$ matrix is systemizable. This has not been used in previous hardware implementations. In addition, the hardware implementation accompanying the Classic McEliece submission does not implement the complete KEM specification but only its Niederreiter core. Hence, all currently existing hardware implementations are not fully specification-compliant.

Contribution. Our contributions are as follows:

- We introduce the three more efficient algorithmic systemizer variants. The systemizers all make use of early-abort to accelerate the systemization. We introduce our hardware designs for the systemizers and compare them to [WSN17, WSN18] in terms of area and speed. All of our designs outperform designs prior art for Classic McEliece by 2.2× to 2.6× in average runtime and by 1.7× to 2.4× in time-area efficiency.

- Based on our improved designs for public-key generation and on the hardware implementation of [WSN17, WSN18] of the core cryptographic functionalities of Classic McEliece, we provide the first complete specification-compliant FPGA implementation of Classic McEliece including seeded key generation, encapsulation, and decapsulation, as well as a joint design of all three operations, adherent to the latest (third-round) Classic McEliece specification.

- Similar to [WSN17, WSN18], our designs are constant time (i.e., the runtime does not depend on any secret information) and provide compile-time parameters for selecting the desired security level and performance.

- We evaluate the resource requirements of our designs on an Xilinx Artix 7 FPGA as recommended by NIST for the evaluation of PQC hardware designs.

Our key generation module implements the systematic variants of key generation. It has been shown that the semi-systematic variants significantly speed up key generation in software implementations [ABC^{+}20, Sect. 2.2.1]. However, we expect that a hardware module for key generation for the semi-systematic public key generation will be more complex than one for the systematic variants, so we consider this as a future work. Since the public key is oblivious of the key generation variants, our encapsulation module works for all variants. Our decapsulation module, for now, only works with the systematic variants, but it can be adapted for the semi-systematic variants with some small changes.

The source code of our hardware designs is available under an open source license at `https://caslab.csl.yale.edu/code/pqc-classic-mceliece/`.

Structure of this paper. We give a brief introduction to code-based cryptography and the relevant algorithms of the Classic McEliece specification in Section 2. We introduce prior systemizer designs in Section 3. We describe and evaluate our performance improvements of the key generation and our three new systemizer variants in Section 4. The modifications and extensions to [WSN17, WSN18] in order to obtain a complete Classic McEliece implementation are described in Section 5. Finally, in Section 6 we describe the overall joint design of the entire Classic McEliece cryptosystem, compare its performance to selected code-based designs, and conclude the paper.

2 Classic McEliece

Code-based cryptography was introduced by McEliece in 1978 [McE78]. The McEliece cryptosystem uses as public key a generator matrix $G \in \mathbb{F}_2^{k \times n}$ with code length n and code rank k of a binary Goppa code $\mathcal{G}$ that can correct up to t errors. Goppa codes are defined using a binary field $\mathbb{F}_q$ with $q = 2^m$ and an irreducible Goppa polynomial g of degree t. The sender encrypts a message by converting it into a vector $m' \in \mathbb{F}_2^k$ and computes the ciphertext $c \in \mathbb{F}_2^n$ as erroneous code word $c = m'G + e$ where $e \in \mathbb{F}_2^n$ is an error vector of weight t. The receiver then uses the secret code structure of the code $\mathcal{G}$ to correct the errors and decode the codeword back to the message m'.

In 1986, Niederreiter proposed a dual-variant of the McEliece scheme [Nie86]: In his version, a parity check matrix $H \in \mathbb{F}_2^{(n-k) \times n}$ is used as public key and the sender encodes the message as an error vector $e \in \mathbb{F}_2^n$ of weight t and encrypts it to a ciphertext $c \in \mathbb{F}_2^{n-k}$ as the syndrome $c = He$. Again, the receiver uses the secret code structure in order to recover the error positions in the syndrome and hence the plaintext. In his proposal, Niederreiter used a code family that later was broken; however, the overall scheme remains secure with binary Goppa codes.

The Classic McEliece submission to NIST [ABC^{+}20] is using the variant by Niederreiter with binary Goppa codes as proposed by McEliece. The parameter sets of Classic McEliece from the third round of the standardization process are shown in Table 1.

Algorithm 1 shows the key generation from a secret random seed as specified in the submission. The function FieldOrdering returns a random permutation of the filed elements from a seed as the secret support $\alpha_1, \ldots, \alpha_n$; for details see [ABC^{+}20, Sect. 2.4.2]. The function Irreducible returns a random irreducible Goppa polynomial g; for details see [ABC^{+}20, Sect. 2.4.1]. Both the support $\alpha_1, \ldots, \alpha_n$ and the Goppa polynomial g are part of the secret key. The public key is generated from the private key using the function MatGen shown in Algorithm 2. It computes a binary matrix $\hat{H}$ from $\alpha_1, \ldots, \alpha_n$ and g

Algorithm 1 SeededKeyGen(δ) algorithm (using PRNG G) [ABC+20, Sect. 2.4.3].

```
Compute $E = \mathsf{G}(\delta)$, a string of $n + \sigma_2 q + \sigma_1 t + \ell$ bits.
Define $\delta'$ as the last $\ell$ bits of $E$.
Define $s$ as the first $n$ bits of $E$.
Compute $\alpha_1, \ldots, \alpha_q$ from the next $\sigma_2 q$ bits of $E$ by the FieldOrdering algorithm.
If this fails, set $\delta \leftarrow \delta'$ and restart the algorithm.
Compute $g$ from the next $\sigma_1 t$ bits of $E$ by the Irreducible algorithm.
If this fails, set $\delta \leftarrow \delta'$ and restart the algorithm.
Define $\Gamma = (g, \alpha_1, \alpha_2, \ldots, \alpha_n)$. (Note that $\alpha_{n+1}, \ldots, \alpha_q$ are not used here.)
Compute $(T, c_{n-k-\mu+1}, \ldots, c_{n-k}, \Gamma') \leftarrow$ MatGen($\Gamma$).
If this fails, set $\delta \leftarrow \delta'$ and restart the algorithm.
Write $\Gamma'$ as $(g, \alpha'_1, \alpha'_2, \ldots, \alpha'_n)$.
Output $T$ as public key and $(\delta, c, g, \alpha, s)$ as private key, where $c = (c_{n-k-\mu+1}, \ldots, c_{n-k})$
and $\alpha = (\alpha'_1, \ldots, \alpha'_n, \alpha_{n+1}, \ldots, \alpha_q)$.
```

Algorithm 2 MatGen(Γ) algorithm (systematic form) [ABC+20, Sect. 2.2.2].

```
Compute the $t \times n$ matrix $\tilde{H} = \{h_{i,j}\}$ over $\mathbb{F}_q$, where $h_{i,j} = \alpha_j^{i-1}/g(\alpha_j)$ for $i = 1, \ldots, t$
and $j = 1, \ldots, n$.
Form an $mt \times n$ matrix $\hat{H}$ over $\mathbb{F}_2$ by replacing each entry $u_0 + u_1 z + \cdots + u_{m-1} z^{m-1}$
of $\tilde{H}$ with a column of $m$ bits $u_0, u_1, \ldots, u_{m-1}$.
Reduce $\hat{H}$ to systematic form $(I_{n-k} \mid T)$ where $I_{n-k}$ is an $(n-k) \times (n-k)$ identity matrix.
If this fails, return $\perp$.
Return $(T, \Gamma)$.
```

Algorithm 3 Encap(T) algorithm with hash-function H [ABC+20, Sect. 2.4.5].

```
Use FixedWeight to generate a vector $e \in \mathbb{F}_2^n$ of weight $t$.
Compute $C_0 =$ Encode($e, T$).
Compute $C_1 = \mathsf{H}(2, e)$. Put $C = (C_0, C_1)$.
Compute $K = \mathsf{H}(1, e, C)$.
Output ciphertext $C$ and session key $K$.
```

Algorithm 4 FixedWeight algorithm [ABC+20, Sect. 2.4.4].

```
Generate $\sigma_1 \tau$ uniform random bits $b_0, b_1, \ldots, b_{\sigma_1 \tau - 1}$.
Define $d_j = \sum_{i=0}^{m-1} b_{\sigma_1 j + i} 2^i$ for each $j \in \{0, 1, \ldots, \tau - 1\}$.
Define $a_0, a_1, \ldots, a_{t-1}$ as the first $t$ entries in $d_0, d_1, \ldots, d_{\tau-1}$ in the range
$\{0, 1, \ldots, n-1\}$. If there are fewer than $t$ such entries, restart the algorithm.
If $a_0, a_1, \ldots, a_{t-1}$ are not all distinct, restart the algorithm.
Define $e = (e_0, e_1, \ldots, e_{n-1}) \in \mathbb{F}_2^n$ as the weight-$t$ vector such that $e_{a_i} = 1$ for each $i$.
Return $e$.
```

Algorithm 5 Encode(e, T) algorithm [ABC+20, Sect. 2.2.3].

```
Define $H = (I_{n-k} \mid T)$.
Compute and return $C_0 = He \in \mathbb{F}_2^{n-k}$.
```

Table 1: Parameter sets of Classic McEliece [ABC+20].

Parameter Set		Parameters			
systematic	semi-systematic	m	n	t	$n-k$
`mceliece348864`	`mceliece348864f`	12	3488	64	768
`mceliece460896`	`mceliece460896f`	13	4608	96	1248
`mceliece6688128`	`mceliece6688128f`	13	6688	128	1664
`mceliece6960119`	`mceliece6960119f`	13	6960	119	1677
`mceliece8192128`	`mceliece8192128f`	13	8192	128	1664

Algorithm 6 Decap $((\delta, c, g, \alpha, s), C)$ algorithm [ABC+20, Sect. 2.3.3].

```
Split the ciphertext $C$ as $(C_0, C_1)$ with $C_0 \in \mathbb{F}_2^{n-k}$ and $C_1 \in \mathbb{F}_2^{\ell}$.
Set $b \leftarrow 1$.
Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_1, \alpha'_2, \ldots, \alpha'_n)$ from the private key.
Compute $e \leftarrow \textsc{Decode}(C_0, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
Compute $C'_1 = \mathsf{H}(2, e)$.
If $C'_1 \neq C_1$, set $e \leftarrow s$ and $b \leftarrow 0$.
Compute $K = \mathsf{H}(b, e, C)$.
Output session key $K$.
```

Algorithm 7 Decode(C_0, Γ') algorithm [ABC+20, Sect. 2.2.4].

```
Extend $C_0$ to $v = (C_0, 0, \ldots, 0) \in \mathbb{F}_2^n$ by appending $k$ zeros.
Find the unique codeword $c$ in the Goppa code defined by $\Gamma'$ that is at distance $\leq t$ from $v$. If there is no such codeword, return $\perp$.
Set $e = v + c$.
If $\mathsf{wt}(e) = t$ and $C_0 = He$, return $e$. Otherwise return $\perp$.
```

and then reduces $\hat{H}$ to its systematic form $(I_{n-k}|T)$. This operation typically is the most expensive operation of the key generation. Reduction of the quasi-random binary matrix $\hat{H}$ might fail; in that case, key generation is repeated with a new seed.

In Classic McEliece, the OW-CPA secure public key encryption (PKE) schemes from McEliece and Niederreiter are converted into an IND-CCA2 secure KEM. Encapsulation is shown in Algorithm 3. First, the function FixedWeight (see Algorithm 4) is used to generate an error vector $e \in \mathbb{F}_2^n$ of weight t. Then this error vector is encoded into a syndrome C_0 using the function Encode shown in Algorithm 5 as described above. The complete parity check matrix is obtained by appending the public key T to the identity matrix I_{n-k}. The error vector is then hashed to obtain $C_1 = \mathsf{H}(2, e)$ and the ciphertext $C = (C_0, C_1)$. The session key is obtained by hashing the error vector e and the ciphertext C; both hash operations use domain separation.

Decapsulation is shown in Algorithm 6. First, the ciphertext C is split into C_0 and C_1. Then, the function Decode (see Algorithm 7) is used to obtain the error vector e from C_0 and to verify that $C_0 = He$. After the hash of e has been compared to C_1, the shared session key K is computed and returned.

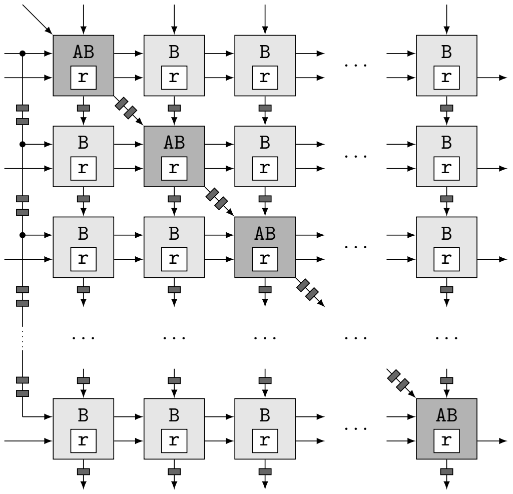


Figure 1: Layout of the processor array from [WSN16].

3 Previous Systemizer Designs

As shown in Algorithm 2, the central computation in public key generation for Classic McEliece is to compute the systematic form of a binary matrix. Generally, Gaussian elimination can be used for this task. There are several hardware designs for Gaussian elimination of matrices over $\mathbb{F}_2$ in literature, e.g., [HQR89, BMP$^+$06, REBG11, YL15]. However, for the systemization of the public key it is not necessary to compute the complete row-echelon matrix if the public key matrix is not systemizable. In this case, computation can be aborted once ~~a~~ there is no pivot found on the left diagonal of the matrix.

The Gaussian elimination designs mentioned above all are using an array of computational nodes organized as systolic lines or systolic networks. They are organized in the shape of an upper triangular matrix of the same dimension as the matrix that needs to be processed and the entire matrix is fed into the array during computation. This approach is fine for relatively small matrices — but for the large matrices of the Classic McEliece cryptosystem, this would require too many resources. Therefore, the hardware implementation of the McEliece cryptosystem in [SWM$^+$10] exploits the fact that the row-echelon form of the public key matrix does not need to be computed during key generation for non-systematic matrices and uses a design where the matrix data is processed in column blocks of a certain width s. This has the advantage that the size of the array can be more freely chosen, and is only as large as the block width s and not as large as the entire matrix, which reduces resource requirements significantly.

The basis for our work is the systemizer[2] hardware design proposed in [WSN16]. This design has been extended for the use in the hardware implantation of Niederreiter key generation in [WSN17] and for the complete hardware implementation of the Niederreiter cryptosystem in [WSN18]. The design from [WSN16] uses a similar approach as [SWM$^+$10] to reduce the public key matrix to systematic form.

In [WSN16], matrix data is stored in on-chip block RAM. The design uses the processor array shown in Figure 1 of processors shown with their input and output ports in Figure 2. The processor array has a width and height of s processor elements. The matrix data is streamed row-wise in row slices of blocks of s columns into the processor array via the `data_in` ports of the processors on the top of the processor array and back out of the processor array via the `data_out` ports of the processors in the bottom during computation.

[2] By systemizer we mean an algorithm or a hardware module that reduces the input matrix to systematic form. The systemizer may return “failure” if the input matrix is not systemizable.

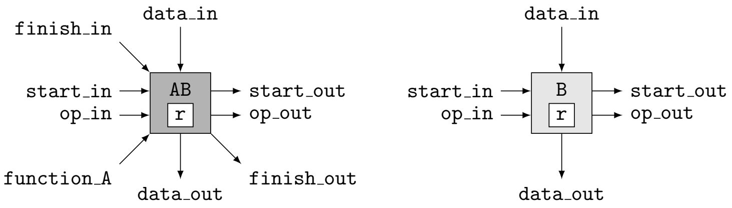


Figure 2: Input and output ports of `processor_AB` and `processor_B` from [WSN16].

Each processor element has a register `r` that stores some row slices of the matrix during the computation. The diagonal nodes in the processor array in Figure 1 of type `processor_AB` (short `AB` in Figure 1) are the pivoting nodes: They have the task to issue operations to the following nodes of type `processor_B` (short `B` in Figure 1) in their processor row. The operations are streamed out of the processor array via the `op_out` ports of the processors on the right side of the processor array into an operation memory. Later, these operations can be streamed back into the processor via the `op_in` processors on the left side of the processor array. The operations are:

- `pass`: The input data is passed on to the next processor row without modification.
- `swap`: The data currently stored in register `r` is passed on to the next processor row and the input data is stored in register `r` instead.
- `add`: The input data is added to currently stored (pivot) data and the result is passed on to the next processor row.

The processors in the processor array are used in a concerted way to operate on the input matrix in order to compute its upper-triangular and systematic form of the matrix as described in the following text.

Computing the upper-triangular form. We say that a matrix is in upper-triangular form, if 1) all the elements below the main diagonal are 0's and 2) all diagonal elements are 1's. Let's first look into how to compute the upper-triangular form of a systemizable matrix: The computation on the entire matrix is divided into several phases with several steps each. In each step, one column block of s columns (as many as the width of the processor array) is fed into the array from the top, one row-slice per cycle. The first column block contains columns 1 to s, the second column block contains columns $s+1$ to $2s$ and so on. Each phase computes s pivots. The process works as follows:

The first step of the first phase is the *pivot step*: The processor nodes of type `processor_B` in the lower left rectangle do not need to contribute to the computation — they constantly receive `pass` as operation. The diagonal pivot nodes (`processor_AB`) are configured to do the pivoting. In the very beginning, all nodes are "empty" — they do not store any matrix data. As the first column block of the matrix is fed into the array, empty nodes are initialized by storing an incoming matrix row-slice in their data registers `r` — the pivot node issues a `swap` operation (the design in [WSN16] uses `start_in` to instruct the pivot nodes to issue the `swap` operation) to the following `processor_B` in its processor row. If the incoming row-slice can be a pivot row, i.e., if it has a 1 at the respective pivot position, then this row-slice can be used right away for reducing the following row-slices (as described below). Otherwise, the following incoming row-slices are checked if they are suitable as pivot row. If not, they are simply passed on — the pivot node issues a `pass` operation. If yes, this row slices is stored in the nodes — the pivot node issues a `swap` operation to the following nodes in its processor row. This evicts the previous non-pivot row, which has the effect that the pivot row and the non-pivot row are swapped.

Once a pivot node has found a suitable matrix row, it can start reducing the following matrix rows: If the incoming data of the corresponding column is a 1, it issues an `add` operation, otherwise it issues a `pass` operation. Hence, the following `processor_B` nodes of the processor row add the incoming matrix row to the stored pivot row or they pass it on without modification correspondingly. Once all row-slices of the current column block have been fed into the processor array, the remaining matrix data, which is still stored in the registers `r` of the processor elements, is read out of the respective processor rows using a `swap` operation (the design in [WSN16] uses the signal `finish_in` to instruct the pivoting nodes to issue the `swap` operation). The data can then be passed out of the processor array using `pass` operations in the following processor rows.

This first pivot step has only processed data from the first column block. In order to process the remaining column blocks, all operations that are issued in the first step are recorded into a dedicated operation memory. In the next step, all nodes including the diagonal nodes act as normal, non-pivoting nodes of type `processor_B`. The operations are read from the operation memory and fed back into the processor array in synchronization with the matrix data of the next column block. Thus, the `pass`, `swap`, and `add` operations from the pivoting step are used to compute the same operations on the next column block. This process is repeated for all remaining column blocks, always replaying the stored operations from the pivoting step.

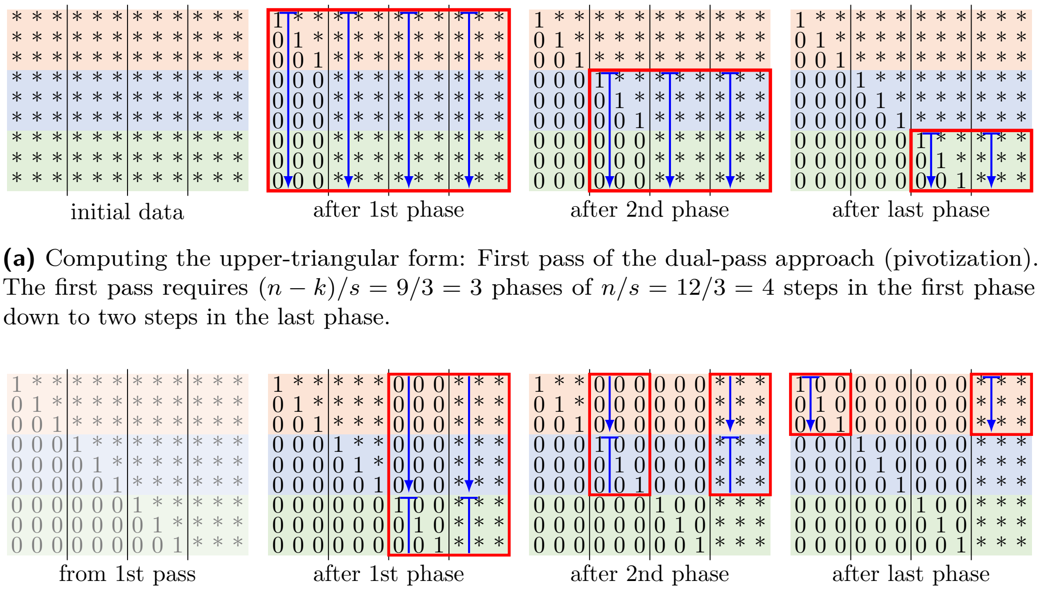
After the last step of the first phase, the matrix has a shape where the first s columns are in upper-triangular form. In the next phase, the same process is repeated starting at the next column block with row s. The operations are recorded in the first step into the operation memory and replayed in the following steps until all column blocks have been processed. These phases are repeated such that phase i starts at row $(i-1)s$ until the entire left square sub-matrix has been converted to upper-triangular form.

Figure 3a shows an illustration of the data memory for the overall process. The figure shows the initial data at the left and then the memory contents after each of the phases (after all steps of the phase have been completed). The parts of the data memory that have been processed in each phase are marked with a red box. Blue arrows indicate in which order row slices of the column block have been fed into the processor array.

Computing the systematic form. Once the matrix is in upper-triangular form, the systematic form can be computed using back substitution. We can use the same processor array as for computing the upper-triangle form: In the first phase, we start with the right-most pivoting column block. First, the pivoting row slices are loaded into the processor array. Then the following rows are echelonized: If the corresponding entry in a pivoting column is a 1, then the pivoting row is added to it — the pivot node issues an `add` operation to the following `processor_B` in its processor row. Otherwise, the row does not need to be modified — the pivot node issues a `pass` operation. Once all rows have been processed, the pivoting rows themselves need to be echelonized. Starting from the first pivoting row, the processor nodes simply evict the row data — the pivot node issues a `swap` operation. The pivoting row data is then processed by the subsequent processor rows as described above which echelonizes the pivot row. As before, the operations of the first step are recorded in the operation memory and replayed in the following steps to the remaining column blocks starting at column $(n-k)+1$.

In the second phase, the same process is repeated for the second-last pivoting column block and so on in the consecutive phases. Observe, that pivoting column blocks that already have been processed do not need to be touched again — since the corresponding entries already are all 0. Each phase needs to compute on s fewer rows until the last phase processed the final batch of s rows.

Figure 3b shows a visualization of the data memory for the process of computing the systematic form using back substitution. Again, the parts of the matrix that are processed



(a) Computing the upper-triangular form: First pass of the dual-pass approach (pivotization). The first pass requires $(n - k)/s = 9/3 = 3$ phases of $n/s = 12/3 = 4$ steps in the first phase down to two steps in the last phase.

(b) Computing the systematic form: Second pass of the dual-pass approach (back substitution). The second pass requires $(n - k)/s = 9/3 = 3$ phases of two steps each.

Figure 3: Visualization of the data memory for the dual-pass approach for an $(n - k) \times n = 9 \times 12$ matrix using a column-block size of $s = 3$. Asterisks denote random data. Corresponding sets of rows are marked in the same color. Red boxes show which matrix parts have been computed.

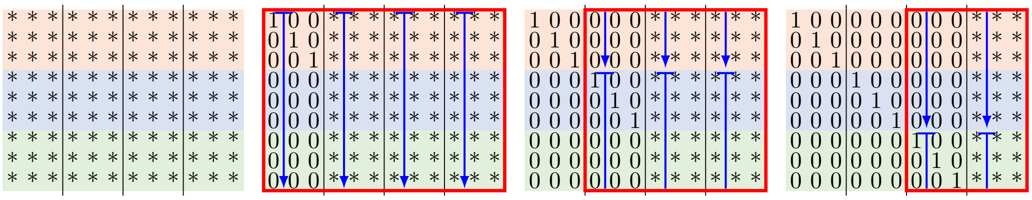
in each phase are marked with a red box and blue arrows indicate the reading order.

Single- and dual-pass approaches. The design in [SWM$^+$10] uses the *dual-pass approach* described above: In the first pass it computes the upper-triangular form of the matrix and in the second pass it uses back substitution to compute the systematic form. Since in each phase of both passes, another s rows are getting echelonized, this approach needs to work on fewer and fewer rows of the matrix in each phase as shown by the red boxes in Figure 3.

The design in [WSN16] is using a *single-pass approach*: At the end of each pivoting step, the stored data is not simply read out of the processor array, but starting from the top processor row, as during back substitution, the data is fed in normal operation to the following processor rows that then fully reduce the rows. The second phase starts from row s, finds pivoting rows, and reduces the second column block including a full reduction of the first s rows. The following phases continue equivalently as shown in Figure 4.

This single-pass approach can be implemented in hardware with simpler logic than the dual-pass approach, but it requires all phases to operate on all matrix rows as illustrated by the red boxes in Figure 4, since each phase computes on all rows. However, as [WSN16] points out, this approach can be more efficient depending on overheads, the dimensions of the input matrix, and the choice for s.

Matrices that are not systemizable. The previous paragraphs assume that the input matrix is systemizable. However, for Classic McEliece, this is not guaranteed. If the matrix is not systemizable, key generation needs to be restarted (if the f-parameter sets are not used). On average, Classic McEliece requires 3.4 key-generation attempts to successfully compute a public key [ABC$^+$20, Sect. 4.2]. The design from [WSN17] provides a failure signal to detect if pivoting failed: In the pivoting step of each phase, the processor nodes



initial data

after 1st phase

after 2nd phase

after last phase

Figure 4: Visualization of the data memory for the single-pass approach for an $(n-k)\times n = 9\times 12$ matrix using a column-block size of $s=3$. The single-pass approach requires $(n-k)/s = 9/3 = 3$ phases of $n/s = 12/3 = 4$ steps in the first phase down to two steps in the last phase. Asterisks denote random data. Corresponding sets of rows are marked in the same color. Red boxes show which matrix parts have been computed.

check if they indeed all found a pivot row during forward elimination. If not, the failure signal is raised and key generation is restarted with a new seed.

An advantage of the dual-pass approach is that it can detect failure already after the first pass without the need to also compute the second pass in case the matrix cannot be systemized. The single-pass approach however, needs to attempt to compute almost the entire systemization of the matrix before it can detect if the matrix indeed can be systemized. Thus, on average, the dual-pass approach can be faster and more efficient for public key generation in the Classic McEliece cryptosystem when the failure rate is taken into account. Therefore, a careful analysis is required for the choice between single- and dual-pass systemization.

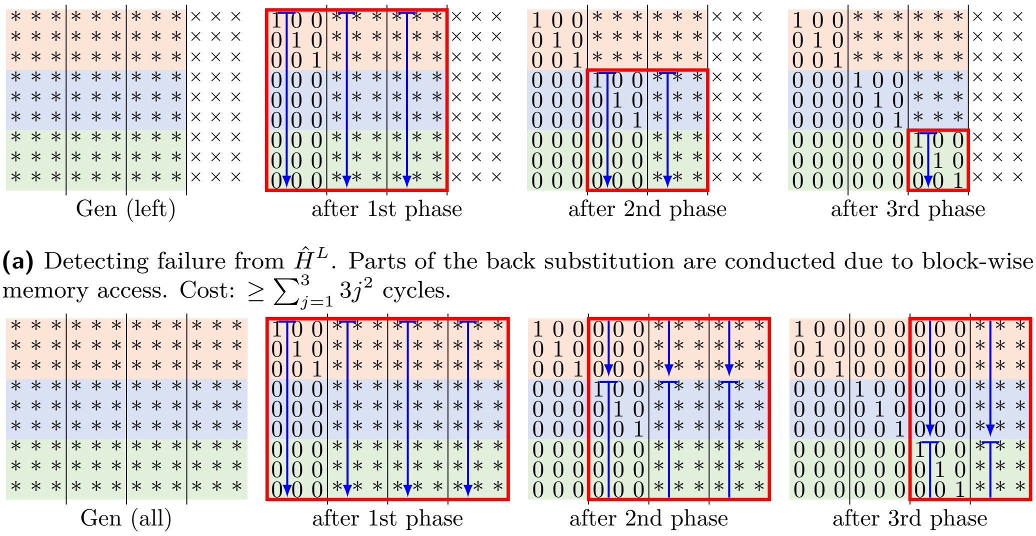
4 Optimization of Public Key Generation

To carry out public-key generation, the offical Classic McEliece hardware implementation described in [WSN17, WSN18] applies single-pass systemization to the whole $\hat{H}$ using the systemizer design from [WSN16]. As public-key generation fails with a high probability (for the non-f parameter sets), the process can be accelerated if the systemizer detects early that $\hat{H}$ is not systemizable and aborts immediately. We ended up with three algorithms supporting such early abort, which we call:

- hybrid early-abort systemizer (HEA),
- single-pass early-abort systemizer (SPEA), and
- dual-pass early-abort systemizer (DPEA).

The algorithms consider $\hat{H}$ as $(\hat{H}^L \mid \hat{H}^R)$, where $\hat{H}^L \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and $\hat{H}^R \in \mathbb{F}_2^{(n-k)\times k}$, and operate on $\hat{H}^L$ (instead of the whole $\hat{H}$) first to check if $\hat{H}$ is systemizable. If not, the algorithms return $\perp$ to indicate that $\hat{H}$ is not systemizable. Once it is confirmed that $\hat{H}$ is systemizable, operations are applied to $\hat{H}^R$ to obtain the public key $T = (\hat{H}^L)^{-1}\hat{H}^R$. The three algorithms make difference decisions on whether to detect if $\hat{H}$ is systemizable using the single-pass or the dual-pass approach and how to compute the final systematic form efficiently once it has been confirmed that that $\hat{H}$ is systemizable.

Below we introduce HEA in Section 4.1, SPEA in Section 4.2, and DPEA in Section 4.3. For each systemier we start from introducing the algorithm and showing an example, and then we introduce the corresponding hardware modules and how they work together to support operations in the systemizer. The sections are written in an "incremental" way: The contents in Section 4.1.2 to Section 4.1.5 apply to not only HEA but also SPEA and DPEA, if not mentioned otherwise. Similarly, the contents in Section 4.2.2 apply to not



(a) Detecting failure from $\hat{H}^L$. Parts of the back substitution are conducted due to block-wise memory access. Cost: $\geq \sum_{j=1}^{3} 3j^2$ cycles.

(b) Computation of the single-pass approach on the entire matrix. Cost: $\geq \sum_{j=1}^{3} 9(j+1)$ cycles.

Figure 5: Visualization of HEA for a 9×12 matrix using a column-block size of $s = 3$. Asterisks denote random data. Corresponding sets of rows are marked in the same color. Red boxes show which matrix parts have been computed. 'Gen (left)' denotes only $\hat{H}^L$ is generated while 'Gen (all)' denotes the same matrix $\hat{H}$ is (re-)generated.

only SPEA but also DPEA, if not mentioned otherwise. In the last subsection Section 4.4, we compare the three systemizers in terms of resource consumption and efficiency.

4.1 Hybrid Early-Abort Systemizer (HEA)

The pseudocode of HEA is shown in Algorithm 8 in Appendix A. HEA first checks if $\hat{H}$ is systemizable by applying the first half of a dual-pass systemization on $\hat{H}^L$ to reduce it to an upper-triangular matrix U. In other words, only forward elimination is carried out for $\hat{H}^L$. If any of the diagonal elements in the upper-triangular matrix is zero, which means that $\hat{H}$ is not systemizable, the algorithm returns $\perp$. Otherwise, a single-pass systemization is applied to $\hat{H}$ to obtain the public key T.

4.1.1 An Example and Lower Bound of Cycles

Figure 5 illustrates how our hardware implementation of HEA with $s = 3$ operates on a 9×12 matrix $\hat{H}$. Figure 5a illustrates generation of $\hat{H}^L$ and the forward elimination on it. As in Figure 3a, the number of processed rows decreases by s in each phase. However, as opposed to Figure 3a and Algorithm 8, parts of back substitution are conducted on the pivoting rows in $\hat{H}^L$. This does not have an impact on the speed of HEA but allows us to share logic with the following single-pass systemization and hence improves the resource requirements of the design. Figure 5b illustrates generation of $\hat{H}$ (including $\hat{H}^L$) and the single-pass systemization on it.

The first pass of the dual-pass approach for checking systemizability operates on $(n-k)/s$ column blocks and all $(n-k)$ rows in the first phase. In the second phase, it operates on $(n-k)/s - 1$ column blocks of $(n-k) - s$ rows and so on. In the last phase, it operates on only one column block and only s rows. Hence, the cost of checking systemizability in HEA is at least $\sum_{j=1}^{(n-k)/s} sj^2$ cycles.

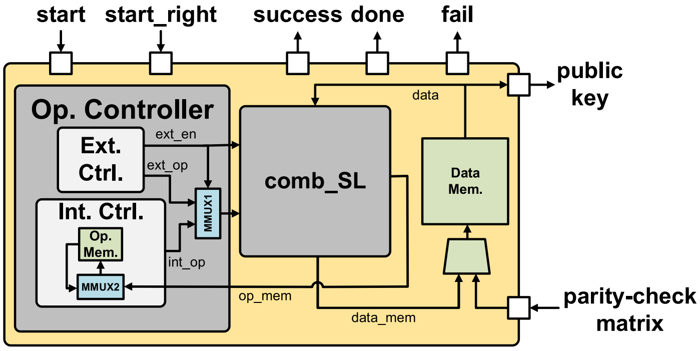


Figure 6: Hardware structure of the HEA systemizer design. Each wire in the diagram is of width s (the column-block width), and the processor array is of dimension $s \times s$.

Once systemizability has been verified, the entire matrix is systemized using the single-pass approach. This requires to operate on n/s column blocks of all $(n-k)$ rows in the first phase. In the second phase, it operates on $n/s-1$ column blocks of all $(n-k)$ rows and so on. In the last phase, it operates on the last column block in $\hat{H}^L$ and all k/s column blocks of $\hat{H}^R$ and all $(n-k)$ rows. Hence, the single-pass systemization on $\hat{H}$ takes at least $\sum_{j=1}^{(n-k)/s}(n-k)(j+k/s)$ cycles.

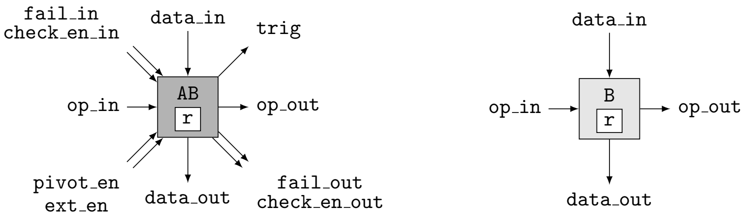
4.1.2 Hardware Architecture

Figure 6 shows an overview of the HEA systemizer architecture. To use the systemizer module, the left part $\hat{H}^L$ of the parity-check matrix $\hat{H}$ needs to be stored in the data memory (Data Mem.). After that, we instruct the systemizer to check if the matrix can be systemized by setting the input signal `start` to high. The $s \times s$ processor array (see Section 4.1.3) then reads $\hat{H}^L$ from the data memory and carries out all the row operations required by HEA to check if $\hat{H}$ can be systemized. If the systemizer detects that the input matrix $\hat{H}$ is not systemizable, it sets the `done` and the `fail` output signals to high. In this case, a new secret key and the corresponding $\hat{H}$ need to be generated and systemization needs to be started again. If the systemizer detects that the matrix can be systemized, it sets the output signal `success` to high. Now, we can initialize the entire $\hat{H}$ and then instruct the systemizer to finish the systemization of $\hat{H}$ by setting the input signal `start_right` to high. After systemization, the systemizer sets the output signal `done` to high again and the public key is available to be read out from the data memory.

There are two s-bit output wires `op_mem` and `data_mem` from the processor array. During a pivot step, the processor array uses `op_mem` to store operations into the operation memory (Op. Mem.), so that they can be replayed in the remaining steps of the same phase. The processor array uses `data_mem` to write reduced column blocks of $\hat{H}$ into the data memory. This overall design is very similar to [WSN16] and [WSN17]. However, we apply several improvements to those prior designs as explained in Sections 4.1.4 and 4.1.5. The internal controller (Int. Ctrl.) is in charge of replaying operations in the operation memory. The external controller (Ext. Ctrl) is in charge of loading data into `comb_SL` at the beginning of a step and reading data from `comb_SL` at the end of a step.

4.1.3 Our Systolic Architecture

The central module of our systemizer is the processor array that implements a systolic architecture in form of a systolic line respective systolic array. The systolic architecture in [WSN16] combines the systolic arrays SQR-SA and TRI-SA from [SWM+10] and hence



(a) Module `processor_AB`. **(b)** Module `processor_B`.

Figure 7: Details of the input and output ports of modules `processor_AB` and `processor_B`. The width of `op_in` and `op_out` is two bits.

calls its processor-array module `comb_SA`. To distinguish our design from [WSN16], we refer to ours as systolic line (architecture) and call the module `comb_SL`. The structure of our processor array `comb_SL` is essentially the same as Figure 1. Our design is similar to that of [WSN16] in respect to the following aspects:

- The processor array consists of two types of processors `processor_AB` that can compute pivots and `processor_B`. The input and output ports of our processors are shown in Figure 7, those of [WSN16] in Figure 2.

- To carry out a step, row slices of the corresponding column block are passed to the topmost processors of the processor array, while the eliminated rows are returned from the bottommost processors.

- In each pivot step, the operations are streamed out of the processor array via the `op_out` ports of the processors on the right side of the processor array into an operation memory. As we will discuss in the later subsections, for SPEA and DPEA this is not the case.

- Control logic (in our case the external and internal controller, see Figure 6) controls the behavior of the processor array by sending operations to the leftmost processors.

Note that the design of the two types of processors, as specified in Table 3 and Table 2, is somewhat different from that of [WSN16].

The output ports `data_out` of the processor elements in the bottom line of the processor array are concatenated together to the bus `data_mem` in Figure 6. Similarly, operation outputs are concatenated together to the bus `op_mem`. The internal and external operation buses `int_op` and `ext_op` in Figure 6 are multiplexed using `MMUX1`, which is a collection of s multiplexers, and fed into `comb_SL`. There, the input bus is split and the individual signals are attached to the input ports `op_in` of the processor elements on the left side of the processor array. The bus `data` from the data memory is split as well and the individual signals are attached to the input ports `data_in` of the processor elements at the top of the processor array.

Design of `processor_B`. Our `processor_B` is quite similar to that of [WSN16] (see Figure 2). As shown in Table 2, depending on the value of `op_in`, our `processor_B` is able to carry out three operations `pass`, `add`, and `swap` as in [WSN16].

Design of `processor_AB`. The truth table for `processor_AB` is shown in Table 3. We use `pivot_en` to configure `processor_AB` to operate in one of the following two modes (similar to the port `function_A` in [WSN16], see Figure 2):

Table 2: The truth table for `processor_B`. The symbol `r'` denotes the new register value in the next cycle.

inputs		state		outputs	
`op_in`	`data_in`	`r`	`r'`	`op_out`	`data_out`
`pass`	d	r	r	`pass`	d
`add`	d	r	r	`add`	r+d
`swap`	d	r	d	`swap`	r

Table 3: The truth table for `processor_AB` in this work.

input				state		output		
`ext_en`	`pivot_en`	`op_in`	`data_in`	`r`	`r'`	`op_out`	`data_out`	`trig`
1	x	`pass`	d	r	r	`pass`	d	0
1	x	`swap`	d	r	d	`swap`	r	0
0	1	x	0	r	r	`pass`	0	0
0	1	x	1	1	1	`add`	1	0
0	1	x	1	0	1	`swap`	0	1
0	0	`pass`	d	r	r	`pass`	d	0
0	0	`add`	d	r	r	`add`	d+r	0
0	0	`swap`	d	r	d	`swap`	r	0

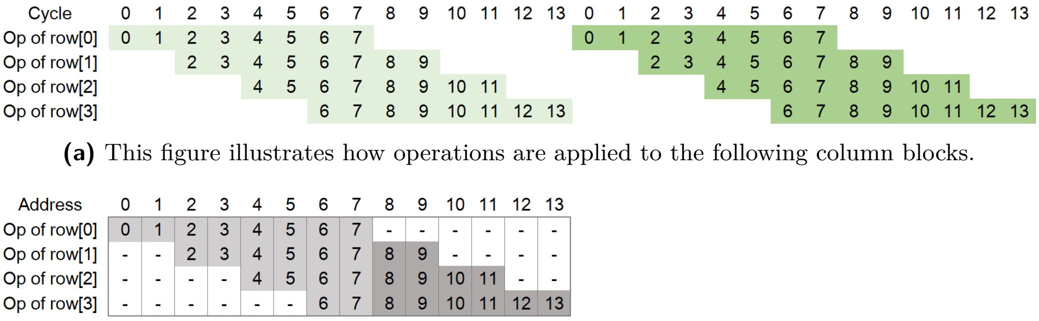
- Mode A (`pivot_en` $= 1$): In pivots steps, we set a `processor_AB` to mode A so that it acts as a pivoting processor ("`processor_A`") to generate commands for the following `processor_B` in the same row of the processor array.
- Mode B (`pivot_en` $= 0$): In non-pivot steps, we set a `processor_AB` to mode B so that it acts like a `processor_B`.

Using the signal `ext_en`, it is possible to pass operations from the control logic to the processor array using `MMUX1` (see Figure 6). This forces `processor_AB` to execute operations provided via `op_in` even if `processor_AB` is in mode A. In HEA, this is used at the beginning of a step to initialize the internal registers of the processors and at the end of a step to flush out pivoting rows from the processor array similar to the signals `start_in` and `finish_in` in [WSN16].

We use an output port `trig` to signal when a pivoting row has been found. To check whether $\hat{H}$ is systemizable, we have to check whether all the diagonal elements of $\hat{H}^L$ are 1's after reduction. We use the ports `fail_in`/`fail_out` in combination with `check_en_in`/`check_en_out` for this purpose. The design in [WSN16] does not provide logic to check if systemization was successful and [WSN17] uses control logic that accesses the contents of the internal register `r` of each `processor_AB` directly.

4.1.4 Reducing Time and Memory Demand by Overlapping Steps

Figure 8a shows an example of two systemizer steps for the design from [WSN16] operating on $n - k = 8$ rows of a column block of width $s = 4$. At the beginning of each step, each processor row in the processor array begins to process data two cycles earlier than the next row. Similarly, at the end of each step each row in the processor array completes all computation two cycles earlier than the next row. It is easy to see that in the example in Figure 8a each step takes $n - k + 2s - 2$ cycles to finish, even though there are only $n - k$ rows to process. The implementation of [WSN16] carries out the steps in each phase sequentially: Step $i + 1$ begins only after the step i has finished completely, which results in a significant overhead.



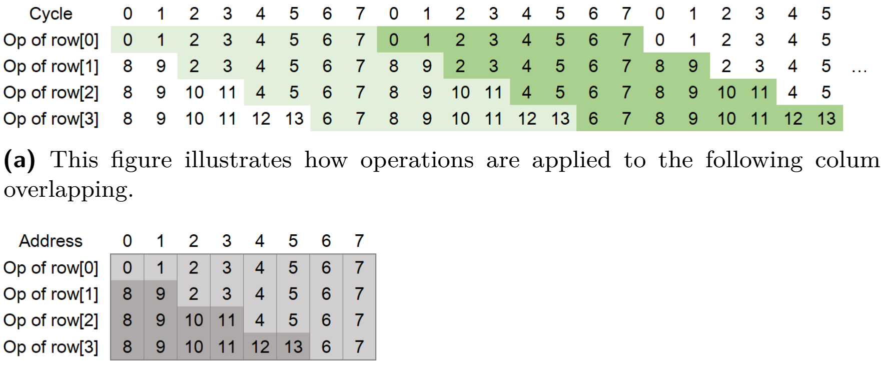
Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Op of row[0]	0	1	2	3	4	5	6	7							0	1	2	3	4	5	6	7						
Op of row[1]			2	3	4	5	6	7	8	9							2	3	4	5	6	7	8	9				
Op of row[2]					4	5	6	7	8	9	10	11							4	5	6	7	8	9	10	11		
Op of row[3]							6	7	8	9	10	11	12	13							6	7	8	9	10	11	12	13

(a) This figure illustrates how operations are applied to the following column blocks.

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Op of row[0]	0	1	2	3	4	5	6	7	-	-	-	-	-	-
Op of row[1]	-	-	2	3	4	5	6	7	8	9	-	-	-	-
Op of row[2]	-	-	-	-	4	5	6	7	8	9	10	11	-	-
Op of row[3]	-	-	-	-	-	-	6	7	8	9	10	11	12	13

(b) This figure indicates how operations in respective cycles are saved in the operation memory (each cell represents two bits in [WSN16]).

Figure 8: Details of the column-block elimination in [WSN16] for $(n - k, s) = (8, 4)$.



Cycle	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	
Op of row[0]	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	
Op of row[1]	8	9	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9	2	3	4	5	...
Op of row[2]	8	9	10	11	4	5	6	7	8	9	10	11	4	5	6	7	8	9	10	11	4	5	
Op of row[3]	8	9	10	11	12	13	6	7	8	9	10	11	12	13	6	7	8	9	10	11	12	13	

(a) This figure illustrates how operations are applied to the following column blocks with overlapping.

Address	0	1	2	3	4	5	6	7
Op of row[0]	0	1	2	3	4	5	6	7
Op of row[1]	8	9	2	3	4	5	6	7
Op of row[2]	8	9	10	11	4	5	6	7
Op of row[3]	8	9	10	11	12	13	6	7

(b) This figure indicates how operations in respective cycles are saved in a condensed way.

Figure 9: Details of the column-block eliminations in this work for $(n - k, s) = (8, 4)$.

This computation pattern also has impact on memory consumption. In a pivot step, the approach of [WSN16] simply records all operations generated by all `processor_AB`s during the step, including operations generated when a `processor_AB` is idle. The layout of the operations when stored in memory is thus as shown in the example in Figure 8b including an overhead marked with "-". Therefore, for the example in Figure 8 this approach requires an operation memory of size $2s(n - k + 2s - 2)$ to store the 2-bit operations, even though there are only $2s(n - k)$ meaningful bits to store.

To increase the utilization rate of our processor array and to reduce the memory requirements, we pipeline the execution of consecutive steps: We change the computation pattern such that each row of the processor array starts to process data of step $i + 1$ of a phase as soon as it has finished processing the data in step i for any $i \geq 2$, as depicted in Figure 9a. In this way, assuming that there are enough column blocks in a phase, the average cycle count for each step is reduced from $n - k + 2s - 2$ to about $n - k$. (There is still some cycle overhead at the beginning and end of the first step of a phase and at the end of the last step of a phase.) To support the new computation pattern, we need to store the operations in the way shown in Figure 9b, so that operations executed in the same cycle (for both step i and step $i + 1$) can be loaded from the same memory entry. With our new memory layout, the size of the operation memory is reduced from $2s(n - k + 2s - 2)$ bits to $2s(n - k)$ bits.

To achieve the memory layout, we first start out as [WSN16] in the beginning of the pivot step of each phase. At the end of a pivot step in cycle $n - k + i$, $i \in \{1, \ldots, 2s - 2\}$, we read out the operations executed in cycle i at the beginning of the pivoting step, combine these operations with the operations generated in the current cycle $n - k + i$, and store the result back to the operation memory. In other words, each operation in the written memory entry comes from either the original memory entry or the processor array.

Whether an operation that is written to operation memory comes from the processor array or from a previous entry in the operation memory is selected by `MMUX2` shown in Figure 6, which is a collection of s multiplexers similar to `MMUX1`. We note that using `MMUX2` increases the length of the critical path and the resource requirements due to additional logic. However, this penalty is a trade-off between performance and memory demand versus logic and path length.

4.1.5 Reducing Memory Demand for the Swap Operations

Observe that in Algorithm 8, there is at most one row swap for each pivot. This fact has been used in the software implementation of [CC21] to reduce memory demand. Below we discuss how we use the same technique as [CC21] for our hardware implementation.

Because of the aforementioned fact, in a pivot step, each of the s `processor_AB`'s, when in mode A, generates at most one `swap`. To make use of this fact, we choose the opcode 00_2 for `pass`, 01_2 for `add`, and 10_2 for `swap`. We follow the approach in the previous subsection but only store the least significant bit of the operations into the operation memory. In this way, we reduce the size of the operation memory from $2s(n - k)$ bits to $s(n - k)$ bits, but we need a way to keep track of the most significant bit of each issued operation, which simply indicates whether the operation is `swap` or not.

Our solution is to add dedicated logic to the internal controller as shown in Figure 10. For each row i of the processor array, there is a register `reg[i]` that records in which cycle of a pivot step the `swap` operation is generated. The registers start to count at the beginning of a step. Due to the systolic structure of the processor array, it takes $n-k+s-1$ cycles before all $n - k$ inputs have had the chance to reach all processor elements (e.g., in case the very last row is the pivoting row for the `processor_AB` in the last processor row). Therefore, each register `reg[i]` needs to be able to count up to $n - k + s - 1$ cycles and hence the width of the registers is $\lceil \log_2 (n - k + s - 1) \rceil$ bits.

When the ith `processor_AB` generates a `swap` operation, it raises the signal `trig` to indicate that the current value of the cycle counter needs be recorded in `reg[i]`. In the following steps of the same phase, a comparator for each row i compares the current cycle-counter value with the recorded cycle-counter value in register `reg[i]`. If they are the same, the ith bit in `perm_op[i]` in Figure 10 is set to 1. By pairwise combining `perm_op` with the s bits stored for `pass` and `add` in the operation memory, we reconstruct the $2s$ operation bits needed for the s rows of the processor array in each cycle.

4.2 Single-Pass Early-Abort Systemizer (SPEA)

The HEA approach has the disadvantage that the operations that are performed on $\hat{H}^L$ to check if the matrix is systemizable are “wasted” — even though we already computed the upper-triangular form during the systemizability check, we recompute the systemization on the entire matrix $\hat{H} = (\hat{H}^L|\hat{H}^R)$ including $\hat{H}^L$. Our second systemizer design SPEA has the goal to re-use the computations used for checking on $\hat{H}^L$ when finishing the systemization on $\hat{H}^R$.

The pseudocode of SPEA is shown in Algorithm 9 in Appendix A. SPEA first checks if $\hat{H}$ is systemizable by performing a single-pass systemization to reduce $\hat{H}^L$ to I_{n-k}. During the elimination, additions between rows are recorded in the space of $\hat{H}^L$ so that they can be replayed to $\hat{H}^R$, and swaps between rows are recorded in a list of $n - k$ row indices. If any of the diagonal elements is zero after the reduction, the algorithm returns $\perp$ to indicate that $\hat{H}$ is not systemizable. Otherwise, the operations that were applied to and recorded in $\hat{H}^L$ are replayed to $\hat{H}^R$ to obtain the public key T.

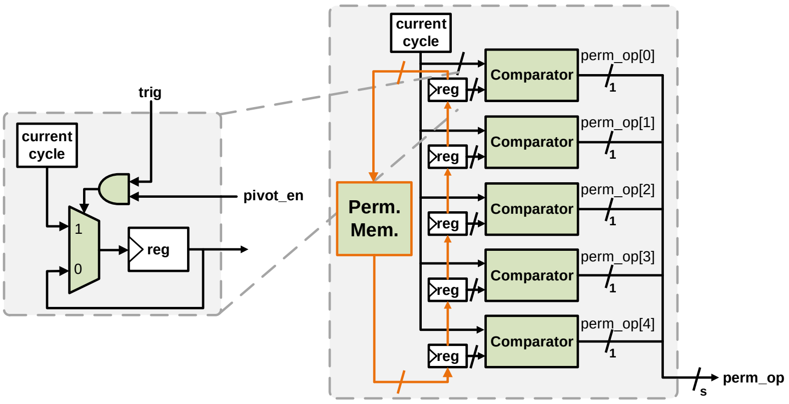


Figure 10: Logic to deal with operation `swap` ($s = 5$). (Orange color indicates components used only for SPEA and DPEA see Section 4.2.2).

4.2.1 An Example and Lower Bound of Cycles

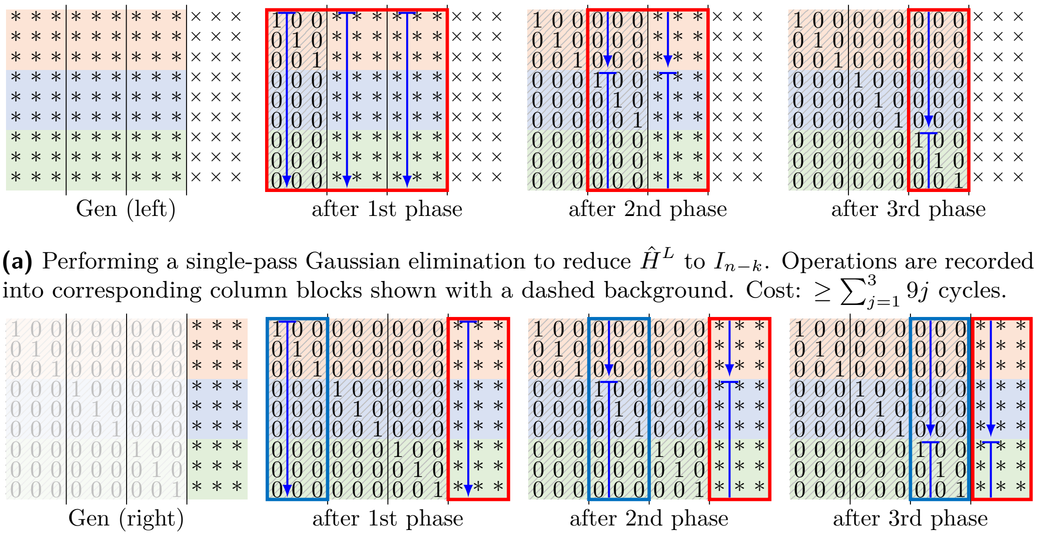
Figure 11 illustrates how our hardware implementation of SPEA with $s = 3$ operates on a 9×12 matrix $\hat{H}$. Figure 11a illustrates generation of $\hat{H}^L$ and how the algorithm operates on it. The column blocks of $\hat{H}^L$ are used to record operations carried out in pivot steps. Figure 11b illustrates generation of $\hat{H}^R$ and how operations recorded in blue boxes are replayed to it (in the red box).

SPEA uses a single-pass approach to detect systemizability. Here, the single-pass approach is working only on $\hat{H}^L$, i.e., the first $(n-k)$ columns of $\hat{H}$. Hence, the cost for this computation is at least $\sum_{j=1}^{(n-k)/s}(n-k)j$ cycles. When operating on $\hat{H}^R$, in each phase SPEA needs to read operations from all rows of one column block of $\hat{H}^L$ and then applies these operations to all rows of all k/s column blocks of $\hat{H}^R$. Hence, the cost for this computation is at least $\sum_{j=1}^{(n-k)/s}(n-k)(1+k/s)$ cycles.

4.2.2 Memory Management for Operations

SPEA stores the operations of all phases that are generated during the computation of $\hat{H}^L$ so that they can be replayed for the computation on $\hat{H}^R$. We thus add a permutation memory ("Perm. Mem.", marked in orange in Figure 10) to the design to store values of `reg[i]`'s generated in all pivot steps. Furthermore, we use the data memory to store the (least significant bit of) operations generated in all pivot steps. The values and operations are loaded later before we operate on $\hat{H}^R$. Below we explain how this is done in our implementation.

Storing/loading `reg[i]`'s. to/from the permutation memory. To simplify the logic for saving and loading the values in the `reg[i]`'s into the permutation memory, we connect the `reg[i]`'s as a shift register, as illustrated by the orange wiring in Figure 10. To store the values in `reg[i]`'s, we store the value of `reg[0]` into the permutation memory while shifting all other values from `reg[i]` to `reg[i-1]`, each time increasing the memory address by one. When loading, we load the permutation indices into `reg[s-1]` and shift all other values from `reg[i]` to `reg[i-1]`. In this way, we avoid a complex decoder/encoder for writing and reading the permutation memory. There are s registers `reg[i]` of width $\lceil \log_2(n-k+s-1) \rceil$ and $(n-k)/s$ phases. Thus, the permutation memory has a size of $(n-k)\lceil \log_2(n-k+s-1) \rceil$ bits.



(a) Performing a single-pass Gaussian elimination to reduce $\hat{H}^L$ to I_{n-k}. Operations are recorded into corresponding column blocks shown with a dashed background. Cost: $\geq \sum_{j=1}^{3} 9j$ cycles.

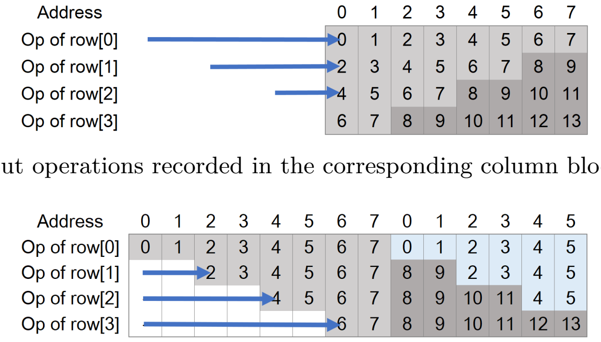
(b) Computation of replaying operations recorded in the $\hat{H}^L$. Blue boxes show which column blocks storing operations are read for restoring recorded operations. Cost: $\geq \sum_{j=1}^{3} 9 \cdot (1+1)$ cycles.

Figure 11: Visualization of SPEA for a 9×12 matrix using a column-block size of $s = 3$. Asterisks denote random data. Corresponding sets of rows are marked in the same color. Red boxes show which matrix parts have been computed. 'Gen (right)' denotes only $\hat{H}^R$ is generated.

Storing/loading operations to/from the data memory. To store (the least significant bit of) operations into the operation memory, we make use of `data_out` of the `processor_AB`'s. As the processors in the same column are connected using `data_out` and `data_in`, we can transfer the operations to the bottommost processors so that they can then be stored into the data memory. The control logic of the external controller sets `op_in` of each of the leftmost processors to `swap`, so that the least significant bit of the operations can be passed from `data_out` of the ith `processor_AB`, via `processor_B`'s (if there are any), to `data_out` of the corresponding processor in the last row and then stored into the data memory. As the `swap` operation is used, there is a two-cycle delay for a bit in row i to reach row $i+1$. Therefore, an operation issued from the ith `processor_AB` of the systolic line takes two more cycles to arrive the last row than an operation issued from the $(i+1)$th `processor_AB`, which results in the memory layout shown in Figure 12a.

In order to carry out one phase for $\hat{H}^R$, we first load operations from the data memory and store them into the operation memory. In order to achieve the memory layout as shown in Figure 8b, we send each memory entry through the systolic line once more, while setting `op_in` of the leftmost processors to `swap` while setting `ext_en` to high. In this way, we force all processors to perform `swap`. Due to the two-cycle delay in the `processor_B`, the operations arrive at the corresponding `processor_AB` in the same order as they were generated in the corresponding pivot step as illustrated in Figure 12b, and each `processor_AB` writes the operation it receives directly to the operation memory. We cannot use `op_out` for this since we need to issue the `swap` operation via `op_out` to the following `processor_B` of this processor row. Instead, we similar to above output the operations through the `data_out` port of all `processor_AB` and connect these ports to `op_mem` directly as shown in Figure 13. Thus both while copying operations and during pivoting, we write the least significant bit of the operation to `op_mem` via `data_out`.

Overall, saving and restoring the operation memory for the computation on $\hat{H}^R$ does



Address	0	1	2	3	4	5	6	7
Op of row[0]	0	1	2	3	4	5	6	7
Op of row[1]	2	3	4	5	6	7	8	9
Op of row[2]	4	5	6	7	8	9	10	11
Op of row[3]	6	7	8	9	10	11	12	13

(a) Output operations recorded in the corresponding column block of $\hat{H}^L$.

Address	0	1	2	3	4	5	6	7	0	1	2	3	4	5
Op of row[0]	0	1	2	3	4	5	6	7	0	1	2	3	4	5
Op of row[1]			2	3	4	5	6	7	8	9	2	3	4	5
Op of row[2]					4	5	6	7	8	9	10	11	4	5
Op of row[3]							6	7	8	9	10	11	12	13

(b) Output operations restored from the corresponding column block of $\hat{H}^L$.

Figure 12: Management for the operation and data memory for $n - k = 8$ and $s = 4$.

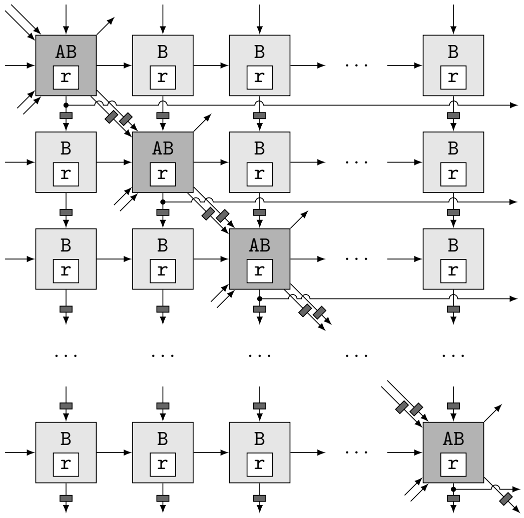


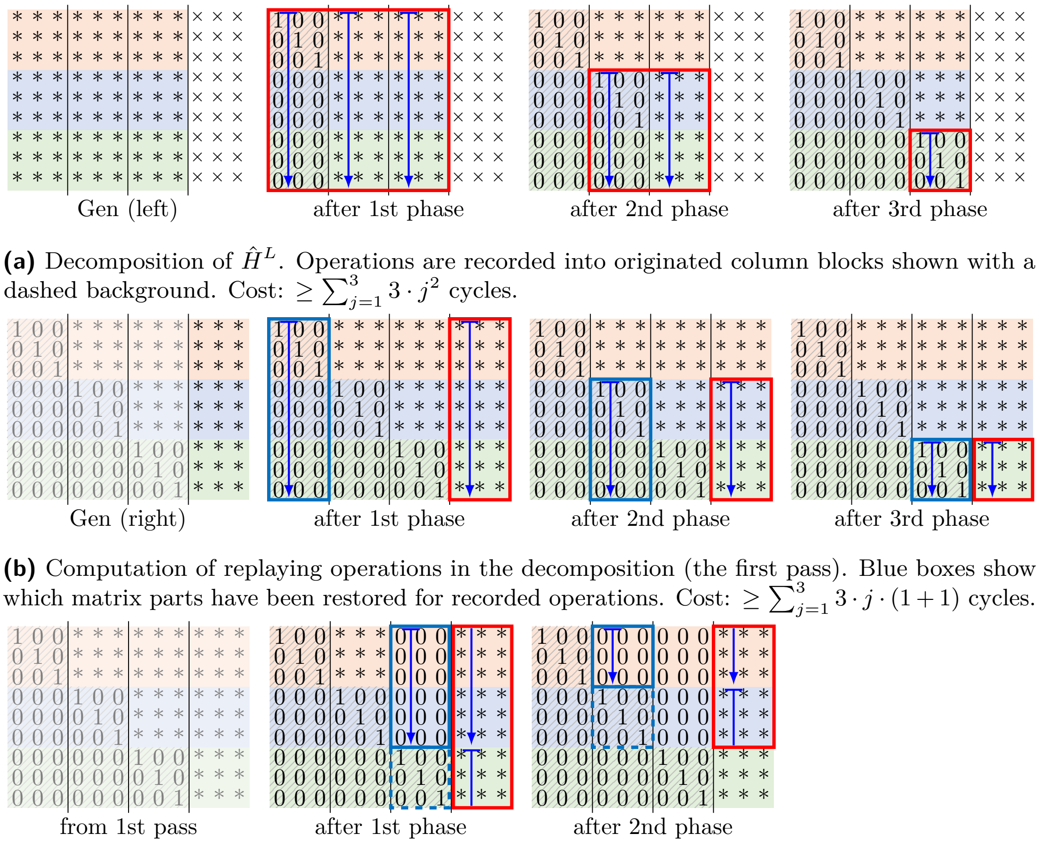
Figure 13: Layout of module `comb_SL` for SPEA.

not require additional memory and only has a small overhead in logic. However, it requires additional cycles in order to transfer the operations from the data memory ($\hat{H}^L$) back to the operation memory.

4.3 Dual-Pass Early-Abort Systemizer (DPEA)

Although SPEA has an advantage over HEA for the finial computation on $\hat{H}^R$, since it is able to record operations from checking $\hat{H}^L$ and to replay them later to $\hat{H}^R$, it also has a disadvantage compared to HEA: For checking if the matrix is systemizable SPEA uses the single-pass approach, while HEA is using the faster first pass of the dual-pass approach. Our third systemizer design DPEA attempts to combine the advantages of both approaches as described in this section.

The pseudocode of DPEA is shown in Algorithm 10 in Appendix A. DPEA first checks if $\hat{H}$ is systemizable by applying the first half of a dual-pass Gaussian elimination on $\hat{H}^L$ to reduce it to an upper-triangular matrix U. During the elimination, additions between rows are recorded in lower-triangular part of $\hat{H}^L$, and swaps between rows are recorded in a list of $n - k$ row indices. If any of the diagonal elements is zero after the reduction, the algorithm returns $\perp$ to indicate that systemization failed. If the diagonal elements



(a) Decomposition of $\hat{H}^L$. Operations are recorded into originated column blocks shown with a dashed background. Cost: $\geq \sum_{j=1}^{3} 3 \cdot j^2$ cycles.

(b) Computation of replaying operations in the decomposition (the first pass). Blue boxes show which matrix parts have been restored for recorded operations. Cost: $\geq \sum_{j=1}^{3} 3 \cdot j \cdot (1+1)$ cycles.

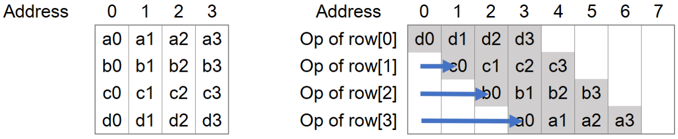
(c) Computation for carrying out the back substitution (the second pass). Dashed red boxes indicate data that is only read but not modified. Cost: $\geq \sum_{j=1}^{2} 3 \cdot (j + (j+1))$ cycles.

Figure 14: Visualization of DPEA for a 9×12 matrix using a column-block size of $s = 3$. Asterisks denote random data. Corresponding sets of rows are marked in the same color. Red boxes show which matrix parts have been computed.

are all 1's, the operations for the forward elimination are be applied to $\hat{H}^R$ to obtain a matrix M. Note that the entries of the upper triangular matrix U (excluding the diagonal elements) then define the remaining row additions that we need to perform on M in order to obtain the public key T (see [CC21, Sect. 3.1] for an explanation). Therefore, DPEA simply applies these operations to M to obtain the public key.

4.3.1 An Example and Lower Bound of Cycles

Figure 14 illustrates how our hardware implementation of DPEA with $s = 3$ operates on a 9×12 matrix $\hat{H}$. Figure 14a illustrates generation of $\hat{H}^L$ and how our implementation operates on it. To detect the failure earlier, each phase i computes on $(i-1)s$ fewer rows. (Again, parts of the back substitution are also conducted due to block-wise memory access.) During the pivot step of each phase, operations are recorded into the corresponding column block in $\hat{H}^L$ (dashed background in the figure). The part of $\hat{H}^L$ without dashed background then forms the matrix U. Figure 14b illustrates generation of $\hat{H}^R$ and from which matrix parts the recorded operations are restored (in blue boxes). By replaying the recorded operations in blue boxes to $\hat{H}^R$ in a sequential order, we can finish the first pass of the elimination on $\hat{H}^R$ to obtain the matrix M.



Address	0	1	2	3
	a0	a1	a2	a3
	b0	b1	b2	b3
	c0	c1	c2	c3
	d0	d1	d2	d3

Address	0	1	2	3	4	5	6	7
Op of row[0]	d0	d1	d2	d3				
Op of row[1]		c0	c1	c2	c3			
Op of row[2]			b0	b1	b2	b3		
Op of row[3]				a0	a1	a2	a3	

Figure 15: Output operations restored from U in $\hat{H}^L$ ($s = 4$). Each column represents a entry of memory while the row represents the bit position.

Figure 14c illustrates the remaining pass where we apply the remaining operations to M. In the first phase, operations in the third column block of U (in the blue box with solid edges) are applied to the corresponding three row blocks of M (in the red box): The operations in the blue box indicate how the rows in the third row block of M should be added to the rows in the first two row blocks. We do not need to load the operations in the dashed blue box, since the corresponding rows in $\hat{H}^R$ simply need to be loaded into the processor array using control logic. In the second phase, operations in the second column block of U (in the blue box with solid edges) are applied to the corresponding two row blocks of M (in the red box): The operations in the blue box indicate how the rows in the second row block of M should be added to the rows in the first row block. As there are no operations in the first column block of U, the algorithm finishes at this moment. Note that unlike the first pass, the second pass involves only two phases.

DPEA first applies the first pass of the dual-pass approach to obtain an upper-triangular matrix. Like in HEA, this requires at least $\sum_{j=1}^{(n-k)/s} sj^2$ cycles. To finish systemization, in each phase DPEA first reads operations from one column block of $\hat{H}^L$ and replays them to all k/s column block of $\hat{H}^R$. In the first phase, all $(n-k)$ rows are involved, in the next phase $(n-k)-s$ rows, and finally in the last phase only s rows. Hence, this computation requires at least $\sum_{j=1}^{(n-k)/s} sj(1+k/s)$ cycles. For the final back-substitution, in each phase DPEA needs to read s rows of the k/s column blocks of $\hat{H}^R$ for eliminating the other rows. In the first phase it also needs to load operations for $(n-k)-s$ rows from one column block in $\hat{H}^L$ and to apply these operations to $(n-k)-s$ rows of all k/s column blocks of $\hat{H}^R$. In the second phase $(n-k)-2s$ rows are involved and in the last phase only s rows. Hence, this computation requires at least $\sum_{j=1}^{(n-k)/s-1} sj + s(j+1)k/s$ cycles.

4.3.2 Back Substitution in DPEA

In DPEA, the elimination on $\hat{H}^R$ is processed in two passes. In the first pass, we replay the operations that were applied to $\hat{H}^L$ for decomposing the matrix. In the second pass, we apply the remaining operations for the back substitution. However, in the second pass of the dual-pass approach, there is no need for permutation since the pivot rows have already been swapped in the first pass. Therefore, after we streamed these pivot rows into the systolic line, the only operations that need to be performed are `pass` and `add`. Since these two kinds of operations are processed within one cycle in each processor row, we delay each operation in the same memory entry accordingly by one cycle using the systolic line when transferring these remaining operations in $\hat{H}^L$ to the operation memory. As shown in Figure 15, by setting the operation of each row in the systolic line to `pass`, the operations for the ith row of the processor array (`Op of row[i]`) are delayed by i cycles as read out from the `data_out` port of the ith `processor_AB`.

We read out the data from $\hat{H}^R$ starting from the s pivot rows. They are used to pivot the following rows according to operations replayed from operation memory. After restoring these operations to the operation memory phase by phase, the remaining operations for back substitution can be applied to $\hat{H}^R$ iteratively in the steps of each phase.

4.4 Performance Evaluation

In this section, we first evaluate the performance of our stand-alone systemizer modules based on the smallest Classic McEliece parameter set `mceliece348864` compared to the prior art [WSN16] and [WSN18] in Section 4.4.1. The difference between the design in [WSN16] and [WSN18] is that [WSN16] only works for matrices over $\mathbb{F}_2$. However, since matrix systemization is also required over $\mathbb{F}_{2^m}$ for private key generation, [WSN17] (which is used in [WSN18]) extended the design such that it can work over any finite field (once modules for addition, multiplication, and inversion in that field are provided). The choice of the field is a parameter that is provided at compile time. The additional logic required for general field arithmetic introduces a significant overhead also when the design is synthesized for $\mathbb{F}_2$.

Since the performance of public key generation does not only depend on the systemizer design but also on the cost for generating the initial matrix entries before systemization, we then analyse the time required for public key generation using our new designs in Section 4.4.2. We then briefly discuss the performance our designs for the remaining, larger parameter sets of Classic McEliece in Section 4.4.3. Finally in Section 4.4.4, we give a brief recommendation on which systemizer design to use under which circumstances.

4.4.1 Systemizer

Table 4 compares the synthesis results for our three $\mathbb{F}_2$ systemizers with that of [WSN16] and [WSN18] on a Xilinx Artix 7 FPGA (xc7a200t) for the parameter set `mceliece348864`. The performance of the designs can be tuned independently of the security parameters by adjusting the width s of the processor array to achieve different levels of parallelism at the cost of computing resources.

The left part of Table 4 shows the performance of the stand-alone systemizer variants. The column “Check” shows the cycles required for checking if the matrix can be systemized and the column “Finish” the cycles required to finish the systemization of the entire matrix after a successful check. (For [WSN16] and [WSN18], there is no separation between checking and finishing, so we list the cost of the entire systemization under “Check”.) We obtained the exact cycle counts from simulated runs of the designs (using Verilator[3]) on a systemizable matrix. The column “Average” shows the average cycle count required to systemize a parity-check matrix successfully including an average of 3.4 failure cases (in which case key generation needs to be repeated, see [ABC$^+$20, Section 4.2] on the failure rate). The average is computed as column “Check” times 3.4 plus column “Finish”. The column “Time” is computed from the average cycle count divided by $F_{\max}$, the column “Time×Area” by multiplying “Time” and “Area”. The time required for generating the private key is not included in this table.

Cycles. HEA performs a complete single-pass systemization on the entire matrix during “Finish”, which is conceptually similar to the single-pass systemization of [WSN16] and [WSN18] listed in their “Check” column — but our HEA design reduces their overheads in time (and memory, see below). This can be seen by comparing the entires of “Check” of [WSN16] and [WSN18] with the entries “Finish” of HEA. Since our approaches perform a complete systemization of the entire matrix only if the check on $\hat{H}^L$ was successful, the average runtime (column “Systemizer — Average”) is significantly lower than that of [WSN16] and [WSN18]. The “Finish” time of SPEA is shorter than that of HEA since SPEA does not need to operate on $\hat{H}^L$ but only replays operations to $\hat{H}^R$, but since it has a larger “Check” time, on average, HEA is faster than SPEA. DPEA has the same “Check” time as HEA and a “Finish” time only slightly longer than SPEA and therefore has the best average speed for smaller values of s. However, the advantages of DPEA over

[3] https://www.veripool.org/verilator/

Table 4: Performance of the $\mathbb{F}_2$ systemizer for the `mceliece348864` parameter set with a 768×3488 matrix $\hat{H}$. All cycle counts, "Time", and "Time×Area" are rounded to four significant figures. Cycles are given in kilocycles (kcyc.). Resources are for a Xilinx Artix 7 FPGA (xc7a200t).

		Systemizer									**Public Key Generation**		
		Cycles					**Resources**				**Cycles**		
Method	**s**	**Check** (kcyc.)	**Finish** (kcyc.)	**Average** (kcyc.)	$\mathbf{F}_{\max}$ (MHz)	**Avg. Time** (ms)	**Area** ($\mathrm{LUT^L}$)	**Memory** (FF)	(BR)	**Time×Area**	**1st Gen.** (kcyc.)	**2nd Gen.** (kcyc.)	**Average** (kcyc.)
[WSN16]	16	7,525	–	25,580	188	136.1	877	843	97	119.4×10^3	201.4	–	26,270
[WSN16]	32	1,961	–	6,667	191	34.91	1,757	2,528	98	61.33×10^3	102.5	–	7,016
[WSN16]	64	535.8	–	1,822	220	8.281	5,206	8,988	132	43.11×10^3	53.46	–	2,004
[WSN16]	128	157.6	–	535.9	215	2.492	18,080	34,223	142.5	45.06×10^3	29.01	–	634.5
[WSN18]	16	7,534	–	25,620	166	154.3	1,107	1,056	97.5	170.8×10^3	201.4	–	26,300
[WSN18]	32	1,963	–	6,675	172	38.81	2,381	2,825	99	92.41×10^3	102.5	–	7,024
[WSN18]	64	536.4	–	1,824	171	10.67	7,435	9,537	133.5	79.30×10^3	53.46	–	2,006
[WSN18]	128	157.8	–	536.4	175	3.065	26,393	35,267	149.5	80.89×10^3	29.01	–	635.0
HEA	16	611.8	7,173	9,253	150	61.69	1,139	1,001	96.5	70.26×10^3	44.35	201.4	9,606
HEA	32	160.0	1,800	2,344	151	15.52	2,154	2,864	97	33.44×10^3	22.56	102.5	2,523
HEA	64	44.63	459.4	611.1	155	3.942	5,967	9,678	130	23.52×10^3	11.66	53.46	704.2
HEA	128	14.51	120.6	169.9	181	0.9389	19,538	35,605	132	18.34×10^3	6.216	29.01	220.1
SPEA	16	906.5	6,307	9,389	153	61.37	1,300	1,012	97	79.78×10^3	44.35	157.1	9,697
SPEA	32	233.6	1,588	2,383	140	17.02	2,523	2,876	97.5	42.94×10^3	22.56	79.90	2,539
SPEA	64	62.93	408.7	622.6	160	3.891	6,538	9,732	130.5	25.44×10^3	11.66	41.80	704.1
SPEA	128	18.99	109.1	173.7	173	1.004	20,969	35,709	132.5	21.05×10^3	6.216	22.79	217.6
DPEA	16	611.8	6,438	8,518	140	60.84	1,527	1,065	97	92.90×10^3	44.35	157.1	8,825
DPEA	32	160.0	1,653	2,197	139	15.81	2,600	2,964	97.5	41.1×10^3	22.56	79.90	2,354
DPEA	64	44.63	441.1	592.9	150	3.952	6,752	9,846	130.5	26.69×10^3	11.66	41.80	674.3
DPEA	128	14.51	125.1	174.5	152	1.148	20,545	35,926	132.5	23.58×10^3	6.216	22.79	218.4

$\mathrm{LUT^L}$ = Lut as logic, FF = flip-flop, BR = BRAM

HEA and SPEA diminish when performing a complete systemization for growing s since DPEA needs to read matrix rows repeatedly during back-substitution. Hence, for $s = 128$ HEA and SPEA outperform DPEA in the average cycle count.

***$F_{\max}$*.** The maximum frequency of all our designs is significantly lower than that of [WSN16] and also slightly lower than that of [WSN18], but our cycle count and resource usage improvements compensate well for that. The maximum frequency of a design does not only depend on the design itself but also on the placement and routing heuristics of the FPGA tool chain. Therefore, it is not easy to compare design frequencies directly. The average $F_{\max}$ of each of our systemizers over the four listed processor-array sizes s is 159.25 MHz for HEA, 156.50 MHz for SPEA, and 145.25 MHz for DPEA. Thus, $F_{\max}$ appears to be quite similar for HEA and SPEA (with a slight advantage for HEA), but the additional logic for back substitution seems to take a slight toll on DPEA.

Average time. We compute the average time ("Avg. Time") from the average cycle count and $F_{\max}$. Our systemizer approaches are more than 2.2 times faster than [WSN16, WSN18] for small s and more than 2.6 times faster for large s despite our lower $F_{\max}$ due to our significant savings in the average cycle count. As seen before, DPEA has some performance advantages over HEA and SPEA for small values of s, while for $s \geq 64$ HEA and SPEA take over due to the diminishing differences in cycle counts and the differences in $F_{\max}$.

Resources. We show the resource consumption in terms of area (in LUTs), registers (flip-flops — "FF"), and block RAM ("BR"). Over all, [WSN16] has the smallest resource requirements due to its simpler design. For small s, the area consumption of [WSN18] lies in between HEA/SPEA and DPEA. However, for growing s, the area consumption of [WSN18] grows much faster than that of our designs due to additional logic that [WSN18] requires to make their systemizer compatible with different finite fields (while our design is restricted to and optimized for $\mathbb{F}_2$). For $s = 128$, our designs require 40%-45% fewer resources than that of [WSN18]. Comparing our three approaches, HEA is the most area efficient one for any value of s.

In regard to the number of required registers, all five designs are quite similar with a small advantage to [WSN16] and the highest register count for DPEA. Since our operation memory is only half the size of that from [WSN16] and [WSN18], we have significant savings in block RAM for larger s over these designs, while the block RAM consumption of our three designs is quite similar.

Time-area product. For the time-area product ("Time×Area") we can see that our designs are 1.7 times more efficient than [WSN16, WSN18] for small s and 2.4 times more efficient for large s since we improve over their designs in terms of cycle count and resource consumption. Comparing our three designs, HEA has the lowest time-area product for all values of s followed by SPEA and then DPEA, since the higher cycle count performance of DPEA comes at a premium in resources and a penalty in maximum frequency.

4.4.2 Public Key Generation

On the right side, Table 4 also shows the cycle counts that are required for the complete public key generation including the cost for generating the initial values in $\hat{H}$ before systemization is performed.

Column "1st Gen." shows the time needed to generate the initial matrix entires before checking if the matrix can be systemized and "2nd Gen." the cycles for generating matrix entries before finishing the systemization. The column "Average" under "Public Key Gen" is the average time needed to compute a complete systemization including failure cases

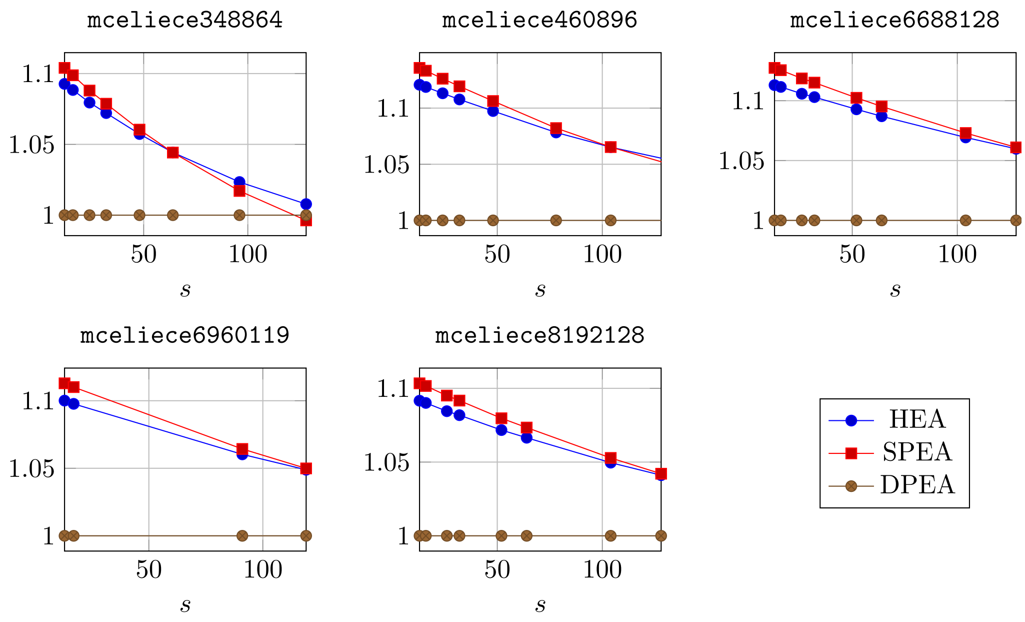


Figure 16: Speed (in terms of cycle count) of public-key generation using our three systemizers HEA, SPEA, and DPEA relative to DPEA for different s for the parameter sets `mceliece348864`, `mceliece460896`, `mceliece6688128`, `mceliece6960119`, and `mceliece8192128`. Compared to DPEA, [WSN16] and [WSN18] (omitted in the graphs) are about three times slower for all s and all parameter sets.

and the time for generating initial matrix entries, i.e., column "1st Gen." times 3.4 plus column "2nd Gen." plus the average time of the systemizer.

Initializing $\hat{H}$. Comparing the time required to generate matrix entries ("1st Gen." and "2nd Gen.") with that required for the following computation ("Check" and "Finish" for the systemizer), we can see that computation dominates the time for small s — but that generation time becomes more significant for larger s, taking up to 30% of the time of "Check" and up to 19% of the time of "Finish" at $s = 128$.

Average cycles. Looking at the average cycle count of public key generation including failure cases and matrix initialization ("Public Key Generation — Average"), we can see that all our approaches HEA, SPEA, and DPEA outperform the designs from [WSN16] and [WSN18], since their designs need to operate on average 3.4 times on the entire matrix, while our approaches abort early in case the matrix cannot be systemized. Since SPEA has a higher workload in "Check", it starts out with a lower performance than HEA for small s but since SPEA does not need to re-generate $\hat{H}^L$ before "Finish", it cuts even and excels over HEA for $s \geq 64$. SPEA also cuts even with DPEA for $s = 128$.

4.4.3 Larger Parameter Sets

Since the main resource consumption of the systemizer designs is due to the size s of the processor array, larger Classic McEliece parameter sets have mainly an impact on the memory resources (due to their larger matrix sizes) but only a small impact on the area for a fixed s, since mainly logic for address calculation is affected by larger parameter sets.

Figure 16 shows the speed of public key generation using all of our systemizer designs relative to DPEA obtained from simulating the designs for all Classic McEliece parameter sets. As described above, for `mceliece348864` DPEA starts out to be about 10% faster than HEA and SPEA for small s. At the beginning, SPEA is the slowest, but it catches up with HEA at $s = 64$. At $s = 128$, HEA and SPEA both are about as fast as DPEA. A similar trend is visible for the larger parameter sets shown in Figure 16. However, the turn-even point between HEA and SPEA is later for larger s. Cutting even with DPEA requires even larger values of s.

4.4.4 Recommendation

We recommend to use the HEA systemizer for area-restricted applications that cannot afford the additional resources of SPEA and DPEA, and the DPEA systemizer for high-performance applications. However, for large values of s, the benefits of DPEA diminish and SPEA becomes the fastest option. Since performance and efficiency depend on s, eventually the most suitable design needs to be chosen based on the specific needs of the application regarding performance requirements and available resources.

5 Encapsulation, Decapsulation, and Key Generation

Classic McEliece KEM consists of three primitives: Key Generation (SeededKeyGen), Encapsulation (Encap), and Decapsulation (Decap). The algorithms for each primitive are shown in Algorithm 1, Algorithm 3, and Algorithm 6 respectively. In this work, using the Classic McEliece PKE code from [WSN18], we design and implement novel hardware designs for all three primitives of Classic McEliece KEM. In the following sub-sections we discuss the hardware design for each primitive on a high level by briefly elaborating on the building blocks involved in their construction. The main building blocks for each primitive are as follows:

- Encap: SHAKE256, FixedWeight, and Encode;
- Decap: SHAKE256, Decode, and FieldOrdering;
- SeededKeyGen: SHAKE256, KeyGen, and $\mathbb{F}_2$ systemizer.

In our hardware design, we re-use the hardware modules implementing FieldOrdering and Decode from [WSN18]. Besides that, we tailor and improve the hardware module implementing SHAKE256 from [WTJ$^+$20] to cater our needs. We also use the improved $\mathbb{F}_2$ systemizer designs discussed in Section 4 to optimize the `KeyGen` hardware module from [WSN18] that we use in the implementation of SeededKeyGen. We design the remaining hardware modules for FixedWeight, Encode, Encap, Decap and SeededKeyGen from scratch. We make our hardware modules parameterizable such that performance parameters can be set based on the targeted time-area trade off.

In the following sections we give a high-level overview of the implementation of our modules. For each of the building blocks and the hardware designs of Encap, Decap, and SeededKeyGen we provide time and area comparison for exemplary performance parameters and a comparison with related work wherever possible.

5.1 SHAKE256

Classic McEliece uses SHAKE256 for several purposes, e.g., for pseudo-random seed expansion in key generation and for hashing in encapsulation in decapsulation. We are also using SHAKE256 as pseudorandom number generator (PRNG) in encapsulation.

We use the `keccak` module from [WTJ$^+$20] to perform SHAKE256 operations in our Classic McEliece design. This module was originally designed as a complete keccak module that can perform all configurations of SHAKE and cSHAKE. We tailor the existing `keccak`

Table 5: Comparison of the time and area for our SHAKE256 module targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

	Resources						
`parallel_slices`	**Area**	**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
	($\mathrm{LUT^L}$)	($\mathrm{LUT^M}$)	(FF)	(cyc.)	(MHz)	(us)	
1	739	25	482	5,010	150	33.40	24.68×10^3
2	878	50	455	2,306	146	15.79	13.86×10^3
4	920	100	360	1,086	147	7.39	6.799×10^3
8	1,169	200	270	542	148	3.66	4.279×10^3
16	1,817	400	226	270	150	1.80	3.271×10^3

LUT^L = LUT as logic, LUT^M = LUT as memory, FF = flip-flop

hardware module as per the requirement of our hardware design. The modifications we perform are as follows:

- Since our design only requires SHAKE256, we removed all surplus logic and further optimized the design for a more efficient area usage.
- We note that the `keccak` hardware design in [WTJ+20] is functionally designed only for 32-bit input data blocks; in our work, we extend its capability to process byte-sized blocks.
- We added a forced exit signal (triggering this signal brings the control back to the loading state and sets all the counters to their initial state) to the control logic of the `SHAKE256` module to support the parallel processing of seed expansion (described in the `FixedWeight` module in Section 5.3.1) and δ expansion (described in the seeded key-generation module in Section 5.2).

The original design presented in [WTJ+20] provides a performance parameter to control time-area trade-offs using parallelization. In our optimized design, we use a similar parameter called `parallel_slices` that provides five different time-area trade-offs as shown in Table 5. The `SHAKE256` design has a 32-bit input interface (for all the variants controlled by `parallel_slices`) that works on a simple valid-ready protocol.

The results targeting a Xilinx Artix 7 `xc7a200t` FPGA for all the variants are shown in Table 5. The clock cycles shown in Table 5 include the cycles required for processing one block of input (where the block size is 1088 bits) and generating a maximum of 1088 bits output. Currently, the design is limited to a maximum `parallel_slices` of 16 due to the structure of the round function of SHAKE256. In all our designs we use `parallel_slices` = 16 as that provides the best time area product.

5.2 Seeded Key Generation

Our hardware design for seeded key generation (described in Algorithm 1) is shown in Figure 17. From Algorithm 1, the seeded key-generation operation can be broken down in to four main components:

1. Expanding δ using a PRNG.
2. Generating permutation using the FieldOrdering.
3. Generating an Irreducible polynomial.
4. Matrix generation using MatGen.

To perform components two, three and four, we are using the key-generation module `KeyGen` from [WSN18]. However, we replace corresponding components from [WSN18] with our optimizations for public key generation and our optimized systemizer modules.

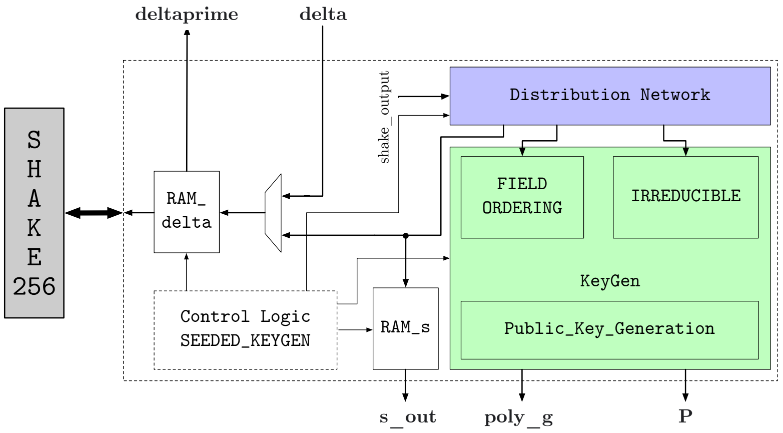


Figure 17: Hardware design of `SeededKeyGen` module interfaced with `SHAKE256` module.

The [WSN18] implementation does not include the first component, i.e., the expansion of δ using a PRNG to the inputs for private key generation (s, α, β, and δ'). Therefore, we extend the existing key generator by adding a wrapper around the `KeyGen` module. The wrapper consists of a distribution network that stores the secret seed s in a single ported RAM (`RAM_s`), distributes the α and β values to the `FieldOrdering` and `Irreducible` modules inside key generation respectively, and stores the δ' value (`deltaprime` in Figure 17) in a single ported RAM (`RAM_delta` in Figure 17). The wrapper also provides an interface for the connection to the `SHAKE256` module described in Section 5.1.

It is possible that the operations Irreducible, FieldOrdering, or MatGen in Algorithm 1 may fail, in which case the key-generation operation needs to be rerun using δ' as new δ. We optimize our design by expanding the δ' values in advance for a potential next iteration of key generation in case of a failure. The process of reseeding and expanding δ' works in parallel with the `KeyGen` module, which hides the time overhead required for expanding δ' in the next attempt of key generation.

The default secret key format in the specification includes 5 components $(\delta, c, g, \alpha, s)$, where α is stored as control bits of a Beněs network. Our module for key generation does compute α (as a list of $\mathbb{F}_{2^m}$ elements) and s, but it does not use the control bits and hence does not generate them to save time and area. In Section 5.4 we will explain that our decapsulation module simply takes (δ, c, g) as input. This is not the default secret key format, but it is explicitly mentioned in the specification as a choice to compress secret keys. We note that using (δ, c, g) reduces the key size by a very large factor.

Table 6 shows the time and area results for our `SeededKeyGen` hardware module. Reported clock cycles are the average cycles for a successful key generation including unsuccessful attempts computed similar as in Table 4 and using $s = t$ (here parameter s is the size of the processor array for Gaussian systemization and t as in Table 1).

The area estimates shown in the Table 6 do not include the area of the `SHAKE256` module, since the `SHAKE256` module is common in the SeededKeyGen, Encap, and Decap modules. Hence, we provide the flexibility of either choosing a common `SHAKE256` module for all operations in an area optimized target or choosing multiple `SHAKE256` modules for parallel processing in a performance optimized target. The first two (comparatively smaller) parameter sets, `mceliece348864` and `mceliece460896` were able to fit on the Xilinx Artix 7 `xc7a200t` FPGA. However, for the other three parameter sets the Block RAM requirement for storing the public key is higher than the memory capacity of target FPGA. Consequently, we use the Xilinx Zynq UltraScale+ ZCU216 evaluation platform `xczu49dr` (which provides more Block RAM resources) as the target.

A noticeable resource difference can be seen between Xilinx Artix 7 and Xilinx Zynq

Table 6: Comparison of the time and area for our `SeededKeyGen` module targeting Xilinx Artix 7 (`xc7a200t`) and Xilinx Zynq UltraScale+ (`xczu49dr`) FPGAs.

	Resources								
Param. Set	**Area**	**Memory**				**ACC**	**F$_{\text{max}}$**	**Time**	**T×A**
	(LUT$^{\text{L}}$)	(LUT$^{\text{M}}$)	(FF)	(BR)	(DSP)	(Mcyc.)	(MHz.)	(ms)	
				Xilinx Artix 7 (`xc7a200t`)					
`mceliece348864`	25,532	290	37,185	165.0	4	1.0	142	6.8	0.17
`mceliece460896`	44,644	515	66,869	271.5	4	1.7	147	11.8	0.53
				Xilinx Zynq UltraScale+ (`xczu49dr`)					
`mceliece348864`	25,119	344	37,245	112.5	4	1.0	186	5.2	0.13
`mceliece460896`	44,631	577	66,894	234.5	4	1.7	178	9.7	0.43
`mceliece6688128`	58,881	408	89,174	365.0	4	2.8	164	17.4	1.02
`mceliece6960119`	55,489	579	85,662	369.0	4	2.7	155	17.2	0.95
`mceliece8192128`	59,127	407	89,200	425.0	4	3.1	158	19.3	1.14

LUT$^{\text{L}}$ = LUT as logic, LUT$^{\text{M}}$ = LUT as memory, FF = flip-flop, BR = BRAM, ACC = average clock cycles, T×A = Time×Area

UltraScale+ FPGAs in terms of Block RAM utilization. This is caused by the synthesis tool (Xilinx Vivado), which synthesises some part of memory using LUTs as memory (distributed RAM) instead of Block RAM on the Zynq UltraScale+ FPGA.

The reported frequency in Table 6 is the maximum clock frequency for the `SeededKeyGen` module standalone. The frequency value is reduced after interfacing it with the `SHAKE256` module since the critical path of the design lies in the `SHAKE256` module. We also observe an improvement in maximum clock frequency, time, and time-area product for the Zynq UltraScale+ (`xczu49dr`) FPGA when compared to the Artix 7 (`xc7a200t`) FPGA due to their different manufacturing processes.

5.3 Encapsulation

As shown in Algorithm 3, the encapsulation function of Classic McEliece uses the functions FixedWeight and Encode. In this section we first describe how we use the `SHAKE256` module from Section 5.1 to generate a fixed-weight error vector implementing the FixedWeight function. Then we describe our re-implementation of the Encode function, replacing the existing implementation from [WSN18]. Finally, we describe how we implement the complete encapsulation function as specified in [ABC+20] using these building blocks.

5.3.1 Fixed-Weight Vector Generation

The FixedWeight function (Algorithm 4) generates a uniform random n-bit error vector e of weight t. The function assumes that there is a random number generator (RNG) that can be used to generate uniformly distributed random bits. The FixedWeight function first generates a string of $\tau\sigma_1$ random bits (where $\tau = t$ for `mceliece8192128` and $\tau = 2t$ for other parameter sets as specified in [ABC+20]). These random bits are arranged into τ m-bit integers. Out of these τ integers, the first t integers of value smaller than n are selected. The selected t integers then indicate the indices of 1's in e. If the number of m-bit integers in the right range (i.e., $< n$) is less than t or if there exist any duplicates among the t selected integers, the whole process needs to start over by generating another string of $\tau\sigma_1$ random bits.

In our hardware implementation, we use a PRNG to generate these uniform random

Table 7: Comparison of the time and area for our FixedWeight hardware module with output word sizes 32-bits and 160-bits targeting Xilinx Artix 7 (xc7a200t) FPGA.

	Resources								
Param. Set	**Area**	**Memory**			**ACC**	**$\mathbf{F_{max}}$**	**Time**	**T×A**	**Prob.**
	(LUT^L)	(LUT^M)	(FF)	(BR)	(kcyc.)	(MHz)	(μs)		
FixedWeight with output word size - 32 bit									
mceliece348864	265	44	148	2.0	1.0	261	3.9	1.02	0.56
mceliece460896	291	58	157	2.0	2.2	261	8.4	2.45	0.36
mceliece6688128	287	58	158	2.0	3.5	259	13.6	3.90	0.29
mceliece6960119	310	84	157	2.0	2.7	262	10.4	3.24	0.36
mceliece8192128	272	32	148	2.0	1.9	262	7.1	1.94	0.37
FixedWeight with output word size - 160 bit									
mceliece348864	488	44	152	5.5	1.0	170	5.9	2.88	0.56
mceliece460896	554	58	156	5.5	2.2	153	14.4	7.98	0.36
mceliece6688128	542	58	159	5.5	3.5	148	23.8	12.92	0.29
mceliece6960119	566	84	158	5.5	2.7	152	18.0	10.18	0.36
mceliece8192128	509	58	158	5.5	1.9	151	12.4	6.30	0.37

LUT^L = LUT as logic, LUT^M = LUT as memory, FF = flip-flop, BR = BRAM, ACC = average clock cycles, T×A = Time×Area, Prob. = Success Probability

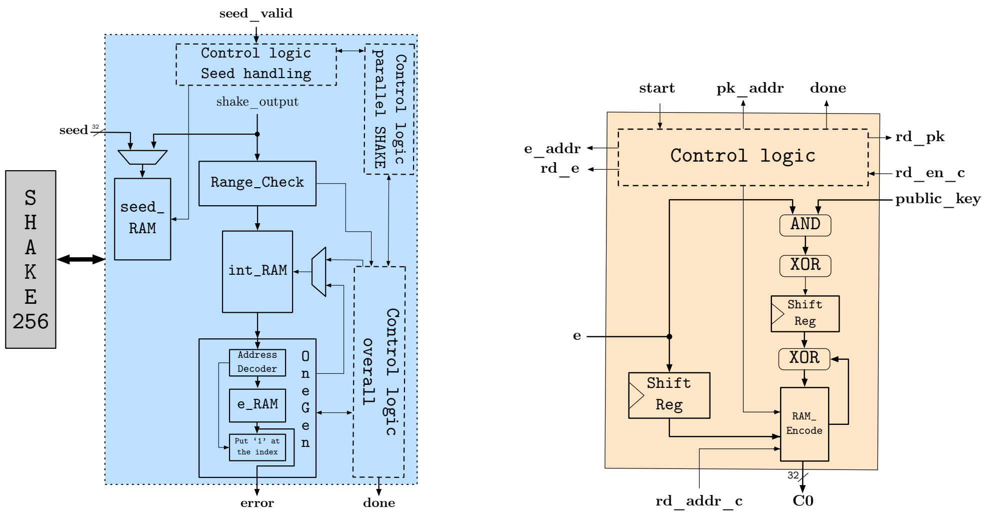
bits from input seeds of length 512 bits. Our hardware module for FixedWeight includes this PRNG, and we assume that the seed will be initialized by another hardware module implementing a true random number generator (TRNG). In our design we use SHAKE256 as a PRNG. To support regeneration of the next $\tau\sigma_1$ random bits, our hardware design actually generates 512 extra bits in addition to the $\tau\sigma_1$ random bits using the PRNG. These 512 bits form a new seed that can be used when the process needs to start over in case the current error-vector generation attempt fails. We note that this specific way of generating random bits for FixedWeight is an implementation choice we made and is not a part of the specification.

The hardware design for the FixedWeight module is shown in the Figure 18a. We use the SHAKE256 module described in Section 5.1 to expand a 512-bit seed to a $(\tau\sigma_1 + 512)$-bit string. Since the SHAKE256 module has a 32-bit interface, we load the new seed in chunks of 32 bits and store it in a single port RAM (seed_RAM) as shown in Figure 18a. The seed_RAM is updated each time a new seed is generated internally.

We use the module RangeCheck to ensure that there are t integers in the right range (i.e., $< n$). The integer values that pass the range checking are stored in a single-ported RAM (int_RAM). Then the OneGen module is used to detect potential duplicates among the integer values stored in int_RAM while it sets the error positions in the error vector e stored in a dual ported RAM (e_RAM). The word width of the e_RAM is parameterizable and can be chosen based on the desired time-area trade-off. This word width also defines the output width of the error port of the OneGen module (shown in 18a).

To reduce the time penalty due to a potential failure during the FixedWeight computation, we reseed and expand the next seed values in advance for the next iteration of FixedWeight in parallel to an ongoing FixedWeight computation. Since the process of reseeding works in parallel to the OneGen module, we are effectively able to hide all the clock cycles required for expanding the seed for the next attempt of FixedWeight error vector generation. Our design is constant-time for successful attempts of error vector generation.

Table 7 shows the results for the FixedWeight hardware module for output widths



(a) FixedWeight module.

(b) Sequential Encode module.

Figure 18: Hardware designs of the FixedWeight error-vector generation module and the sequential Encode module.

32 bits and 160 bits targeting an Xilinx Artix 7 xc7a200t FPGA. With an increasing output width, the required number of BRAMs increases as well, because for a larger output width more BRAMs need to be used for our e_RAM. Also, with the increase in output width, the maximum clock frequency is reduced, because of some combinatorial logic overhead from the address decoder in the OneGen module (shown in Figure 18a). The clock cycles shown in Table 7 are the average cycles computed based on the success probability of the FixedWeight error vector generation process. We obtain the success probabilities (provided in column "Prob." of Table 7) for each parameter set using the methodology described in [ABC+20, Sect. 4.4, p. 31].

The area estimates shown in the Table 7 do not include the area of the SHAKE256 module for the same reasons as described in Section 5.2. The reported frequency values from Table 7 shows the maximum clock frequency for the FixedWeight module standalone. The overall frequency when combining the FixedWeight module and the SHAKE256 module is limited by the SHAKE256 module as described in Section 5.2.

5.3.2 Encoding Function

The Encode function (Algorithm 5) takes a weight-t error vector $e \in \mathbb{F}_2^n$ generated by the FixedWeight function and a public key $T \in \mathbb{F}_2^{(n-k)\times k}$ as an input and generates a ciphertext $C_0 = (I_{n-k} \mid T)e^T \in \mathbb{F}_2^{n-k}$. We first analyzed the hardware implementation of the encoding module provided in [WSN18]. Although that design of the encoding module performs well in terms of cycles and frequency as shown in Table 8, the module requires inputs of the full length of public key columns per clock cycle. This results in a significant resource cost. Furthermore, [WSN18] stores the public key in column major format, which introduces additional effort to import and export a key adherent to the specification, since the specification requires the public key to be represented in row major format.

We address these issues by implementing a sequential Encode module. The hardware design for our sequential Encode module is shown in Figure 18b. We follow a RAM-based approach in order to avoid the usage of large registers as in [WSN18]. Since the first $n-k$ columns of the public key matrix H are always the identity matrix, we efficiently perform the multiplication of the error vector e with this sub-matrix by copying the first $n-k$

Table 8: Comparison of the time and area for our sequential `Encode` module with two exemplary column-block widths of 32 and 160 bit vs. the full-width hardware design from [WSN18] targeting a Xilinx Artix 7 `xc7a200t` FPGA.

	Resources						
Param. Set	**Area**	**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
	(LUT^L)	(FF)	(BR)		(MHz)	(us)	
32-bit design (Our Design)							
`mceliece348864`	139	167	1	66,053	337	195.9	15.87×10^3
`mceliece460896`	144	173	1	132,293	329	401.9	35.37×10^3
`mceliece6688128`	150	175	1	262,917	337	780.3	71.01×10^3
`mceliece6960119`	160	196	1	264,542	319	828.6	72.08×10^3
`mceliece8192128`	145	176	1	341,125	335	1,019	83.55×10^3
160-bit design (Our Design)							
`mceliece348864`	313	321	1	13,289	197	67.46	10.55×10^3
`mceliece460896`	313	326	1	27,461	201	136.5	21.02×10^3
`mceliece6688128`	322	393	1	54,917	196	279.9	47.03×10^3
`mceliece6960119`	333	350	1	54,150	190	284.6	47.24×10^3
`mceliece8192128`	320	394	1	69,893	199	351.8	54.89×10^3
Full-width implementation [WSN18]							
`mceliece348864`	4,267	3,504	0	2,720	312	8.718	37.20×10^3
`mceliece460896`	5,866	4,624	0	3,360	330	10.18	59.73×10^3
`mceliece6688128`	8,365	6,705	0	5,024	322	15.60	130.5×10^3
`mceliece6960119`	8,519	6,977	0	5,413	310	17.46	148.8×10^3
`mceliece8192128`	9,869	8,209	0	6,528	321	20.33	200.7×10^3

LUT^L = LUT as logic, FF = flip-flop, BR = BRAM

elements (i.e., bits) of e directly to the `RAM_Encode` with the help of a shift register (shown in Figure 18b).

We are using the same storage format for the right part T of the public key matrix as for the generation of the public key by storing the matrix in column blocks. On the one hand, this simplifies loading the public key in row-major format into the memory and on the other hand, this simplifies to share the large memory of the public key between key generation and encoding for a joint design. Hence, for processing the right side of the matrix H, we load the rows of the public key in chunks of the width of the column blocks in each clock cycle. The column-block size for the computation is parameterizable and can be chosen freely depending on the targeted time-area trade-off (or according to the choice made for key generation in a joint design). However, in our design, the ciphertext is always consolidated in 32-bit words and stored in "`RAM_Encode`" irrespective of the block size chosen for the public key matrix. Our hardware design is constant-time, compatible with all recommended parameter sets given in the third-round specification document [ABC+20], and is parameterizable in terms of the column-block size for the public key matrix and the error vector input width.

Table 8 shows performance results for our sequential `Encode` module for the column-block sizes 32-bits and 160-bits in comparison to the reference implementation of [WSN18] targeting an Xilinx Artix 7 `xc7a200t` FPGA for all the recommended parameter sets. From the area results shown in Table 8 it can be seen that our sequential implementation is highly optimized in terms of area, while the full-width module from [WSN18] requires much fewer cycles at a significant cost in resources. However, when increasing the column-block size, the efficiency of our design improves in regard to both clock cycles and time-area

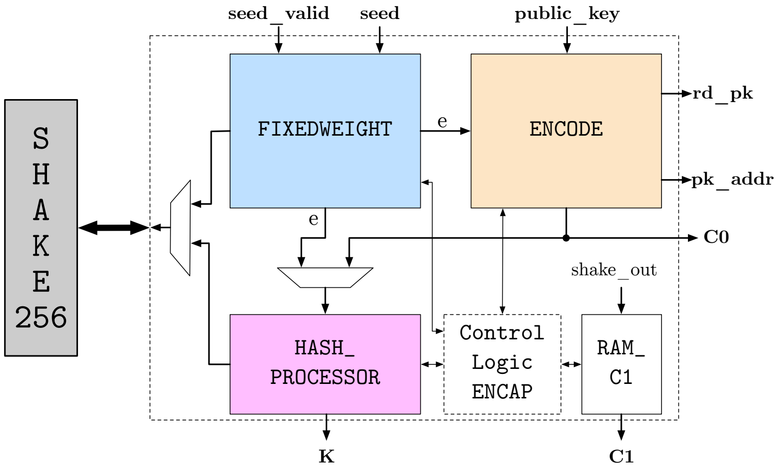


Figure 19: Hardware design of `Encap` module interfaced with `SHAKE256`.

product. We also observe that as the column-block size is increased the maximum clock frequency decreases because the depth of the combinatorial logic performing addition and multiplication increases in the `Encode` module (shown in Figure 18b).

5.3.3 $\mathsf{H}(2, e)$ and $\mathsf{H}(b, e, C)$ Functions

As specified in [ABC$^+$20] we use SHAKE256 as the hash function H in Algorithm 3. For $\mathsf{H}(2, e)$, we prepend the byte `0x02` to the most significant part of the error vector e and calculate the hash value as directed in the specification [ABC$^+$20]. For $\mathsf{H}(b, e, C)$, we prepend the byte `0x0`b to the most significant part of the error vector e and append the ciphertext C. The resulting bit vector is sent to the hash function and a hash value is calculated as directed in the specification [ABC$^+$20]. To compute the aforementioned hash values efficiently, we design a `Hash_Processor`. In this design, we interface a block RAM (which we refer to as `Hash_RAM`) with the `SHAKE256` module such that the specified number of bytes are fetched from the block RAM and the hash computation is performed on them afterwards. We use this approach to eliminate complex multiplexing logic at the input of the `SHAKE256` module that would potentially impose negative effects on the overall maximum clock frequency.

5.3.4 Complete Encapsulation Module

The hardware design for the complete encapsulation module implementing Algorithm 3 is shown in Figure 19. We are using the `FixedWeight`, `Encode`, and `Hash_Processor` modules described in the previous paragraphs as building blocks in the implementation. In order to be able to share the `SHAKE256` module with other Classic McEliece functions (e.g., key generation), we are using a 32-bit interface that is compatible with the `SHAKE256` module and multiplex all inputs going to `SHAKE256` module via this interface. We start with computing the FixedWeight error vector. Then, we compute Encode and $\mathsf{H}(2, e)$ operations (to generate ciphertext C_0 and C_1 respectively) in parallel completely hiding the cycles taken for C_1. We achieve this by storing e inside a dual-port RAM in the `OneGen` module (within `FixedWeight` module, described in Section 5.3.1). Then we compute $\mathsf{H}(1, e, C)$ to generate the session key K.

Our design is constant-time and parameterizable across all the recommended parameter sets described in the third-round specification [ABC$^+$20]. We take advantage of the parameterizable column-block width of the `Encode` module (described in Section 5.3.2) and the

Table 9: Comparison of the time and area for our `Encap` hardware module for column-block sizes 32-bits and 160-bits targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

	Resources							
Parameter Set	**Area**	**Memory**			**Cycles**	**Freq.**	**Time**	**T×A**
	(LUT$^{\mathrm{L}}$)	(LUT$^{\mathrm{M}}$)	(FF)	(BR)	(kcyc.)	(MHz)	(μs)	
`Encap` **with column-block size = 32-bits**								
`mceliece348864`	679	76	423	4	67.98	215	316.2	214.7×10^3
`mceliece460896`	713	64	427	4	135.6	219	619.1	441.4×10^3
`mceliece6688128`	731	90	446	4	267.9	204	1,313	959.9×10^3
`mceliece6960119`	809	116	482	4	268.9	217	1,239	$1{,}002 \times 10^3$
`mceliece8192128`	718	90	414	4	344.8	204	1,690	$1{,}214 \times 10^3$
`Encap` **with column-block size = 160-bits**								
`mceliece348864`	1,110	76	577	7.5	15.75	174	90.52	100.5×10^3
`mceliece460896`	1,209	90	591	7.5	28.19	144	195.8	236.7×10^3
`mceliece6688128`	1,190	90	664	7.5	32.55	142	229.2	272.8×10^3
`mceliece6960119`	1,240	116	636	7.5	58.50	147	398.0	493.5×10^3
`mceliece8192128`	1,181	90	677	7.5	70.02	146	479.6	566.4×10^3

LUT$^{\mathrm{L}}$ = LUT as logic, LUT$^{\mathrm{M}}$ = LUT as memory, FF = flip-flop, BR = BRAM, T×A = Time×Area

parameterizable error-vector output width of the `FixedWeight` (described in Section 5.3.1) and add a similar parameterizable capability to our `Encap` hardware module. Since the `Encap` module has the public key as an input, our design allows a free choice of the key column-block size depending upon on the desired time-area trade-off. Based on the choice of the column-block size, the error vector output width from `FixedWeight` is adjusted internally to support the Encode operation.

Table 9 shows the area and time utilization results for our `Encap` hardware module for key column-block widths of 32-bits and 160-bits targeting a Xilinx Artix 7 `xc7a200t` FPGA. The clock cycles in Table 9 include the average cycles taken by `FixedWeight` error vector generation module (computed based on the success probability as described in Section 5.3.1), cycles taken by `Encode` module and hash computation for K. The area estimates shown in Table 9 do not include the area of the `SHAKE256` module. The reported frequency in Table 9 shows the maximum clock frequency of `Encap` module standalone for all the parameter sets. As discussed before, the frequency is lower when the `Encap` module is interfaced with the `SHAKE256`. Across all the parameter sets we observe that as the column-block size is increased, the efficiency of our design improves. This can be observed in terms of a decrease in the number of clock cycles for the encapsulation operation and a better time-area product.

5.4 Decapsulation

In this section, we present our efficient, modular, and constant-time hardware implementation of the Decap operation defined in Algorithm 6. Our implementation uses the `Decode` module from [WSN18] as building block. An overview of our `Decap` hardware module is shown in Figure 20.

The Decap function takes a ciphertext C (C_0, C_1) and a secret key as inputs and outputs the session key K (see Algorithm 6). The default secret key format includes 5 components $(\delta, c, g, \alpha, s)$, but our decapsulation module takes (δ, c, g) as input. Therefore, our decapsulation module regenerates α (as a list of $\mathbb{F}_{2^m}$ elements) and s by expanding δ so that decapsulation can be carried out. The corresponding decapsulation process can thus be broken down into four main components:

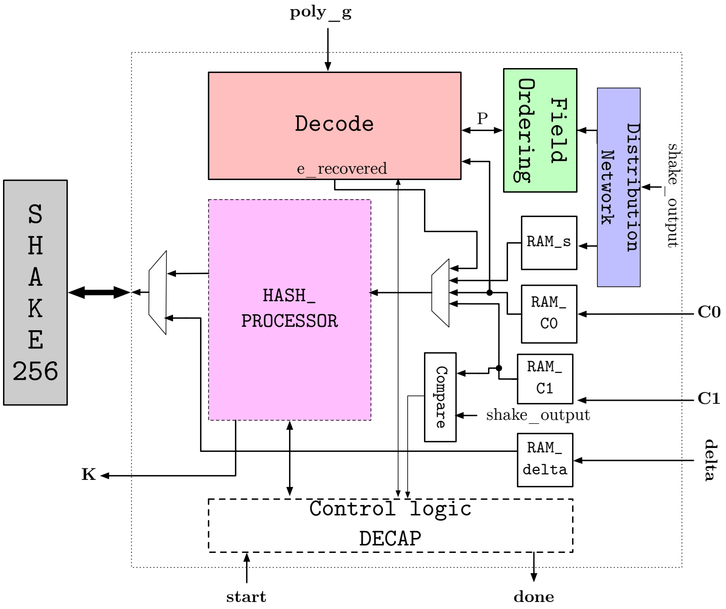


Figure 20: Hardware design of `Decap` module interfaced with `SHAKE256` module.

1. Expand δ (the secret seed) using the PRNG into $n+\sigma_2 q$ bits, use the most significant n bits as S, and rest of the bits (i.e. $\sigma_2 q$ bits) will be used to generate α.
2. Compute α as a list of field elements from the $\sigma_2 q$ bits using FieldOrdering.
3. Decode the fixed-weight error vector from the permutation output and C_0.
4. Compute the H.

We perform the δ expansion using the `SHAKE256` module described in Section 5.1 and generate a total of $n + \sigma_2 q$ pseudorandom bits. As described in Section 5.1, the `SHAKE256` module has a 32-bit interface and therefore generates 32-bits of output per cycle. We build a distribution network to distribute the generated psuedorandom bits to appropriate modules (as shown in Figure 20). Out of the generated $n + \sigma_2 q$-bits, the first n-bits are stored as s in the Block RAM (`RAM_s` as shown in Figure 20). The word size of `RAM_s` is 32-bits. The following $\sigma_2 q$-bits are broken down into two 16-bit numbers. From each 16-bit number j_i the m least significant bits are used as input for the `FieldOrdering` module.

The `FieldOrdering` module computes the q field elements of the support α. After the FieldOrdering step is completed, we the use permutation output, the polynomial g (`poly_g` in Figure 20), and the first part of the ciphertext (i.e., C_1) to decode the error vector using the `Decode` module (shown in Figure 20). We use the `FieldOrdering` and `Decode` hardware modules from the implementations provided in [WSN18]. After the error vector has been decoded, the error vector is loaded into the `Hash_RAM` and the functions $\mathsf{H}(2, e)$ and $\mathsf{H}(b, e, C)$ are computed as described in Section 5.3.3. In case of a decoding failure, we load s into the `Hash_Processor` instead of the error vector as described in Algorithm 6 and perform the same steps as above.

We use a 32-bit interface that is compatible with the `SHAKE256` module to multiplex inputs from δ expansion and H calculation into the `SHAKE256` module. The `Decode` module from [WSN18] uses the number of multipliers inside the Berlekamp-Massey decoder as a performance parameter, which is defined using parameters '`mul_sec_BM`' and '`mul_sec_BM_step`'. We set both these parameters to 20 to obtain a good time-area balance.

Within the `Decode` module, after the error vector is recovered, a `ReEncrypt` module gets triggered to check the validity of the recovered error. Specifically, as shown in Algorithm 7,

Table 10: Comparison of the time and area for our `Decap` hardware module targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

	Resources							
Parameter Set	**Area**	**Memory**			**Cycles**	**Freq.**	**Time**	**T×A**
	(LUT$^{\text{L}}$)	(LUT$^{\text{M}}$)	(FF)	(BR)	(kcyc.)	(MHz)	(ms)	
`mceliece348864`	15,557	314	29,984	34.5	100.2	180	0.56	8.711×10^3
`mceliece460896`	24,698	540	46,509	70.5	201.7	176	1.15	28.36×10^3
`mceliece6688128`	25,848	330	54,527	52.5	216.0	175	1.24	31.96×10^3
`mceliece6960119`	29,546	546	58,126	70.5	210.9	171	1.23	36.36×10^3
`mceliece8192128`	26,633	330	59,048	52.5	219.1	174	1.26	33.43×10^3

LUT$^{\text{L}}$ = LUT as logic, LUT$^{\text{M}}$ = LUT as memory, FF = flip-flop, BR = BRAM, T×A = Time×Area

a validity check ensures that the hamming weight of e is t and $Hv = He$. The first step within re-encryption is to scan the error vector e to extract its hamming weight. This step also packs the indexes of the t nonzero bits of e to a vector `error_bits_indexes`.

A direct check of $Hv = He$ requires the parity check matrix H and hence the large public key, which is actually not necessary. As described in the specification [ABC+20, Sect. 2.2.4], we use the double-size parity check matrix $H^{(2)}$ as the parity check matrix and compare $H^{(2)}v$ with $H^{(2)}e$ in our design. Since the computation of the double-size syndrome $H^{(2)}v$ [WSN18] is already a sub-module within the `Decode` module, the computation of $H^{(2)}e$ can directly reuse this sub-module with the `error_bits_indexes` signal provided as input. Since `error_bits_indexes` always encodes the information of t indexes of e, it is ensured that the re-encryption step is constant-time. Using this approach, the overhead for re-encryption is very small, both in terms of area utilization and clock cycles.

Table 10 shows results for the `Decap` hardware module for all the parameter sets. The area estimates shown in Table 10 do not include the area of the `SHAKE256` module. We observe that more than 80% of the cycles for the Decap operation are taken by the δ expansion and FieldOrdering steps. This overhead can be reduced by buffering α between consecutive decoding operations that using the same private key. The frequency values reported in Table 10 are the maximum clock frequency of our `Decap` module standalone. However, the maximum clock frequency is limited by the `SHAKE256` module when interfaced with our `Decap` module as explained before.

6 Classic McEliece KEM — Joint Design

In this section, we present our hardware design of a joint Classic McEliece design combining our `Encap`, `Decap`, and `SeededKeyGen` modules described in Section 5 into one overall design. In order to build a resource-efficient joint design we start with identifying the sub-modules that can be shared among these three primitives:

1. SHAKE256: As discussed in Sections 5.2 to 5.4, the `SHAKE256` module is common among all three primitives key generation, encapsulation, and decapsulation. The resource utilization for the `SHAKE256` module is reported in Table 5.
2. FieldOrdering: The FieldOrdering operation is common among the Decap and SeededKeyGen algorithms as described in Section 5.4. For the parameter set `mceliece348864`, the `FieldOrdering` hardware module takes up 94% and 14% of the Block RAM resources of the `Decap` and `SeededKeyGen` modules respectively.
3. Additive FFT: The `KeyGen` and `Decode` modules described in [WSN18] use similar `AdditiveFFT` modules. For the parameter set `mceliece348864`, the `AdditiveFFT` module takes up to 17% of the resources of `Decap` and up to 28% of `SeededKeyGen`.

Table 11: Comparison of the time and area for our joint hardware design of Classic McEliece with other code-based schemes (as there is no other complete hardware implementation of Classic McEliece KEM to compare with) targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

	Resources										
Design	**Logic**		**Memory**		**F**	**Encap**		**Decap**		**KeyGen**	
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(Mcyc.)	(ms)	(Mcyc.)	(ms)	(Mcyc.)	(ms)
`mceliece348864` **(our design)**											
LW	23,890	5	45,658	138.5	112	0.13	1.1	0.17	1.5	8.88	79.2
HS	40,018	4	61,881	177.5	113	0.03	0.3	0.10	0.9	0.97	8.6
BIKE - L1 [RBMG20]											
LW	12,868	7	5,354	17.0	121	0.20	1.2	1.62	13.3	2.67	21.9
HS	52,967	13	7,035	49.0	96	0.01	0.1	0.19	1.9	0.26	2.6
HQC - L1 (HLS design) [AAB$^+$20]											
LW	8,900	0	6,400	14.0	132	1.50	11.4	2.10	15.9	0.63	4.7
HS	20,000	0	16,000	12.5	148	0.09	0.6	0.19	1.3	0.04	0.3

LW = LightWeight, *HS* = HighSpeed, FF = flip-flop, F = $F_{\max}$, BR = BRAM

4. Public Key Memory: As discussed in Section 5.2, the public key memory has huge impact on the Block RAM usage in the `SeededKeyGen` module. Duplicating it for Encap would double the number of required Block RAM resources.

To save the resource overhead that would result from duplicating these hardware components, we decided to share them between the corresponding modules. To differentiate between the three operations SeededKeyGen, Decap, or Encap, we add a 2-bit `instruction` port to our joint hardware design to indicate which of the three operations should be performed.

Table 11 shows the time and area results for our joint Classic McEliece design in two flavors, lightweight (*LW*) and high-speed (*HS*). Since there exists no other Classic McEliece hardware design to compare to, we compare our design to existing hardware designs of the code-based cryptography KEM schemes BIKE from [RBMG20] and HQC from [AAB$^+$20] (high-level synthesis from C code) at NIST security level 1. For our *HS* design we choose the modules and performance parameters as described in Section 5, whereas for our *LW* design, we select the `KeyGen` module with DPEA systemizer (described in Section 4) with $s = 12$, the `Encap` module with column-block size = 12 (Section 5.3.4), and we choose the smallest possible performance parameters for the `Decode` module (from [WSN18]), i.e., `mul_sec_BM` = 1 and `mul_sec_BM_step` = 1.

We observe that the area footprint for our *HS* Classic McEliece hardware design is smaller than that of the *HS* BIKE design in terms of logic utilization and lies in between BIKE and HQC. Time taken by our `Encap` module is faster in all the cases except in case of *HS* implementation of BIKE. Our *HS* and *LW* `Decap` module is 9× and 2× faster than *HS* and *LW* BIKE implementation and 11× and 1.5× faster than the *HS* and *LW* HLS implementation of HQC respectively, even though our design includes re-computation of the support α. We also observe that the overall maximum clock frequency of our *LW* and *HS* joint designs is limited due to the `SHAKE256` module as described in Section 5.

Conclusion. Overall, our design has a relatively high resource cost for the *LW* variant but shows overall a very good performance at a good cost for the *HS* variant. Hence, in regard to hardware implementation Classic McEliece competes well with other code-based schemes. In particular the relatively high cost of key generation can be compensated well using optimized systemizer designs if sufficient resources are available.

Acknowledgments

This work was partially funded through the Taiwan Ministry of Science and Technology (MoST) grant 109-2222-E-001-001-MY3, a grant from the Technology Innovation Institute, the United States National Science Foundation grant 1716541, and the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. We would like to thank Victor Mateu for helpful discussions.

References

[AAB+20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABC+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 250–272. Springer, August 2013.

[BMP+06] Andrey Bogdanov, Marius C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. A parallel hardware architecture for fast Gaussian elimination over GF(2). In *Field-Programmable Custom Computing Machines - 14th IEEE Symposium, FCCM 2006*, pages 237–248. IEEE, April 2006.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR TCHES*, 2021(3):125–148, 2021.

[Cho17] Tung Chou. McBits revisited. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 213–231. Springer, September 2017.

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. MicroEliece: McEliece for embedded devices. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 49–64. Springer, September 2009.

[GDUV12] Santosh Ghosh, Jeroen Delvaux, Leif Uhsadel, and Ingrid Verbauwhede. A speed area optimized embedded co-processor for McEliece cryptosystem. In *Application-Specific Systems, Architectures and Processors - 23rd IEEE International Conference, ASAP 2012*, pages 102–108. IEEE, July 2012.

[HG13] Stefan Heyse and Tim Güneysu. Code-based cryptography on reconfigurable hardware: tweaking Niederreiter encryption for performance. *Journal of Cryptographic Engineering*, 3(1):29–43, April 2013.

[HQR89] Bertrand Hochet, Patrice Quinton, and Yves Robert. Systolic Gaussian elimination over GF(p) with partial pivoting. *IEEE Trans. Computers*, 38(9):1321–1324, 1989.

[LGCN20] Mariano López-García and Enrique Cantó-Navarro. Hardware-software implementation of a McEliece cryptosystem for post-quantum cryptography. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Proceedings of the 2020 Future of Information and Communication Conference (FICC) - Advances in Information and Communication*, volume 1130 of *AISC*, pages 814–825. Springer, March 2020.

[MBR15] Pedro M.C. Massolino, Paulo S.L.M. Barreto, and Wilson V. Ruggiero. Optimized and scalable co-processor for McEliece with binary Goppa codes. *ACM Trans. Embed. Comput. Syst.*, 14(3):45:1–45:32, 2015.

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. Technical report, NASA, 1978. https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[Nie86] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[RBMG20] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding BIKE: Scalable hardware implementation for reconfigurable devices. Cryptology ePrint Archive, Report 2020/897, 2020. https://eprint.iacr.org/2020/897.

[REBG11] Andy Rupp, Thomas Eisenbarth, Andrey Bogdanov, and Oliver Grieb. Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalytic applications. *Integr.*, 44(4):290–304, 2011.

[SWM+10] Abdulhadi Shoufan, Thorsten Wink, Gregor Molter, Sorin A. Huss, and Eike Kohnert. A novel cryptoprocessor architecture for the McEliece public-key cryptosystem. *IEEE Trans. Computers*, 59(11):1533–1546, 2010.

[WSN16] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Solving large systems of linear equations over GF(2) on FPGAs. In Peter M. Athanas, René Cumplido, Claudia Feregrino, and Ron Sass, editors, *ReConFigurable Computing and FPGAs - International Conference, ReConFig 2016*, pages 1–7. IEEE, November 2016.

[WSN17] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 253–274. Springer, September 2017.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter cryptosystem using binary Goppa codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, volume 10786 of *LNCS*, pages 77–98. Springer, April 2018.

[WTJ+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the HW/SW co-design of qTESLA. *IACR TCHES*, 2020(3):269–306, 2020.

[YL15] Haibo Yi and Weijian Li. Small FPGA implementations for solving systems of linear equations in finite fields. In *Software Engineering and Service Science - 6th IEEE International Conference, ICSESS 2015*, pages 561–564. IEEE, September 2015.

A Appendix

Algorithm 8 Hybrid Early-Abort Systemizer (HEA)

Input: $\hat{H} = \left(\hat{H}^L \mid \hat{H}^R\right)$, where $\hat{H}^L \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and $\hat{H}^R \in \mathbb{F}_2^{(n-k)\times k}$

Output: $T \in \mathbb{F}_2^{(n-k)\times k}$ such that $\left(I_{n-k} \mid T\right)$ is the systematic form of $\hat{H}$, or $\perp$

```
$A \leftarrow \hat{H}^L$
for $i = 0$ to $n-k-1$ do
    for $j = i+1$ to $n-k-1$ do
        if $A_{i,i} = 0$ and $A_{j,i} = 1$ then
            for $c = i$ to $n-k-1$ do
                swap $A_{i,c}$ with $A_{j,c}$
            end for
        else if $A_{i,i} = 1$ and $A_{j,i} = 1$ then
            for $c = i$ to $n-k-1$ do
                $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
            end for
        end if
    end for
    if $A_{i,i} \neq 1$ then
        return $\perp$                                        ▷ $\hat{H}$ is not systemizable.
    end if
end for
$B \leftarrow \hat{H}$
for $i = 0$ to $n-k-1$ do
    for $j = i+1$ to $n-k-1$ do
        if $B_{i,i} = 0$ and $B_{j,i} = 1$ then
            for $c = i$ to $n-1$ do
                swap $B_{i,c}$ with $B_{j,c}$
            end for
        else if $B_{i,i} = 1$ and $B_{j,i} = 1$ then
            for $c = i$ to $n-1$ do
                $B_{j,c} \leftarrow B_{j,c} + B_{i,c}$
            end for
        end if
    end for
    for $j = 0$ to $i-1$ do
        if $B_{j,i} = 1$ then
            for $c = i$ to $n-1$ do
                $B_{j,c} \leftarrow B_{j,c} + B_{i,c}$
            end for
        end if
    end for
end for
return the matrix formed by the last $k$ columns of $B$
```

Algorithm 9 Single-Pass Early-Abort Systemizer (SPEA)

Input: $\hat{H} = \left(\hat{H^L} \mid \hat{H^R}\right)$, where $\hat{H^L} \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and $\hat{H^R} \in \mathbb{F}_2^{(n-k)\times k}$

Output: $T \in \mathbb{F}_2^{(n-k)\times k}$ such that $\left(I_{n-k} \mid T\right)$ is the systematic form of $\hat{H}$, or $\perp$

```
for $\ell = 0$ to $n-k-1$ do
    $p_\ell = \ell$
end for
$A \leftarrow H^L$
for $i = 0$ to $n-k-1$ do
    for $j = i+1$ to $n-k-1$ do
        if $A_{i,i} = 0$ and $A_{j,i} = 1$ then
            for $c = i$ to $n-k-1$ do
                swap $A_{i,c}$ with $A_{j,c}$
            end for
            $p_i \leftarrow j$
        else if $A_{i,i} = 1$ and $A_{j,i} = 1$ then
            for $c = i+1$ to $n-k-1$ do
                $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
            end for
        end if
    end for
    if $A_{i,i} \neq 1$ then
        return $\perp$                                  ▷ $\hat{H}$ is not systemizable.
    end if
    for $j = 0$ to $i-1$ do
        if $A_{j,i} = 1$ then
            for $c = i+1$ to $n-k-1$ do
                $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
            end for
        end if
    end for
end for
$B \leftarrow H^R$
for $i = 0$ to $n-k-1$ do
    for $j = i+1$ to $n-k-1$ do
        if $p_i = j$ then
            for $c = i$ to $k-1$ do
                swap $B_{i,c}$ with $B_{j,c}$
            end for
        else if $\hat{H^L}_{j,i} = 1$ then
            for $c = i$ to $k-1$ do
                add $B_{i,c}$ to $B_{j,c}$
            end for
        end if
    end for
    for $j = 0$ to $i-1$ do
        if $A_{j,i} = 1$ then
            for $c = i$ to $k-1$ do
                add $B_{i,c}$ to $B_{j,c}$
            end for
        end if
    end for
end for
return $B$
```

Algorithm 10 Dual-Pass Early-Abort Systemizer (DPEA)

Input: $\hat{H} = \left(\hat{H^L} \mid \hat{H^R}\right)$, where $\hat{H^L} \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and $\hat{H^R} \in \mathbb{F}_2^{(n-k)\times k}$

Output: $T \in \mathbb{F}_2^{(n-k)\times k}$ such that $\left(I_{n-k} \mid T\right)$ is the systematic form of $\hat{H}$, or $\perp$

```
for $\ell = 0$ to $n - k - 1$ do
    $p_\ell = \ell$
end for
$A \leftarrow H^L$
for $i = 0$ to $n - k - 1$ do
    for $j = i + 1$ to $n - k - 1$ do
        if $A_{i,i} = 0$ and $A_{j,i} = 1$ then
            for $c = i$ to $n - k - 1$ do
                swap $A_{i,c}$ with $A_{j,c}$
            end for
            $p_i \leftarrow j$
        else if $A_{i,i} = 1$ and $A_{j,i} = 1$ then
            for $c = i + 1$ to $n - k - 1$ do
                $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
            end for
        end if
    end for
    if $A_{i,i} \neq 1$ then
        return $\perp$                                        ▷ $\hat{H}$ is not systemizable.
    end if
end for
$B \leftarrow H^R$
for $i = 0$ to $n - k - 1$ do
    for $j = i + 1$ to $n - k - 1$ do
        if $p_i = j$ then
            for $c = i$ to $k - 1$ do
                swap $B_{i,c}$ with $B_{j,c}$
            end for
        else if $A_{j,i} = 1$ then
            for $c = i$ to $k - 1$ do
                add $B_{i,c}$ to $B_{j,c}$
            end for
        end if
    end for
end for
for $i = n - k - 1$ to $0$ do
    for $j = i - 1$ to $0$ do
        if $A_{j,i} = 1$ then
            for $c = i$ to $k - 1$ do
                add $B_{i,c}$ to $B_{j,c}$
            end for
        end if
    end for
end for
return $B$
```

Errata

The following points have been fixed compared to the CHES version of this paper (IACR TCHES, Volume 2022, Issue 3):

Sept. 5, 2022: Removed “a” on page page 6.

Sept. 5, 2022: Changed “H” to “$\hat{H}$” twice on page page 10.

Sept. 5, 2022: Added “-1” to “$n/s - 1$”on page page 12.

Sept. 5, 2022: Changed “H” to “$\hat{H}$” on page page 14.

Sept. 5, 2022: Changed “dual-pass” to “single-pass” twice on page page 17.

Aug. 25, 2022: Added a clarification on the choice of parameter s (size of the processor array) in Section 5.2 on page 28.

Aug. 25, 2022: Fixed metric suffix in Table 6 on page page 29 from “kcyc.” to ”Mcyc.”.