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. 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 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

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

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]. This hardware implementation does not implement the complete KEM specification but only its Niederreiter core. Hence, all currently existing hardware implementations are not fully specification-compliant.

To address the need, our work presents the first fully specification-compliant hardware design. 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.

Contribution. Our contributions are as follows:

- Based on the improved designs for public-key generation described in [CCD+22] 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.
- 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. The modifications and extensions to [WSN17, WSN18] in order to obtain a complete Classic McEliece implementation are described in Section 3. Finally, in Section 4 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

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

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 $\alpha_1, \dots, \alpha_n$; 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 $\alpha_1, \dots, \alpha_n$ 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 $\alpha_1, \dots, \alpha_n$ and g 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.

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

```
1: Compute $E = \mathsf{G}(\delta)$, a string of $n + \sigma_2 q + \sigma_1 t + \ell$ bits.
2: Define $\delta'$ as the last $\ell$ bits of $E$.
3: Define $s$ as the first $n$ bits of $E$.
4: 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.
5: 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.
6: Define $\Gamma = (g, \alpha_1, \alpha_2, \ldots, \alpha_n)$. (Note that $\alpha_{n+1}, \ldots, \alpha_q$ are not used here.)
7: 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.
8: Write $\Gamma'$ as $(g, \alpha'_1, \alpha'_2, \ldots, \alpha'_n)$.
9: 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].

```
1: 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 $j = 1, \ldots, n$.
2: 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 $u_0, u_1, \ldots, u_{m-1}$.
3: 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 $\perp$.
4: Return $(T, \Gamma)$.
```

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

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

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

```
1: Generate $\sigma_1 \tau$ uniform random bits $b_0, b_1, \ldots, b_{\sigma_1 \tau - 1}$.
2: Define $d_j = \sum_{i=0}^{m-1} b_{\sigma_1 j + i} 2^i$ for each $j \in \{0, 1, \ldots, \tau - 1\}$.
3: 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.
4: If $a_0, a_1, \ldots, a_{t-1}$ are not all distinct, restart the algorithm.
5: 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 $i$.
6: Return $e$.
```

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

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

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

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

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

```
1: Extend $C_0$ to $v = (C_0, 0, \dots, 0) \in \mathbb{F}_2^n$ by appending $k$ zeros.
2: 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 $\perp$.
3: Set $e = v + c$.
4: If $\mathsf{wt}(e) = t$ and $C_0 = He$, return $e$. Otherwise return $\perp$.
```

3 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 [CCD+22, Sect. 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.

3.1 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

Table 2: Comparison of the time and area for our SHAKE256 module targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

`parallel_slices`	**Resources**			**Cycles**	**Freq.**	**Time**	**Time×Area**
	Area	**Memory**					
	(LUT^{L})	(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

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` 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 3.3.1) and δ expansion (described in the seeded key-generation module in Section 3.2).

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 2. 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 2. The clock cycles shown in Table 2 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.

3.2 Seeded Key Generation

Our hardware design for seeded key generation (described in Algorithm 1) is shown in Figure 1. 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.

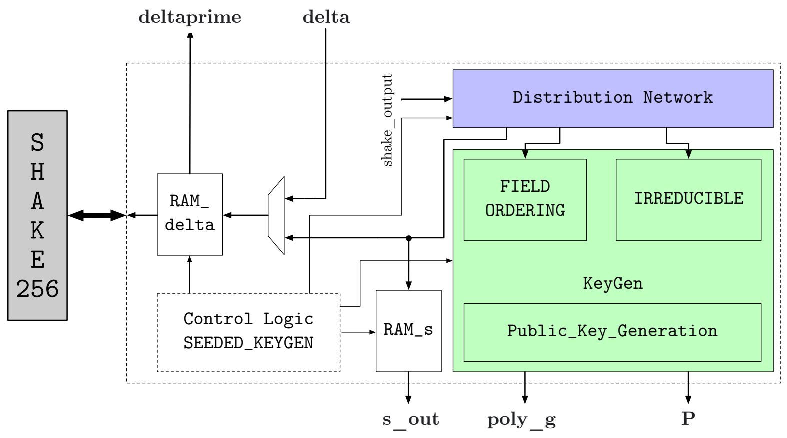


Figure 1: Hardware design of `SeededKeyGen` module interfaced with `SHAKE256` module.

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.

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 1) in a single ported RAM (`RAM_delta` in Figure 1). The wrapper also provides an interface for the connection to the `SHAKE256` module described in Section 3.1.

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 3.4 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 3 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 to as described in [CCD+22, 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 3 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

Table 3: 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_{max}**	**Time**	**T×A**
	(LUT^{L})	(LUT^{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^{L} = LUT as logic, LUT^{M} = LUT as memory, FF = flip-flop, BR = BRAM, ACC = average clock cycles, T×A = Time×Area

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 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 3 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.

3.3 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 3.1 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.

3.3.1 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 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 2a. We use the `SHAKE256` module described in Section 3.1 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 2a. 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 2a).

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 4 shows the results for the `FixedWeight` hardware module for output widths 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 2a). The clock cycles shown in Table 4 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 4) for each parameter set using the methodology described in [ABC+20, Sect. 4.4, p. 31].

The area estimates shown in the Table 4 do not include the area of the `SHAKE256` module for the same reasons as described in Section 3.2. The reported frequency values from Table 4 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 3.2.

Table 4: 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.

Param. Set	Resources				ACC	F_{max}	Time	T×A	Prob.
	Area	Memory							
	(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

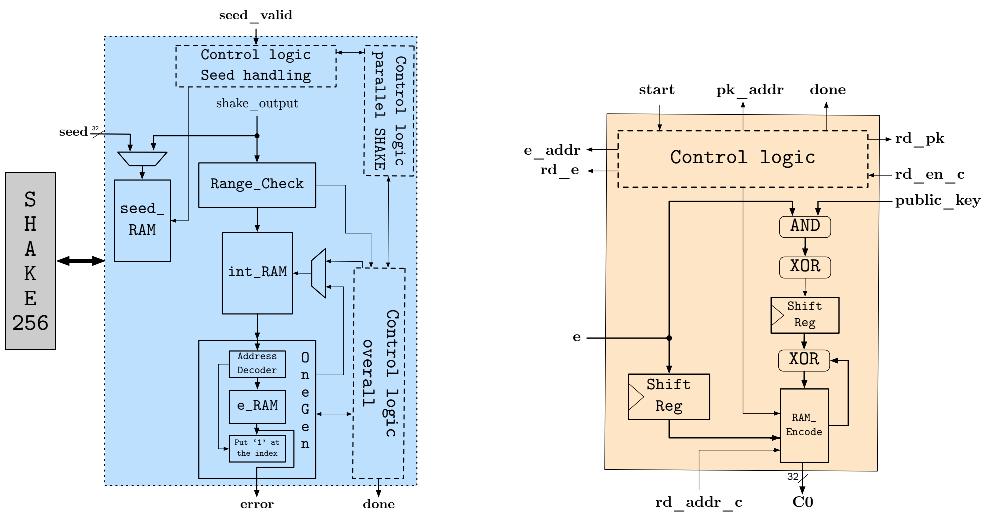
LUT^L = LUT as logic, LUT^M = LUT as memory, FF = flip-flop, BR = BRAM, ACC = average clock cycles, T×A = Time×Area, Prob. = Success Probability

3.3.2 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 5, 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 2b. 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 $n-k$ elements (i.e., bits) of e directly to the `RAM_Encode` with the help of a shift register (shown in Figure 2b).

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



(a) `FixedWeight` module. **(b)** Sequential `Encode` module.

Figure 2: Hardware designs of the `FixedWeight` error-vector generation module and the sequential `Encode` module.

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 5 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 5 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 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 2b).

3.3.3 $\mathsf{H}(2, e)$ 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.

Table 5: 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.

Param. Set	**Resources**			**Cycles**	**Freq.**	**Time**	**Time×Area**
	Area	**Memory**					
	($\mathrm{LUT}^{\mathrm{L}}$)	(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

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

3.3.4 Complete Encapsulation Module

The hardware design for the complete encapsulation module implementing Algorithm 3 is shown in Figure 3. 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 3.3.1). 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 3.3.2) and the parameterizable error-vector output width of the `FixedWeight` (described in Section 3.3.1) 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 6 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`

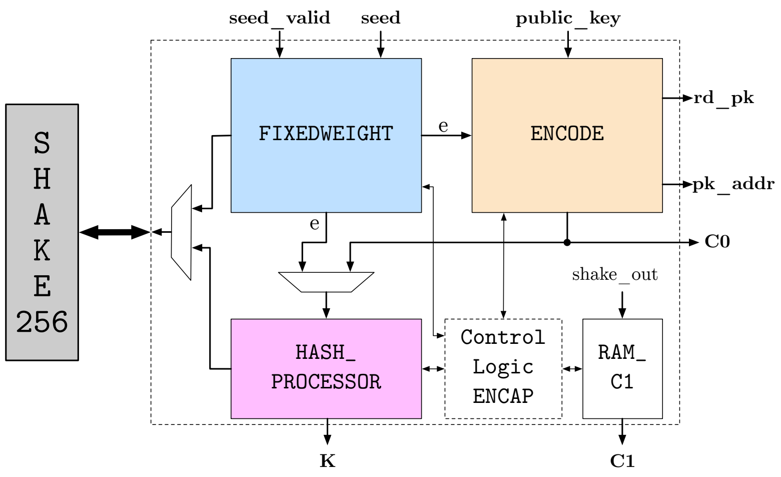


Figure 3: Hardware design of `Encap` module interfaced with `SHAKE256`.

FPGA. The clock cycles in Table 6 include the average cycles taken by `FixedWeight` error vector generation module (computed based on the success probability as described in Section 3.3.1), cycles taken by `Encode` module and hash computation for K. The area estimates shown in Table 6 do not include the area of the `SHAKE256` module. The reported frequency in Table 6 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.

3.4 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 4.

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:

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 3.1 and generate a total of $n + \sigma_2 q$ pseudorandom bits. As described in Section 3.1, 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 4). Out of the generated $n + \sigma_2 q$-bits, the first n-bits are

Table 6: 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^{L})	(LUT^{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^{L} = LUT as logic, LUT^{M} = LUT as memory, FF = flip-flop, BR = BRAM, T×A = Time×Area

stored as s in the Block RAM (`RAM_s` as shown in Figure 4). 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 4), and the first part of the ciphertext (i.e., C_1) to decode the error vector using the `Decode` module (shown in Figure 4). 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 3.3.3. 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, 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

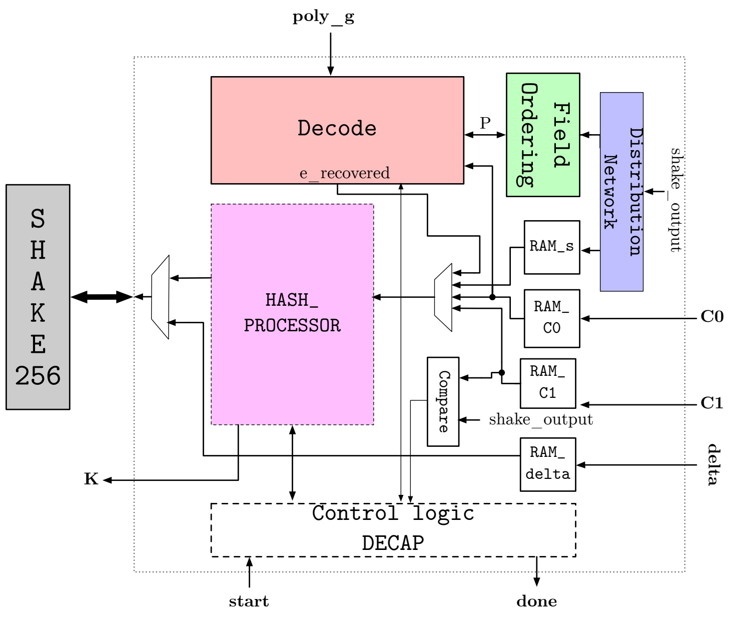


Figure 4: Hardware design of `Decap` module interfaced with `SHAKE256` module.

Table 7: 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^{L})	(LUT^{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^{L} = LUT as logic, LUT^{M} = LUT as memory, FF = flip-flop, BR = BRAM, T×A = Time×Area

for re-encryption is very small, both in terms of area utilization and clock cycles.

Table 7 shows results for the `Decap` hardware module for all the parameter sets. The area estimates shown in Table 7 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 7 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.

4 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 3 into one overall design. In order to build a resource-efficient joint design we start with identifying the

Table 8: 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.

Design	Resources				F	Encap		Decap		KeyGen	
	Logic		Memory								
	(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

sub-modules that can be shared among these three primitives:

1. SHAKE256: As discussed in Sections 3.2 to 3.4, 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 2.
2. FieldOrdering: The FieldOrdering operation is common among the Decap and SeededKeyGen algorithms as described in Section 3.4. 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`.
4. Public Key Memory: As discussed in Section 3.2, 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 8 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 3, whereas for our *LW* design, we select the `KeyGen` module with DPEA systemizer (described in [CCD+22, Sect. 4]) with $s = 12$, the `Encap` module with column-block size = 12 (Section 3.3.4), 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 $9\times$ and $2\times$ faster than HS and LW BIKE implementation and $11\times$ and $1.5\times$ 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 3.

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.

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

[CCD+22] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved fpga implementation of classic mceliece. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):71–113, Jun. 2022.

[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.

[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.

[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.

[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.

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 $\perp$

```
1:  $A \leftarrow \hat{H}^L$
2:  for $i = 0$ to $n-k-1$ do
3:      for $j = i+1$ to $n-k-1$ do
4:          if $A_{i,i} = 0$ and $A_{j,i} = 1$ then
5:              for $c = i$ to $n-k-1$ do
6:                  swap $A_{i,c}$ with $A_{j,c}$
7:              end for
8:          else if $A_{i,i} = 1$ and $A_{j,i} = 1$ then
9:              for $c = i$ to $n-k-1$ do
10:                 $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
11:             end for
12:         end if
13:     end for
14:     if $A_{i,i} \neq 1$ then
15:         return $\perp$                                  ▷ $\hat{H}$ is not systemizable.
16:     end if
17: end for
18: $B \leftarrow \hat{H}$
19: for $i = 0$ to $n-k-1$ do
20:     for $j = i+1$ to $n-k-1$ do
21:         if $B_{i,i} = 0$ and $B_{j,i} = 1$ then
22:             for $c = i$ to $n-1$ do
23:                 swap $B_{i,c}$ with $B_{j,c}$
24:             end for
25:         else if $B_{i,i} = 1$ and $B_{j,i} = 1$ then
26:             for $c = i$ to $n-1$ do
27:                 $B_{j,c} \leftarrow B_{j,c} + B_{i,c}$
28:             end for
29:         end if
30:     end for
31:     for $j = 0$ to $i-1$ do
32:         if $B_{j,i} = 1$ then
33:             for $c = i$ to $n-1$ do
34:                 $B_{j,c} \leftarrow B_{j,c} + B_{i,c}$
35:             end for
36:         end if
37:     end for
38: end for
39: 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 $\perp$

```
1: for $\ell = 0$ to $n-k-1$ do
2:     $p_\ell = \ell$
3: end for
4: $A \leftarrow H^L$
5: for $i = 0$ to $n-k-1$ do
6:     for $j = i+1$ to $n-k-1$ do
7:         if $A_{i,i} = 0$ and $A_{j,i} = 1$ then
8:             for $c = i$ to $n-k-1$ do
9:                 swap $A_{i,c}$ with $A_{j,c}$
10:            end for
11:            $p_i \leftarrow j$
12:        else if $A_{i,i} = 1$ and $A_{j,i} = 1$ then
13:            for $c = i+1$ to $n-k-1$ do
14:                $A_{j,c} \leftarrow A_{j,c} + A_{i,c}$
15:            end for
16:        end if
17:    end for
18:    if $A_{i,i} \neq 1$ then
19:        return $\perp$                              ▷ $\hat{H}$ is not systemizable.
20:    end if
21: end for
22: $B \leftarrow H^R$
23: for $i = 0$ to $n-k-1$ do
24:     for $j = i+1$ to $n-k-1$ do
25:         if $p_i = j$ then
26:             for $c = i$ to $k-1$ do
27:                 swap $B_{i,c}$ with $B_{j,c}$
28:             end for
29:         else if $A_{j,i} = 1$ then
30:             for $c = i$ to $k-1$ do
31:                 add $B_{i,c}$ to $B_{j,c}$
32:             end for
33:         end if
34:     end for
35: end for
36: for $i = n-k-1$ to $0$ do
37:     for $j = i-1$ to $0$ do
38:         if $A_{j,i} = 1$ then
39:             for $c = i$ to $k-1$ do
40:                 add $B_{i,c}$ to $B_{j,c}$
41:             end for
42:         end if
43:     end for
44: end for
45: return $B$
```