Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices

Jan Richter-Brockmann[1], Johannes Mono[1] and Tim Güneysu[1,2]

[1] Ruhr-Universität Bochum, Horst-Görtz Institute for IT-Security, Germany
[2] DFKI, Germany
firstname.lastname@rub.de

Abstract. Contemporary digital infrastructures and systems use and trust Public-Key Cryptography to exchange keys over insecure communication channels. With the development and progress in the research field of quantum computers, well established schemes like RSA and ECC are more and more threatened. The urgent demand to find and standardize new schemes – which are secure in a post-quantum world – was also realized by the National Institute of Standards and Technology which announced a Post-Quantum Cryptography Standardization Project in 2017. Recently, the round three candidates were announced and one of the alternate candidates is the Key Encapsulation Mechanism scheme BIKE.
In this work we investigate different strategies to efficiently implement the BIKE algorithm on Field-Programmable Gate Arrays (FPGAs). To this extend, we improve already existing polynomial multipliers, propose efficient strategies to realize polynomial inversions, and implement the Black-Gray-Flip decoder for the first time. Additionally, our implementation is designed to be scalable and generic with the BIKE specific parameters. All together, the fastest designs achieve latencies of 2.69 ms for the key generation, 0.1 ms for the encapsulation, and 1.9 ms for the decapsulation considering the first security level.

Keywords: BIKE · QC-MDPC · PQC · Reconfigurable Devices · FPGA.

1 Introduction

In our contemporary life Public-Key Cryptography (PKC) plays a crucial part to exchange keys over insecure communication channels. However, established schemes like RSA [RSA78] and ECC [Mil85] are threatened by the advancing development of quantum computers [Gam20]. In 1999, Peter Shor already presented an algorithm breaking currently used PKC schemes in polynomial time on quantum computers [Sho99]. Therefore, there was extensive research to find new schemes which are secure even in the presence of quantum adversaries. A promising research area is code-based cryptography where hard problems from coding-theory are used to create cryptographic schemes. The first such scheme based on linear error codes was proposed by McEliece in 1978 [McE78]. Even though the McEliece cryptosystem is assumed to be secure against classical and quantum-based attacks, one disadvantage is its large public key.

In order to decrease the key size (and the corresponding memory requirements and transmission bandwidth), a new class of linear codes were designed called Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes. They were first presented in [MTSB13] and gained more and more attention in the recent years due to performance and security features. In 2017, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization Project aiming to find and standardize suitable Post-Quantum Cryptography (PQC) schemes. One of the submissions was Bit

Flipping Key Encapsulation (BIKE) which is built upon QC-MDPC codes. With the submission to the third round, the BIKE team reduced the number of algorithms proposed in earlier specifications [ABB$^+$19] to one single algorithm now just called BIKE. The remaining algorithm (called BIKE-2 in earlier submissions) is based on the Niederreiter framework [Nie86] including some tweaks [ABB$^+$20].

After the announcement of the Post-Quantum Cryptography Standardization Project, NIST published a list of selection-criteria including security, cost and performance as well as algorithm and implementation characteristics on various platforms [AAAS$^+$19]. Current software implementations are the reference implementation of BIKE, an optimized software implementation for Intel **CPU!**s [DGK20a], and an efficient microcontroller implementation [BOG19]. Until now, there is no complete hardware implementation of the third round submission of BIKE. In this work, we propose an optimized hardware design for FPGAs.

Related Work. After the introduction of QC-MDPC codes by Misoczki et *al.*, the authors of [HVMG13] were the first researchers who implemented the McEliece cryptosystem with QC-MDPC codes on FPGAs. Besides an exploration of different decoders suited for efficient hardware implementations, they decided to follow a design strategy targeting a high-speed implementation. To this end, they stored all keys and intermediate results directly in the FPGA logic and did not use any external or internal memories. Following this strategy, the implementation can process an entire vector at a time allowing to instantiate high-speed submodules.

One year later, von Maurich and Güneysu presented a lightweight implementation of McEliece using QC-MDPC codes [VMG14]. They divided each vector into chunks of 32 bit and processed them separately. This approach also included internal memory (i.e., Block-RAM (BRAM)) to keep the amount of registers as low as possible.

The authors of [HC17] proposed an area time efficient hardware implementation for QC-MDPC codes and achieved better results than Heyse et *al.* in [HVMG13]. The improvements were mainly gained by a custom designed decoder equipped with a hardware module estimating the Hamming weight of larger vectors.

With the submission to the second round, the BIKE team presented an FPGA implementation of one of the discarded algorithms called BIKE-1 including the key generation and encapsulation [ABB$^+$19]. Their design strategy was very similar to the one presented in [VMG14] but included two optimization levels which basically parallelized the encoding process.

Recently, Reinders et *al.* proposed an efficient hardware design with a constant-time decoder which was also designed for the older BIKE-1 algorithm [RMGS20]. However, the decoder proposed in their work differs from the introduced decoder of the current BIKE specification and reference implementation. Additionally, as they opted for BIKE-1, they did not implemented any polynomial inversion.

An efficient algorithm to accomplish polynomial inversions was presented in [HGWC15] and is based on the classic Itoh-Tsujii Algorithm (ITA) [IT88]. Here and in many other parts of BIKE, polynomial multiplications are an essential building block which can be realized by different design strategies. Two of them – i.e., a row-by-row strategy and a strategy dividing the vectors into chunks – were described in the above mentioned works [HVMG13] and [VMG14]. Another strategy was recently introduced by Hu et *al.* in [HWCW19] where the authors decomposed the quasi-cyclic matrix (constructed from one of the polynomials) into sub-matrices achieving an enhanced area-time product.

Contribution. In our work we present the first hardware implementation of the entire BIKE algorithm selected as alternate candidate in the NIST PQC competition. The first challenge lies in the realization of the polynomial inversion required for the key generation.

In our work we investigate different optimization strategies realizing a polynomial inversion in hardware, which eventually leads to a highly optimized design. The inversion module as well as other parts of BIKE require a polynomial multiplier. We slightly improve the multiplier proposed in [HWCW19] and reduce the overall latency. Additionally, we provide the first hardware implementation of the Black-Gray-Flip (BGF) decoder originally proposed in [DGK20c]. The implementation is constant-time with respect to secret values (i.e., the operation time of all modules is independent of any secret values) and is thus secure against timing attacks.

By implementing a parameterized design, we can scale our implementation down to small devices (resulting in higher latency) or scale up for low-latency applications (resulting in a bigger implementation). Additionally, we provide SageMath scripts to achieve a design which is completely generic with respect to all parameters used in BIKE. The scripts also generate testvectors using the software reference implementation. All HDL-files are available at `https://github.com/Chair-for-Security-Engineering/BIKE`.

Outline. The remainder of this work is structured as follows: In Section 2 we formally introduce the BIKE algorithm, describe the underlying decoder, and state our design considerations. Afterwards, we present the hardware implementations of each individual building block required to compose BIKE in Section 3. Following, Section 4 provides an overview about the whole design. We evaluate our implementation in Section 5 and conclude our work in Section 6.

2 Preliminaries

In this chapter we briefly summarize the mathematical background of QC-MDPC codes and describe the BIKE algorithm. We closely follow the notations of [ABB+20].

We define $|v|$ as the Hamming weight of a given polynomial v. A uniform random sampling of v is denoted by $v \xleftarrow{\$} \mathcal{U}$. When writing h_j, we access the j-th row of a matrix H. The notation $\{0,1\}^l_{[t]}$ describes the set of all l-bit strings with Hamming weight t. Throughout this work, we will use $\log()$ as the base 2 logarithm $\log_2()$.

2.1 QC-MDPC Codes

Definition 1. An (n, k)-linear code $\mathcal{C}$ of length n, dimension k, and co-dimension $r = (n - k)$ is a k-dimensional subspace of $\mathbb{F}_q^n$.

Note, BIKE considers only binary linear codes so that $q = 2$ is used throughout this work.

Definition 2. A matrix $G \in \mathbb{F}_2^{k\times n}$ is called a generator matrix of a binary (n, k)-linear code $\mathcal{C}$ if $\mathcal{C} = \left\{mG | m \in \mathbb{F}_2^k\right\}$. A matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is called parity-check matrix of $\mathcal{C}$ if $\mathcal{C} = \left\{c \in \mathbb{F}_2^n | Hc^{\mathrm{T}} = 0\right\}$.

Definition 3. A vector $c \in \mathcal{C}$ is called codeword and is generated from a vector $m \in \mathbb{F}_2^{n-r}$ by $c = mG$. Given a vector $c' \in \mathbb{F}_2^n$, the vector $s \in \mathbb{F}_2^r$ gained by $s^{\mathrm{T}} = Hc'^{\mathrm{T}}$ is called syndrome.

Given a valid codeword $c \in \mathcal{C}$ and a vector $e \in \mathbb{F}_2^n$ such that $c' = c \oplus e$, than the syndrome can be expressed by $s^{\mathrm{T}} = Hc'^{\mathrm{T}} = Hc^{\mathrm{T}} \oplus He^{\mathrm{T}} = He^{\mathrm{T}}$.

Definition 4. A binary square matrix A is called circulant matrix if each row is the rotation of one element to the right of the preceding row.

As a result, a circulant matrix is completely defined by its first row. Additionally, a block-circulant matrix is composed of circulant square blocks of identical size called *order*. The number of circulant blocks in a row is called *index*. A formal definition is given below.

Definition 5. A (binary) Quasi-Cyclic (QC) code of index n_0 and order r is a linear code which admits as generator matrix a block-circulant matrix of order r and index n_0. A (n_0, k_0)-QC code is a quasi-cyclic code of index n_0, length $n_0 r$ and dimension $k_0 r$.

A binary $r \times r$ matrix A can be expressed by an element from a quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[X]/(X^r - 1)$. The mapping between A and $\mathcal{R}$ is described by a natural ring isomorphism φ which maps the first row of A, represented by $(a_0, a_1, ..., a_{r-1})$, to the polynomial $\varphi(A) = a_0 + a_1 X + ... + a_{r-1} X^{r-1}$. Therefore, all matrix operations can be seen as polynomial operations.

Definition 6. The transposition of a polynomial $a_0 + a_1 X + ... + a_{r-1} X^{r-1} = a \in \mathcal{R}$ is defined as $a^{\mathrm{T}} = a_0 + a_{r-1} X + ... + a_1 X^{r-1}$.

This definition ensures $\varphi(A^{\mathrm{T}}) = \varphi(A)^{\mathrm{T}}$. Furthermore, the isomorphism φ can be extended to any binary vector $(v_0, v_1, ..., v_{r-1}) = \mathbf{v} \in \mathbb{F}_2^r$ such that $\varphi(\mathbf{v}) = v_0 + v_1 X + ... + v_{r-1} X^{r-1}$. To stay consistent with Definition 6, the transposition of $\varphi(\mathbf{v})$ is defined as $\varphi(\mathbf{v}^{\mathrm{T}}) = v_0 + v_{r-1} X + ... + v_1 X^{r-1}$ resulting in $\varphi(\mathbf{v}A) = \varphi(\mathbf{v})\varphi(A)$ and $\varphi(A\mathbf{v}^{\mathrm{T}}) = \varphi(A)\varphi(\mathbf{v})^{\mathrm{T}}$.

Definition 7. A quasi-cyclic code of length $n = n_0 r$, dimension $k = k_0 r$, order r and a parity-check matrix with constant row weight $w = \mathcal{O}(\sqrt{n})$ is called an (n_0, k_0, r, w)-QC-MDPC code.

Due to their structure, iterative decoders can be applied to QC-MDPC codes as proposed by Gallager in 1962 for Low-Density Parity-Check (LDPC) codes [Gal62].

2.2 BIKE

BIKE consists of three algorithms namely the *key generation*, *encapsulation*, and *decapsulation.* Besides the security level λ, three parameters r, w, and t are defined. The parameter r defines the block length and needs to be prime such that $(X^r - 1)/(X - 1) \in \mathbb{F}_2[X]$ is irreducible. The row weight w defines the number of bits set in the private key and is chosen such that $d = w/2$ is odd. The parameter t is a positive integer and determines the decoding radius, i.e., the Hamming weight of an error vector $e = (e_0, e_1)$. As an additional parameter the shared secret size ℓ is defined as a positive integer. Note that the code length n is set to $n = 2r$.

Additionally, BIKE defines a set of three functions $\mathbf{H}, \mathbf{K}, \mathbf{L}$ modeled as random oracles. The functions are defined with the following domains and ranges.

$$\begin{aligned}\mathbf{H} :& \{0,1\}^{\ell} \to \{0,1\}^{2r}_{[t]} \\ \mathbf{K} :& \{0,1\}^{r+2\ell} \to \{0,1\}^{\ell} \\ \mathbf{L} :& \{0,1\}^{2r} \to \{0,1\}^{\ell}\end{aligned}$$

Algorithm 1, Algorithm 2, and Algorithm 3 formally describe the three algorithms key generation, encapsulation, and decapsulation, respectively. Table 1 lists the suggested parameters for the security levels 1 and 3. The shared secret size ℓ is fixed to 256 for both cases. For more details, we refer the interested reader to the specification of BIKE [ABB+20].

2.3 Efficient Decoder

The decapsulation of BIKE invokes a `decoder` (cf. Algorithm 3) trying to determine the error vector sampled in the encapsulation process in order to recover the message m. An efficient algorithm for this task was presented in [DGK20c] and is called Black-Gray-Flip

Algorithm 1: Key Generation.

```
Input  : BIKE parameters $n, w, t, \ell$.
Output : Private key $(h_0, h_1, \sigma)$ and public key $h$.
1  Generate $(h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ both of odd weight $|h_0| = |h_1| = w/2$.
2  Generate $\sigma \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
3  Compute $h \leftarrow h_1 h_0^{-1}$.
4  Return $(h_0, h_1, \sigma)$ and $h$.
```

Algorithm 2: Encapsulation.

```
Input  : Public key $h$.
Output : Encapsulated key $K$ and ciphertext $C = (c_0, c_1)$.
1  Generate $m \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
2  Compute $(e_0, e_1) \leftarrow \mathbf{H}(m)$.
3  Compute $C = (c_0, c_1) \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$.
4  Compute $K \leftarrow \mathbf{K}(m, C)$.
5  Return $(C, K)$.
```

Algorithm 3: Decapsulation.

```
Input  : Private key $(h_0, h_1, \sigma)$ and ciphertext $C = (c_0, c_1)$.
Output : Decapsulated key $K$.
1  Generate $(e'_0, e'_1) \xleftarrow{\$} \mathcal{R}^2$ with $|e_0| + |e_1| = t$.
2  Compute syndrome $s \leftarrow c_0 h_0$.
3  Compute $\{(e''_0, e''_1), \perp\} \leftarrow \texttt{decoder}(s, h_0, h_1)$.
4  if $(e''_0, e''_1) \leftarrow \texttt{decoder}(s, h_0, h_1)$ and $|(e''_0, e''_1)| = t$ then
5      Set $(e'_0, e'_1) \leftarrow (e''_0, e''_1)$.
6  end
7  Compute $m' \leftarrow c_1 \oplus \mathbf{L}(e'_0, e'_1)$.
8  if $\mathbf{H}(m') \neq (e'_0, e'_1)$ then
9      Compute $K \leftarrow \mathbf{K}(\sigma, C)$.
10 else
11     Compute $K \leftarrow \mathbf{K}(m', C)$.
12 Return $K$.
```

Decoder. With the submission to the third round of the NIST PQC competition, the BGF decoder was included in the BIKE scheme. A formal description of the decoder can be found in Algorithm 4. The decoder is an iterative algorithm, running for *NBIter* iterations, taking (s, h_0, h_1) as input, and returning an error vector $e = (e_0, e_1)$ in case of a successful decoding or $\perp$ in case the decoding fails. Based on the Hamming weight of the sum $s + eH^{\mathrm{T}}$, a threshold T is computed by

$$\texttt{threshold}(x) = \max(\lfloor f_0 \cdot x + f_1 \rfloor, c) \tag{1}$$

where f_0, f_1 and c are constants associated with the security level. The procedure `BFIter` counts the Unsatisfied-Parity-Check (UPC) equations by invoking `ctr` (i.e., the Hamming weight of $h_j \cdot s$ where h_j is the j-th column of the matrix H) and flips all bits in the error vector that were indicated by counter values exceeding the threshold T. Additionally, `BFIter` generates two lists – *black* and *gray* – which mark all positions where the counter exceeds T and $T - \tau$, respectively. In the first iteration of the decoder these two lists are

Table 1: BIKE parameters.

	BIKE Specific			**Decoder Specific**				
Security	r	w	t	f_0	f_1	c	$NBIter$	τ
Level 1	12 323	142	134	0.0069722	13.530	36	5	3
Level 3	24 659	206	199	0.005265	15.2588	52	5	3

Algorithm 4: Black-Gray-Flip Decoder [DGK20c, ABB+20].

```
Data: $H \in \mathbb{F}_2^{r \times n}$, $s \in \mathbb{F}_2^r$
1  $e \leftarrow 0^n$
2  for $i = 1$ to $NBIter$ do
3      $T \leftarrow$ threshold $\left(\left|s + eH^{\mathrm{T}}\right|\right)$
4      $e, black, gray \leftarrow$ BFIter $\left(s + eH^{\mathrm{T}}, e, T, H\right)$
5      if $i = 1$ then
6          $e \leftarrow$ BFMIter $\left(s + eH^{\mathrm{T}}, e, black, (d+1)/2+1, H\right)$
7          $e \leftarrow$ BFMIter $\left(s + eH^{\mathrm{T}}, e, gray, (d+1)/2+1, H\right)$
8      end
9  end
10 if $s = eH^T$ then
11     return $e$
12 else
13     return $\perp$
14 end
15 procedure BFIter$(s, e, T, H)$
16 for $j = 0$ to $n - 1$ do
17     if ctr$(H, s, j) \geq T$ then
18         $e_j \leftarrow e_j \oplus 1$
19         $black_j \leftarrow 1$
20     else if ctr$(H, s, j) \geq T - \tau$ then
21         $gray_j \leftarrow 1$
22 end
23 return $e, black, gray$
24 procedure BFMIter$(s, e, mask, T, H)$
25 for $j = 0$ to $n - 1$ do
26     if ctr$(H, s, j) \geq T$ then
27         $e_j \leftarrow e_j \oplus mask_j$
28     end
29 end
30 return $e$
```

used to adjust the error vector by applying the procedure `BFMIter`. All parameters used to define the decoder are summarized in Table 1 for both security levels.

2.4 Design Considerations

In general, our implementation tries to keep the footprint as small as possible while providing a reasonable throughput. This goal is achieved by storing all polynomials in BRAMs instead of using registers even if that means forgoing the possibility to access all bits of a polynomial at the same time. This strategy drastically reduces the amount of required registers (and consequently slices). Nevertheless, we decided to use registers whenever values of ℓ bits (e.g., m or c_1) need to be stored as spending an entire BRAM would waste hardware resources.

Besides these trade-offs, our implementation is developed to be generic with the BIKE specific parameters in case they need to be adapted (e.g., for security reasons). We also use a parameter b to set the internally applied data bus width, to determine the bus width of all BRAMs, and to scale several submodules. All polynomials are divided into chunks of b bits which will be further processed by the required submodules (e. g., multiplier or inversion). By writing $a[i]$, we denote b bits of the polynomial a which are stored at address i where the Least Significant Bit (LSB) a_0 of a is stored in the LSB at address $i = 0$. In our evaluation we consider $b \in \mathcal{B} = \{32, 64, 128\}$ as these values are common for bus widths and as larger values would exceed the available hardware resources on Xilinx's Artix-7 FPGAs[1]. All submodules (except BRAMs with a wrapper for each b) are implemented generically such that they can be adapted to specific use-cases and instantiated with arbitrary b.

The generations of (h_0, h_1), m and (e'_0, e'_1) in the key generation, encapsulation and decapsulation, respectively, require a source of randomness. In our design we assume that the target device is equipped with an appropriate Random Number Generator (RNG) since the implementation of a secure RNG is out of scope of this work. All modules requiring such randomness have implemented ports which could be connected to an available source of randomness.

Our goal is to comply with the BIKE specification and implement everything as proposed. Thus, we can generate and extract testvectors from the reference implementation and can validate the output of our design.

3 Efficient Hardware Implementation

In this section we present our design strategies for all submodules required to assemble the complete BIKE algorithm. We discuss our approaches and state implementation results for each submodule separately. All results were generated for a Xilinx Artix-7 FPGA.

3.1 Sampler

With Predefined Hamming Weight The first step in the key generation (cf. Algorithm 1) is to sample the polynomials (h_0, h_1) representing the first part of the secret key. Since both polynomials are defined to have a Hamming weight of $w/2$, they can be sampled in parallel. The samplers are realized by rejection sampling [DG19] both expecting a $\lceil \log_2(r) \rceil$-bit input $x_{\text{rand},i}$ of fresh randomness every two clock cycles with $i \in \{0, 1\}$. The input $x_{\text{rand},i}$ determines the non-zero positions in the polynomial h_i. Since this procedure works on bit level, we decided to fix b to 32 bits as increasing b would not improve the throughput of the sampler and just would produce additional overhead in terms of hardware resources. Based on this, the sampler uses the lower 5 bits of $x_{\text{rand},i}$ to identify the desired bit in a target chunk which is read from an address determined by the remaining bits of $x_{\text{rand},i}$. Within the first clock cycle of sampling one single bit, the sampler reads the target address. In the 32-bit output the desired bit is checked and if it is zero it is set to one otherwise it is left unaltered. The result is written back in the second clock cycle. In case a bit is set to one and $x_{\text{rand},i} < r$ a valid signal enables a counter which monitors the Hamming weight of the sampled polynomial.

Rejection sampling avoids biased random values obtained by e.g., reducing $x_{\text{rand},i}$ modulo r but the sample process does not finish in constant time. This behavior does not reveal confidential information [DG19] but in order to estimate the latency of the

[1]Note that the NIST recommended to use Artix-7 FPGAs for PQC hardware implementations.

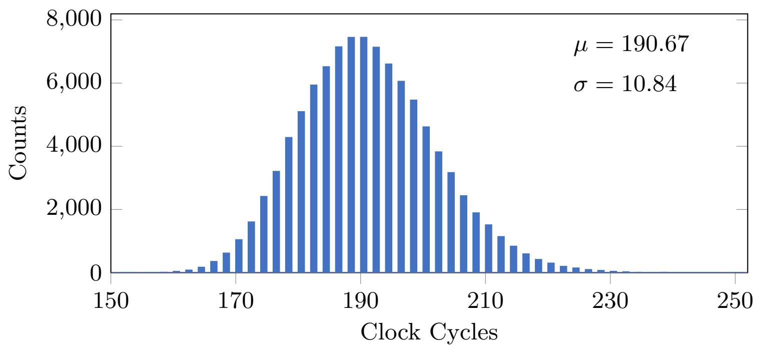


Figure 1: Distribution of required clock cycles to sample one polynomial of the secret key for $r = 12\,323$ and $w = 142$ based on 100 000 simulations.

sampler, we provide a formula to calculate the average clock cycles in the following. The probability of not getting rejected, i.e., the success probability is $s = \frac{r}{2^{\lceil \log(r) \rceil}}$. However, this term needs to be adjusted as collisions getting more likely with an increased number of bits already set in $h_{i \in \{0,1\}}$ which is done by $(1 - \frac{j-1}{r})$ where j indicates the number of bits already been set. Finally, Equation 2 is used to calculate the average clock cycles $N_{\text{sample,avg}}$ required to finish the sample process for the polynomials h_i. The leading factor of two is due to the read and write accesses to the BRAM mentioned above.

$$N_{\text{sample,avg}} = 2 \cdot \sum_{j=1}^{w/2} \frac{1}{s \cdot (1 - \frac{j-1}{r})} \tag{2}$$

For the first security level $N_{\text{sample,avg}} = 2 \cdot 94.67 = 189.34$. In order to verify our hardware implementation, we performed 100 000 simulations and plotted the number of required clock cycles to finish the sampling in Figure 1. The results confirm a correct functionality of our implemented sampler and show the expected average number of clock cycles.

One sampler generating a single polynomial $h_{i \in \{0,1\}}$ consumes around 25 slices partitioned into 66 Look-Up Tables (LUTs) and 19 registers. Our final implementation instantiates two samplers to generate (h_0, h_1) in parallel.

Uniform Sampler The sampling process of the second half of the secret key σ is done in a straightforward way by using a 32-bit input providing fresh randomness. The random bits are stored in registers because σ only consists of $\ell = 256$ bit.

3.2 Multiplication

Polynomial multiplication is a basic building block for each of the three algorithms involved in BIKE. In the key generation h_1 is multiplied by h_0^{-1} (and several multiplications are executed in the inversion), in the encapsulation e_1 is multiplied by the public key h, and in the decapsulation a multiplier is required to compute the syndrome s. Our multiplier focuses on optimal BRAM usage as well as a good area-time product and is formally defined in Algorithm 6 in Appendix A. For this, we use the vector-matrix representation of the multiplication. Although the runtime of our multiplication is $\mathcal{O}\left(\left\lceil \frac{r}{b} \right\rceil^2\right)$, we benefit from carry-less and reduction-less multiplication in $\mathbb{F}_2$. We also considered using Karatsuba multiplication but current research shows that this is expensive with respect to the footprint [ZGF20].

A multiplication $c = m \cdot h$ requires the constant overhang $O = r \bmod b$, that is the number of bits in the polynomial's most significant word. Following our design strategy

$$\begin{array}{l}
c_0 = m_0 \cdot h_0 + m_1 \cdot h_9 + m_2 \cdot h_8 + m_3 \cdot h_7 + \ldots \\
c_1 = m_0 \cdot h_1 + m_1 \cdot h_0 + m_2 \cdot h_9 + m_3 \cdot h_8 + \ldots \\
c_2 = m_0 \cdot h_2 + m_1 \cdot h_1 + m_2 \cdot h_0 + m_3 \cdot h_9 + \ldots \\
\hline
c_3 = m_0 \cdot h_3 + m_1 \cdot h_2 + m_2 \cdot h_1 + m_3 \cdot h_0 + \ldots \\
c_4 = m_0 \cdot h_4 + m_1 \cdot h_3 + m_2 \cdot h_2 + m_3 \cdot h_1 + \ldots \\
c_5 = m_0 \cdot h_5 + m_1 \cdot h_4 + m_2 \cdot h_3 + m_3 \cdot h_2 + \ldots \\
\hline
c_6 = m_0 \cdot h_6 + m_1 \cdot h_5 + m_2 \cdot h_4 + m_3 \cdot h_3 + \ldots \\
c_7 = m_0 \cdot h_7 + m_1 \cdot h_6 + m_2 \cdot h_5 + m_3 \cdot h_4 + \ldots \\
c_8 = m_0 \cdot h_8 + m_1 \cdot h_7 + m_2 \cdot h_6 + m_3 \cdot h_5 + \ldots \\
\hline
c_9 = m_0 \cdot h_9 + m_1 \cdot h_8 + m_2 \cdot h_7 + m_3 \cdot h_6 + \ldots
\end{array}$$

Figure 2: Exemplary decomposition of the partial products for a multiplication with $r = 10$ and $b = 3$.

of processing b bits in parallel, the multiplier reads b bits of m and b bits of h such that $b \cdot b$ partial products are computed at the same time. This leads to a multiplication which is conducted column-wise, i.e., all partial products including the message's bits $m[i]$ are calculated before the next b bits of m are read from the BRAM. As an example, we graphically depicted the multiplication process for $r = 10$ and $b = 3$ in Figure 2. For every column consisting of $r \cdot b$ partial products, there are two initial steps: the first step computes the partial products of the upper triangle (in our example only $m_2 \cdot h_8$), the second step computes all partial products that include the current most O significant bits of h and all bits from $m[i]$ excluding the first bit (in our example $m_1 \cdot h_9$ and $m_2 \cdot h_9$). Afterwards, the algorithm proceeds with a regular flow. In each clock cycle the multiplier reads $h[j]$ and $c[j]$ from the BRAMs and computes the related partial products in the next clock cycle (illustrated by connected background colors). The lower b bits of the result are added to the intermediate result which was gained by the upper $b-1$ bits of the previous multiplication's result. These intermediate results are stored in registers in order to have direct access. As the authors in [VMG14], we also use the *read-first* setting of the BRAM modules. This setting allows to read a result from a specific address and afterwards to write a new value to the same address in the same clock cycle. Hence, new results from the multiplication engine, which are added to the current intermediate result $c[j]$, are stored in the BRAM at position $(j + 1) \bmod r$. However, since there are $\lceil r/b \rceil$ columns, the final result c is stored in the correct layout, i.e., $c[0]$ contains the LSBs of the final polynomial. Besides the result c, the polynomial h also rotates through the BRAM and needs to be tracked by the implementation. A special case is to determine $h[0]$ as it consists partly of $h[r-1]$ and partly of $h[r-2]$ (in our example $h[0] = (h_7, h_8, h_9)$ for the second column).

This structure performs a multiplication within $\lceil r/b \rceil \cdot (\lceil r/b \rceil + 3) + 1$ clock cycles. The additional three clock cycles in every column originate from the two initial steps described above and one additional clock cycle to read $h[0]$. The last additional clock cycle is only required to switch to a `DONE` state.

To this end, our design slightly outperforms the polynomial multiplier recently proposed by Hu et *al.* [HWCW19] whose implementation conducts a multiplication within $\lceil \frac{r}{b} \rceil^2 + 18 \lceil \frac{r}{b} \rceil - 9$ clock cycles. Our multiplier achieves a latency of $\lceil \frac{r}{b} \rceil^2 + 3 \lceil \frac{r}{b} \rceil + 1$ clock cycles with a slightly decreased linear part. Table 2 compares our design to the design by Hu et *al.* and to the multiplier proposed within the Round-2 submission of the BIKE specifications [ABB+19]. All results were generated for a Xilinx Artix-7 FPGA and for $r = 10\,163$ since Hu et *al.* reported their results for the parameter set of the second round submission of BIKE. While our implementation consumes slightly more hardware resources, the latency clearly decreases. However, the area-time product only shows considerably better results for $b = 32$ and $b = 64$.

Table 2: Comparison between different multiplier generated for an Artix-7 FPGA with $r = 10\,163$.

	Resources				**Performance**			
	Logic	Memory		Area	Cycles	Freq.	Latency	Area-Time
	LUT	FF	BRAM	Slices	Cycles	MHz	ms	Slices × ms
Round-2 Implementation [ABB+19]								
32 bit	87	53	3	40	3 252 161	416	7.818	312.72
Multiplier by Hu et al. [HWCW19]								
32 bit	N/A	N/A	2.5	219	106 839	205	0.521	114.099
64 bit	N/A	N/A	5	654	28 134	180	0.156	102.024
128 bit	N/A	N/A	7.5	1 596	7 831	150	0.052	82.992
This work								
32 bit	886	90	1.5	274	102 079	312	0.327	89.598
64 bit	2 384	119	3	740	25 759	277	0.093	68.82
128 bit	8 864	248	6	2 519	6 641	147	0.045	113.355

3.3 Inversion

With the decision of the BIKE team to only rely on the BIKE version being built upon the Niederreiter framework, a new challenge of implementing a polynomial inversion in hardware arose. Since BIKE is designed to work with ephemeral keys, an efficient implementation of an inversion algorithm is even more critical to achieve reasonable throughput. To this end, we decided to implement the inversion of a polynomial a in $\mathcal{R}$ using Fermat's Little Theorem as

$$a^{-1} = a^{2^{r-1}-2} \tag{3}$$

holds for every $a \in \mathcal{R}^*$ with $\mathrm{ord}(a) \mid 2^{r-1} - 2$. To exponentiate a target polynomial a with $2^{r-1} - 2$, we first rewrite the exponent as $2\left(2^{r-2} - 1\right)$. Eventually, the exponentiation is accomplished by Algorithm 5 which is based on the classic ITA [IT88] and a slightly adapted version of Algorithm 1 defined in [HGWC15]. Note that we do not follow the recently proposed algorithm by Drucker et *al.* [DGK20b] (which is used in the software reference implementation of BIKE) as it performs slightly worse in hardware. The number of required multiplications is the same for both algorithms but the number of squarings differs. Assuming that an exponentiation f^{2^t} is divided into a chain of operations of the form f^{2^k} with $k \in \mathcal{K}$ and $k \leq t$ where each operation has the same runtime (more details are given below), Algorithm 5 requires less of these operations than Algorithm 2 from [DGK20b] as shown in Table 3 for different sets $\mathcal{K}$. Additionally, the proposed algorithm by Drucker et *al.* would require one additional BRAM to hold the intermediate results res (cf. Algorithm 2, line 8 in [DGK20b]).

However, Algorithm 5 first executes the exponentiation of $\left(2^{r-2} - 1\right)$ described by lines 2-11 and eventually the final squaring from line 12. To this end, the inversion consists of exponentiations of the form f^{2^t}, of polynomial squarings, and of polynomial multiplications. The latter operation is realized by using the multiplier described in Section 3.2. The strategies to implement a squaring module and to realize the exponentiation with 2^t are described in the following.

Squaring Module for Fixed k An exponentiation of a polynomial f with 2^t for arbitrary t can always be accomplished by dividing the exponentiation into a chain of t squarings.

Algorithm 5: nversion based on the classic ITA [IT88, HGWC15].

```
Data: $r - 2 = (r_{q-1}, ..., r_0)$ with $r_i \in 0, 1$ and $a \in \mathcal{R}^*$
Result: $a^{-1}$
1  $f \leftarrow a$, $t \leftarrow 1$
2  for $i \leftarrow q - 2$ to $0$ do
3      $g \leftarrow f^{2^t}$
4      $f \leftarrow f \cdot g$
5      $t \leftarrow 2t$
6      if $r_i = 1$ then
7          $g \leftarrow f^2$
8          $f \leftarrow a \cdot g$
9          $t \leftarrow t + 1$
10     end
11 end
12 return $f^2$
```

Table 3: Comparison between Algorithm 5 and Algorithm 2 from [DGK20b] indicating the total amount of squaring operations.

$\mathcal{K}$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	Sum
Algorithm 2 from [DGK20b]					
$\{1\}$	12 355	0	0	0	12 355
$\{1, 2\}$	5	6 175	0	0	6 180
$\{1, 2, 3\}$	10	6	4 111	0	4 127
$\{1, 2, 3, 4\}$	5	1	0	3 087	3 093
*Algorithm 5 (used in **this work**)*					
$\{1\}$	12 321	0	0	0	12 321
$\{1, 2\}$	7	6 157	0	0	6 164
$\{1, 2, 3\}$	8	2	4 103	0	4 113
$\{1, 2, 3, 4\}$	6	2	1	3 077	3 086

One possibility to speed up the calculation is to implement a module which is able to perform $k < t$ squarings in the same time as a single squaring. Hence, a squaring chain would consist of $\lfloor t/k \rfloor$ k-squarings and $t \bmod k$ single squarings.

The strategy implementing squaring modules with fixed k pursues our global design consideration to achieve submodules which scales with b. A polynomial squaring $g = f^{2^k}$ for arbitrary k can be realized by a simple bit-permutation and is mathematically described by

$$g_i = f_{i \cdot 2^{-k} \bmod r} \tag{4}$$

where i denotes the i-th element in the target polynomial. Equation 4 indicates that for each b bits of the target polynomial g, bits from at least 2^k different addresses of the source polynomial f are required where the maximum number of different addresses is bounded by $2 \cdot 2^k - 1$. As an example, Figure 3 shows a draft of the permutation and corresponding memory pattern for a squaring with $k = 1$, $b = 8$, and $r = 59$. It is shown that bits from three different addresses are required in order to combine them to the correct result written to the first address implying that all necessary bits from f need to be loaded from the BRAM first. This is done in an initial phase which is automatically calculated to

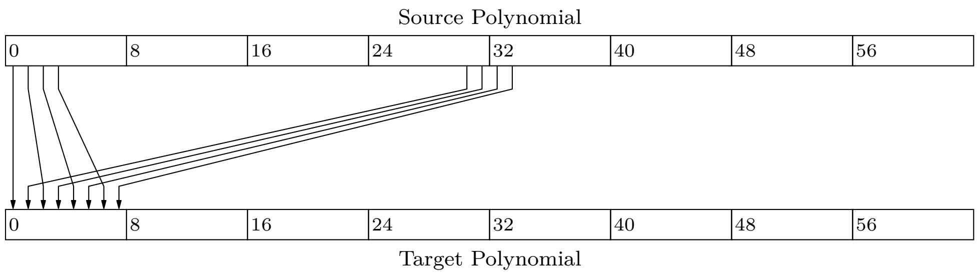


Figure 3: Exemplary permutation for a squaring module with $k = 1$, $r = 59$, and $b = 8$.

Table 4: Hardware utilization for different k-squaring modules setting $r = 12\,323$ and $b = 32$.

Resources	$k = 1$	$k = 2$	$k = 3$	$k = 4$
LUT	81	161	236	4 070
Register	105	186	346	820
Slices	38	62	118	1 124

be optimal by our scripts. Additionally, the scripts ensure that all upcoming results can be directly written to the BRAM containing the target polynomial by determining an optimal read sequence of bits from the source polynomial. The amount of clock cycles required for the initial phase also determines the number of b-bit registers holding the already read parts from the source polynomial. Note, after the initial phase, which depends on k and r, the squaring finishes within $\lceil r/b \rceil$ clock cycles.

Table 4 shows implementation results for four different squaring modules with fixed k and $b = 32$. Using modules generated for larger k linearly decreases the latency as described above but the implementation costs drastically increase. To this end, we only consider k-squaring modules with a maximum of $k = 4$.

Squaring Module for Arbitrary k Besides the above described strategy, we explore another approach implementing a squaring module which can accomplish a k-squaring (i.e., $g = f^{2^k}$) for arbitrary k within r clock cycles. For Algorithm 5 this approach is especially interesting for larger t as the exponentiation has not to be decomposed into a squaring chain but rather can directly be carried out. Figure 4 shows a schematic drawing of the hardware implementation and the corresponding operations required to compute the addresses of the source and target polynomial and the output data for the target polynomial g. The bits of the target polynomial are determined in an ascending order so that the corresponding bits from the source polynomial need to be computed by the implementation. Therefore, the module requires an input `INC` which needs to be assigned to $2^{-k} \bmod r$. Starting with 0, the implementation adds (modulo r) every clock cycle `INC` to the current value where the upper bits determine the address and the lower $\log(b)$ bits are used as a selection signal for a b-to-1 multiplexer. The input of the multiplexer is the current b-bit chunk of the source polynomial. After selecting the desired bit from the input, a barrel shifter is used to shift the desired bit to the correct position. The resulting b bits are than added (xored) to the current intermediate result destined for the target polynomial. After all b bits for a target address of g are collected and shifted to the correct position, the implementation writes the result to the BRAM.

For $r = 12\,323$ and $b = 32$ the above described approach requires just 45 slices partitioned into 96 LUTs and 80 registers. The utilization is very similar to the squaring

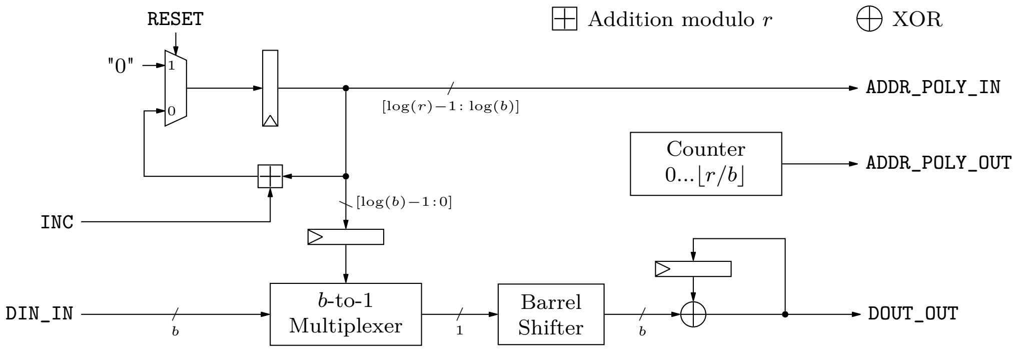
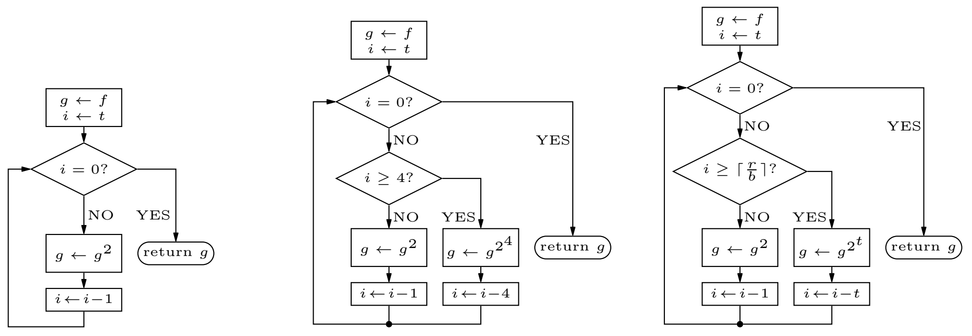


Figure 4: Schematic drawing of a module being able to perform a k-squaring for arbitrary k in r clock cycles.



(a) Strategy 1. (b) Strategy 2. (c) Strategy 3.

Figure 5: Different strategies to implement $g = f^{2^t}$ required for the polynomial inversion.

module for a fixed $k = 1$ proposed in the paragraph before.

Squaring Strategies Given the two different modules to compute a k-squaring, we investigate three optimization strategies to implement the exponentiation $g = f^{2^t}$ in Algorithm 5, line 3. The three approaches are depicted in flow charts in Figure 5. The first strategy only utilizes a squaring module for a fixed $k = 1$. In this case all exponentiations are carried out by chains of simple squarings. The second strategy implements two different but fixed squaring modules: one with $k = 1$ and the other one with $k = 4$. Hence, as long as t and the remaining exponent of the squaring chain is larger or equal four, the faster module is used. If the remaining exponent is smaller the squaring module with $k = 1$ is applied. The last strategy uses a combination of a fixed squaring module with $k = 1$ and the module being able to perform arbitrary k-squarings. In this way, all k-squarings with $k \geq b$ are executed by the latter module.

Note that all strategies have implemented a fixed squaring module with $k = 1$ because of two reasons: (1) simple squarings are always needed in the inversion process (cf. Algorithm 5, line 7 and line 12), and (2) it consumes just 42 slices and speeds up the computation notably.

Independently of the strategy, the inversion process requires four BRAMs. One BRAM stores the private key, i.e., (h_0, h_1). The other three BRAM modules are interchangeably used to perform a squaring chain (two BRAMs are used in alternation as source and target polynomial) and a subsequent multiplication by the squaring chain's input polynomial (cf. Algorithm 5, line 3 and line 4).

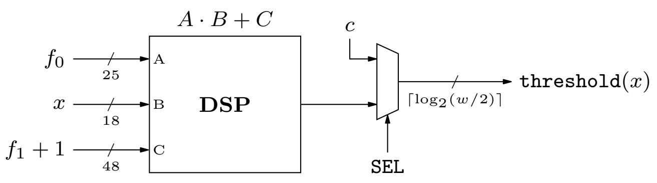


Figure 6: Schematic of the threshold computation.

3.4 Decoder

The BGF decoder mainly consists of three submodules. The first module is the threshold function described in Equation 1. Its argument $|(s + eH^{\mathrm{T}})|$ is computed by the second module. The third module flips the bits of the error vector e and generates the black and gray lists. In the following, we describe our implementation of these three modules.

Threshold Function The threshold for flipping a bit in the error vector is calculated with a multiplication followed by an addition with a constant term. We use Digital Signal Processor (DSP) instantiated with an output register stage as a straightforward implementation choice. In order to ensure that the bus widths of the input ports are used as optimal as possible, the corresponding VHDL-code is generated by a Sage script producing binary representations of the constants f_0 and f_1. The floor-function is realized by omitting all fractional digits from the result. As this procedure sustains a loss of precision, the script also checks if the result is still correct for all possible inputs x. Figure 6 illustrates the threshold module and the usage of the DSP, a multiplexer is used for the max()-function.

Hamming Weight The implementation of the Hamming weight module follows our design strategy to scale submodules with the parameter b. As for the threshold computation, we utilize DSPs with one register stage to add up all non-zero bits. To do so, each b-bit chunk $a = g[i]$ of a target polynomial g is separately feed into the module depicted in Figure 7. In $\log(b)$ stages all bits are added up where each stage consists of $\left\lceil \frac{b/2^j \cdot (j+1)}{48} \right\rceil$ DSPs[2] with $1 \leq j \leq \log(b)$. Hence, for each stage the full width of each DSP is utilized. For example, in the first stage each DSP can add up $48/2 \cdot 2$ bits so that for $b = 32$ only one DSP is necessary in the first stage. In total, the Hamming weight computation requires

$$1 + \sum_{j=1}^{\log(b)} \left\lceil \frac{b/2^j \cdot (j+1)}{48} \right\rceil$$

DSPs where the additional DSP is used to add up all intermediate results at the end.

Bit-Flipping The last module of the decoder is responsible for the bit-flipping of the error vector's bits, i.e., the functions `BFIter` and `BFMIter` from Algorithm 4. In our implementation we realize both functions in one module and select the modes of operations (i.e., `BFIter` producing the black and gray lists, `BFIter` without producing the lists, `BFMIter` processing the black mask, and `BFMIter` processing the gray mask) with a control signal `MODE`. The most interesting part is the process of counting the UPC equations which is depicted in Figure 8. We follow our design strategy and instantiate b counters in parallel where the `ENABLE` (`EN`) signals depend on the current part of the syndrome and the secret key. For storing the secret key, we decided to rely on a compact representation, i.e., only the positions of non-zero bits are stored instead of the entire polynomial. Hence,

[2]DSPs of Xilinx's 7-series are able to add at maximum two 48-bit numbers.

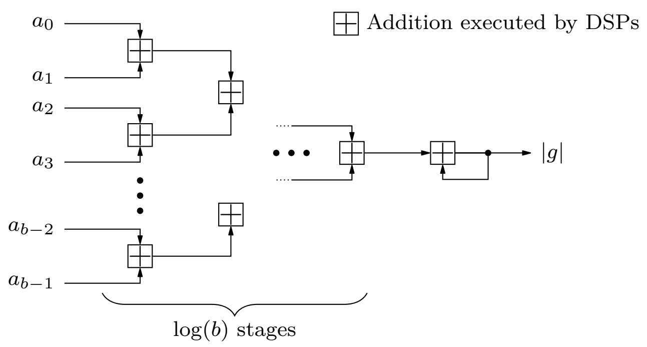


Figure 7: Hamming weight computation of a polynomial g divided into b-bit chunks a. In each stage, as many as possible additions are carried out by one DSP.

to determine the enable signals of all b counters in the same clock cycle, we compute the positions of the currently considered non-zero bit for the next $b-1$ columns (considering the secret key in its matrix representation) by adding the corresponding offsets (white adders) which would be gained when shifting the polynomial to the right. The position of the non-zero bit of the secret key is also used to read the corresponding chunk of the syndrome (depicted at the top in Figure 8). Here, we decide to duplicate the syndrome s and store a copy in a separate BRAM. This is necessary since we need b successive bits from s starting at the bit position determined by the current non-zero bit of the secret key which is not aligned with the layout of the BRAMs. For $r = 17$ and $b = 4$ this behavior is shown in the following example.

$$s_2\ s_1\ s_0\ s_{16}\ |\ s_{15}\ s_{14}\ s_{13}\ s_{12}\ |\ s_{11}\ s_{10}\ \underline{s_9\ s_8\ |\ s_7\ \overset{\downarrow}{s_6}}\ s_5\ s_4\ |\ s_3\ s_2\ s_1\ s_0$$

The arrow indicates the position of the current non-zero bit of the secret key and the underlined bits are required to determine the enable signals of the counters. As we can only read one chunk within one clock cycle, we decided to create the aforementioned copy of the BRAM storing the syndrome to achieve a lower latency and read both chunks within one clock cycle from two different memories. The careful reader may notice that the least significant bits of the syndrome in the example are also stored in the most significant chunk such that the chunk is completely filled with data. The least significant bits from s are copied to the most significant chunk in an initial phase each time the `BFIter` module is evoked. This is necessary in case the non-zero bit of the secret key (the arrow in the example) would point for example to s_{15}.

After a non-zero bit position is read from the BRAM, b is added and the result is written back to the memory for the next iteration, i.e., the next b columns. At the end of each `BFIter` execution the original secret key is restored as it is required for the next invoking. When the entire decoding module is receiving a secret key from the key generation, it stores the non-zero bit positions two times: in the lower 16 bits and in the upper 16 bits of each address position in the BRAM. Therefore, the restoring can easily be handled by copying the upper 16 bits to the lower 16 bits.

However, after each non-zero bit position is read once from the BRAM, the counter values can be evaluated and compared to the threshold T. In case a counter value exceeds T the corresponding bit is set. The resulting b bit vector is added to the current chunk of the error vector or is used to set the bits in the black list. The same procedure is applied for the gray list but with a threshold reduced by τ.

The `BFIter` function finishes in constant time and only depends on r, w, and b as shown in Equation 5.

$$\frac{w}{2} \cdot 2 \cdot \left\lceil \frac{r}{b} \right\rceil + 6 \cdot 2 \cdot \left\lceil \frac{r}{b} \right\rceil + 5 \tag{5}$$

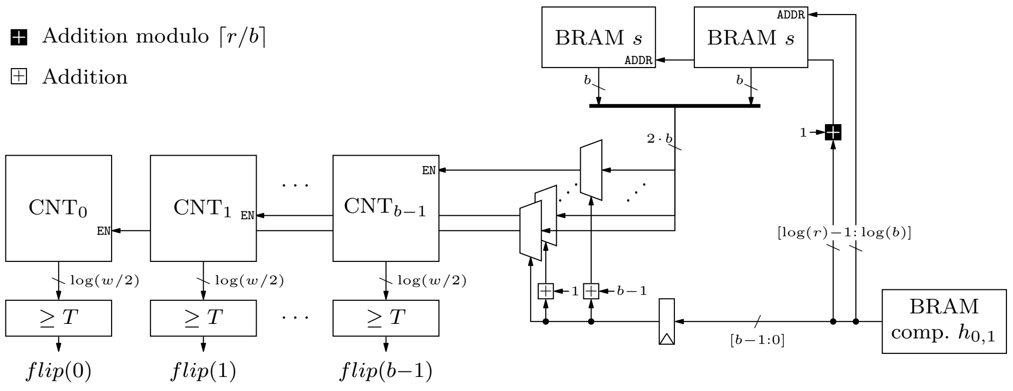


Figure 8: Extract of the bit-flipping module.

The hardware utilization for $r = 12\,323$ and $b = 32$ adds up to 355 slices composed of 280 registers and 1 125 LUTs.

3.5 Random Oracles

The BIKE specifications defines the three functions $\mathbf{H}, \mathbf{K}$, and $\mathbf{L}$ as random oracles [ABB+20]. $\mathbf{K}$ and $\mathbf{L}$ rely on a standard SHA384 core hashing m concatenated with C and hashing (e_0, e_1), respectively. It is assumed that all data is stored in byte arrays so that the input size to the SHA function is a multiple of eight. For our hardware design we implemented the SHA core in a straightforward way, i.e., as a round-based approach including retiming. The core consumes 1 162 slices (3 620 LUTs and 2 115 registers).

The $\mathbf{H}$ function relies on an AES256 core (instantiated in counter mode) where the input to $\mathbf{H}$ serves as 256-bit key. After one execution of the AES, the resulting ciphertext is used as randomness generating the error vectors. More precisely, the 128-bit output is divided into four 32-bit words which serve as inputs to the sampler described in Section 3.1. The realization of $\mathbf{H}$ utilizes 607 slices composed of 457 registers and 1 873 LUTs.

4 Implementation of the BIKE Scheme

This section covers the composition of the above described submodules to assemble the key generation, encapsulation, and decapsulation.

4.1 Key Generation

On the top level, the key generation consists of two samplers generating the private key (h_0, h_1). The resulting key is written to a generic BRAM module which automatically picks and connects the minimum number of required BRAM tiles based on the selected parameters r and b. The private key σ is generated by the sampler described in Section 3.1 and is stored in a 256-bit register. In order to generate the public key $h = h_1 h_0^{-1}$, one of the above introduced inversion modules is instantiated. The multiplication is also performed inside the inversion module as it already contains a multiplication engine. One additional task of the inversion module is to track the BRAM containing the final public key and to return it to the output.

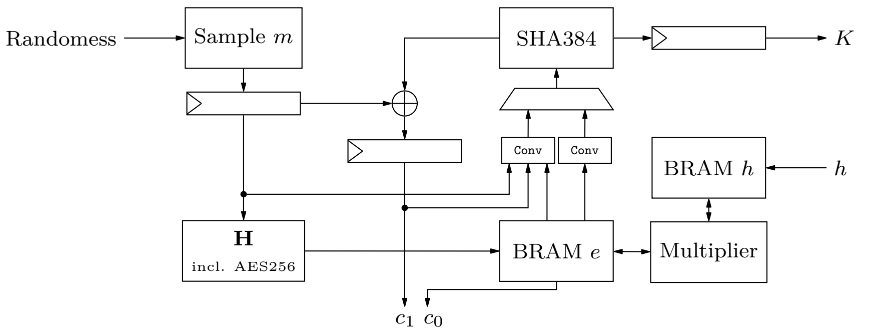


Figure 9: Top level view of the encapsulation module.

4.2 Encapsulation

Figure 9 shows a schematic of the top level of the encapsulation. To sample and store m, a uniform sampler and a 256-bit register is instantiated. The message m is used as input to $\mathbf{H}$ generating the error vector $e = (e_0, e_1)$. Afterwards, $c_0 = e_0 + e_1 h$ and $c_1 = m \oplus \mathbf{L}(e_0, e_1)$ are computed in parallel. A parallel computation is only possible due to an additional BRAM which is placed in the conversion module and stores a copy of e as input to $\mathbf{L}$. The final result of the multiplication is stored in the part of the BRAM which initially holds e_0. After $C = (c_0, c_1)$ was computed, the cryptogram and the message m is fed into a conversion module to generate the input to the SHA core realizing $\mathbf{K}$.

4.3 Decapsulation

The decapsulation uses most of the submodules including a sampler, multiplier, the decoder, and all three random oracles (see Figure 10). After transmitting the private key (h_0, h_1, σ) and the ciphertext $C = (c_0, c_1)$, the decapsulation is started by randomly sampling an error vector $e' = (e'_0, e'_1)$ with Hamming weight t. The syndrome $s = c_0 h_0$ is computed by using the multiplier introduced in Section 3.2. Afterwards, the algorithm invokes the decoder by using the `BFIter` module which forms the center of the decapsulation since it is connected to the most submodules. In case the Hamming weight module returns zero in the last iteration of `BFIter`, a *success* flag is raised indicating that the content of e'' is copied to e' and e'' is reseted (the current content is overwritten by zeros). However, if the *success* flag is zero, only the content of e'' is overwritten by zeros and no copying is performed. The content of e' is converted, forwarded to the SHA core, and added to c_1. The resulting message m' serves as key for the AES256 core generating an error vector which is stored in e'' and compared to the content in e'. In case the polynomials are equal, the implementation forwards m' to $\mathbf{K}$. Otherwise σ (part of the secret key) is used as input to $\mathbf{K}$.

5 Analysis

Based on the designs presented in Section 4, we now provide implementation results for Xilinx Artix-7 FPGAs for Level 1. The implementation results for Level 3 can be found in Appendix B. In the last paragraph of this section we compare our design to other hardware implementations of Key Encapsulation Mechanism (KEM) schemes.

5.1 Key Generation

Table 5 summarizes the implementation results for the key generation for all three introduced design strategies. Starting with Strategy 1, which utilizes only one squaring

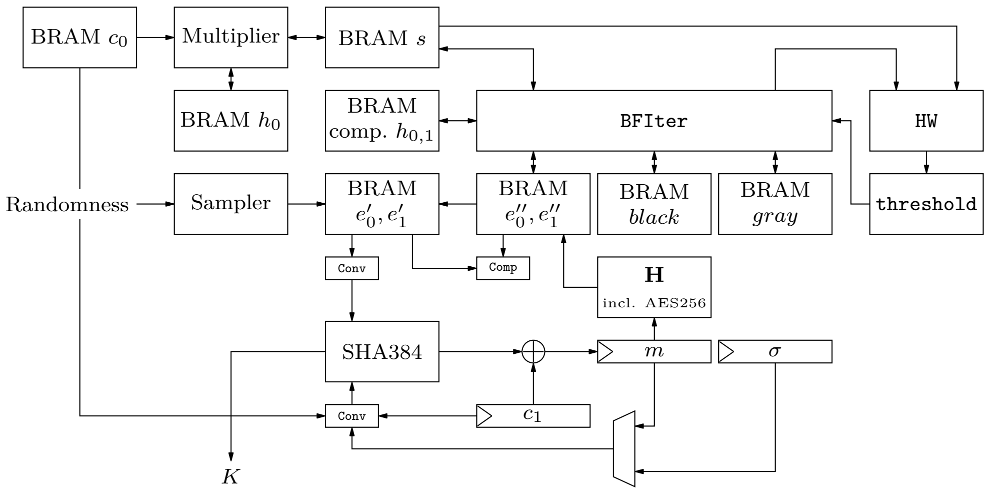


Figure 10: Top level view of the decapsulation module.

module, the implementation requires for $b = 32$ in average 7.37 million clock cycles[3] which corresponds to a latency of 56.75 ms for a maximum possible frequency of 129.87 MHz. The latency can roughly be decreased by a factor of four setting $b = 128$. However, the hardware utilization scales with a factor of five resulting in an area footprint of 3 354 slices. A better ratio between latency and resource utilization is achieved with Strategy 3. The utilization is very similar to the first strategy but the latency is notably decreased so that the implementation for $b = 128$ requires just 2.69 ms to finish one key generation by consuming 3 554 slices and 10 BRAMs. Hence, a distinct superiority is clearly visible.

5.2 Encapsulation

Table 6 summarizes the implementation results for the encapsulation module for $b \in \mathcal{B}$. Since the main part of the encapsulation is the multiplication to generate c_0, the implementation perfectly scales with b. For $b = 32$ the design requires 3 BRAMs and 2 133 slices while performing one encapsulation within 1.25 ms. Switching to $b = 128$, increases the hardware utilization roughly by a factor of two while the latency is decreased by a factor of twelve. The small increase of the hardware utilization originates from the relative large footprints of the SHA384 and the AES256 which stay constant for each b. Both modules consume together roughly 1 770 slices (cf. Section 3.5) which are 83 % of the whole design when setting $b = 32$.

5.3 Decapsulation

Table 7 summarizes the implementation results for the decapsulation. The main parts, i.e., the multiplier and the bit-flipping module, perfectly scale with the parameter b. Hence, increasing b from 32 bits to 128 bits lowers the latency from 13.02 ms to 1.90 ms by spending roughly three times more hardware recourses.

5.4 Comparison to Related Work and Discussion

Recently, Dang et *al.* published a paper comparing round 2 candidates of the NIST PQC standardization process [DFA+20]. We adjust their table which compares KEM

[3]The average number of clock cycles was determined by performing a simulation with one set of testvectors and adjust the resulting number of clock cycles by applying Equation 2.

Table 5: Implementation results for the key generation exploring the introduced strategies for $r = 12\,323$.

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Freq.	Latency
	LUT	FF	BRAM	Slices	Cycles	MHz	ms
Strategy 1							
32 bit	2 092	589	4	669	7 370 429	129.87	56.75
64 bit	3 607	631	5	1 046	3 070 613	125	24.56
128 bit	11 838	861	10	3 354	1 409 621	104	13.53
Strategy 2							
32 bit	6 982	1 396	4	1 986	3 804 192	131.58	28.91
64 bit	9 140	2 303	5	2 570	1 295 190	123.46	10.49
128 bit	23 801	4 567	10	6 742	520 374	106.38	4.89
Strategy 3							
32 bit	2 074	659	4	649	2 671 076	131.58	20.30
64 bit	4 432	735	5	1 285	748 964	113.64	6.59
128 bit	12 654	1 044	10	3 554	258 750	96.15	2.69

Table 6: Implementation results of the encapsulation module for Level 1 ($r = 12\,323$).

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Freq.	Latency
	LUT	FF	BRAM	Slices	Cycles	MHz	ms
32 bit	6 730	3 298	3	2 143	152 694	121.95	1.25
64 bit	8 253	3 327	5	2 538	40 368	121.95	0.33
128 bit	14 829	3 471	10	4 540	12 240	121.95	0.10

schemes by adding our work and listing only the code-based approaches. The evaluation is shown in Table 8. As for McEliece, we provide numbers for a lightweight and high-speed implementation. However, we did not create a composed design but determined the implementation results by using the prior presented results and just accumulating the numbers. Here, we assume that the AES and SHA modules are only instantiated once on the chip and that the encapsulation and decapsulation share them. For the lightweight approach we selected the designs for $b = 32$ and for the high-speed approach the designs generated for $b = 128$. The maximum frequency was determined by the slowest implementation among the three corresponding modules. Note that the BIKE-2 design from [HWCW19] is based on an older set of parameters (i.e., $r = 10\,163$). Furthermore, BIKE-2 did not specify any random oracles so that the footprint is notably smaller.

In case a hardware implementation of BIKE does not have to perform the key generation, encapsulation, and decapsulation in parallel, the design could further be optimized. Besides instantiating the AES and SHA module only once, one could also just implement one shared multiplier, shared register banks and shared BRAMs. This would drastically reduce the required hardware resources.

In Section 5.2 we already discussed the huge footprint of the random oracles. Hence, the choice of using AES and SHA as underlying building blocks appears not to be optimal for hardware implementations. To this end, we would suggest to use other standardized

Table 7: Implementation results of the decapsulation module for Level 1 ($r = 12\,323$).

	Resources					**Performance**		
	Logic		Memory		Area	Cycles	Freq.	Latency
	LUT	DSP	FF	BRAM	Slices	Cycles	MHz	ms
32 bit	9 557	7	3 969	10	3 055	1 627 942	125	13.02
64 bit	16 349	9	4 331	15	4 887	519 170	113.64	4.57
128 bit	30 977	13	5 092	29	8 862	189 615	100	1.90

cores like KECCAK which could be used as hash function (for **K** and **L**) and as random number generator (for **H**). This should reduce the overall footprint of a BIKE hardware implementation.

Table 8: Comparison to other code-based KEM schemes based on [DFA+20, Table 2].

							Key Gen		**Encaps**		**Decaps**	
Design	LUT	FF	Slices	DSP	BRAM	Freq.*	cycles†	μs	cycles†	μs	cycles†	μs
mceliece348864 [WSN18]	81 339	132 190	–	0	236	106	202.7	1 920.3	2.7	25.8	12.7	120.7
mceliece348864 [WSN18]	25 327	49 383	–	0	168	108	1 600	14 800	2.7	25.2	18.3	169.8
BIKE-2 [HWCW19]	3 874	2 141	1 312	0	10	160	2 150	13 437	–	–	–	–
This work	12 868	5 354	4 078	7	17	121	2 671	21 903	153	1 252	1 628	13 349
This work	52 967	7 035	15 187	13	49	96	259	2 691	12	127	189	1 972

* in MHz. † in thousand.

6 Conclusion

In this work we present a complete hardware implementation of the round 3 candidate BIKE submitted to the NIST PQC standardization process. Our implementation is scalable with respect to the used hardware resources and the corresponding latency while performing all operations in constant time (i.e., there is no dependency on secret values). As polynomial multiplications mainly determines the speed of the key generation and encapsulation, we use carry-less vector-matrix-multiplication with a short feedback path. For the key generation, we investigate three different implementation strategies where we demonstrate that one of them clearly outperforms the other ones. Additionally, we propose the first hardware implementation of the BGF decoder required in the decapsulation of BIKE. With all these improvements and optimizations we are able to implement a key generation which only takes 2.69 ms, an encapsulation which can be accomplished in 0.1 ms, and a decapsulation which finishes in 1.9 ms. Since multiplication is the most important operation with respect to performance, we suggest to investigate other approaches such as Karatsuba for high-speed implementations in future work.

References

[AAAS+19] Gorjan Alagic, Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process.* US Department of Commerce, National Institute of Standards and Technology, 2019.

[ABB+19] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 2 Submission. 2019. https://bikesuite.org/files/round2/spec/BIKE-Spec-Round2.2019.03.30.pdf.

[ABB+20] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Submission for Round 3 Consideration. 2020. https://bikesuite.org/files/v4.0/BIKE_Spec.2020.05.03.1.pdf.

[BOG19] Mario Bischof, Tobias Oder, and Tim Güneysu. Efficient Microcontroller Implementation of BIKe. In *International Conference on Information Technology and Communications Security*, pages 34–49. Springer, 2019.

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. Cryptology ePrint Archive, Report 2020/795, 2020. https://eprint.iacr.org/2020/795.

[DG19] Nir Drucker and Shay Gueron. A Toolbox for Software Optimization of QC-MDPC Code-Based Cryptosystems. *Journal of Cryptographic Engineering*, 9(4):341–357, 2019.

[DGK20a] Nir Drucker, Shay Gueron, and Dusan Kostic. Additional Implementation of BIKE (Bit Flipping Key Encapsulation). github, 2020. https://github.com/awslabs/bike-kem.

[DGK20b] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast Polynomial Inversion for Post Quantum QC-MDPC Cryptography. Cryptology ePrint Archive, Report 2020/298, 2020. https://eprint.iacr.org/2020/298.

[DGK20c] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC Decoders with Several Shades of Gray. In *International Conference on Post-Quantum Cryptography*, pages 35–50. Springer, 2020.

[Gal62] Robert Gallager. Low-Density Parity-Check Codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[Gam20] Jay Gambetta. IBMâĂŹs Roadmap For Scaling Quantum Technology. IBM Research Blog, 2020. https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/.

[HC17] Jingwei Hu and Ray CC Cheung. Area-Time Efficient Computation of Niederreiter Encryption on QC-MDPC Codes for Embedded Hardware. *IEEE Transactions on Computers*, 66(8):1313–1325, 2017.

[HGWC15] Jingwei Hu, Wei Guo, Jizeng Wei, and Ray CC Cheung. Fast and Generic Inversion Architectures Over $GF(2^m)$ Using Modified Itoh–Tsujii Algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, 2015.

[HVMG13] Stefan Heyse, Ingo Von Maurich, and Tim Güneysu. Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 273–292. Springer, 2013.

[HWCW19] Jingwei Hu, Wen Wang, Ray CC Cheung, and Huaxiong Wang. Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKe. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 231–234. IEEE, 2019.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF (2m) Using Normal Bases. *Information and computation*, 78(3):171–177, 1988.

[McE78] Robert J McEliece. A Public-Key Cryptosystem Based on Algebraic. *Coding Thv*, 4244:114–116, 1978.

[Mil85] Victor S Miller. Use of Elliptic Curves in Cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto. MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes. In *2013 IEEE international symposium on information theory*, pages 2069–2073. IEEE, 2013.

[Nie86] Harald Niederreiter. Knapsack-type Cryptosystems and Algebraic Coding Theory. *Prob. Control and Inf. Theory*, 15(2):159–166, 1986.

[RMGS20] Andrew Reinders, Rafael Misoczki, Santosh Ghosh, and Manoj Sastry. Efficient BIKE Hardware Design with Constant-Time Decoder. Cryptology ePrint Archive, Report 2020/117, 2020. https://eprint.iacr.org/2020/117.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho99] Peter W Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM review*, 41(2):303–332, 1999.

[VMG14] Ingo Von Maurich and Tim Güneysu. Lightweight Code-Based Cryptography: QC-MDPC McEliece Encryption on Reconfigurable Devices. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter cryptosystem using binary Goppa codes. In *International Conference on Post-Quantum Cryptography*. Springer, 2018.

[ZGF20] Davide Zoni, Andrea Galimberti, and William Fornaciari. Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials. *IEEE Access*, 8:75809–75821, 2020.

A Supplementary Material

Algorithm 6 formally describes our approach to implement the polynomial multiplication. The two initialization phases require each one clock cycle. Everything inside the *for*-loop iterating over j is executed in parallel.

Algorithm 6: Polynomial Multiplication.

```
Data: Input polynomials $h, m \in \mathcal{R}$ stored in BRAMs with a bus width of $b$ bits. Accessing $h[i]$
      corresponds to reading from address $i$ from the BRAM.
Result: Product $c = m \cdot h \in \mathcal{R}$ which is written to a BRAM.
$O \leftarrow r \bmod b$, $mask \leftarrow (2^b - 1)$, $addr \leftarrow \lceil r/b \rceil$
for $i \leftarrow 0$ to $addr - 1$ do
    $temp \leftarrow 0$
    /* Initialization Phase 1                                    */
    for $u \leftarrow O + 1$ to $b - 1$ do
        $temp \leftarrow temp \oplus ((m[i] >> u) \ \& \ 1) \cdot (h[addr - 2] >> (b + O - u))$
    end
    /* Initialization Phase 2                                    */
    $t \leftarrow (h[addr - 1] \ \& \ (2^O - 1)) << (b - O - 1)$;
    for $u \leftarrow 1$ to $b - 1$ do
        $temp \leftarrow temp \oplus ((m[i] >> u) \ \& \ 1) \cdot (t >> (b - 1 - u))$
    end
    /* Regular Flow                                              */
    $h' \leftarrow h[0]$, $tmp\_c\_add \leftarrow c[i]$
    for $j \leftarrow 0$ to $addr - 1$ do
        /* Parallel execution.                                   */
        $temp2 \leftarrow temp$
        $temp \leftarrow 0$
        for $u \leftarrow 0$ to $b - 1$ do
            $p \leftarrow \Big( ((m[i] >> u) \ \& \ 1) \cdot h' \Big) << u$
            $temp2 \leftarrow temp2 \oplus (p \ \& \ mask)$
            $temp \leftarrow temp \oplus ((p >> b) \ \& \ mask)$
        end
        $tmp\_c \leftarrow c[(j + i + 1) \bmod addr]$
        if $j = (addr - 1)$ then
            $c[(j + i + 1) \bmod addr] \leftarrow tmp\_c\_add \oplus \Big( temp2 \ \& \ \big(2^O - 1\big) \Big)$
            $h[0] \leftarrow \Big( \big( h' << (b - O) \big) \ | \ (h[j] >> O) \Big) \ \& \ mask$
        else
            $c[(j + i + 1) \bmod addr] \leftarrow tmp\_c\_add \oplus temp2$
            $tmp\_h \leftarrow h'$
            $h' \leftarrow h[j + 1]$
            $h[j + 1] \leftarrow tmp\_h$
        end
        $tmp\_c\_add \leftarrow tmp\_c$
    end
end
return $c$
```

B Implementation Results for Level 3

Table 9 shows the Level 3 implementation results for the third strategy investigated for the key generation, for the encapsulation, and the decapsulation.

Table 9: Implementation results for Level 3 ($r = 24\,659$).

	Resources					**Performance**		
	Logic		Memory		Area	Cycles	Freq.	Latency
	LUT	DSP	FF	BRAM	Slices	Cycles	MHz	ms
Key Generation (Strategy 3)								
32 bit	1 757	0	628	5	561	11 600 207	135.14	85.84
64 bit	4 580	0	801	5	1 303	3 089 329	111.11	27.80
128 bit	12 193	0	970	10	3 491	930 179	96.15	9.67
Encapsulation								
32 bit	6 436	0	3 305	5	1 982	601 099	121.95	4.93
64 bit	8 329	0	3 366	5	2 508	154 499	119.05	1.30
128 bit	15 004	0	3 441	10	4 376	42 173	125	0.34
Decapsulation								
32 bit	8 913	7	3 974	16	2 964	5 970 413	125	47.76
64 bit	16 606	9	4 377	16	4 949	1 805 881	121.95	14.81
128 bit	30 772	13	5 096	30	8 997	610 645	100	6.11