Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices

Jan Richter-Brockmann[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.
In our daily lives we constantly 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. Currently, this project is in the third round and one of the submitted candidates is the Key Encapsulation Mechanism scheme BIKE.
In this work we investigate different strategies to efficiently implement the BIKE algorithm on 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 104.04 ms for the decapsulation considering the first security level.

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

1 Introduction

Public-Key Cryptography (PKC) plays a crucial part of our contemporary life to exchange keys over insecure communication channels. However, established schemes like RSA [RSA78] and ECC [Mil85] are threatened by the far advanced development of quantum computers [Mos09]. In 1999, Peter Shor already presented an algorithm breaking PKC schemes in polynomial time on quantum computers [Sho99]. Knowing this threat, there was extensive research to find new schemes which are secure even in the presence of quantum computers. One research area is characterized by code-based cryptography where hard problems from coding-theory are used to achieve the desired resistance against quantum adversaries. The first cryptographic algorithm based on linear error codes was proposed by McEliece in 1978 [McE78]. Even though the McEliece cryptosystem is viewed to be secure against classical and quantum-based attacks, it suffers from the drawback of using large generator matrices serving as 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 its 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 the second round submission) is based on the Niederreiter framework including some tweaks [ABB+20].

After the announcement of the Post-Quantum Cryptography Standardization Project, the NIST published a list of selection-criteria including security, cost and performance, and algorithm and implementation characteristics on various platforms [AAAS+19]. Besides the reference implementation of BIKE, researchers implemented highly optimized software implementations for Intel CPUs [DGK20a] as well as an efficient microcontroller implementation [BOG19]. Until now, there is no complete hardware implementation of the third round submission of BIKE. In this work we will fill this gap proposing an optimized hardware design for Field-Programmable Gate Arrays (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 MyEcliece 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 resulting in a multiplication that finishes in r clock cycles where r defines the size of the public key.

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 includes internal memory (i.e., Block-RAM (BRAM)) to keep the amount of required 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 a hardware implementation for FPGAs 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 mainly 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 corresponding 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]. In the inversion algorithm 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 designs [HVMG13, 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 for the entire BIKE algorithm submitted to the third round of 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 us to a highly optimized design. The inversion module as well as other parts of BIKE require a polynomial multiplier. We slightly improved the multiplier proposed in [HWCW19] and achieved a design which is to the best of our knowledge the fastest one reported in literature. Additionally, we provide the first hardware implementation of the Black-Gray-Flip (BGF) decoder originally proposed in [DGK20c].

In our work we try to develop a hardware implementation which on the one hand can be instantiated on small devices (with a higher latency) and on the other hand is design for low-latency applications. This is accomplished by introducing a parameter which allows to scale our design. Additionally, we provide scripts written in SageMath in order to achieve a design which is completely generic with respect to all parameters used in BIKE. Besides the generation of the HDL-files, the scripts invoke the software reference implementation of BIKE to produce appropriated testvectors ensuring a correct functionality of our hardware design.

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. This section is followed by an overview about the entire hardware design described in Section 4. Before we conclude our work in Section 6, we evaluate our implementation with respect to required hardware resources and latencies in Section 5.

2 Preliminaries

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

Starting with specific notations, we define for a given polynomial v, $|v|$ as the Hamming weight of v. An uniform random sampling of a polynomial 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.

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 in the following.

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 $\mathcal{C} = \{mG | m \in \mathbb{F}_2^k\}$. A matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is called parity-check matrix of $\mathcal{C}$ if $\mathcal{C} = \{c \in \mathbb{F}_2^n | Hc^{\mathrm{T}} = 0\}$.

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.

Assuming a given 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 descried 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 $\varphi(A) = a_0 + a_1 X + ... + a_{r-1} X^{r-1}$. To this end, 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 $(v_0, v_1, ..., v_{r-1}) = \mathbf{v} \in \mathbb{F}_2^r$ so 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.

The structure of QC-MDPC codes allows to apply iterative decoders as proposed by Gallager in 1962 for Low-Density Parity-Check (LDPC) codes [Gal62].

2.2 BIKE

BIKE consists of three main algorithms namely the *key generation*, *encapsulation* and *decapsulation*. Dependent on the security level λ, three parameters r, w, and t are determined. 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 $w/2$ 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 $n = 2r$.

Besides this set of parameters, 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.

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

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 secrete size ℓ is fixed to 256 for both cases. For a more detailed description of the scheme, we refer the interested reader to the specification of BIKE [ABB+20].

Algorithm 1: Key Generation.

Input : BIKE parameters n, w, t, ℓ.
Output : Private key (h_0, h_1, σ) and public key h.

Generate $(h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ both of odd weight $|h_0| = |h_1| = w/2$.
Generate $\sigma \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
Compute $h \leftarrow h_1 h_0^{-1}$.
Return (h_0, h_1, σ) and h.

Algorithm 2: Encapsulation.

Input : Public key h.
Output : Encapsulated key K and ciphertext $C = (c_0, c_1)$.

Generate $m \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
Compute $(e_0, e_1) \leftarrow \mathbf{H}(m)$.
Compute $C = (c_0, c_1) \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$.
Compute $K \leftarrow \mathbf{K}(m, C)$.
Return (C, K).

Algorithm 3: Decapsulation.

Input : Private key (h_0, h_1, σ) and ciphertext $C = (c_0, c_1)$.
Output : Decapsulated key K.

Generate $(e'_0, e'_1) \xleftarrow{\$} \mathcal{R}^2$ with $|e_0| + |e_1| = t$.
Compute syndrome $s \leftarrow c_0 h_0$.
Compute $\{(e''_0, e''_1), \perp\} \leftarrow \texttt{decoder}(s, h_0, h_1)$.
if $(e''_0, e''_1) \leftarrow \texttt{decoder}(s, h_0, h_1)$ and $|(e''_0, e''_1)| = t$ **then**
  Set $(e'_0, e'_1) \leftarrow (e''_0, e''_1)$.
end
Compute $m' \leftarrow c_1 \oplus \mathbf{L}(e'_0, e'_1)$.
if $\mathbf{H}(m') \neq (e'_0, e'_1)$ **then**
  Compute $K \leftarrow \mathbf{K}(\sigma, C)$.
else
  Compute $K \leftarrow \mathbf{K}(m', C)$.
Return K.

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 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

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

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` and flipping all

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

bits in the error vector that were indicted by counter values exceeding the threshold T. Additionally, `BFIter` generates two lists, namely *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 used to adjust the error vector by applying the procedure `BFMaskedIter`. All parameters used to define the decoder are summarized in Table 1 for both security levels.

We would like to note that the description of the algorithm slightly differs from that presented in the specifications of BIKE [ABB+20]. The algorithm proposed in [DGK20c] and the software reference implementation of BIKE recomputes $s + eH^{\mathrm{T}}$ after a bit in e was flipped. Therefore, `ctr` always works with an updated sum (cf. line 17, 20 and 26) effecting the behavior and success rate of the decoder.

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 registers and therefore slices required to realize all submodules. 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 for e.g., security reasons. Additionally, we introduce a parameter b which is used to set the internally applied data width, to determine the bus width of all BRAMs, and to scale several submodules. Hence, 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 larger values would exceed the available hardware resources on Xilinx's Artix-7 FPGAs[1].

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.

Eventually, we do not aim to change or modify any specifications made for BIKE. Instead we implement everything as proposed in the software reference implementation meaning that we use the same realizations for e.g., the random oracles and the decoder. Thus, we can generate and extract testvectors from the reference implementation and can validate the output of our design.

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

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

Data: $H \in \mathbb{F}_2^{r \times n}$, $s \in \mathbb{F}_2^r$

$e \leftarrow 0^n$
for $i = 1$ **to** $NBIter$ **do**
  $T \leftarrow \texttt{threshold}(|s + eH^{\mathrm{T}}|)$
  $e, black, gray \leftarrow \texttt{BFIter}(s, e, T, H)$
  if $i = 1$ **then**
    $e \leftarrow \texttt{BFMaskedIter}(s, e, black, (d+1)/2+1, H)$
    $e \leftarrow \texttt{BFMaskedIter}(s, e, gray, (d+1)/2+1, H)$
  end
end
if $s = eH^T$ **then**
  return e
else
  return $\perp$
end
procedure $\texttt{BFIter}(s, e, T, H)$
for $j = 0$ **to** $n-1$ **do**
  if $\texttt{ctr}(H, s + eH^{\mathrm{T}}, j) \geq T$ **then**
    $e_j \leftarrow e_j \oplus 1$
    $black_j \leftarrow 1$
  else if $\texttt{ctr}(H, s + eH^{\mathrm{T}}, j) \geq T - \tau$ **then**
    $gray_j \leftarrow 1$
end
return $e, black, gray$
procedure $\texttt{BFMaskedIter}(s, e, mask, T, H)$
for $j = 0$ **to** $n-1$ **do**
  if $\texttt{ctr}(H, s + eH^{\mathrm{T}}, j) \geq T$ **then**
    $e_j \leftarrow e_j \oplus mask_j$
  end
end
return e

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 two 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 $x_{\mathrm{rand},i}$ of fresh randomness every two clock cycles with $i \in \{0, 1\}$. The input $x_{\mathrm{rand},i}$ determines the position in the polynomial h_i that should be set to one. 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

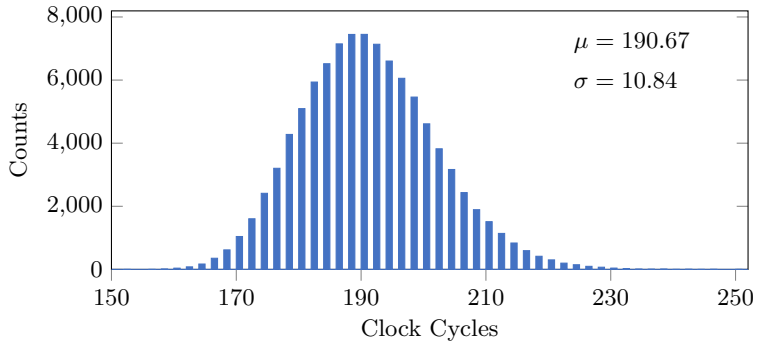


Figure 1: Distribution of required clock cycles to sample one polynomial of the secret key for $r = 12\,323$ and $w = 142$ based on 100 000 simulations.

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 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 a term $(1 - \frac{j-1}{r})$ 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 which are 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 written directly to 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 multiplication are

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

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

executed in the inversion), in the encapsulation the error vector e_1 is multiplied by the public key h, and in the decapsulation a multiplier is required to compute the syndrome s. In this work we present a multiplier which is to the best of our knowledge the fastest one reported in literature and formally defined in Algorithm 6 in Appendix A. As a constant parameter, the multiplication – computing $c = m \cdot h$ – requires the overhang $O = r \bmod b$ indicating the number of valid bits in the polynomial's most significant word stored in BRAM, e.g., $h\,[\lfloor r/b \rfloor]$. Following our design strategy 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 where the first step computes the partial products of the upper triangle which consists in our example only of $m_2 \cdot h_8$. 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 these are the products on the upper right emphasized by a darker background, i.e., $m_1 \cdot h_9$ 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 $\lceil r/d \rceil \cdot (\lceil r/d \rceil + 3) + 1$ clock cycles. The additional three clock cycles in every column originate from the two initial steps described above and one additional clock cycles 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 optimized 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	Frequency	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 $\text{ord}(a) \mid 2^{r-1} - 2$. 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 a slightly adapted version of the algorithm presented in [HGWC15] in order to work in GF(2) and not as originally proposed in GF(2^m). 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 on hardware. 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. One possibility to speed up the calculation is to implement a module which is able to

Algorithm 5: Polynomial Inversion.

Data: $r-2=(r_{q-1},...,r_0)$ with $r_i \in 0,1$ and $a \in \mathcal{R}^*$
Result: a^{-1}

```
1  f ← a, t ← 1
2  for i ← q − 2 to 0 do
3  |   g ← f^(2^t)
4  |   f ← f · g
5  |   t ← 2t
6  |   if r_i = 1 then
7  |   |   g ← f^2
8  |   |   f ← a · g
9  |   |   t ← t + 1
10 |   end
11 end
12 g ← f^2
13 return g
```

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 mod k single squarings.

The strategy implementing squaring modules with fixed k pursues our global design consideration to achieve submodules which are scalable 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. Hence, to write the result to the first address, all necessary bits from f need to be loaded from the BRAM first. This is done in an initial phase which is automatically calculated by our scripts to be optimal. 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 that after the initial phase, which depends on k and r, the squaring finishes within $\lceil r/b \rceil$ clock cycles.

Table 3 shows implementation results for four different squaring modules with fixed k and $b = 32$. Applying modules generated for larger k, linearly decrease 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

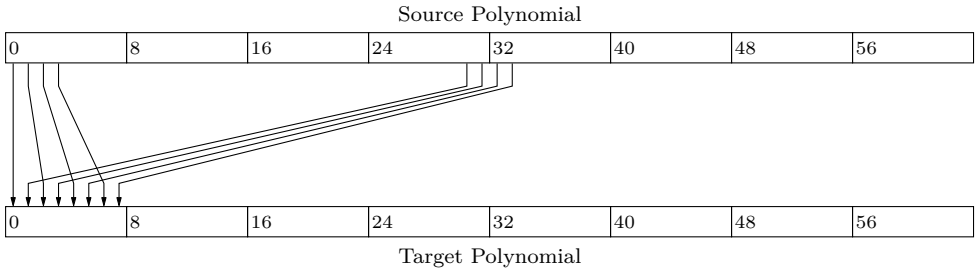


Figure 3: Exemplary permutation for a squaring module with $k = 1$, $r = 59$, and $b = 8$.

Table 3: 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

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} mod 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 which are partitioned into 96 LUTs and 80 registers. The utilization is very similar to the squaring module for a fixed $k = 1$ proposed in the paragraph before.

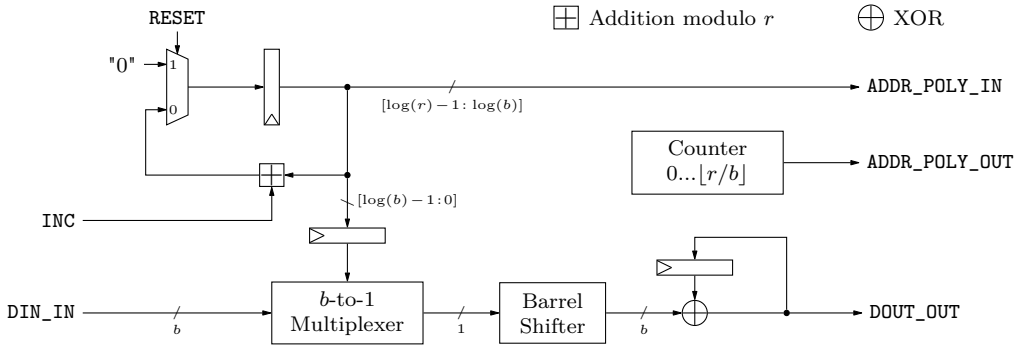
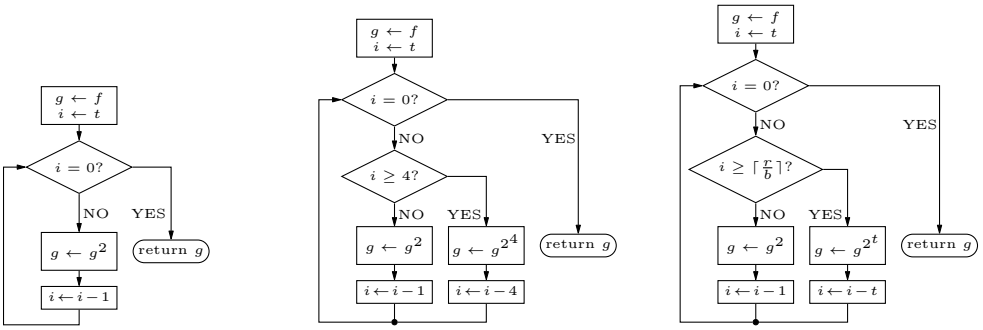


Figure 4: Schematic drawing of a module being able to perform a k-squaring for arbitrary k in r clock cycles.



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

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

Squaring Strategies Given the two different modules to compute a k-squaring, we investigate three different 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 than four 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 $k \geq \lceil r/b \rceil$ 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) the simple squaring module 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).

3.4 Decoder

The BGF decoder mainly consists of three different modules needed to be implemented in hardware. The first module is the threshold function described in Equation 1. The argument for the threshold function is the Hamming weight of the sum $(s + eH^{\mathrm{T}})$ which is computed by the second module. The third module is responsible for flipping the bits of the error vector e and to generate the black and gray lists. In the following we describe the hardware realizations of these three modules.

Threshold Function In order to determine the threshold, deciding whether a bit in the error vector is flipped or not, Equation 1 is applied. Since it consists of a multiplication followed by an addition with a constant term, using a Digital Signal Processor (DSP) instantiated with an output register stage seems to be the most promising option. 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 both constants. The ceiling-function is realized by producing the binary representation

of $f_1 + 1$ instead of just f_1 and by omitting all fractional digits from the result. As this procedure sustains a loss of precision, the script also checks if for all possible inputs x the result is still correct.

Hamming Weight The implementation of the Hamming weight module follows our design strategy scaling 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 6. 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 $1 \leq j \leq \log(b)$. All together, 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.

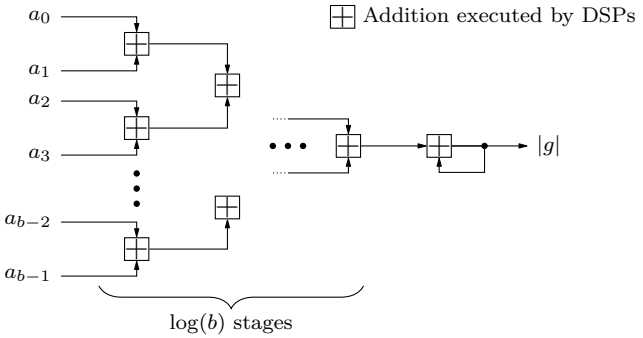


Figure 6: Hamming weight computation of a polynomial g divided into b-bit chunks a.

Bit-Flipping The last part of the decoder is a module which is responsible for the bit-flipping of the error vector's bits, i.e., the functions `BFIter` and `BFMaskedIter` 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, `BFMaskedIter` processing the black mask, and `BFMaskedIter` processing the gray mask) with a control signal `MODE`. In order to count the UPC equations, we decided to rely on a compact representation of the secret key. This allows a faster processing of `ctr` as the implementation just has to check $w/2$ addresses (non-zero bits in $h = (h_0, h_1)$). Unfortunately, the BGF decoder is a bit-oriented algorithm so that we cannot follow our design consideration of parallelizing and scaling the decoder with b. Instead we set all bus widths of related BRAMs to $b = 32$. However, in the following we briefly outline the hardware implementation of the decoder module which is sketched in Figure 7. For sake of clarity, we only draft the generation of the data and address signals of the private key, the syndrome, and the error vector and only show the process for one polynomial, i.e., for e_0 and h_0. The specified sizes for r allow to store the addresses of non-zero bits of h within 16 bits (for the specified values of r even less bits are required) which is done in the lower 16 bits of `SK_DIN`. Since the columns of h are processed each by each in an ascending order, the succeeding columns can be generated by the current column shifting them one position to the left (for the compact representation each value is subtracted by one modulo r). Note that we have to perform a left shift instead of a right

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

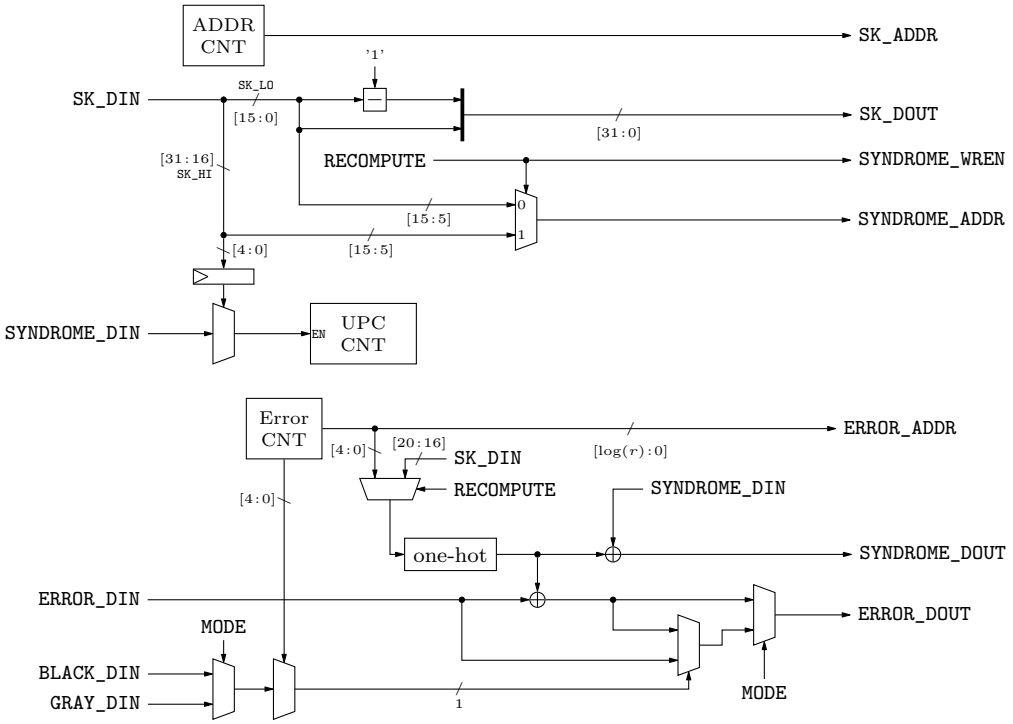


Figure 7: Extract of the bit-flipping module.

shift as we do not store the syndrome in a transposed form. In the upper part of the 32 bits, the current values are saved as they are required to recompute the syndrome in case `ctr` exceeds the threshold T. The addresses to read the current positions of non-zero bits of h are determined by an increasing counter.

The syndrome is used at two points: for calculating the UPC equations and in case `ctr` exceeds T and the syndrome needs to be updated. In case the UPC equations are computed, the address for reading 32-bit chunks of the syndrome are determined by the upper eleven bits of `SK_LO`. In case the syndrome – to be more precisely the sum $(s + eH^{\mathrm{T}})$ – needs to be updated, the address is determined by the upper eleven bits of `SK_HI`. The lower five bits of `SK_HI` are used as selection signal for a 32-to-1 multiplexer receiving `SYNDROME_DIN` as input and they are required to determine the bit position in a target chunk of the syndrome which need to be flipped. This is done by the *one-hot* encoding module and the subsequent addition with `SYNDROME_DIN`.

Every time `ctr` exceeds the threshold T, one bit in the error vector needs to be flipped. The address of the corresponding 32-bit chunk is determined by the *error counter* which is initialized with 0 and is decremented by one for every column. Again, this is necessary as the syndrome as well as the error vector are not stored transposed in the BRAM. Starting at zero and decrementing the counter, results in stepping through the error vector as it would be transposed (cf. Definition 6). To determine the output for the error vector `E_DOUT`, the implementation distinguishes between three cases: (1) the error bit is flipped due to `BFIter`, (2) the error bit is flipped caused by the black mask, and (3) the error bit is flipped caused by the gray mask. Eventually, the target bit needed to be flipped is determined by the lower five bits of the error counter and the subsequent *one-hot* encoding. Note that the black and gray masks are stored as polynomials, i.e., non-zero bits indicate the positions in the lists.

All in all, Equation 5 states the order of required clock cycles where N_{flips} denotes the

number of error bit flips conducted within the decoding procedure.

$$BF_{\text{cycle}} = (\tau + 2) \cdot n \cdot \frac{w}{2} + N_{\text{flips}} \cdot \frac{w}{2} \cdot 2 \tag{5}$$

Independent of the security level, the module invokes `BFIter` and `BFMaskedIter` seven times $(\tau + 2)$. In each iteration the algorithm iterates over all n columns of h where for each column $\frac{w}{2}$ clock cycles are required to access the stored positions of non-zero bits of h. For each bit flip, the implementation requires $\frac{w}{2} \cdot 2$ additional clock cycles. Note, this equation just provides a rough estimation and neglects additional clock cycles for initial phases to access memory or to switch states. Considering the high latency, the implementation is very lightweight and just consumes 159 slices divided into 415 LUTs and 80 registers.

3.5 Random Oracles

In the BIKE specifications the three functions $\mathbf{H}, \mathbf{K}, \mathbf{L}$ are defined as random oracles [ABB+20]. The functions $\mathbf{K}$ and $\mathbf{L}$ rely on a standard SHA384 core where $\mathbf{K}$ is used to hash m concatenated with the ciphertext C and $\mathbf{L}$ to hash the error vectors e_0 and e_1. 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. Since we did not optimize the core, it consumes 1 162 slices (3 620 LUTs and 2 115 registers).

The $\mathbf{H}$ function relies on an AES-256 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.

4.2 Encapsulation

Figure 8 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

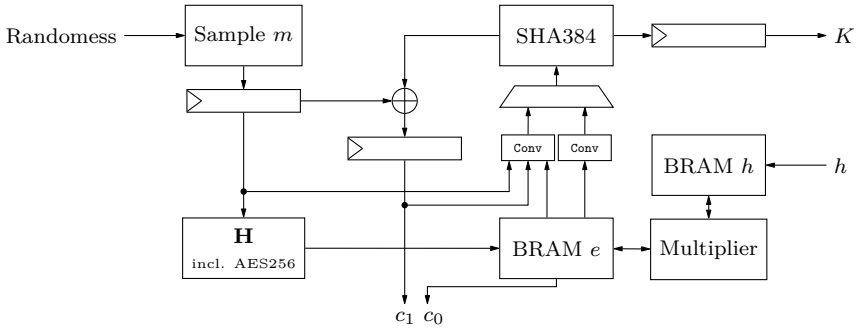


Figure 8: Top level view of the encapsulation module.

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 which realizes **K**.

4.3 Decapsulation

The decapsulation is composed of the most submodules including a sampler, multiplier, the decoder, and all three random oracles (see Figure 9). 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 **K**. Otherwise σ (part of the secret key) is used as input to **K**.

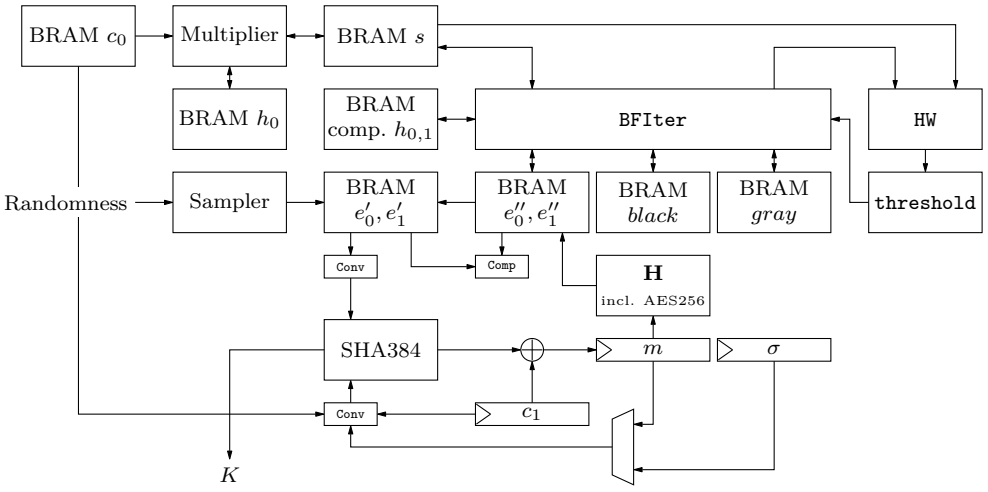


Figure 9: Top level view of the decapsulation module.

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 also compare our design to other Key Encapsulation Mechanism (KEM) schemes submitted to the NIST PQC project.

5.1 Key Generation

Table 4 summarizes the implementation results for the key generation for all three introduced design strategies. Starting with Strategy 1, which utilizes only one squaring module, the implementation requires for $b = 32$ in average 7.37 million clock cycles[3] which corresponds to 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 hardware 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.

The superiority of Strategy 3 is also shown in the Pareto plot in Figure 10 where the dependency between the required latencies in milliseconds and the amount of slices is depicted. Strategy 1 and Strategy 2 are clearly Pareto dominated by Strategy 3 for all considered b.

Table 4: Implementation results for the key generation exploring the introduced strategies for $r = 12\,323$.

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles (average)	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

[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.

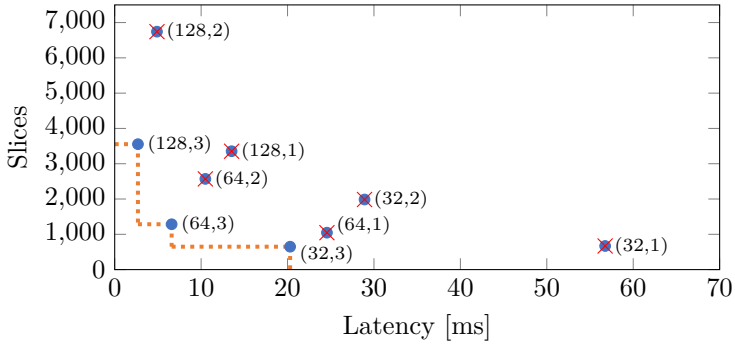


Figure 10: Dependency between the latency and required hardware resources. The first number in parentheses denotes b and the second number the optimization strategy. The dotted line marks the Pareto-front while all crossed points are Pareto dominated.

5.2 Encapsulation

Table 5 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 $b = 32$.

Table 5: Implementation results of the encapsulation module for Level 1 ($r = 12\,323$).

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles (average)	MHz	ms
32 bit	6 728	3 304	3	2 133	152 694	121.95	1.25
64 bit	8 291	3 333	5	2 560	40 368	121.95	0.33
128 bit	14 894	3 477	10	4 313	12 240	121.95	0.10

5.3 Decapsulation

Table 6 summarizes the implementation results for the decapsulation. Since only the multiplier scales with b and this computation represents only a small part of the entire decapsulation, the latency does not noticeably improve with increasing b. Actually, for $b = 128$ the latency increases as the maximum frequency decreases due to a more challenging routing of the multiplier. Hence, for the decapsulation the module with $b = 32$ achieves the best efficiency considering latency and hardware utilization.

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 extend their table which compares KEM schemes by adding our work but only listing the code-based approaches. The evaluation is shown in Table 7. As for McEliece, we provide numbers for a lightweight and high-speed

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

	Resources					**Performance**		
	Logic		Memory		Area	Cycles	Frequency	Latency
	LUT	DSP	FF	BRAM	Slices	Cycles (average)	MHz	ms (average)
32 bit	7 393	7	3 699	8	2 321	13 119 558	125	104.96
64 bit	9 005	9	3 727	8	2 774	13 005 385	125	104.04
128 bit	15 796	13	3 864	11	4 637	12 976 784	121.95	106.41

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 design we decided to select the key generation and encapsulation modules for $b = 128$ but use the decapsulation module for $b = 32$ since an increased b does not remarkably increase the throughput. 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 cores like KECCAK which could be used as hash function (for **K** and **L**) and as random number generator (for **H**). This would notably reduce the overall footprint of our implementation.

Table 7: 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	10 702	4 940	3 334	7	15	121	2 671	21 903	153	1 252	13 120	107 580
This work	29 448	5 498	8 419	7	28	96	259	2 691	12	127	13 120	136 443

* in MHz. † in thousand.

6 Conclusion

In this work we presented 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. As polynomial multiplications mainly determine the speed of the key generation and encapsulation, we designed an optimized multiplier achieving the lowest latency of proposed multipliers in the literature. For the key generation we investigated three different implementation

strategies where we demonstrated 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 were 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 104.96 ms.

Even if we highly optimized most parts of BIKE, we still see potential to decrease the latency of the BGF decoder which could be investigated in future work. In the current implementation the decoder does not significantly scale with the parameter b which, however, is a challenging task since all operations are bit-oriented and depend on previous steps.

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.

[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.

[Mos09] T Moses. Quantum computing and cryptography-their impact on cryptographic practice. *Entrust Inc., Dallas, TX*, 2009.

[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.

[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*, pages 77–98. Springer, 2018.

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**
    `/* The following block is executed in parallel on hardware. */`
    $temp2 \leftarrow temp$
    $temp \leftarrow 0$
    for $u \leftarrow 0$ **to** $b - 1$ **do**
      $p \leftarrow (((m[i] >> u)\ \&\ 1) \cdot h') << u$
      $temp2 \leftarrow temp2 \oplus (p\ \&\ mask)$
      $temp \leftarrow temp \oplus ((p >> b)\ \&\ mask)$
    end
    $tmp\_c \leftarrow c[(j + i + 1) \bmod addr]$
    if $j = (addr - 1)$ **then**
      $c[(j + i + 1) \bmod addr] \leftarrow tmp\_c\_add \oplus \left(temp2\ \&\ \left(2^O - 1\right)\right)$
      $h[0] \leftarrow ((h' << (b - O))\ |\ (h[j] >> O))\ \&\ mask$
    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 8 shows the Level 3 implementation results for all three strategies introduced in Section 3.3 for the key generation. Some of the reported numbers are slightly smaller than for the design generated for the first level. This mainly relates to different structures of the squaring modules in the inversion. For different r the number of initial phases and corresponding registers to hold the loaded chunks can vary (cf. Figure 3).

Table 9 summarizes the Level 3 implementation results for the encapsulation. The fastest design finishes within 0.35 ms while consuming 4 278 slices and 10 BRAMs.

Finally, the implementation results for the decapsulation are stated in Table 10. As for the first security level, the design for $b = 32$ achieves the best efficiency considering the latency and area utilization. To this end, the decapsulation process lasts 300.77 ms.

Table 8: Implementation results of the key generation for Level 3 ($r = 24\,659$).

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles (average)	MHz	ms
Strategy 1							
32 bit	1 522	557	5	507	30 447 899	138.89	219.33
64 bit	4 309	696	5	1 256	12 494 211	119.05	104.95
128 bit	11 856	794	10	3 288	5 576 559	104.17	53.53
Strategy 2							
32 bit	5 979	1 434	5	1 744	16 218 309	131.58	123.26
64 bit	12 653	2 263	5	3 489	5 354 781	119.05	44.98
128 bit	30 818	3 939	10	8 480	2 022 249	100.00	20.22
Strategy 3							
32 bit	1 757	628	5	561	11 600 207	135.14	85.84
64 bit	4 580	801	5	1 303	3 089 329	111.11	27.80
128 bit	12 193	970	10	3 491	930 179	96.15	9.67

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

	Resources				**Performance**		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles (average)	MHz	ms
32 bit	6 454	3 310	5	2 020	601 099	121.95	4.93
64 bit	8 350	3 371	5	2 606	154 499	121.95	1.27
128 bit	14 994	3 446	10	4 278	42 162	121.95	0.35

Table 10: Implementation results of the decapsulation module for Level 3 ($r = 24\,659$).

	Resources					**Performance**		
	Logic		Memory		Area	Cycles	Frequency	Latency
	LUT	DSP	FF	BRAM	Slices	Cycles (average)	MHz	ms (average)
32 bit	7 090	7	3 709	13	2 184	37 596 111	125	300.77
64 bit	8 894	9	3 769	13	2 778	37 147 593	125	297.18
128 bit	15 532	13	3 838	16	4 642	37 034 309	121.95	303.68