FPGA Key Generation Architectures for Post-Quantum Key Encapsulation Based on Quasi-Cyclic Codes

Jingwei Hu[1], Wen Wang[2], Ray Cheung[3], San Ling[1], and Huaxiong Wang[1]

[1] School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore, {davidhu, lingsan, HXWang}@ntu.edu.sg
[2] Department of Electrical Engineering, Yale University, USA, wen.wang.ww349@yale.edu
[3] Department of Electronic Engineering, City University of Hong Kong, Hong Kong, r.cheung@cityu.edu.hk

Abstract. In this paper, we present a constant-time FPGA-based key generator for the quantum-safe key encapsulation mechanism based on MDPC codes called BIKE which is among round-2 candidates for the NIST post-quantum standardization project. Existing hardware implementations focus on the encryption/decryption or key encapsulation/decapsulation feature while few of them take into account performance efficiency for the key generation which is particularly crucial for KEM-based applications. The solution proposed here aims at maximizing the metric of timing efficiency under a given resource restriction. New parallelized polynomial multiplication algorithms are proposed to achieve this goal. We show for instance that our highly optimized implementations for 128-bit security require 9.5×10^5 cycles (0.597 ms), 2.1×10^6 cycles (14.334 ms) and 9.8×10^5 cycles (0.492 ms) to generate a key pair for BIKE-1/2/3. The proposed key generator is also area-efficient, uses only 544/1272/423 slices for BIKE-1/2/3 on a small-size low-end Xilinx Spartan-7 FPGA.

Keywords: Post-Quantum Cryptography, Key Encapsulation Mechanism, Key Generator, QC-MDPC Code, FPGA Implementation

Introduction

Key encapsulation mechanisms (KEMs) are a class of encryption techniques designed to secure symmetric cryptographic key material for transmission using asymmetric (public-key) algorithms. Efficient and robust quantum-safe KEM design is a crucial and urgent topic in the cryptographic community. Recent years witnessed the NIST call for efficient and secure post-quantum KEMs in the post-quantum cryptography standardization competition [**?**]. The construction of a commercial quantum computer in not-so-distant future is a desperate threat to quantum-vulnerable primitives which are still relying on the hardness of the integer factorization problem or discrete logarithm problems such as the Diffie-Hellman key exchange, the Rivest-Shamir-Adleman (RSA) and Elliptic Curve

Cryptography. Shor's algorithm [**?**] can be deployed on a quantum computer to solve both the integer factorization problem and the discrete logarithm problem in polynomial time. Code-based cryptosystems, which build their security on the hardness of decoding general linear codes, are among the most promising quantum-resisting candidates for which no known polynomial time attack on a quantum computer exists.

McEliece proposed the first code-based cryptosystem in 1978 [**?**], which uses Goppa codes [**?**] as the underlying coding system. Goppa code-based schemes yield large public keys which limit the deployment of such systems in resource-constrained scenarios. Niederreiter proposed in 1986 another code-based system [**?**] which exploits the same trapdoor, but uses the syndromes and parity-check matrices to construct the ciphertext and the key. It has been proved that Niederreiter and McEliece are equivalent [**?**] and achieve the same security levels if the same family of codes is used.

Active research is focused on replacing Goppa codes with other families of structured codes that might lead to key size reduction. Nevertheless, this attempt may also compromise the system security. For example, some McEliece variants based on low-density parity-check (LDPC) codes [**?**], quasi-cyclic low-density parity-check (QC-LDPC) codes [**?**], quasi-dyadic (QD) codes [**?**], convolutional codes [**?**], generalized Reed-Solomon (GRS) codes [**?**], and rank-metric codes [**?**,**?**] have been successfully attacked. Nevertheless, some variants exploiting the quasi-cyclic form [**?**,**?**,**?**] have been shown to counteract the currently existing crypt-analysis attacks while achieving small-sized keys. A variety of KEMs built on the Niederreiter cryptosystem and LDPC/MDPC codes such as LEDAkem [**?**], CAKE [**?**], BIKE [**?**], also start to appear in the literature. BIKE and LEDAkem are currently on the round-2 candidate list for the post-quantum cryptography standardization. Recently, a new statistical attack, called reaction attack [**?**,**?**] has been devised to recover the key by exploiting the information leaked from decoding failures in QC-LDPC and QC-MDPC code-based systems. The reaction attack is further enhanced in [**?**] where the error pattern chaining method is introduced to generate multiple undecodable error patterns from an initial error pattern, thus improving the performance of such reaction attack in the CPA case. Nevertheless, the reaction attack only works for CPA systems and long-term keys, therefore, the deployment of a CCA2-secure conversion or ephemeral keys can resist the attack.

Related work. Cryptographic hardware for the classical McEliece/Niederreiter schemes based on Goppa codes has been extensively studied in the last decade. In 2009, the first FPGA-based implementation of McEliece's cryptosystem was proposed targeting a Xilinx Spartan-3 FPGA and encrypted and decrypted data in 1.07 ms and 2.88 ms, using security parameters achieving equivalence of 80-bit symmetric security [**?**]. The authors of [**?**] presented another accelerator for McEliece encryption with binary Goppa codes on a more powerful Virtex5-LX110T, which is capable to encrypt and decrypt a message in 0.5 ms and 1.3 ms providing a similar level of security, respectively. Another publication [**?**] based on hardware/software co-design on a Spartan3-1400AN

decrypts a message in 1 ms at 92 MHz with the same level of security. Heyse and Güneysu in [**?**] report that a Goppa code-based Niederreiter decryption operation consumes 58.78 μs on a Virtex6-LX240T FPGA for $n = 2048$ and $t = 27$ for 80-bit symmetric security. Wen Wang *et.al.* present an FPGA-based key generator for the Goppa code-based Niederreiter cryptosystem [**?**] and later a full implementation [**?**] which includes modules for encryption, decryption, and key generation. Their designs generate a key pair in $3.98ms$ for 256-bit security on a Xilinx Ultrascale+ FPGA.

The first implementation of MDPC code-based McEliece cryptosystem on embedded devices was presented in [**?**] in 2013. For 80-bit security, it is reported to run decryption in 125 μs with over 10,000 slices on Xilinx Virtex-6. A lightweight MDPC-McEliece has been implemented on FPGAs by sequentially manipulating cyclic rotations of the private key in block RAMs [**?**]. This lightweight design achieves circuit compactness which costs 64 slices for encryption and 148 slices for decryption on low-end Xilinx Spartan-6 device. An area-time efficient MDPC-Niederreiter has been implemented on FPGAs, which exploits a resource balanced MDPC decoding unit [**?**]. Their experimental results show the new architecture decrypts a message in 65 μs by using about 8,000 slices on a Virtex-6 FPGA.

Contributions. Current progress focuses on key encapsulation/decpasulation. However, it should be considered to have a closer look at the key generation as well. KEMs are used for ephemeral key encapsulation and not for public-key encryption. Therefore, the key generation is used as often as the encapsulation and decapsulation, *i.e.*, we do have a strong motivation to accelerate the key generation since KEMs use ephemeral keys which require updating keys frequently. As a consequence, this paper studies for the first time-efficient and compact FPGA-based key generation for a promising post-quantum KEM using MDPC codes (BIKE). The contributions include:

- Constant time executions for all critical operations protect the proposed design from potential timing attacks.
- The timing scores (9.55×10^4, 2.15×10^6, and 9.85×10^4 clock cycles for BIKE-1, BIKE-2, and BIKE-3, respectively) of our design outperform other embedded hardware and even powerful CPU with the dedicate carry-less multiplication instruction in terms of cycle counts.
- A fast and lightweight hardware architecture of BIKE for all secure parameters from 128-bit to 256-bit security.
- New polynomial multiplication algorithms (sparse-times-dense multiplication and generic algorithm) over ring $\mathbb{F}_2[x]/\langle x^r + 1\rangle$ optimize the timing performance of key generation.

Outline. The paper is organized as follows. In Section 2, we describe BIKE. In Section 3, we present our design methodologies for implementing BIKE on reconfigurable devices. In particular, our new approaches for polynomial multiplication and squaring are detailed to achieve time efficiency and area efficiency. In Section 4, a generic lightweight architecture for BIKE hardware is presented and discussed. In Section 5, experimental results on Xilinx FPGAs are shown to

Table 1: Parameters for BIKE according to NIST defined security category, referenced from [**?**]

Category		**n**	**r**	**w**	**t**	**SL**	**DFR**
1	BIKE-[1,2]	20,326	10,163	142	134	128	$\leq 10^{-7}$
	BIKE-3	22,054	11,027	134	154		
2-3	BIKE-[1,2]	39,706	19,853	206	199	192	$\leq 10^{-7}$
	BIKE-3	43,366	21,683	198	226		
4-5	BIKE-[1,2]	65,498	32,749	274	264	256	$\leq 10^{-7}$
	BIKE-3	72,262	36,131	266	300		

demonstrate the efficiency of the proposed techniques and hardware. In Section 6, some conclusions are drawn.

1 Preliminaries for BIKE/LEDAkem Key Generation

1.1 General Definitions

Polynomial View:	
$\mathbb{F}_2$	Finite field of 2 elements
$\mathcal{R}$	The cyclic polynomial ring $\mathbb{F}_2[x]/\langle x^r+1\rangle$
$\lvert x\rvert$	The Hamming weight of a binary polynomial x
$u \xleftarrow{\$} U$	Variable u is sampled uniformly at random from set U
Matrix View:	
$\mathcal{V}$	The vector space of dimension n over $\mathbb{F}_2$
$\mathbf{m}$	The vector over $\mathbb{F}_2$
$\mathbf{G}$	The matrix over $\mathbb{F}_2$

Table 2: Notations

Definition 1. *(Circulant Matrix) Let* $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{F}^n$. *The circulant matrix induced by* $\mathbf{x}$ *is defined and denoted as follows:*

$$\mathbf{rot}(\mathbf{x}) = \begin{bmatrix} x_0 & x_{n-1} & \cdots & x_1 \\ x_1 & x_0 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1} & x_{n-2} & \cdots & x_0 \end{bmatrix}$$

As a consequence, the product of any two polynomials x and y in $\mathcal{R}$ can be viewed as a usual vector-matrix (or matrix-vector) product using the operator $\mathbf{rot}(\cdot)$ as:

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x} \cdot \mathbf{rot}(\mathbf{y})^T = \mathbf{y} \cdot \mathbf{rot}(\mathbf{x})^T = \mathbf{y} \cdot \mathbf{x}$$

Definition 2. *(Linear Code) A Binary Linear Code $\mathcal{C}$ of length n and dimension k (denoted as $[n, k]$) is a subspace of $\mathcal{V}$ of dimension k. Elements of $\mathcal{C}$ are referred to as codewords.*

Definition 3. *(Generator and Parity-Check Matrix)* $\mathbf{G} \in \mathbb{F}^{k \times n}$ *is a Generator Matrix for the* $[n, k]$ *code* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{\mathbf{mG} | \mathbf{m} \in \mathbb{F}_2^k\}$$

$\mathbf{H} \in \mathbb{F}^{(n-k) \times n}$ *is called a Parity-Check Matrix of* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{\mathbf{c} \in \mathbb{F}_2^n | \mathbf{Hc}^T = 0\}$$

Definition 4. *(Quasi-Cyclic Codes) 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 $k_0 r$*

For instance, the generator matrix of a $(3, 1)$-QC code is formed by three circulant blocks $\mathbf{G_i}$:

$$\mathbf{G} = [\mathbf{G_0}|\mathbf{G_1}|\mathbf{G_2}]$$

In polynomial view, each block $\mathbf{G_i}$ can be represented (one-to-one mapping) as a polynomial g_i over $\mathcal{R}$ and thus the generator matrix $\mathbf{G}$ can be viewed as the polynomial matrix $[g_0|g_1|g_2]$ over $\mathcal{R}$. In all aspects, any codeword $\mathbf{c} = \mathbf{m} \cdot \mathbf{G}$ over $\mathbb{F}^n$ can be represented in its polynomial form as $[mg_0|mg_1|mg_2]$ over $\mathcal{R}$. All the above notations are collected in Table **??**.

1.2 BIKE

A novel code-based key encapsulation mechanism using MDPC codes, called BIKE [**?**], was submitted by Aragon *et. al* in 2017. The use of a QC-MDPC code $C(n_0, k_0)$ in BIKE allows key size reduction with respect to unstructured codes. Specifically, BIKE uses the double-circulant structure for the parity check matrix where two smaller cyclic matrices are included. The authors propose nine instances of BIKE (BIKE-1, BIKE-2, BIKE-3) as shown in Table **??**, according to the NIST call for Post-Quantum Cryptography Standardization which defines 5 security categories characterized by increasing security strength [**?**]. QC construction brings a beneficial reduction of memory storage required for public keys, for which only the first row/column of each circulant block is stored and the remaining part can be recovered by cyclic rotations of this row/column, minimizing the actual key size to be comparable with other post-quantum cryptographic candidates. BIKE currently is one of the second round candidate algorithms of the NIST post-quantum standardization project.

The primitives of BIKE are described next. Algorithm **??** describes the key generation part of BIKE-1. The private keys comprise of two 'sparse' polynomials h_0 and h_1 where $|h_0|$ and $|h_1|$ is restricted to $w/2$. The public keys (f_0, f_1) are exposed for public use and therefore must scramble the structure of h_0 and h_1.

This is done by first multiplying h_0 or h_1 by a random dense polynomial g to get gh_0 and gh_1.

Algorithm **??** describes BIKE-2 key generation. Note that BIKE-2 does not generate any random polynomial to hide the secret key h_0 or h_1. Instead, it computes $h = h_0^{-1}h_1$ as the public key by using polynomial inverse. As a result, the public key size is reduced by half, however, the price to pay is the computational involving operation for inverse.

Algorithm **??** describes the key generation part of BIKE-3. To further improve BIKE-1, BIKE-3 uses only one multiplication and one addition to compute the first half of public key $f_0 = h_1 + gh_0$. The second half f_1 equals to the random dense polynomial g which requires no multiplication to compute.

Input: λ, the target quantum security level.
Output: the sparse private key $\mathbf{PK} = (h_0, h_1)$ and the dense public key $\mathbf{SK} = (f_0, f_1)$

1 Given λ, set the parameters r, w as described above
2 Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
3 Generate $g \leftarrow \mathcal{R}$ of odd weight (so $|g| \approx r/2$)
4 Compute $(f_0, f_1) \leftarrow (gh_1, gh_0)$
5 **return** *PK and SK*

Algorithm 1: BIKE-1 Key Generation in Polynomial View [**?**]

Input: λ, the target quantum security level.
Output: the sparse private key $\mathbf{PK} = (h_0, h_1)$ and the dense public key $\mathbf{SK} = h$

1 Given λ, set the parameters r, w as described above
2 Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
3 Compute $h \leftarrow h_1 h_0^{-1}$
4 **return** *PK and SK*

Algorithm 2: BIKE-2 Key Generation in Polynomial View [**?**]

2 Design parameters for BIKE on lightweight oriented configurable devices

In this section, we introduce techniques to realize BIKE key generation on small, embedded systems. In particular, we thoroughly describe 1. the memory organization for polynomials used in BIKE 2. constant-time yet fast polynomial inverse, squaring and multiplication algorithms used in BIKE 3. efficient generation of sparse/dense polynomials with prescribed Hamming weight.

Input: λ, the target quantum security level.
Output: the sparse private key $\mathbf{PK} = (h_0, h_1)$ and the dense public key $\mathbf{SK} = (f_0, f_1)$
1 Given λ, set the parameters r, w as described above
2 Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
3 Generate $g \leftarrow \mathcal{R}$ of odd weight (so $|g| \approx r/2$)
4 Compute $(f_0, f_1) \leftarrow (h_1 + gh_0, g)$
5 **return** *PK and SK*

Algorithm 3: BIKE-3 Key Generation in Polynomial View [**?**]

2.1 Data Structure

In this paper, two formats are used to represent polynomials in the quotient ring $\mathbb{F}_2[x]/(x^r+1)$. One is the dense format which stores every coefficient of the given polynomial $f = f_0 + f_1 x + \ldots + f_{r-1}x^{r-1}$. For example, the public keys (f_0, f_1) are dense polynomials and can be described by the dense format. Small endian notation is used here, *i.e.*, f_0 is first stored, then f_1, and finally f_{r-1}. For FPGA-based implementations, the coefficients f_i of f are typically stored in block RAMs, each address of which keeps multiple bits (word) of data. Let us define such data structure for a polynomial f as $\overline{\mathbf{f}}$ where $\overline{\mathbf{f}[i]}$ denotes the word (d bits in total) stored in the i-th address:

$$(f_{id}, f_{id+1}, \ldots, f_{id+d-1})_2$$

By using this form, a total of $\lceil r/d \rceil$ words is needed to represent a polynomial over $\mathbb{F}_2[x]/(x^r+1)$ but the last word $\overline{\mathbf{f}}[\lceil r/d \rceil - 1]$ is incomplete, which is padded by zeros as $(f_{(\lceil r/d \rceil - 1)d}, \ldots, f_r, \ldots, 0)_2$. Fig **??** illustrates the data structure of $\overline{\mathbf{f}}$.

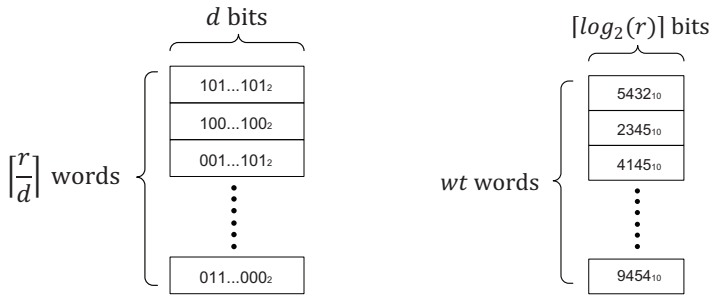
The other is the sparse format which only saves the bit positions of non-zero coefficients of f. For example, the secret keys (h_0, h_1) are sparse polynomials and can be described well by the sparse format. Let us define this sparse form for a polynomial f as $\hat{\mathbf{f}}$ where $\hat{\mathbf{f}}[i]$ denotes the non-zero bit position (belongs to the set $\{i | f_i = 1\}$, each item of this set is $log_2(r)$ bits) stored in the i-th address. Such format is memory-efficient which costs $|f|$ words and each word is $log_2(r)$ bits long to represent the given polynomial f if the polynomial weight is small.

2.2 Polynomial Arithmetics

Polynomial Inverse

Theorem 1. *Let a be an invertible polynomial in the quotient ring $\mathbb{F}_2[x]/(x^r+1)$ where r is a prime, then the inverse of polynomial a can be computed as $a^{-1} = (a^{2^{r-2}-1})^2$*

According to Theorem 1, we can compute the polynomial inverse over ring by exponentiation. Next, we describe our method to fast compute such exponentiation. Let us define the function $\beta_k(a) = a^{2^k-1}$, it is easy to see that



(a) Polynomial represented in dense format

(b) Polynomial represented in sparse format

Fig. 1: Polynomial in $\mathbb{F}_2[x]/(x^r+1)$ represented in different formats

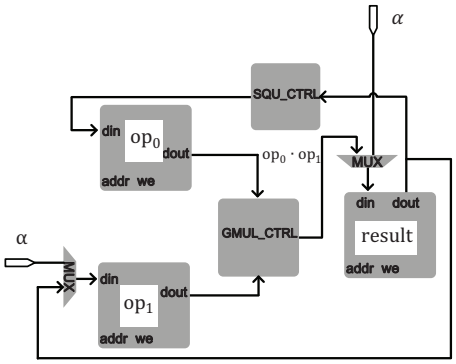


Fig. 2: Detailed polynomial inversion architecture

$a^{-1} = (\beta_{r-2}(a))^2$ in the quotient ring $\mathbb{F}_2[x]/(x^r+1)$. The following recursive formula holds:

$$\beta_{k+j}(a) = (\beta_k(a))^{2^j}\beta_j(a)$$

With the recursion above, we can generate $\beta_{r-2}(a)$ from $\beta_1(a) = a$ via an addition chain in constant steps, which is known as Itoh-Tsujii Inversion algorithm (ITA). More importantly, the building blocks for ITA are squaring and multiplication which are highly optimized in our work. Another approach for inverse is via Extended Euclidean algorithm (EEA). In [**?**], EEA with constant flow is proposed which may be exploited for secure polynomial inverse. However, the steps in the proposed algorithm are data-dependant and thus is not suitable to parallelize on hardware. In summary, we implement the algorithmic descrip-

Input: $r-2=(r_{q-1}\dots r_0)_2$ and $\alpha \in \mathbb{F}_2[x]/(x^r+1)$.
Output: $\beta = \alpha^{-1}$
1 $\beta \leftarrow \alpha$
2 $t \leftarrow 1$
3 **for** $i \leftarrow q-2$ ***to*** 0 **do**
4 $\quad \beta \leftarrow (\beta)^{2^t} \cdot \beta$
5 $\quad t \leftarrow 2t$
6 $\quad$ **if** $r_i = 1$ **then**
7 $\quad\quad \beta \leftarrow (\beta)^2 \cdot \alpha$
8 $\quad\quad t \leftarrow t+1$
9 $\beta \leftarrow \beta^2$
10 **return** β

Algorithm 4: Itoh-Tsujii Inversion Algorithm (ITA) [**?**]

tion for ITA from [**?**] as shown in Alg. **??**. An illustrative example for computing the inverse used in BIKE-2, 128-bit security is listed in Table **??**. From this table, it is seen that ITA is not only fast (20 multiplications + 20 squarings) but also performs in constant time. Fig. **??** depicts the overview of the ITA inverter. The initial polynomial β is uploaded to RAM result and raised to β^{2^t} via SQU_CTRL which is the module for squaring. This result is then stored to RAM op0. GMUL_CTRL performs the generic multiplication (step 4,7, Alg. **??**). After the final squaring (step 10), the inverted polynomial appears in RAM op0. In the next subsections, we detail the squaring and the generic multiplication used in ITA.

Squaring in $\mathbb{F}_2[x]/(x^r+1)$ First, consider the direct squaring $a^2(x)$ for any polynomial $a(x) \in \mathcal{R}$:

$$a^2(x) = (a_{r-1}x^{r-1} + a_{r-2}x^{r-2} + \cdots + a_1 x + a_0)^2 \tag{1}$$
$$= a_{r-1}x^{2(r-1)} + a_{r-2}x^{2(r-2)} + \cdots + a_1 x^2 + a_0 \tag{2}$$
$$= \widetilde{a_{r-1}}x^{r-1} + \widetilde{a_{r-2}}x^{r-2} + \cdots + \widetilde{a_1}x + \widetilde{a_0} \tag{3}$$

To compute the coefficients $\widetilde{a_i}$, the matrix multiplication is used, the key is the base transformation matrix BT, shown as follows:

$$a^2(x) = [a_0, a_1, \cdots, a_{r-1}] \begin{bmatrix} x^0 \\ x^2 \\ \vdots \\ x^{2(r-2)} \\ x^{2(r-1)} \end{bmatrix} = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}_{r\times r} \cdot \begin{bmatrix} x^0 \\ x^1 \\ \vdots \\ x^{r-2} \\ x^{r-1} \end{bmatrix}$$

Table 3: An illustrative example for computing polynomial inverse a^{-1} in $\mathbb{F}_2[x]/(x^r+1)$ with $r = 10163$ used in BIKE-2. This table explicitly demonstrates step by step a constat-time execution for inverse.

i	u_i	rule	$[\beta_{u_{i_1}}(a)]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_{i_i}}(a) = a^{2^{u_i}-1}$
0	1	—	—	$\beta_{u_{i_0}}(a) = a^{2^1-1}$
1	2	$2u_{i-1}$	$[\beta_{u_{i_0}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_1}}(a) = a^{2^2-1}$
2	4	$2u_{i-1}$	$[\beta_{u_{i_1}}(a)]^{2^{u_{i_1}}} \cdot \beta_{u_{i_1}}(a)$	$\beta_{u_{i_2}}(a) = a^{2^4-1}$
3	8	$2u_{i-1}$	$[\beta_{u_{i_2}}(a)]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_{i_3}}(a) = a^{2^8-1}$
4	9	$u_{i-1}+u_0$	$[\beta_{u_{i_3}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_4}}(a) = a^{2^9-1}$
5	18	$2u_{i-1}$	$[\beta_{u_{i_4}}(a)]^{2^{u_{i_4}}} \cdot \beta_{u_{i_4}}(a)$	$\beta_{u_{i_5}}(a) = a^{2^{18}-1}$
6	19	$u_{i-1}+u_0$	$[\beta_{u_{i_5}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_4}}(a) = a^{2^{19}-1}$
7	38	$2u_{i-1}$	$[\beta_{u_{i_6}}(a)]^{2^{u_{i_6}}} \cdot \beta_{u_{i_6}}(a)$	$\beta_{u_{i_7}}(a) = a^{2^{38}-1}$
8	39	$u_{i-1}+u_0$	$[\beta_{u_{i_7}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_8}}(a) = a^{2^{39}-1}$
9	78	$2u_{i-1}$	$[\beta_{u_{i_8}}(a)]^{2^{u_{i_8}}} \cdot \beta_{u_{i_8}}(a)$	$\beta_{u_{i_9}}(a) = a^{2^{78}-1}$
10	79	$u_{i-1}+u_0$	$[\beta_{u_{i_9}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{10}}}(a) = a^{2^{79}-1}$
11	158	$2u_{i-1}$	$[\beta_{u_{i_{10}}}(a)]^{2^{u_{i_{10}}}} \cdot \beta_{u_{i_{10}}}(a)$	$\beta_{u_{i_{11}}}(a) = a^{2^{158}-1}$
12	316	$2u_{i-1}$	$[\beta_{u_{i_{11}}}(a)]^{2^{u_{i_{11}}}} \cdot \beta_{u_{i_{11}}}(a)$	$\beta_{u_{i_{12}}}(a) = a^{2^{316}-1}$
13	317	$u_{i-1}+u_0$	$[\beta_{u_{i_{12}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{13}}}(a) = a^{2^{317}-1}$
14	634	$2u_{i-1}$	$[\beta_{u_{i_{13}}}(a)]^{2^{u_{i_{13}}}} \cdot \beta_{u_{i_{13}}}(a)$	$\beta_{u_{i_{14}}}(a) = a^{2^{634}-1}$
15	635	$u_{i-1}+u_0$	$[\beta_{u_{i_{14}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{15}}}(a) = a^{2^{635}-1}$
16	1270	$2u_{i-1}$	$[\beta_{u_{i_{15}}}(a)]^{2^{u_{i_{15}}}} \cdot \beta_{u_{i_{15}}}(a)$	$\beta_{u_{i_{16}}}(a) = a^{2^{1270}-1}$
17	2540	$2u_{i-1}$	$[\beta_{u_{i_{16}}}(a)]^{2^{u_{i_{16}}}} \cdot \beta_{u_{i_{16}}}(a)$	$\beta_{u_{i_{17}}}(a) = a^{2^{2540}-1}$
18	5080	$2u_{i-1}$	$[\beta_{u_{i_{17}}}(a)]^{2^{u_{i_{17}}}} \cdot \beta_{u_{i_{17}}}(a)$	$\beta_{u_{i_{18}}}(a) = a^{2^{5080}-1}$
19	10160	$2u_{i-1}$	$[\beta_{u_{i_{18}}}(a)]^{2^{u_{i_{18}}}} \cdot \beta_{u_{i_{18}}}(a)$	$\beta_{u_{i_{19}}}(a) = a^{2^{10160}-1}$
20	10161	$u_{i-1}+u_0$	$[\beta_{u_{i_{19}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{20}}}(a) = a^{2^{10161}-1}$

where $[\widetilde{a_0}, \widetilde{a_1}, \cdots, \widetilde{a_{r-1}}] = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}$ and BT is a $r \times r$ permutation matrix defined as:

$$\mathrm{BT} = \begin{bmatrix} \mathbf{1}_0 & 0 & \cdots & \cdots & \cdots \\ \cdots & \mathbf{1}_2 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \mathbf{1}_{2r-4} & 0 & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{2r-2} & 0 \end{bmatrix}$$

where $\mathbf{1}_i$ represents the (i mod r)-th (counting from zero) non-zero entry in the corresponding row of the matrix BT.

More generally, we derive the base transformation matrix $\mathrm{BT}_n = \mathrm{BT}^n$ for continuous square $a(x)^{2^n} = \widetilde{a_{r-1}}x^{r-1} + \widetilde{a_{r-2}}x^{r-2} + \cdots + \widetilde{a_1}x + \widetilde{a_0}$, which is

particularly useful in computing the polynomial inverse.

$$\mathrm{BT}^n = \begin{bmatrix} \mathbf{1}_0 & 0 & \cdots & \cdots & \cdots \\ \cdots & \mathbf{1}_{2^n} & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \mathbf{1}_{(r-2)2^n} & 0 & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{(r-1)2^n} & 0 \end{bmatrix}$$

In order to compute the coefficient vector $[\widetilde{a_0}, \widetilde{a_1}, \cdots, \widetilde{a_{r-1}}] = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}^n$, BT^n must be represented in column view, *i.e.*, the row number of the non-zero entry for every column of BT^n. In this form, the squaring of $a(x)$ is essentially the substitutions of the coefficients of $a(x)$:

$$\mathrm{BT}^n = \begin{bmatrix} \mathbf{1}_0 & \vdots & \vdots & \vdots & \vdots \\ \vdots & \mathbf{1}_{2^{-n}} & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{(r-2)2^{-n}} & \vdots \\ \vdots & \vdots & \vdots & \vdots & \mathbf{1}_{(r-1)2^{-n}} \end{bmatrix}$$

Theorem 2. *For squaring* $(a(x))^{2^n} = \sum \widetilde{a_i} x^i$ *of a polynomial* $a(x) = \sum a_i x^i \in \mathcal{R}$*, each coefficient* $\widetilde{a_i}$ *is updated by* a_j *in* $[a_{r-1}, a_{r-2}, \cdots, a_0]$ *as:*

$$\widetilde{a_i} = a_{i2^{-n} \bmod r}$$

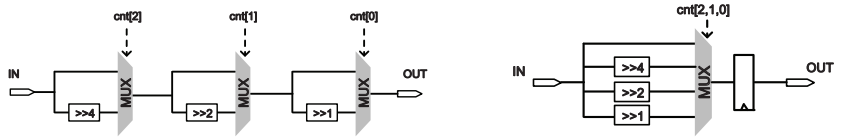
Note that the data is typically processed digit by digit on hardware. However, sometimes we want to extract the precise bit in the digit. Squaring is such an example where the result is indeed computed bit by bit. To preserve the constant time execution while avoiding the timing performance degration, Barrel shifter is used to rotate the desired bit within $\lceil log_2 d \rceil$ cycles.

Fig. **??** depicts the pipeline stages for computing the continuous square. The basic pattern is: read from memory the entry which contains $a_{i2^{-n}}$, then load it to the Barrel shifter, rotate to correct bit position, and finally write the result back to memory. In general, such pattern repeats r times and hence the cycle counts for continuous square are

$$r(\lceil log_2 d \rceil + 3)$$

Next, the above pipeline can be further improved based on the observation that read and rotate can be parallelized, and load and write can be parallelized as well. To sum up, the optimized cycle counts are

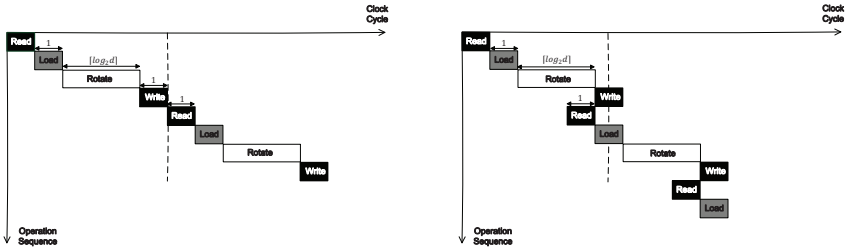
$$r(\lceil log_2 d \rceil + 1) + 2$$



(a) Barrel shifter constructed by combinational logic

(b) Barrel shifter constructed by sequential logic

Fig. 3: Barrel shifter used in our key generator to ensure constant execution of multiplication and squaring



(a) Primitive operation pipelining for continuous square

(b) Optimised operation pipelining for continuous square

Fig. 4: Timing diagram used in continuous square

Generic Multiplication in $\mathbb{F}_2[x]/(x^r+1)$ For two dense polynomials $a(x)$ and $b(x)$, their multiplication is represented as:

$$a(x)\cdot b(x) = (a_{r-1}x^{r-1}+\cdots+a_1x+a_0)\cdot(b_{r-1}x^{r-1}+\cdots+b_1x+b_0) \quad (4)$$

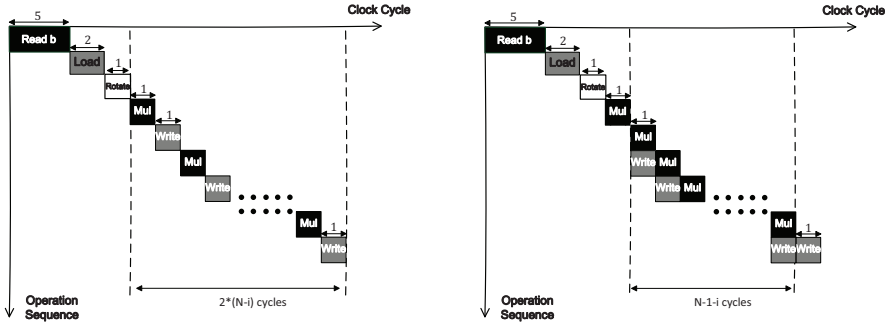
$$= \widetilde{c_{r-1}}x^{r-1}+\widetilde{c_{r-2}}x^{r-2}+\cdots+\widetilde{c_1}x+\widetilde{c_0} \quad (5)$$

Likewise, another base transformation matrix BT_{mul} is used to compute the coefficients $\widetilde{c_i}$ of $a(x)b(x)$, shown as follows:

$$a(x)b(x) = [\widetilde{c_0},\widetilde{c_1},\cdots,\widetilde{c_{r-1}}]\begin{bmatrix}x^0\\x^1\\\vdots\\x^{r-2}\\x^{r-1}\end{bmatrix} = [a_0,a_1,\cdots,a_{r-1}]\cdot \mathrm{BT}_{mul,r\times r}\cdot\begin{bmatrix}x^0\\x^1\\\vdots\\x^{r-2}\\x^{r-1}\end{bmatrix}$$

BT_{mul} is a $r\times r$ matrix where each column/row is a cyclic form of the vector $[b_{r-1},b_{r-2},\ldots,b_1,b_0]$:

$$\mathrm{BT} = \begin{bmatrix} b_0 & b_1 & \cdots & b_{d-1} & \cdots & b_{r-1}\\ b_{r-1} & b_0 & \cdots & b_{d-2} & \cdots & b_{r-2}\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\ b_2 & b_3 & \cdots & b_{d+1} & \cdots & b_1\\ b_1 & b_2 & \cdots & b_d & \cdots & b_0 \end{bmatrix}$$



(a) Primitive operation pipelining for generic multiplication

(b) Optimised operation pipelining for generic multiplication

Fig. 5: Timing diagram used in generic multiplication

BT is redundant since the blocks along each diagonal is identical (for example, $BT[0:d-1,0:d-1] = BT[d:2d-1,d:2d-1] = \ldots$). This way, BT is split into three parts: the central diagonal, the upper triangle, and the lower triangle. Fig. **??**, **??**, and **??** illustrate how to compute them. By eliminating such redundancy, the communication overhead for loading BT_{mul} is minimized and therefore, faster pipeline scheduling is possible in the generic multiplication.

A formal description for the proposed generic multiplication can be found in Alg. **??**. It is seen that the core function is the multiplication of a d-bit vector $A[k]$ of $a(x)$ and a block matrix $\mathrm{BT}[i,j]$ from BT:

$$[a_j, a_{j+1}, \ldots, a_{j+d-1}] \cdot \begin{bmatrix} b_i & b_{i+1} & b_{i+2} & \cdots & b_{i+d-1} \\ b_{i-1} & b_i & b_{i+1} & \cdots & b_{i+d-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{i-d+1} & b_{i-d+2} & b_{i-d+3} & \cdots & b_i \end{bmatrix}_{d \times d}$$

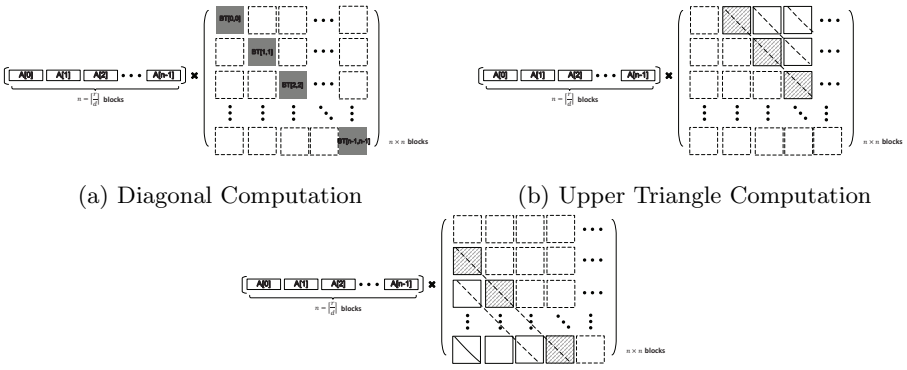
We design a dedicate core unit for the multiplication above in one clock cycle, depicted in Fig **??**. With this module, the proposed generic multiplication algorithm can be scheduled as Fig. **??** shows. Read takes 5 cycles to balance the critical path for reading $B[i,j]$. Again the dual-port RAM is used such that the target vector $B[i,j]$ can be loaded within two cycles. Here a fixed rotation is performed to move $A[k]$ to the correct bit positions. MUL performs the core multiplication $A[k] \cdot B[i,j]$ and later writes back the result.

An even better schedule is depicted in Fig. **??** where Mul and Write perform simultaneously. To handle the diagonal computation, it takes $5+2+1+2\lceil r/d \rceil$ cycle counts where $i = 0$. $\lceil r/d \rceil - 1$ iterations are required for the upper triangle computation. A particular $i = 1, 2, \ldots, d-1$-th iteration consumes $5+2+1+\lceil r/d \rceil - 1 - i + 1$ cycle counts and thus the entire upper triangle computation costs $(\lceil r/d \rceil^2 + 17\lceil r/d \rceil - 18)/2$ cycle counts. The same cycle counts are needed for the lower triangle computation.

Input: dense polynomials $a(x), b(x) \in \mathbb{F}_2[x]/(x^r + 1)$.
Output: $a(x) \cdot b(x) \in \mathbb{F}_2[x]/(x^r + 1)$

Reformulate $a(x)$ to block matrix view as $A[0], A[1], A[2], \ldots, A[n-1]$ where $n = \lceil r/d \rceil$, d is the number of bits used in the row vector $A[i]$.
Compute partial blocks from the transformation matrix BT associated with $b(x)$: $BT[0,0]$, $BT[0,1], BT[0,2], \ldots, BT[0,n-1]$ and $BT[1,0]$, $BT[2,0], \ldots, BT[n-1,0]$
The multiplication result is presented again in block matrix view $C[0], C[1], C[2], \ldots, C[n-1]$
/*Diagonal Computation*/
for $i \leftarrow 0$ ***to*** $n-1$ **do**
    $C[i] \leftarrow A[i] \cdot BT[0,0]$
/*Upper Triangle Computation*/
for $i \leftarrow 1$ ***to*** $n-1$ **do**
    for $j \leftarrow 0$ ***to*** $n-1-i$ **do**
        $C[i+j] \leftarrow C[i+j] + A[j] \cdot BT[0,i]$
/*Lower Triangle Computation*/
for $i \leftarrow 1$ ***to*** $n-1$ **do**
    for $j \leftarrow 0$ ***to*** $n-1-i$ **do**
        $C[j] \leftarrow C[j] + A[i+j] \cdot BT[i,0]$
return *the vector* $C = [C[0], C[1], \ldots, C[n-1]]$ *as the coefficients of* $c(x) = a(x)b(x)$

Algorithm 5: Proposed generic multiplication algorithm



(a) Diagonal Computation

(b) Upper Triangle Computation

(c) Lower Triangle Computation

Fig. 6: Illustration for the proposed generic multiplication algorithm

In general, the optimised cycle counts for generic multiplication are

$$\lceil r/d \rceil^2 + 18\lceil r/d \rceil - 9$$

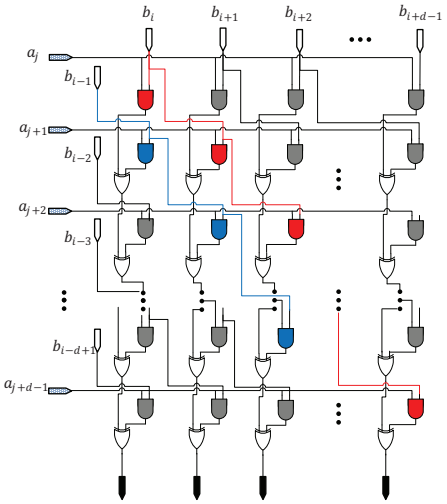


Fig. 7: architectural overview of the core unit in the generic multiplication

Table 4: Cycle counts for extracting d bits of $x^i b(x)$ in Sparse-Times-Dense Multiplication

Operation	Read_h	Read_g	Load	Rotation	Add	Write	Total
	3	3	2	$\lceil log_2(d) \rceil$	1	1	$10+\lceil log_2(d) \rceil$

2.3 Sparse-Times-Dense Multiplication in $\mathbb{F}_2[x]/(x^r+1)$

In BIKE, a special form of multiplication called sparse-times-dense multiplication, is frequently used. In sparse-times-dense multiplication, one of the operands is a sparse polynomial whereas the other is a dense polynomial. Such multiplication can be efficiently computed as follows: The sparse polynomial, *e.g*, $a(x)$ is first represented by an array of indices in $I = \{i | a_i = 1\}$. Then, the multiplication result can be obtained by extracting $i \in I$ rows of BT_{mul} and then accumulating them, *i.e.*, $\sum_{i \in I} x^i b(x)$ and the i-th row of the matrix BT_{mul} represents the coefficients of $x^i b(x)$. Formally, the coefficient vector of sparse-times-dense multiplication can be calculated as follows:

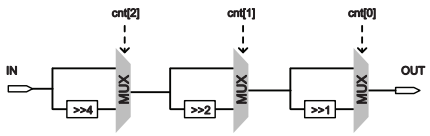
$$\sum_{i \in I} [b_{-i}, b_{-i+1}, \cdots, b_{-i-1}]$$

Fig. **??** depicts the basic pipeline schedule where the total cycle counts for one complete sparse-times-dense multiplication are

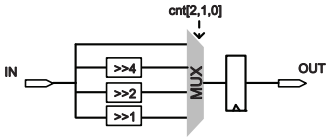
$$|I| \cdot \lceil r/d \rceil \cdot (10 + \lceil log_2(d) \rceil)$$

Table 5: Cycle counts comparison between unoptimized and optimized sparse-times-dense multiplication. The pipelined architecture introduces about 50% reduction of cycle counts for all BIKE instances

Category		d	r	$\lvert I\rvert = w/2$	unoptimized version	optimized version
128-bit	BIKE-[1,2]	64	10163	71	$180,624$	$90,880$
	BIKE-[3]	64	11027	67	$185,456$	$93,264$
192-bit	BIKE-[1,2]	64	19853	103	$512,528$	$257,088$
	BIKE-[3]	64	21683	99	$536,976$	$269,280$
256-bit	BIKE-[1,2]	64	32749	137	$1,122,304$	$562,248$
	BIKE-[3]	64	36131	133	$2,404,640$	$1,203,384$

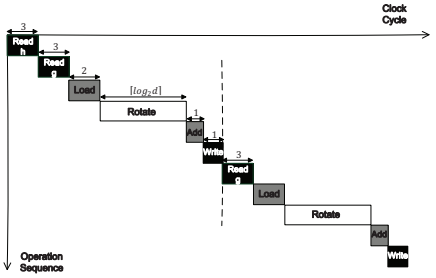


(a) Barrel shifter constructed by combinational logic

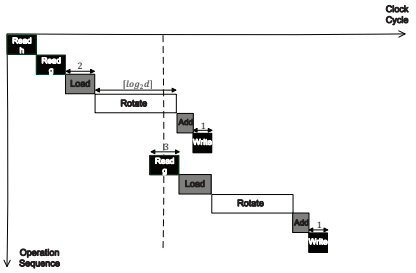


(b) Barrel shifter constructed by sequential logic

Fig. 8: Barrel shifter used in our key generator to ensure constant execution of multiplication and squaring



(a) Primitive operation pipelining for sparse-times-dense multiplication



(b) Optimised operation pipelining for sparse-times-dense multiplication

Fig. 9: Timing diagram used in sparse-times-dense multiplication

and optimized cycle counts are

$$|I|(\lceil r/d \rceil \cdot (2 + \lceil log_2(d) \rceil) + 8)$$

2.4 Generation of Polynomials in $\mathbb{F}_2[x]/(x^r + 1)$ with Prescribed Weight

Strictly speaking, the execution of the random polynomial generations proposed in Alg. **??**,**??** is non-constant. Alg. **??** performs an optional step if the Ham-

Input: PRNG seed, polynomial length r, data width d
Output: binary vector $\overline{\mathbf{f}}$ representing
$f = f_0 + \ldots + f_{r-1}x^{r-1} \in \mathbb{F}_2[x]/(x^r + 1)$, with odd weight $|f| \approx r/2$

1 $wt \leftarrow 0$
2 **for** $i \leftarrow 0$ ***to*** $\lceil r/d \rceil - 1$ **do**
3 $\quad \overline{\mathbf{f}[i]} \leftarrow \text{truncate}_d(SecureRand(seed))$
4 $\quad wt \leftarrow wt + |\overline{\mathbf{f}[i]}|$
5 **if** *wt is even* **then**
6 $\quad$ flip the first bit in $\overline{\mathbf{f}[i]}$
7 **return** $\overline{\mathbf{f}}$

Algorithm 6: Generation of a random polynomial with odd weight

Input: PRNG seed, polynomial length r, data width d, polynomial weight $wt = w/2$
Output: integer vector $\hat{\mathbf{f}}$ contains non-zero bit-positions $(\{i|f_i \neq 0\}$ to represent $f = f_0 + \ldots + f_{r-1}x^{r-1} \in \mathbb{F}_2[x]/(x^r + 1)$, with odd weight $|f| = wt$

1 **for** $i \leftarrow 0$ ***to*** $wt - 1$ **do**
2 $\quad$ **do**
3 $\quad\quad rand \leftarrow \text{truncate}_d(SecureRand(seed))$
4 $\quad$ **while** $rand \geq r$ *or* $rand \in \hat{\mathbf{f}}$
5 $\quad \hat{\mathbf{f}}[i] \leftarrow rand$
6 **return** $\hat{\mathbf{f}}$

Algorithm 7: Generation of a random sparse polynomial with given (odd) weight

ming weight of the polynomial is an even number; Alg. **??** keeps generating new random numbers unless the number is smaller than the system parameter r and non-repeating. Nevertheless, we validate that these non-constant behaviors do not leak any useful information on secrets for an adversary to exploit: The optional step in Alg. **??** flips the first bit-position of $\overline{\mathbf{f}}$ and the probabilistic distribution of this bit position does not change as long as the PRNG itself is unbiased (uniformly distributed). An adversary has no advantage to estimate the value of this specific bit even if he knows the timing difference. For Alg. **??**, the timing difference observed by an adversary is actually the total amount of time introduced by re-generation of random numbers if overflow ($rand \geq r$) or repetition ($rand \in \hat{\mathbf{f}}$) occurs. Overflow does not leak any information about $\hat{\mathbf{f}}$ as overflow only suggests the generated number is above r which leaks no information on $\hat{\mathbf{f}}$. The probability that the generated random number is smaller than r is $Pr(\text{rand} < r) = r/2^{\lceil log_2(r) \rceil}$. To eliminate the possible timing leakage from the repetition, we introduce a method in which we change the while-loop in step 4 to while $rand \geq r$. In other terms, we generate w_t random numbers of which each is smaller than r without considering whether they collide or not.

Table 6: Cycle counts for generating one entry of a random polynomial with odd weight

Operation	PRNG	Write	Total
	T_{PRNG}	1	T_{PRNG}+1

Table 7: Cycle counts for generating the i-th entry (i=1,2,...) of a random sparse polynomial with odd weight

Operation	PRNG	Rd_FIFO	CHECK	Wr_RAM	Total
	T_{PRNG}	4	i	1	$T_{\text{PRNG}} + 5 + i$

By doing this, no timing information is leaked on repetition but we might occur a few repetitions which reduce the Hamming weight of the sparse polynomial generated. However, a few repetitions are tolerable because BIKE is not sensitive to a slight loss of Hamming weight w. To reach λ bits of classical security, $\lambda \approx w - log_2 r$ for BIKE-1 and BIKE-2 and $\lambda \approx w - \frac{1}{2} log_2 r$ for BIKE-3, and 1 or 2 repetitions only decrease the security level 2^λ to $2^{\lambda-2}$. More importantly, the probability that a single run of Alg. **??** without considering collision succeeds if we tolerate one or two repetitions is very high (at least 0.997, data are collected in Table **??**):

$$Pr(\text{Alg. ?? succeeds}) = Pr(0 \text{ or } 1 \text{ or } 2 \text{ repetition occur}) \tag{6}$$

$$= \frac{r(r-1)\cdots(r-w_t+1)}{r^{w_t}} + \frac{\binom{w_t}{2} r(r-1)\cdots(r-w_t+2)}{r^{w_t}} \tag{7}$$

$$+ \frac{[\binom{w_t}{3} + \binom{w_t}{2}\binom{w_t-2}{2}/2] r(r-1)\cdots(r-w_t+3)}{r^{w_t}} \tag{8}$$

Note that the design uses a simple PRNG based on linear congruence to enable deterministic generation of random numbers (*i.e.*, the function of $SecureRand(\cdot)$ in Alg. **??**,**??**). For real deployment, such PRNG must be replaced with a cryptographically secure random number generator, *e.g.*, [**?**,**?**]. We require at most d random bits per T_{PRNG} clock cycles where $T_{\text{PRNG}} = 13$ in our settings. The average clock cycles for generating a qualified rand number is $T'_{\text{PRNG}} = T_{\text{PRNG}}/Pr(\text{rand} < r)$

Therefore, the total cycle counts for generating a random polynomial with odd weight (Alg. **??**) are

$$\lceil r/d \rceil \cdot T_{\text{PRNG}} + 1$$

We use the average case for Alg. **??** to estimate the clock cycle counts for generating a random sparse polynomial with given (odd) weight as:

$$\left(w_t T'_{\text{PRNG}} + \frac{w_t(w_t+1)}{2} + 5w_t \right) / Pr(\text{Alg. ?? succeeds})$$

Table 8: Success Rate for generating a random polynomial with odd weight using Alg. **??**

Category		no repetition	1 repetition	2 repetitions	success rate
128-bit	BIKE-[1,2]	0.7826	0.1926	0.0228	0.9981
	BIKE-[3]	0.8179	0.1649	0.0159	0.9989
192-bit	BIKE-[1,2]	0.7671	0.2040	0.0264	0.9976
	BIKE-[3]	0.7992	0.1796	0.0196	0.9985
256-bit	BIKE-[1,2]	0.7521	0.2148	0.0300	0.9970
	BIKE-[3]	0.7840	0.1911	0.0228	0.9981

Table 9: Cycle counts for generating the sparse polynomial h_0/h_1 in BIKE KeyGen

Category		Total	r	w_t	$Pr(\text{rand} < r)$	$Pr(\text{Alg. ?? succeeds})$
128-bit	BIKE-[1,2]	1279	10163	71	0.62	0.9981
	BIKE-[3]	1062	11027	67	0.67	0.9989
192-bit	BIKE-[1,2]	2120	19853	103	0.60	0.9976
	BIKE-[3]	1847	21683	99	0.66	0.9985
256-bit	BIKE-[1,2]	2801	32749	137	0.99	0.9970
	BIKE-[3]	3350	36131	133	0.55	0.9981

By using the FIFO-based architecture, random number generation and Hamming weight check are parallelized since FIFO buffers the random numbers without break whenever Hamming weight check is idle or not. As a result, the count above is reduced to

$$\left(T'^2_{\text{PRNG}} - 4T'_{\text{PRNG}} + \frac{T'_{\text{PRNG}} + w_t + 6}{2}(w_t - T'_{\text{PRNG}} + 5)\right) / Pr(\text{Alg. ?? succeeds})$$

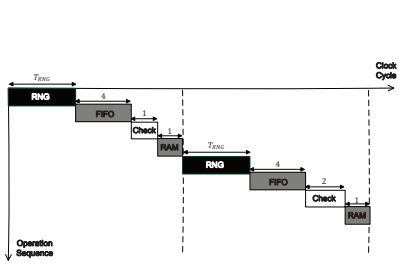
If the dual-port RAM is used, the above estimation can be further improved to

$$\left(T''^2_{\text{PRNG}} - 4T''_{\text{PRNG}} + \frac{T''_{\text{PRNG}} + \lceil w_t/2 \rceil + 6}{2}(\lceil w_t/2 \rceil - T''_{\text{PRNG}} + 5)\right) / Pr(\text{Alg. ?? succeeds})$$

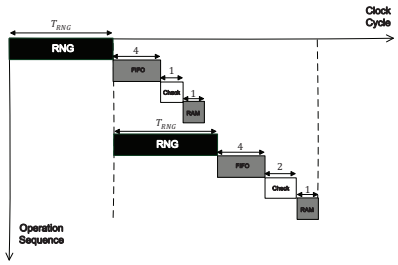
where $T''_{\text{PRNG}} = T_{\text{PRNG}}/Pr^2(\text{rand} < r)$

3 Key Generator Architecture

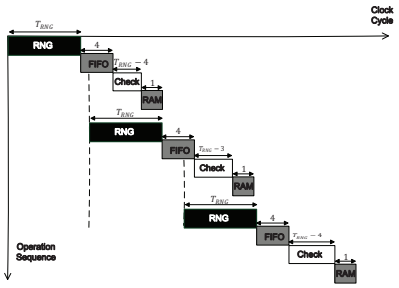
Our key generator performs the computations in BIKE-1/2/3 as shown in Algorithm **??**, Algorithm **??**, and Algorithm **??**. The top level architecture is depicted in Fig. **??**, **??**, **??**. BIKE-1 and BIKE-2 share a similar architecture where sparse-times-dense multiplication (MUL_CTRL) is the core functionality. The difference is that BIKE-1 requires two sparse-times-dense multiplications whereas BIKE-3 requires only one sparse-times-dense multiplication plus one addition. In our



(a) Primitive operation pipelining for Alg. **??**



(b) Optimised operation pipelining for Alg. **??** (initial phase)



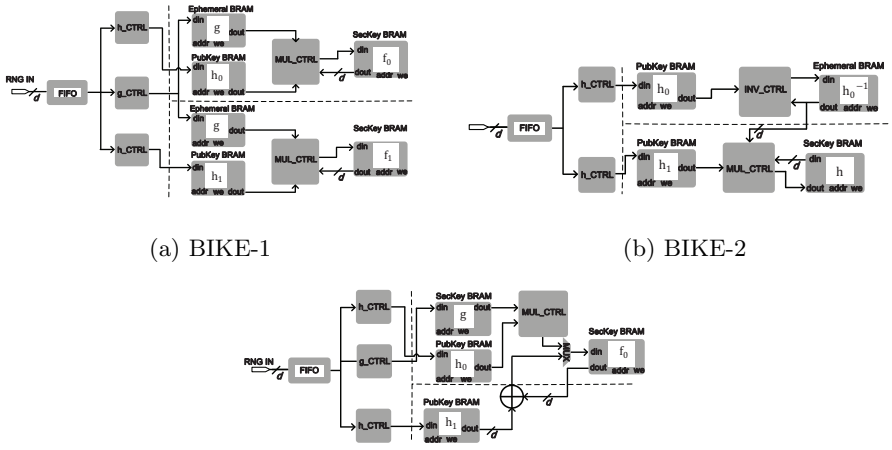
(c) Optimised operation pipelining for Alg. **??** (final phase)

Fig. 10: Timing diagram used for generating a random polynomial with odd weight

design, the two multiplications in BIKE-1 are performed simultaneously to improve the timing score and thus the total cycle count of BIKE-1 is close to that of BIKE-3. However, BIKE-3 is the lightest design with the shortest critical path since only one copy of MUL_CTRL is instantiated here. The module h_ctrl and g_ctrl assist to generate the sparse polynomials h_0, h_1 and the dense polynomial g as we have discussed in Section 2.6.

BIKE-2 is significantly more computationally intensive than BIKE-1 and BIKE-3 due to the polynomial inversion. The inversion is performed on the module INV_CTRL as Alg. **??** describes. INV_CTRL consists of the arithmetic core for continuous square and generic multiplication as illustrated in Section 2.3 and 2.4. After inversion, the dense polynomial h_0^{-1} is obtained and finally the sparse-times-dense multiplication is performed on MUL_CTRL to extract the secret key $h = h_0^{-1}h_1$.

The pipeline optimization technique is applied to g_ctrl, h_ctrl, MUL_CTRL, INV_CTRL for enhancing the timing performance. The detailed schedules have been depicted in Fig. **??**, **??**, **??**. As dual-port RAM is instantiated, we are able to load two successive data from secret/public key RAM in one clock cycle to slice registers for fast processing.



(a) BIKE-1 (b) BIKE-2

(c) BIKE-3

Fig. 11: Overview of the proposed BIKE architectures

Table 10: Implementation results of BIKE on a Xilinx Virtex-7 XC7V585T FPGA and a Xilinx Spartan-7 XC7S50 FPGA after the place and route process.

Aspect	BIKE-1 KeyGen		BIKE-2 KeyGen		BIKE-3 KeyGen	
	Virtex-7	Spartan-7	Virtex-7	Spartan-7	Virtex-7	Spartan-7
FFs	1055	993	2159	2098	955	911
LUTs	1970	1897	4445	3872	1482	1402
Slices	612	544	1483	1272	472	423
BRAM	7	7	10	10	4	4
Frequency	350 MHz	160 MHz	320 MHz	150 MHz	360 MHz	200 MHz
Time/Op	0.273 ms	0.597 ms	6.719 ms	14.334 ms	0.273 ms	0.492 ms
Compute h_0	1279 cycles		1279 cycles		1062 cycles	
Compute h_1	1279 cycles		1279 cycles		1062 cycles	
Compute g	2068 cycles		—		2250 cycles	
Compute ${h_0}^{-1}$	—		$2,056,683$ cycles		—	
Compute f_0	$90,880$ cycles		—		$94,135$ cycles	
Compute f_1	$90,880$ cycles		—		—	
Compute h	—		$90,880$ cycles		—	
Overall	$95,506$ cycles		$2,150,121$ cycles		$98,509$ cycles	

4 Performance Evaluations

In this section, we present our BIKE implementation results in FPGA. All the results are obtained post place-and-route for a Xilinx high-end FPGA device — Virtex XC7V585T and a low-end FPGA device — Spartan XC7S50 FPGA using Xilinx Vivado v2018.1 which demonstrate the compactness of our design

on both platforms. To the best of our knowledge, this work is the first one that reports the performance results of BIKE key generation on an FPGA platform. As shown in Table **??**, our BIKE-1 key generator on the Virtex-7 device runs at 350 MHz and generates one key pair in $95,506$ cycles, that is 0.273 ms of operation time. It runs on the Spartan-7 device at 160 MHz and generates one key pair in 0.597 ms. The area footprint is low (612 slices and 544 slices on Virtex-7 and Spartan-7 respectively) as our proposed architecture executes all operations in digit level (here in this concrete experiment, 64-bits of data are operated per time), which costs limited hardware resources. Our BIKE-3 key generator performs even better: It runs faster in 0.273 ms and 0.492 ms on Virtex-7 and Spartan-7, respectively. The slice overheads are also lower as BIKE-3 carries one copy of the core unit, sparse-times-dense multiplication whereas BIKE-1 has two of them. The cycle count is slightly worse than that of BIKE-1 as BIKE-3 requires an extra polynomial addition to compute the secret key f_0.

On the other hand, our BIKE-2 key generator runs at 320 MHz and 150 MHz on the Virtex-7 and Spartan-7 device respectively, and generates a key pair in $2,150,121$ cycles. This cycle count is equivalent to 6.719 ms and 14.334 ms of operating time for Virtex-7 and Spartan-7, respectively. It is noteworthy to mention that BIKE-2 timing performance is two orders of magnitude worse than BIKE-1 and BIKE-3. Such performance gap is due to the costly polynomial inversion in BIKE-2. Nevertheless, by using the proposed generic multiplication algorithm and continuous square algorithm, the BIKE-2 timing is significantly improved and preserves the constant-time execution. The slice usage is also conservative, increasing by 2-3 times compared with BIKE-1 and BIKE-2. The memory overhead is larger than that of BIKE-1/2 since inversion requests to save some ephemeral variables.

In the following, we compare our work with state-of-arts software implementations on either CPUs or micro-controllers. It is worth noting that comparison of our results with other software implementations is not fair since the platform used differs significantly. We attempt to compare the performance in the metric of cycle counts which is an platform independent indicator, given in Table **??**. First, we compare with the standard BIKE specification implementation [**?**] which is also submitted for NIST PQC standardization. The experiments from [**?**] is equipped with an Intel i5 CPU running at 1.8 GHz and 32 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and 4096K L3 cache. Their implementation is non-constant time and is much slower than ours. The micro-controller implementation from [**?**] is performed on ARM Cortex-M4 and uses obsolete BIKE-2 system parameters with 80-bit security level which is not recommended in the NIST PQC standardization. Despite the insufficiently 80-bit SL parameters used in [**?**], their implementation is non-constant time and costs huge cycle counts (148.5 million). An optimized software implementation for BIKE is reported in [**?**]. This additional implementation is equipped with an Intel i5 CPU running at 3.0 GHz and 70 GB RAM, 32K L1d and L1i cache, 1024K L2 cache, and 25,344K L3 cache. The core functionality is written in x86 assembly, and wrapped by assisting C code. The implementation uses the PCLMULQDQ architecture ex-

Table 11: Performance comparison of our FPGA implementation with dedicate software implementations.

Scheme	SL [bit]	Implementation	Platform	Performance [in millions of cycles]
BIKE-1	128	Ours, constant-time	FPGA	0.095
	128	[**?**]	CPU	≈ 0.73
	128	[**?**]	CPU	0.09
BIKE-2	128	Ours, constant-time	FPGA	2.15
	128	[**?**]	CPU	≈ 6.3
	128	[**?**]	CPU	4.38
	80	[**?**]	Micro-controller	≈ 148.5
BIKE-3	128	Ours, constant-time	FPGA	0.098
	128	[**?**]	CPU	≈ 0.43

Table 12: Performance comparison of our FPGA implementation with other code-based key generators.

Scheme	SL [bit]	Platform	f[MHz]	Time/Op	Cycles	Slices	BRAM
MDPC code:							
Ours (BIKE-1)			**350**	**0.273ms**	9.55×10^4	**612**	**7**
Ours (BIKE-2)	**128**	Virtex-7	**320**	**6.719ms**	2.15×10^6	**1483**	**10**
Ours (BIKE-3)			**360**	**0.273ms**	9.85×10^4	**472**	**4**
Goppa code:							
[**?**]	256	Virtex Ultrascale+	225	$3.98ms$	$\approx 1.01 \times 10^{11}$	112,845	375
[**?**]	103	Virtex 5	168	$16ms$	$\approx 2.72 \times 10^6$	8171	89
[**?**]	103	Virtex 5	163	$90ms$	$\approx 1.47 \times 10^7$	17,280	148

tension to enable carry-less multiplication for performance gain. Despite these dedicate and platform-dependent optimizations, our results are comparable and even better than [**?**]. The performance gain of our design comes from two aspects: first, the pipelining parallelization is proposed for core functions including squaring, multiplication. second, the proposed continuous squaring, generic multiplication, and sparse-times-dense multiplication has lower communication overheads to external memory which facilitate computational accelerations on FPGA.

Second, we compare our work with other existing hardware solutions based on Goppa codes [**?**,**?**]. As the state-of-art reference we consider the implementation reported in [**?**], where scalability is tested with different system parameters and FPGA platforms. We extract from [**?**] the experimental results at 256-bit SL and 103-bit SL on Xilinx FPGAs to provide direct and fair comparison. It is seen that for almost the same security level, Goppa codes require much more memory and computation time to generate the public/secret key. Such memory overhead is inevitable in Goppa code schemes as Goppa code cannot benefit from quasi-cyclic construction to compress the key size. Conversely, our work based

on quasi-cyclic codes is identified as a scalable lightweight design which shares high-speed and low-storage characteristics.

5 Conclusions

This paper presented a lightweight yet fast hardware-based key generator of a key encapsulation mechanism based on MDPC, called BIKE, both of which are second-round candidates for the NIST PQC competition. Novel multiplication and squaring algorithms were proposed to accelerate the key generation with low logic overhead. This work also presented optimized pipeline scheduling for fast execution of the core functions. The methods above are generic and applicable to other code-based schemes if the underlying code is quasi-cyclic.