Fast polynomial inversion for post quantum QC-MDPC cryptography

Nir Drucker[1,2], Shay Gueron[1,2], and Dusan Kostic[3]

[1]University of Haifa, Israel, [2]Amazon, USA, [3]EPFL Switzerland

Abstract. The NIST PQC standardization project evaluates multiple new designs for post-quantum Key Encapsulation Mechanisms (KEMs). Some of them present challenging tradeoffs between communication bandwidth and computational overheads. An interesting case is the set of QC-MDPC based KEMs. Here, schemes that use the Niederreiter framework require only half the communication bandwidth compared to schemes that use the McEliece framework. However, this requires costly polynomial inversion during the key generation, which is prohibitive when ephemeral keys are used. One example is BIKE, where the BIKE-1 variant uses McEliece and the BIKE-2 variant uses Niederreiter. This paper shows an optimized constant-time polynomial inversion method that makes the computation costs of BIKE-2 key generation tolerable. We report a speedup of 11.8× over the commonly used NTL library, and 55.5× over OpenSSL. We achieve additional speedups by leveraging the latest Intel's Vector-`PCLMULQDQ` instructions on a laptop machine, 14.3× over NTL and 96.8× over OpenSSL. With this, BIKE-2 becomes a competitive variant of BIKE.

Keywords: Polynomial inversion, BIKE, QC-MDPC codes, constant-time algorithm, constant-time implementation

1 Introduction

Bit Flipping Key Encapsulation (BIKE) [3] is a code-based KEM that uses Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes. It is one of the Round-2 candidates of the NIST PQC Standardization Project [20]. BIKE submission includes three variants: BIKE-1 and BIKE-3 that follow the McEliece [17] framework and BIKE-2 that follows the Niederreiter [19] framework. The main advantage of BIKE-2 is communication bandwidth (in both directions) that is half the size compared to BIKE-1 and BIKE-3. Another advantage is that BIKE-2 IND-CCA has a tighter security reduction compared to the other variants. However, it is currently not the popular BIKE variant (e. g., only BIKE-1 is integrated into LibOQS [21] and s2n [2]). The reason is that BIKE-2 key generation involves polynomial inversion (over $\mathbb{F}_2$) with computational cost that shadows the cost of decapsulation (see [18]). This is especially prominent when protocols are designed to achieve forward-secrecy through using ephemeral keys.

Polynomial inversion over a finite field is a time-consuming operation in several post-quantum cryptosystems (e. g., BIKE [3], HQC [1], ntruhrss701 [15],

LEDAcrypt [4]). The literature includes different approaches for inversions, depending on the polynomial degree and the field/ring over which the polynomials are defined. For example, the Itoh-Tsuji inversion (ITI) algorithm [16] is efficient when the underlying field is $\mathbb{F}_{2^k}$ for some k. Safegcd [5] implements inversion through a fast and constant-time Extended GCD algorithm. It is demonstrated in [5] as a means for speeding up ntruhrss701 [15] and for ECC with Curve25519. It is also used in the latest implementation of LEDAcrypt [4]. Algorithms for inversion of sparse polynomials over binary fields are discussed in [13, 14]. These algorithms are based on the division algorithm of [7].

There are (at least) two popular open-source libraries that provide polynomial inversion over $\mathbb{F}_2$: a) NTL [24], compiled with the GF2X library [22]; b) OpenSSL [25]. We note that the Additional code of BIKE (BIKE-2) [9] can be compiled to use either NTL or OpenSSL. We use this as our comparison baseline. For this research, we implemented a variant of the ITI algorithm (see also [6]) for polynomial inversion that leverages the special algebraic structure in our context, and runs in constant-time.

The paper is organized as follows. Section 2 offers some background and notation. In Section 3 we briefly explain our polynomial inversion method. Section 5 provides our performance results and Section 6 concludes this paper with several concrete proposals.

2 Preliminaries and notation

In this paper, we indicate hexadecimal notation with a `0x` prefix, and place the LSB on the right-most position. Let Y be a string of bits. We use $Y[j]$ to refer to the j^{th} bit of Y. Let $\mathbb{F}_2$ be the finite field of characteristic 2. Let $\mathcal{R}$ be the polynomial ring $\mathbb{F}_2[x]/\langle x^r - 1\rangle$ for some *block size* r and let $\mathcal{R}^*$ denote the set of invertible elements in $\mathcal{R}$. We treat polynomials, interchangeably, as vectors of bits. For every element $v \in \mathcal{R}$ its Hamming weight is denoted by $wt(v)$, its bit length by $|v|$, and its support (i.e., the positions of the non-zero bits) by $supp(v)$. In other words, if an element $a \in \mathcal{R}$ is defined by $a = \sum_{i=0}^{r-1} \alpha_i x^i$ then $supp(a)$ is the set of positions of the non-zero bits, $supp(a) = \{i : \alpha_i = 1\}$. Uniform random sampling from a set U is denoted by $u \xleftarrow{\$} U$. Uniform random sampling of an element with fixed Hamming weight w from a set U is denoted by $u \xleftarrow{w} U$.

2.1 BIKE

Table 1 shows the key generation of the variants of BIKE. The computations are executed over $\mathcal{R}$, and the block size r is a parameter. The weight of the secret key (sk) is w and we denote the public key by pk. For example, the parameters of BIKE-1-CCA for NIST Level-1 as defined in the specification [3] are: $r = 11779$, $|pk| = 23558$, $w = 142$. Table 1 shows that the key generation for BIKE-2 requires polynomial inversion. This is a heavy operation that can be a barrier for adoption when targeting forward-secrecy via ephemeral keys. On

the other hand, BIKE-2 has half the communication cost compared to BIKE-1 (and $\sim 2/3$ the communication cost compared to the bandwidth-optimized version of BIKE-3). Specifically, the initiator in BIKE KEM sends pk to the responder, i.e., f_0 for BIKE-2 versus (f_1, f_0) for BIKE-1. In the other direction, the responder sends a ciphertext to the initiator (not shown in Table 1). The length of BIKE-2's ciphertext is half the length of BIKE-1's ciphertext (see [3]). Therefore, reducing the computational cost of polynomial inversion can place BIKE-2 in an advantageous position.

Table 1: BIKE key generation. Polynomial inversion is required with BIKE-2.

	BIKE-1-CPA CPA	BIKE-1 CCA	BIKE-2 CPA	BIKE-2 CCA	BIKE-3 CPA	BIKE-3 CCA
	$h_0, h_1 \xleftarrow{w/2} \mathcal{R}$					
	$g \xleftarrow{\approx r/2,\ \text{odd}} \mathcal{R}$ $(f_0, f_1) = (gh_1, gh_0)$		$f_0 = h_1 h_0^{-1}$		$g \xleftarrow{\approx r/2,\ \text{odd}} \mathcal{R}$ $(f_0, f_1) = (h_1 + gh_0, g)$	
		$\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$		$\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$		$\sigma_0, \sigma_1, \sigma_2 \xleftarrow{\$} \mathcal{R}$
$sk =$	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1)$	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1)$	(h_0, h_1)	$(h_0, h_1, \sigma_0, \sigma_1, \sigma_2)$
$pk =$	(f_0, f_1)		f_0		(f_0, f_1)	

3 Optimized polynomial inversion in $\mathbb{F}_2[x]/\langle (x-1)h \rangle$ with irreducible h

In this paper, we propose to use an algorithm that is similar to the ITI algorithm [16]. In both cases, the essence is that raising an element a to the power 2^k (referred to as k-squaring hereafter), can be done efficiently. The ITI algorithm inverts an element of $\mathbb{F}_{2^k}$, where the field elements are represented in normal basis where computing a^{2^k} consists of k cyclic shifts of a's vector representation. This results in fast k-squaring. However, we note that the ITI algorithm can be generalized to other cases where k-squaring is efficient. One example is the set of polynomial rings that are used in BIKE and in other QC-MDPC based schemes.

Our inversion algorithm is Algorithm 2. It applies Algorithm 1 that computes a^{2^k-1} for some $k = 2^t$. Algorithm 1 is analogous to [16][Algorithm 2] that computes $a^{-1} \in \mathbb{F}_{2^\ell}$ for $\ell = 2^t + 1$ through Fermat's Little Theorem as

$$a^{-1} = a^{2^\ell - 2} = (a^{2^{\ell-1}-1})^2 = (a^{2^{2^t}-1})^2$$

BIKE, on the other hand, operates in the polynomial ring $\mathcal{R}$ with a value r for which

$$\mathcal{R} = \mathbb{F}_2[x]/\langle x^r - 1 \rangle = \mathbb{F}_2[x]/\langle (x-1)h \rangle$$

Algorithm 1 Computing a^{2^k-1} where $k = 2^t$

Input: a
Output: a^{2^k-1}
1: **procedure** CUSTOM_EXPONENTIATION(a)
2: $f = a$
3: **for** $i = 0$ to $t-1$ **do**
4: $g = f^{2^{2^i}}$
5: $f = f \cdot g$
6: **return** f

and h is an irreducible polynomial of degree $r-1$. In this ring, $ord(a) \mid 2^{r-1}-1$ for every $a \in \mathcal{R}^*$, and therefore

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

Here, Algorithm [16][Algorithm 2] cannot be used directly because $a^{2^{r-1}-2} = (a^{2^{r-2}-1})^2$ and $r-2$ is not a power of 2. Therefore, we use the following decomposition.

Decomposition of $2^{r-1}-2$. In order to apply Algorithm 1, we write $s = supp(r-2)$ and rewrite $z = 2^{r-1}-2$ in a convenient way:

$$z = 2 \cdot (2^{r-2}-1) = 2 \cdot \sum_{i \in s} \left((2^{2^i}-1) \cdot \left(2^{(r-2) \bmod 2^i} \right) \right) \tag{2}$$

Algorithm 2 uses Algorithm 1 and the decomposition (2) as follows.

Algorithm 2 Inversion in $\mathcal{R} = \mathbb{F}_2[x]/\langle (x-1)h \rangle$ with an irreducible h

Input: $a \in \mathcal{R}^*$
Output: a^{-1}
1: **procedure** INVERT(a)
2: $f = a$
3: $res = a$
4: **for** $i = 1$ to $\lfloor \log(r-2) \rfloor$ **do**
5: $g = f^{2^{2^{(i-1)}}}$ ▷ As in Alg. 1
6: $f = f \cdot g$
7: **if** $((r-2)[i] = 1)$ **then** ▷ i^{th} bit of $r-2$
8: $res = res \cdot f^{2^{(r-2) \bmod 2^i}}$
9: $res = res^2$
10: **return** res

Algorithm 2 requires $\lfloor \log(r-2) \rfloor + wt(r-2) - 1$ multiplications plus $\lfloor \log(r-2) \rfloor + wt(r-2) - 1$ k-squarings and 1 squaring (in $\mathcal{R}$). The performance depends

on $|r-2|$ and on $wt(r-2)$ and choices of r with smaller $|r-2|$ and $wt(r-2)$ lead to better performance.

Remark 1. The last square in line 9 of Algorithm 2 can be saved by changing line 6 therein to the following line

$$res = res \cdot f^{2^{1+(r-2) \bmod 2^i}}$$

This optimization is omitted from the algorithm's description for clarity.

Example 1. The recommended block size (r) for BIKE-1-CCA / BIKE-2-CCA, Level-1, is $r = 11779$. Here, $2^{r-1} - 2$ can be written as:

$$\begin{aligned} 2^{11778} - 2 = 2 \cdot (1 + 2(2^{512} - 1) + 2^{513}(2^{1024} - 1) + \\ 2^{1537}(2^{2048} - 1) + 2^{3585}(2^{8192} - 1)) \end{aligned}$$

With this decomposition, Algorithm 2 requires 17 polynomial multiplications, 17 k-squarings and 1 squaring.

For implementation efficiency, our method leverages the following observation.

Observation 1 *Let $a = \sum_{j \in supp(a)} x^j \in \mathcal{R}^*$. Then,*

$$\begin{aligned} a^{2^k} = \left(\sum_{j \in supp(a)} x^j \right)^{2^k} &= \sum_{j \in supp(a)} (x^j)^{2^k} \\ &= \sum_{j \in supp(a)} x^{j \cdot 2^k} = \sum_{j \in supp(a)} x^{j \cdot 2^k \bmod r} \end{aligned} \tag{3}$$

The first step in (3) is an identity in a ring with characteristic 2. The last step uses the fact that $ord(x) = r$ in $\mathcal{R}$. Using Observation 1, we can compute the k-square of $a \in \mathcal{R}^*$ as a permutation of the bits of a.

4 Our implementation.

This section discusses our implementation and further optimizations for Algorithm 2. Some explanatory code snippets are provided in Appendices A, B and C.

Speeding up the implementation with precomputed tables. The actual values of k in all the k-squarings of Algorithm 2 depend on r but not on a. Therefore, if r is fixed, the permutation $p_0 : j \to j \cdot 2^k \bmod r$ can be pre-computed for all the relevant values of k (which depends only on r). This speeds up the implementation. The required storage is $\lfloor \log(r-2) \rfloor + 1 + wt(r-2)$ tables where each one holds $|r|$ values.

Inverted permutation. The BIKE implementation stores the polynomials in a *dense* representation, i.e., an array of $\lceil r/word\_size \rceil$ words where each word holds $word\_size$ bits of the polynomial. The straightforward way to permute is to go over all the words of the data, extract all the $word\_size$ bits, and store every one of them in the required position of the output polynomial (as defined by the permutation map). This approach requires one memory read and $word\_size$ writes to random locations in the output data, per word of the input. However, when we apply the inverted permutation map, the k-square requires $word\_size$ random memory reads from the input data and only one memory write to the output array, per word of the input array. This speeds up the k-squaring in a noticeable way.

Using regular polynomial square. Squaring a polynomial in $\mathcal{R}$ is very efficient (significantly faster than a k-squaring. See Appendix A). This leads to the following optimization for small values of k: execute a chain of k single squarings instead of executing a k-square routine. The k value for preferring a k-square over a chain of squares depends on the implementation. We provide Table 6 in Appendix A to this end. Consequently, in addition to $r-2$ and $wt(r-2)$, the efficiency of inversion depends on the number of k-squares that can be replaced with regular squares. For example, consider $r_1 = 11779$ and $r_2 = 12347$. Here, inverting a polynomial of degree r_1 is expected to be faster than for r_2, because $wt(r_1 - 2) = 5 < 6 = wt(r_2 - 2)$. However, from the binary representations $r_1 - 2 =$ `0b10111000000001` and $r_2 - 2 =$ `0b11000000111001`, we see that the set bits in $r_2 - 2$ are positioned close to the LSB, and the set bits in $r_1 - 2$ are positioned close to the MSB. If the k-square threshold is 64, then for r_1 we can replace (only) one k-square with a chain of (regular) squares, and for r_2 we can replace 4 such k-squares.

Constant-time considerations. Algorithm 2 involves a constant number of steps for every given (fixed) r because the number and the order of multiplications and k-squarings are independent of the input. However, to achieve a constant-time implementation, the multiplication and k-squaring have to be constant-time routines. The Additional code of BIKE [9] already implements multiplication in constant-time. Since the k-squaring operation is merely a permutation of bits, it is straightforward to implement it in constant-time as follows: scan every bit of the input and update the appropriate bit in the output polynomial. This approach enjoys also constant memory access because the permutation is determined only by the (fixed) value of r.

Using Vector-PCLMULQDQ. Modern CPUs offer a fast carry-less multiplication instruction (`PCLMULQDQ`) that can be used for multiplication in binary fields. We note that `PCLMULQDQ` can be a bottleneck when algorithms that involve polynomial multiplication run on modern architectures: while AVX512 architectures can use wider 512-bit registers (`zmm`), `PCLMULQDQ` operates only on 128-bit registers (`xmm`). In the recent 10th generation CPUs (codename "Ice Lake") Intel® introduced a vector `PCLMULQDQ` instruction, and we leverage this

feature to our advantage. We replaced the 4×4 64-bit words schoolbook implementation that is used in the Additional code of BIKE, with the 8×8 64-bit words schoolbook algorithm of [11]. This yields some improvements. We further optimized the code to use a 16×16 64-bit words Karatsuba multiplication and observed a total speedup by a factor of 1.08 with this architecture. The details are explained in Appendix B.

Using binary-recursive-Karatsuba. In [8] we recommended to use the binary recursive-Karatsuba for multiplication for polynomials whose degree is slightly smaller than a power of two (e. g., $r = 32749$ for BIKE-1-CPA Level-5). This allows some optimizations and simpler code because implementation of a 4×4 schoolbook suffices (in addition to the binary recursive Karatsuba code). However, the values of r for the IND-CCA variants of BIKE and for IND-CPA BIKE in Level-1 and Level-3 are not close to a power of two. Here, padding every multiplicand to the closest power of 2 can be costly. For example, padding $r = 11779$ (as in BIKE-2-CCA Level-1) to the 16384 increases the multiplicands size by $\sim 40\%$. To this end, we replaced the lower 8×8 multiplication (using Vector-`PCLMULQDQ`) with $\beta \times \beta$ ($9 \leq \beta \leq 16$) multiplication. This yields multiplications of sizes $2^\alpha \cdot \beta$, for some integer $\alpha > 1$. Table 2 shows the exact values. For example, for $r = 11779$, with $\beta = 12$, the closest value of the form $2^\alpha \cdot \beta$ is 12288 ($\alpha = 10$) with only $\sim 5\%$ increase in the overall multiplicands size. Our experiments for BIKE-2-CCA Level-1 show that the fastest implementation is as in [8] with $\beta = 16$ and $\alpha = 10$.

In this case, we can make the implementation faster by avoiding multiplications of the higher parts of the multiplicands, which are zero. This optimization depends on the values of r and β.

Table 2: The sizes of the multiplicands ($2^\alpha \cdot \beta$) for different choices of α and β. Boldface values are the closest from above to $r = 11779$ (BIKE-2-CCA Level-1). Italic values are the closest from above to $r = 24821$ (BIKE-2-CCA Level-3).

α	β=9	10	11	12	13	14	15	16
10	9216	10240	11264	**12288**	**13312**	**14336**	**15360**	**16384**
11	**18432**	**20480**	**22558**	24576	*26624*	*28672*	*30720*	*32768*
12	*36864*	*40960*	*45056*	*49152*	53248	57344	61440	65536
13	73728	81920	90112	98304	106496	114688	122880	131072

5 Results

This section provides performance results and compares them to the specified baseline.

The platforms. We carried out performance measurements on two different platforms, which we call "laptop" and "server" platforms:

- The laptop platform is a Dell XPS 13 7390 2-in-1 laptop. It has a 10^{th} generation Intel®Core^{TM} processor (microarchitecture codename "Ice Lake"[ICL]). The specifics are Intel®Core^{TM} i7-1065G7 CPU 1.30GHz. This platform has 16 GB RAM, 48K L1d cache, 32K L1i cache, 512K L2 cache, and 8MiB L3 cache and it supports AVX512 and Vector-`PCLMULQDQ` instructions. For the experiments, we turned off the Intel® Turbo Boost Technology (in order to work with a fixed frequency and measure performance in cycles).
- The server platform is an AWS EC2 `m5.24xlarge` instance with the 6^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Sky Lake"[SKL]) Xeon®Platinum 8175M CPU 2.50GHz. This platform has 384 GB RAM, 32K L1d and L1i cache, 1MiB L2 cache, and 32MiB L3 cache that only have AVX512 capabilities.

Measurements methodology. The performance reported hereafter is measured in processor cycles (per single core), where lower count is better. We obtain the results using the following methodology. Every measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the `RDTSC` instruction) and averaged. To minimize the effect of background tasks running on the system, every experiment was repeated 10 times, and the minimum result was recorded.

The code. Our code is written mainly in C with some x86-64 assembly routines. Some versions use the Vector-`PCLMULQDQ` and other AVX512 instructions. On the Ice Lake machine we compiled the code with gcc (version 9.2.1), using the "-O3 -march=native" optimization flags and ran it on a Linux OS (Ubuntu 19.04). On the server platform the code is compiled with gcc (version 7.4.0) in 64-bit mode, using the "-O3 -march=native" optimization flags and ran on Ubuntu 18.04.2 LTS.

The comparison baseline. Our comparison baseline are the implementations of the popular open-source libraries NTL (compiled with GF2X) [22, 24] and OpenSSL [25]. We do not compare to [7, 13, 14, 16] because they are all slower than NTL: a) the inversion algorithm of [13] is reported to be 2× faster than [7], 12× faster than [16], but 1.7× slower than NTL; b) the implementation in [14] is reported to be 3× slower than NTL. We also measured the inversion function of the LEDAcrypt optimized code [4] that implements safegcd [5]. This code uses AVX2, and our implementation uses AVX512. For fair comparison, we compiled our code with AVX2 instructions only. The performance of the LEDAcrypt inversion (on "laptop") is: a) using gcc: 4.05/12.43/27.32 million cycles for Level-1/3/5, respectively; b) using clang: 3.29/10.30/22.94 million cycles for Level-1/3/5, respectively. The performance of our inversion on the same platform is: 0.65/2.36/5.37 million cycles for Level-1/3/5, respectively. The code of [4] runs in constant time and is faster than NTL. On the other hand, it is slower than our implementation even when we use only the AVX2 code.

Blinding a non-constant time inversion. Binary polynomial inversion does not operate in constant-time for either OpenSSL [25] or NTL [24] because these libraries use the extended GCD based algorithms to compute the inverse. To address this issue, a recent change in OpenSSL, (between version 1.0.2 to version 1.1.0) protects the implementation by *blinding* the inversion as follows. The function `BN_GF2m_mod_inv`(a, s) computes a^{-1} mod s by the following sequence: 1) choose a random b; 2) compute $c = ab$; 3) invert c); 4) multiply by b. Unfortunately, this does not work in the general case, where s is not necessarily an irreducible polynomial (see discussion in [12]). If s is reducible, $c = ab$ may be non-invertible modulo s. This is exactly the case of BIKE-2 where $x^r - 1$ is reducible. Although the OpenSSL function `BN_GF2m_mod_inv`(a, $x^r - 1$) is called with invertible a, the blinding may select a random non-invertible polynomial b and then inverting $c = ab$ would fail. In the polynomial ring $\mathcal{R}$ a randomly selected b has probability $\frac{1}{2}$ to be non-invertible. For a fair comparison (of constant-time implementations), we use the same blinding technique for NTL as well. For correctness, we always choose b such that $wt(b)$ is odd, and therefore b is invertible in $\mathcal{R}$.

The results are summarized in Table 3 (for "laptop"), and Table 4 (for "server"). In all cases, our implementation outperforms the baseline. The relative speedups for BIKE-2 are higher for Level-1 than for Level-5. This is quite fortunate because our focus is anyway on Level-1. Note that NIST has announced that Level-5 is not critical for standardization (we provide Level 5 performance for the sake of comparison with other works).

We observe that the relative speedup on "laptop" is only slightly better than on "server" despite the fact that the laptop has a newer (10^{th} generation) CPU with Vector-`PCLMULQDQ`. In fact, we expect to see additional speedup as soon as Intel releases servers with the 10^{th} generation processor.

Table 3: BIKE-2 key generation when the inversion uses NTL with GF2X [22,24], OpenSSL [25], and our method. The platform is "laptop" (see text). Columns 2-5 count cycles in millions, and lower is better. The r values correspond to the IND-CCA variants of BIKE for Level-1/3/5.

r	NTL [22, 24]	OpenSSL [25]	This work	This work (w/ tables)	Speedup NTL/Our	Speedup NTL/T (w/ tables)	Speedup OpenSSL/T (w/ tables)
11779	6.28	42.51	0.47	0.44	13.46	14.31	96.86
24821	9.29	164.95	1.71	1.65	5.44	5.62	99.86
40597	16.37	515.21	4.08	3.85	4.02	4.25	133.91

The use of precomputed permutation tables (see Section 3) provides an interesting tradeoff. It improves the overall performance at a cost of occupying some memory space. The tables that we need to store hold $r \cdot (\lfloor \log(r-2) \rfloor + 1 + wt(r-2))$ entries of size r bits (for all security levels of BIKE the entries can be stored

Table 4: BIKE-2 key generation when the inversion uses NTL with GF2X [22,24], OpenSSL [25], and our method. The platform is "server" (see text). Columns 2-5 count cycles in millions, and lower is better. The r values correspond to the IND-CCA variants of BIKE for Level-1/3/5.

r	NTL [22,24]	OpenSSL [25]	This work	This work (w/ tables)	Speedup NTL/Our	Speedup NTL/T (w/ tables)	Speedup OpenSSL/T (w/ tables)
11779	4.93	23.22	0.43	0.42	11.51	11.79	55.53
24821	7.64	121.86	1.61	1.59	4.75	4.82	76.78
40597	15.24	342.61	3.89	3.80	3.92	4.01	90.13

in 2 bytes of memory). For example, for BIKE-2-CCA the required memory is 450KB for Level-1, 1.1MB for Level-3, and 2MB for Level-5.

Table 5 shows relative speedups in the BIKE-2 key generation for different values of r and illustrates the effect of $wt(r-2)$ on the performance of the key generation. All the values of r are legitimate choices for BIKE (i.e., $x^r - 1 = (x-1)h$, where h is irreducible). We chose one representative for every value of $wt(r-2)$ and the table includes the recommended parameters from the [3] specification.

6 Discussion

The effect of different choices of r on BIKE-2 performance. In general, the parameter r determines the sizes of the public key, the ciphertext and thus the overall latency and bandwidth. So far, r was chosen as the minimum value that satisfies the security target [3] and the target Decoding Failure Rate (DFR) of the decoder [10, 23]. We propose an additional consideration, namely $wt(r-2)$ (recall how the inversion Algorithm 2 depends on $wt(r-2)$). The currently recommended r for Level-1 is $r = 11779$ for which $wt(r-2) = 5$. Interestingly, a considerably larger $r = 12323$ has $wt(r-2) = 4$. Note that [10] shows that $\sim r = 12323$ is needed and sufficient in order to achieve a DFR of 2^{-128}.

Two considerations are pointed out in [8]: a) rejection sampling is faster for values of r that are close (from below) to a power of 2 (e. g., $r = 32749$ is close to $2^{15} = 32768$ and the rejection rate is $32749/32768 \approx 1$); b) it is useful to pad multiplicands to the nearest power of two. It follows that the three considerations should be taken into account together. For example, $wt(32749-2) = 13$ is quite large and the slightly larger r=32771 has $wt(32771-2) = 2$ and seems to be preferable. However, key generation with $r = 32749$ takes 4.2M compared to 5.3M cycles with $r = 32771$.

BIKE-2 versus BIKE-1 Until now, BIKE-1 seemed to be a more appealing option than BIKE-2. This is the result of the prohibitive cost of BIKE-2 key generation that seemed to be an obstacle for adoption, especially when ephemeral keys are

Table 5: Relative speedups in the BIKE-2 key generation for different values of r. The table shows how the performance depends on $wt(r-2)$. The values in boldface corresponds to the recommended parameters in [3].

r	$wt(r-2)$	Speedup over NTL		Speedup over NTL (with tables)		Speedup over OpenSSL (with tables)	
		server	laptop	server	laptop	server	laptop
12323	4	11.60	13.89	11.81	14.27	62.08	99.13
11779	**5**	**11.51**	**13.46**	**11.79**	**14.31**	**55.53**	**96.86**
12347	6	10.46	12.62	10.63	12.96	59.64	87.97
11789	7	10.85	12.91	11.09	13.30	70.09	79.63
11821	8	10.37	12.43	10.46	12.72	62.04	68.48
11933	9	9.89	11.96	10.09	12.45	52.86	72.74
12149	10	9.56	11.20	9.76	11.56	52.71	63.69
12157	11	9.23	10.91	9.43	11.39	42.76	64.20
25603	4	5.87	6.70	5.97	7.08	92.06	142.05
24659	5	5.40	6.12	5.49	6.34	87.44	122.43
24677	6	5.18	5.96	5.25	6.17	65.65	107.28
24733	7	5.00	5.89	5.08	6.09	83.44	91.20
24821	**8**	**4.75**	**5.44**	**4.82**	**5.62**	**76.78**	**99.86**
25453	9	4.65	5.31	4.71	5.54	81.13	101.41
24547	10	4.01	4.59	4.08	4.74	70.83	79.28
24533	11	3.90	4.49	3.97	4.63	58.92	82.64
24509	12	4.49	5.18	4.58	5.45	72.75	93.95
40973	5	4.51	5.18	4.59	5.37	103.48	133.27
41051	6	4.28	4.82	4.33	5.07	107.19	128.89
41077	7	4.06	4.61	4.12	4.77	84.17	132.74
40709	8	3.71	4.23	3.81	4.50	80.66	117.54
40597	**9**	**3.92**	**4.02**	**4.01**	**4.25**	**90.13**	**133.91**
40763	10	3.41	3.91	3.48	4.13	86.89	105.37
40637	11	3.33	3.71	3.39	3.96	84.55	111.01
40829	12	3.19	3.61	3.27	3.84	83.73	107.99

desired. This left out BIKE-2's bandwidth advantage. BIKE specification [3] addresses this difficulty by using a "batch inversion" approach that requires precomputation of a batch of key pairs. Such solutions require that other protocols are adapted to using batched key pairs, and this raises additional complications.

Our improved inversion and hence faster key generation avoids the difficulty. For Level-1 ($r = 11779$) BIKE-2 has key generation / encapsulation / decapsulation at 440K/180K/1.2M cycles, and communication bandwidth of 1.4KB in each direction. By comparison, BIKE-1 (after using our latest multiplication implementation) has key generation / encapsulation / decapsulation at 67K/230K/1.3M cycles, and communication bandwidth of 2.8KB in each direction. We believe that our results position BIKE-2 as an appealing design choice among the BIKE variants.

Acknowledgements

This research was partly supported by: NSF-BSF Grant 2018640; The BIU Center for Research in Applied Cryptography and Cyber Security, and the Center for Cyber Law and Policy at the University of Haifa, both in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

We would also like to thank Thorsten Kleinjung for his valuable comments on this work.

References

1. Aguilar Melchor, C., Aragon, N., Bettaieb, S., Lo ic, B., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zmor, G.: Hamming Quasi-Cyclic (HQC) (2017), https://pqc-hqc.org/doc/hqc-specification_2017-11-30.pdf
2. Amazon Web Services: s2n. https://github.com/awslabs/s2n (2020), last accessed 16 Feb 2020
3. Aragon, N., Barreto, P.S.L.M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Güneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.P., Zmor, G.: BIKE: Bit Flipping Key Encapsulation (2017), https://bikesuite.org/files/round2/spec/BIKE-Spec-2019.06.30.1.pdf
4. Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., Santini, P.: LEDAcrypt (2019), https://www.ledacrypt.org/
5. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 340–398 (May 2019). https://doi.org/10.13154/tches.v2019.i3.340-398
6. Bos, J.W., Kleinjung, T., Niederhagen, R., Schwabe, P.: ECC2K-130 on Cell CPUs. In: Bernstein, D.J., Lange, T. (eds.) Progress in Cryptology – AFRICACRYPT 2010. pp. 225–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12678-9_14
7. Chien-Hsing Wu, Chien-Ming Wu, Ming-Der Shieh, Yin-Tsung Hwang: High-speed, low-complexity systolic designs of novel iterative division algorithms in $gf(2^m)$. IEEE Transactions on Computers **53**(3), 375–380 (March 2004). https://doi.org/10.1109/TC.2004.1261843
8. Drucker, N., Gueron, S.: A toolbox for software optimization of QC-MDPC code-based cryptosystems. Journal of Cryptographic Engineering pp. 1–17 (jan 2019). https://doi.org/10.1007/s13389-018-00200-4
9. Drucker, N., Gueron, S., Kostic, D.: Additional implementation of BIKE. https://bikesuite.org/additional.html (2019)
10. Drucker, N., Gueron, S., Kostic, D.: QC-MDPC decoders with several shades of gray. Tech. Rep. Report 2019/1423 (Dec 2019), https://eprint.iacr.org/2019/1423
11. Drucker, N., Gueron, S., Krasnov, V.: Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH). pp. 115–119 (jun 2018). https://doi.org/10.1109/ARITH.2018.8464777
12. Gueron, Shay: https://github.com/open-quantum-safe/openssl/issues/42#issuecomment-433452096 (October 2018)

13. Guimar, A., Borin, E., Aranha, D.F., Guimarães, A., Borin, E., Aranha, D.F.: Introducing Arithmetic Failures to Accelerate QC-MDPC Code-Based Cryptography. Code-Based Cryptography **2**, 44–68 (2019). https://doi.org/10.1007/978-3-030-25922-8
14. Guimarães, A., Aranha, D.F., Borin, E.: Optimized implementation of QC-MDPC code-based cryptography. Concurrency and Computation: Practice and Experience **31**(18), e5089 (2019). https://doi.org/10.1002/cpe.5089
15. Hülsing, A., Rijneveld, J., Schanck, J., Schwabe, P.: High-Speed Key Encapsulation from NTRU. In: Fischer, Wieland and Homma, Naofumi (ed.) Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 232–252. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_12
16. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases. Information and Computation **78**(3), 171–177 (1988). https://doi.org/10.1016/0890-5401(88)90024-7
17. McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report **44**, 114–116 (Jan 1978), https://ui.adsabs.harvard.edu/abs/1978DSNPR..44..114M
18. Misoczki, R.: BIKE - Bit-Flipping Key Encapsulation. https://csrc.nist.gov/CSRC/media/Presentations/bike-round-2-presentation/images-media/bike-misoczki.pdf (2019), last accessed 18 Feb 2020
19. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Contr. Inform. Theory **15**(2), 157–166 (1986), https://ci.nii.ac.jp/naid/80003180051/en/
20. NIST: Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography (2019), last accessed 20 Aug 2019
21. Open Quantum Safe Project: liboqs. https://github.com/open-quantum-safe/liboqs (2020), last accessed 16 Feb 2020
22. Pierrick Gaudry, Richard Brent, P.Z., Thome, E.: gf2x-1.2. https://gforge.inria.fr/projects/gf2x/ (July 2017)
23. Sendrier, N., Vasseur, V.: On the Decoding Failure Rate of QC-MDPC Bit-Flipping Decoders. In: Ding, J., Steinwandt, R. (eds.) Post-Quantum Cryptography. vol. 2, pp. 404–416. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25510-7
24. Shoup, V.: Number theory c++ library (ntl) version 11.3.2. http://www.shoup.net/ntl (November 2018)
25. The OpenSSL Project: OpenSSL 1.1.1: The open source toolkit for SSL/TLS. https://github.com/openssl/openssl

A Squaring using PCLMULQDQ and VPCLMULQDQ

This appendix describes our C implementation for squaring in $\mathcal{R}$, using PCLMULQDQ. For brevity, we replace the long names of the C intrinsics with shorter macros as follows.

```
#define PERM64(a, mask)        _mm512_permutex_epi64(a, mask)
#define PERM64X2(a, mask, b)   _mm512_permutex2var_epi64(a, mask, b)
#define PERM64VAR(mask, a)     _mm512_permutexvar_epi64(mask, a)
#define MUL(a, b, imm8)        _mm512_clmulepi64_epi128(a, b, imm8)
#define MXOR(src, mask, a, b)  _mm512_mask_xor_epi64(src, mask, a, b)
#define ALIGN(a, b, count)     _mm512_alignr_epi64(a, b, count)
#define STORE(mem, reg)        _mm512_storeu_si512(mem, reg)
#define LOAD(mem)              _mm512_loadu_si512(mem)
#define EXPANDLOAD(mask, mem) _mm512_maskz_expandloadu_epi64(mask, mem)

#define LOAD128(mem)           _mm_loadu_si128(mem)
#define STORE128(mem, reg)     _mm_storeu_si128(mem, reg)
#define MUL128(a, b, imm8)     _mm_clmulepi64_si128(a, b, imm8)
```

When PCLMULQDQ (and not vector-PCLMULQDQ) is available, the square function is

```
void gf2x_sqr(uint64_t *res, const uint64_t *a)
{
  for (size_t i = 0; i < ceil(R/128); i++)
  {
      __m128i va = LOAD128((__m128i*)(a+i*2));
      STORE128((__m128i*)(&res[i*4]), MUL128(va, va, 0x00););
      STORE128((__m128i*)(&res[i*4+2]), MUL128(va, va, 0x11););
  }
}
```

When vector-PCLMULQDQ is available, four multiplications can be executed in parallel and the code is

```
void gf2x_sqr_vpclmulqdq(uint64_t *res, const uint64_t *a)
{
  __m512i vm = _mm512_set_epi64(7, 3, 6, 2, 5, 1, 4, 0);
  for (int i = 0; i < ceil(R/512); i++)
  {
      __m512i va  = LOAD(&a[i*8]);
      va  = PERM64VAR(vm, va);

      STORE(&res[i*16], MUL(va, va, 0x00));
      STORE(&res[i*16+8], MUL(va, va, 0x11));
  }
}
```

The permutation and thus some of the flow's serialization can be removed by using the _mm512_maskz_expandloadu_epi64 instruction.

```
void gf2x_sqr_vpclmulq(uint64_t *res, const uint64_t *a)
{
  for (int i = 0; i < ceil(R/512); i++)
  {
      __m512i va1  = EXPANDLOAD(0x55, &a[i*8]);
      __m512i va2  = EXPANDLOAD(0x55, &a[i*8+1]);

      STORE(&res[i*16], MUL(va1, va1, 0x00));
      STORE(&res[i*16+8], MUL(va2, va2, 0x00));
  }
}
```

However, our experiments show slower results with this instruction.

Table 6 compares squaring and k-squaring in $\mathcal{R}$ using our code. Our implementation starts with squaring up to the described threshold and then continues with k-squaring. The threshold depends on the platform.

Table 6: Squaring and k-squaring in $\mathcal{R}$ using our code. Columns 2 and 3 count cycles, where lower is better (threshold=floor(k-square/square). The r values correspond to the IND-CCA variants of BIKE for Level-1/3/5.

(a) Laptop

r	k-square	square	threshold
11779	16000	230	69
24821	35000	510	68
40597	65000	790	82

(b) Server

r	k-square	square	threshold
11779	20000	350	57
24821	42000	680	61
40597	68000	1100	61

B A 16×16 quad-words multiplication using `VPCLMULQDQ`

This appendix describes the C code of our recursive Karatsuba multiplication. The `mul128x4` function performs four 128-bit Karatsuba multiplications in parallel.

```
static inline void mul128x4(__m512i *h, __m512i *l, __m512i a, __m512i b)
{
    const __m512i mask_abq = _mm512_set_epi64(6, 7, 4, 5, 2, 3, 0, 1);
    __m512i       s1 = a ^ PERM64(a, _MM_SHUFFLE(2, 3, 0, 1));
    __m512i       s2 = b ^ PERM64(b, _MM_SHUFFLE(2, 3, 0, 1));

    __m512i lq  = MUL(a, b, 0x00);
    __m512i hq  = MUL(a, b, 0x11);
    __m512i abq = lq ^ hq ^ MUL(s1, s2, 0x00);
    abq         = PERM64VAR(mask_abq, abq);
    *l          = MXOR(lq, 0xaa, lq, abq);
    *h          = MXOR(hq, 0x55, hq, abq);
}
```

Then, we define the `mul512` function that receives two 512-bit `zmm` registers (a, b) as input, multiplies them and writes the result into the two registers $zh||zl$. The function performs several permutations to reorganize the quad-words. The relevant masks are:

```
const __m512i mask0   = _mm512_set_epi64(13, 12,  5,  4,  9,  8, 1, 0);
const __m512i mask1   = _mm512_set_epi64(15, 14,  7,  6, 11, 10, 3, 2);
const __m512i mask2   = _mm512_set_epi64( 3,  2,  1,  0,  7,  6, 5, 4);
const __m512i mask3   = _mm512_set_epi64(11, 10,  9,  8,  3,  2, 1, 0);
const __m512i mask4   = _mm512_set_epi64(15, 14, 13, 12,  7,  6, 5, 4);
const __m512i mask_s2 = _mm512_set_epi64( 3,  2,  7,  6,  5,  4, 1, 0);
const __m512i mask_s1 = _mm512_set_epi64( 7,  6,  5,  4,  1,  0, 3, 2);
```

The `__m512i` variables that are used in this function are: a) `xl`, `xh`. These hold the lower and upper parts of the 128-bit Karatsuba sub-multiplications; b) `xabl`, `xabh`, `xab`, `xab1`, `xab2`. These are used for the middle term of the 256-bit Karatsuba sub-multiplications; c) `yl`, `yh`, `yabl`, `yabh`, `yab`. These are used for middle term of the top 512-bit Karatsuba multiplication; d) `t[4]` that holds all the temporary products to `mul128` of the middle words.

Define

$$AX[i] = a[128(i+1)-1 : 128i]$$
$$BX[i] = b[128(i+1)-1 : 128i]$$
$$AY[i] = a[256(i+1)-1 : 256i]$$
$$BY[i] = b[256(i+1)-1 : 256i]$$

Then set

$$t[0] = AX1 \oplus AX3||AX2 \oplus AX3||AX0 \oplus AX2||AX0 \oplus AX1$$
$$t[1] = BX1 \oplus BX3||BX2 \oplus BX3||BX0 \oplus BX2||BX0 \oplus BX1$$

where

$$AX1 \oplus AX3||AX0 \oplus AX2 = (AX1||AX0) \oplus (AX3||AX2) = AY0 \oplus AY1$$
$$BX1 \oplus BX3||BX0 \oplus BX2 = (BX1||BX0) \oplus (BX3||BX2) = BY0 \oplus BY1$$

and set the lower 128 bits of `t[2]`, `t[3]` to (ignoring the upper bits)

$$t[2][127:0] = AX1 \oplus AX3 \oplus AX0 \oplus AX2$$
$$t[3][127:0] = BX1 \oplus BX3 \oplus BX0 \oplus BX2$$

```
    t[0] = PERM64VAR(mask_s1, a) ^ PERM64VAR(mask_s2, a);
    t[1] = PERM64VAR(mask_s1, b) ^ PERM64VAR(mask_s2, b);
    t[2] = t[0] ^ ALIGN(t[0], t[0], 4);
    t[3] = t[1] ^ ALIGN(t[1], t[1], 4);
```

The implementation invokes `mul128x4` three times: a) for calculating the lower and the upper 512-bit words; b) for the two middle 256-bit words; c) for the middle 512-bit word in the top-level Karatsuba. The number of invocations of `VPCLMULQDQ` for the entire `mul512` is only 9.

```
    mul128x4(&xh, &xl, a, b);
    mul128x4(&xabh, &xabl, t[0], t[1]);
    mul128x4(&yabh, &yabl, t[2], t[3]);
```

Finally, we complete the four 128-bit Karatsuba by

```
    xab  = xl ^ xh ^ PERM64X2(xabl, mask0, xabh);
    yl   = PERM64X2(xl, mask3, xh);
    yh   = PERM64X2(xl, mask4, xh);
    xab1 = ALIGN(xab, xab, 6);
    xab2 = ALIGN(xab, xab, 2);
    yl   = MXOR(yl, 0x3c, yl, xab1);
    yh   = MXOR(yh, 0x3c, yh, xab2);
```

and the 512-bit result is

```
    __m512i oxh = PERM64X2(xabl, mask1, xabh);
    __m512i oxl = ALIGN(oxh, oxh, 4);
    yab         = oxl ^ oxh ^ PERM64X2(yabl, mask0, yabh);
    yab         = MXOR(oxh, 0x3c, oxh, ALIGN(yab, yab, 2));
    yab ^= yl ^ yh;

    yab = PERM64VAR(mask2, yab);
    *zl = MXOR(yl, 0xf0, yl, yab);
    *zh = MXOR(yh, 0x0f, yh, yab);
```

For higher efficiency, our `mul1024` Karatsuba implementation holds the data in `zmm` registers in order to save memory operations when invoking `mul512`.

```
void mul1024(uint64_t *cp, const uint64_t *ap, const uint64_t *bp) {
    const __m512i a0 = LOAD(ap);
    const __m512i a1 = LOAD(ap + 8);
    const __m512i b0 = LOAD(bp);
    const __m512i b1 = LOAD(bp + 8);
    __m512i       hi[2], lo[2], ab[2];

    mul512(&lo[1], &lo[0], a0, b0);
```

```
    mul512(&hi[1], &hi[0], a1, b1);
    mul512(&ab[1], &ab[0], a0 ^ a1, b0 ^ b1);

    __m512i middle = lo[1] ^ hi[0];

    STORE(cp, lo[0]);
    STORE(cp + 8, ab[0] ^ lo[0] ^ middle);
    STORE(cp + 16, ab[1] ^ hi[1] ^ middle);
    STORE(cp + 24, hi[1]);
}
```

C Fast permutation

The inverted bit permutation ($a = map(b)$) of Section 4 can be implemented in a straightforward way as follows. We first convert the map to two maps `bytes_map` and `bits_map`, where `bytes_map`[i] is the byte index of $map(b[i])$ and `bits_map`[i] is the position of the relevant bit inside this byte.

```
idx = 0;
for(int i = 0; i < r; i++)
{
  uint8_t t = 0;
  for (size_t j = 0; j < 8; j++) {
      uint8_t bit = (a[pos_byte[idx]] >> pos_bit[idx]) & 1;
      t |= (bit << j);
      idx++;
  }
  b[i] = t;
}
```

A simpler way to apply the map is possible if we store every bit in a byte (that has the value `0x00` or `0x01`).

```
for(int i = 0; i < r; i++)
  b[i] = a[map[i]];
```

This can involve a costly conversion to and from across the representations but fortunately, we can speed it up with AVX512 (when available)

```
// Converting a binary array (B) to a bytes array (A)
for(size_t i = 0; i < qw_len; i++)
  STORE(&A[i*8], _mm512_maskz_set1_epi8(B[i], 1));
```

```
// Converting a bytes array (A) to a binary array (B)
__m512i first_bit_mask = _mm512_set1_epi8(1);
for(size_t i = 0; i < qw_len; i++)
    B[i] = _mm512_cmp_epi8_mask(LOAD(&A[i*8]), first_bit_mask, 0);
```

Note that the `_mm512_bitshuffle_epi64_mask` instruction can also be used for the latter conversion (see next). This instruction requires the `AVX512_BITALG` extension while the `_mm512_cmp_epi8_mask` instruction requires only `AVX512F` which is more common.

```
// Converting a bytes array (A) to a binary array (B)
__m512i first_bit_mask = _mm512_set1_epi64(0x3830282018100800);
for(int i=0; i < qw_len; i++)
  B[i] = _mm512_bitshuffle_epi64_mask(LOAD(&A[i*8]), first_bit_mask);
```