Information and Computation 281 (2021) 104799

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Fast polynomial inversion for post quantum QC-MDPC cryptography

Nir Drucker [a,b], Shay Gueron [a,b], Dusan Kostic [c,*]

[a] University of Haifa, Israel
[b] Amazon, USA
[c] EPFL, Switzerland

ARTICLE INFO

Article history:
Received 11 January 2021
Received in revised form 19 August 2021
Accepted 29 August 2021
Available online 3 September 2021

Keywords:
Polynomial inversion
BIKE
QC-MDPC codes
Constant-time algorithm
Constant-time implementation
NIST PQC round-3

ABSTRACT

New post-quantum Key Encapsulation Mechanism (KEM) designs, evaluated as part of the NIST PQC standardization Project, pose challenging tradeoffs between communication bandwidth and computational overheads. Several KEM designs evaluated in Round-2 of the project are based on QC-MDPC codes. BIKE-2 uses the smallest communication bandwidth, but its key generation requires a costly polynomial inversion. In this paper, we provide details on the optimized polynomial inversion algorithm for QC-MDPC codes (originally proposed in the conference version of this work). This algorithm makes the runtime of BIKE-2 key generation tolerable. It brings a speedup of 11.4× over the commonly used NTL library, and 83.5× over OpenSSL. We achieve additional speedups by leveraging the latest Intel's Vector-PCLMULQDQ instructions, 14.3× over NTL and 103.9× over OpenSSL. Our algorithm and implementation were the reason that BIKE team chose BIKE-2 as the only scheme for its Round-3 specification (now called BIKE).

1. Introduction

Bit Flipping Key Encapsulation (BIKE) [1] is a code-based KEM that uses Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes. It is one of the Round-3 candidates of the NIST PQC Standardization Project [2] selected by NIST as an alternative finalist. BIKE Round-2 submission [3] included three variants: BIKE-1 and BIKE-3 that follow the McEliece [4] framework and BIKE-2 that follows the Niederreiter [5] 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, at the time of writing the conference (short) version of this paper [6,7], it was not the popular BIKE variant (e.g., only BIKE-1 is integrated into LibOQS [8] and s2n [9]). 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 [10]). 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 [1], HQC [11], ntruhrss701 [12], LEDAcrypt [13]). The literature includes different approaches for inversions, depending on

* Corresponding author.
E-mail address: dkostic@protonmail.com (D. Kostic).

https://doi.org/10.1016/j.ic.2021.104799

the polynomial degree and the field/ring over which the polynomials are defined. For example, the Itoh-Tsuji inversion (ITI) algorithm [14] is efficient when the underlying field is $\mathbb{F}_{2^k}$ for some k. Safegcd [15] implements inversion through a fast and constant-time Extended GCD algorithm. It is demonstrated in [15] as a means for speeding up ntruhrss701 [12] and for ECC with Curve25519. It is also used in the latest implementation of LEDAcrypt [13]. Algorithms for inversion of sparse polynomials over binary fields are discussed in [16,17]. These algorithms are based on the division algorithm [18].

There are (at least) two popular open-source libraries that provide polynomial inversion over $\mathbb{F}_2$: a) NTL [19], compiled with the GF2X library [20]; b) OpenSSL [21]. We note that the Additional code of BIKE Round-2 (BIKE-2) [22] 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 [23]) for polynomial inversion that leverages the special algebraic structure in our context, and runs in constant-time.

Additions over the original conference version. This extended version of [6,7] includes the following new materials

- A detailed description of how to optimize k-squaring using vector instructions.
- Optimization for the implementation of the permutation map generation, and analysis of "precompute" vs. "generate on the fly".
- Polynomial multiplication and squaring optimizations with `(V)PCLMUL` instruction.
- Fully detailed performance studies of our inversion algorithm (previously, we reported only the performance of the entire key generation).

For Round-3 of NIST's PQC project, the BIKE team decided to make BIKE-2 as their only proposed design, which is now called simply BIKE [1]. This is due to the smallest communication bandwidth (from the three original versions) and the tolerable key generation performance that is achieved by our work. Subsequently, NIST accepted BIKE as the selected QC-MDPC design among the other QC-MDPC-based proposals, and promoted BIKE to Round-3, as an alternative finalist (to be potentially standardized although not in the first pass).

The paper is organized as follows. Section 2 offers some background and notation. In Section 3, we briefly explain our polynomial inversion method. Our implementation is described in Section 4. 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 Round-2. 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 Round-2 specification [3] are: $r = 11779$, $|pk| = 23558$, $w = 142$. Table 1 shows that the key generation for BIKE-2 requires polynomial inversion. This heavy operation 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.

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

In this paper, we propose to use an algorithm for inversion that is similar to the ITI algorithm [14]. 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 $a \in \mathbb{F}_{2^k}$ where the field elements are represented in normal basis. With such representation computing the k-squaring operation, a^{2^k}, consists of k cyclic shifts of a's vector representation. This results in fast implementation of 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.

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
	$h_0, h_1 \xleftarrow{w/2} \mathcal{R}$					
	$g \xleftarrow{\approx r/2,\ \text{odd}} \mathcal{R}$ $(f_0, f_1) = (gh_1, gh_0)$		$f_0 = h_1 h_0^{-1}$		$g \xleftarrow{\approx r/2,\ \text{odd}} \mathcal{R}$ $(f_0, f_1) = (h_1 + gh_0, g)$	
		$\sigma_{0/1} \xleftarrow{\$} \mathcal{R}$		$\sigma_{0/1} \xleftarrow{\$} \mathcal{R}$		$\sigma_{0/1/2} \xleftarrow{\$} \mathcal{R}$
$sk =$	(h_0, h_1)	$(h_0, h_1, \sigma_{0/1})$	(h_0, h_1)	$(h_0, h_1, \sigma_{0/1})$	(h_0, h_1)	$(h_0, h_1, \sigma_{0/1/2})$
$pk =$	(f_0, f_1)		f_0		(f_0, f_1)	

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

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

Our inversion algorithm is Algorithm 2. It applies Algorithm 1 that computes a^{2^k-1} for some $k = 2^t$ using the following recursive equation (with $t \geq 0$)

$$\begin{aligned} S_t(a) &= a^{2^{2^t}-1} \\ &= a^{2^{2^t}-2^{2^{t-1}}+2^{2^{t-1}}-1} = a^{(2^{2^{t-1}}-1)2^{2^{t-1}}+2^{2^{t-1}}-1} \\ &= (a^{2^{2^{t-1}}-1})^{2^{2^{t-1}}} a^{2^{2^{t-1}}-1} = (S_{t-1}(a))^{2^{2^{t-1}}} S_{t-1}(a) \end{aligned}$$

Algorithm 2 is analogous to [14][Algorithm 2] that computes $a^{-1} \in \mathbb{F}_{2^\gamma}$ for $\gamma = 2^t + 1$ through Fermat's Little Theorem as

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

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

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

where h is the cyclotomic irreducible polynomial Φ_{r-1} of degree $r-1$. In this ring, $ord(a) \mid 2^{r-1} - 1$ for every $a \in \mathcal{R}^*$, and by Fermat's theorem,

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

Here, Algorithm [14][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:

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

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:

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

Algorithm 2 Inversion in $\mathcal{R} = \mathbb{F}_2[x]/((x-1)h)$ 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 Algorithm 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 of the inversion 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. Applying Algorithm 2 on Example 1 results in 17 polynomial multiplications, 17 k-squarings and 1 squaring.

Remark 1. By changing line 8 of Algorithm 2 into

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

the last square, in line 9, can be removed. This optimization is omitted from the algorithm's description for clarity.

Efficient k-squaring. The straightforward way to implement the k-squaring routine is as a series of k regular squares. The operation of squaring a binary polynomial is very efficient, i.e., it can be done in r bit operations, whereas multiplication takes r^2 bit-operations if implemented naively, or $r^{\log_2 3}$ if the Karatsuba algorithm is used (note that the bit-operation numbers are correct up to a multiplication by a constant). However, the size of k in our k-squaring operations is $O(r)$ which means that the k-squaring would require r^2 bit-operations. Moreover, modern CPU architectures offer an instruction that multiplies two 64-bit words that represent two binary polynomials in just a few cycles. With this instruction the runtime of multiplication and squaring in $\mathcal{R}$ is actually $(r/64)^{\log_2 3}$ and $r/64$ word-operations, respectively. Note that this improvement does not fully translate to the k-squaring since the number k of required consecutive squares stays the same, resulting in a total of $r^2/64$ word-operations. Therefore, this approach does not yield an efficient algorithm. Furthermore, it underlines the imbalance of the performance of the two operations required for the inversion – multiplication and k-squaring.

Fortunately, in the context of QC-MDPC codes used in post-quantum cryptographic schemes we can perform the k-squaring more efficiently by exploiting the following observation. Let $a = \sum_{j \in supp(a)} x^j \in \mathcal{R}^*$. Then we have that

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

The first step in Equation (3) is an identity for polynomials with binary coefficients. The last step stems from the fact that the order of $x \in \mathcal{R}$ is $ord(x) = r$. Therefore, k-square of an element in $\mathcal{R}$ can be computed as a permutation of the bits of the element. The only remaining question is how performant can be a secure implementation of the permutation, while at the same time the implementation admits the standard properties of side-channel protection, i.e., it is constant-time and constant-memory access.

4. Our implementation

This section discusses our implementation and further optimizations for Algorithm 2. In addition, it provides details on the optimization of the polynomial multiplication code used by BIKE Additional code package [24].

In our implementation [24], we represent elements of $\mathcal{R}$ as arrays of $r_{size} = \lceil r/8 \rceil$ bytes, where every byte represents eight consecutive coefficients. Algorithm 3 shows a naíve implementation of k-squaring an element a using a permutation map. The algorithm starts by generating the permutation map

$$\pi_k(i) : \; i \longrightarrow i \cdot 2^k \bmod r$$

for $i \in [0, r-1]$ according to Equation (3). Subsequently, it iterates over the bits of a and moves them to their destination according to this map.

Algorithm 3 Computing k-square as permutation.

Input: a as an array of r_{size} bytes, k
Output: $c = a^{2^k}$ as an array of r_{size} bytes

```
1:  procedure K_SQUARE(a, k)
2:      for i = 0 to r − 1 do                          ▷ Generate the permutation map
3:          map[i] = (i · 2^k) % r
4:      for i = 0 to r_size − 1 do                     ▷ Apply the permutation map
5:          byte = a[i]
6:          for j = 0 to 7 do
7:              bit = (byte >> j) & 1
8:              pos = map[i · 8 + j]
9:              c[pos/8] |= (bit << (pos%8))
10:     return c
```

"Inverted" permutation. Algorithm 3 performs one memory read (line 5) and eight memory writes (line 9) per byte of a. The read operations are performed on consecutive memory locations ($a[i]$, $0 \le i < r_{size}$) while the memory writes are to random locations in c that are determined by pos. In contrast, Algorithm 4 completes the same task using the inverse permutation map

$$\pi_k^{-1}(i) : \; i \cdot 2^{-k} \bmod r \longrightarrow i$$

that allows us to perform one memory write (line 11) and eight memory reads (line 7) per byte of c. This approach improves the performance of the implementation in a noticeable way and thus we chose to use it in [24].

Algorithm 4 Computing k-square as "inverted" permutation.

Input: a as an array of r_{size} bytes, k
Output: $c = a^{2^k}$ as an array of r_{size} bytes

```
1:  procedure K_SQUARE(a, k)
2:      for i = 0 to r − 1 do                          ▷ Generate the permutation map
3:          inverse_map[(i · 2^k) % r] = i
4:      for i = 0 to r_size − 1 do                     ▷ Apply the permutation map
5:          val = 0
6:          for j = 0 to 7 do
7:              pos = inverse_map[i · 8 + j]
8:              byte = a[pos/8]
9:              bit = (byte >> (pos%8)) & 1
10:             val |= (bit << j)
11:         c[i] = val
12:     return c
```

Remark 2. Our implementation of Algorithms 3 and 4 performs bit operations bitwise shift ($\ll 3$) and bitwise and (&8) instead of the slow division by 8 and modulo reduction by 7, respectively.

Efficient generation of π_k. Computing the values $\pi_k(i) = i \cdot 2^k \pmod r$, $0 \le i < r$, can be optimized by precomputing $\ell = 2^k \pmod r$ and then replacing the costly $i \cdot \ell$ multiplication with a sequence of additions, i.e., $\pi_k(i+1) = \pi_k(i) + \ell \pmod r$. The reduction can be done by subtracting r if $\pi_k(i+1) > r$. We use this optimization also for π_k^{-1} using $\ell = 2^{-k}$. This optimization can be further vectorized using SIMD instructions (see Section 4.1).

Precomputed maps. In BIKE, the values of k in Algorithm 2 depend only on the fixed public parameter r. Thus, the maps π_k can be precomputed and the performance of the permutation is now bounded by the time it take to access the maps' memory. The required memory storage is $\lfloor \log(r-2) \rfloor + 1 + wt(r-2)$ tables where each one holds r values. The performance-memory trade-off is discussed in Section 5.

k-square versus k squares. Squaring an element in $\mathcal{R}$ can be done efficiently on modern processors. Thus, for small enough values of k, it is faster to perform k square operations than performing one k-square operation. The actual choice of the threshold k_{thr} depends on the actual performance of the square and k-square implementations on a specific processor. Table D.5 in Appendix D provides examples for optimal values of k_{thr}. As a consequence, the efficiency of the inversion algorithm depends on the values of $r-2$ and $wt(r-2)$ in addition to the value of k_{thr}.

Example 2. We expect that inverting a polynomial in $\mathcal{R}_1$ (with $r_1 = 11779$) will be faster then inverting a polynomial in $\mathcal{R}_2$ (with $r_1 = 12347$). The reason is that $wt(r_1 - 2) = 5 < 6 = wt(r_2 - 2)$, and the number of required k-squares is smaller. However, from the binary representations $r_1 - 2 = \texttt{0b10111000000001}$ and $r_2 - 2 = \texttt{0b11000000111001}$, 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. Therefore,

if $k_{thr} = 64$, then for the inversion in $\mathcal{R}_1$ we can replace only one k-square with a chain of squares, while in case of $\mathcal{R}_2$ we can replace 4 such k-squares. This is another consideration that should be taken into account when choosing the r parameter for the scheme (as discussed in Section 6).

4.1. Generating permutation map with SIMD instructions

Modern Intel x86-64 processors are equipped with vector instruction-sets such as AVX that operates on 16, 128-bit registers, AVX2 that operates on 32, 256-bit registers and AVX512 that operates on 32, 512-bit registers. Except for the large registers, AVX512 brings another advantage: every instruction can be used together with a mask that indicates which vector elements are considered for the operation.

Listing 1 shows an implementation of the map generation function using the AVX512 instruction-set. The function receives $\ell = 2^{-k} \bmod r$ as its input and outputs a permutation map with r elements, each fits in a 16-bit entry. The implementation relies on two properties: a) in BIKE, $r < 2^{15}$, and the sum of two entries fits in another 16-bit entry without encountering a carry overflow; b) a single AVX512 register can hold 32 map elements.

The code loads the first 32 elements ($i \cdot \ell \pmod r$, $0 \le i < 32$) to *curr* and broadcast the values $32\ell \pmod r$ and r to *inc* and *rval*, respectively (all registers are 512-bit registers). Subsequently, the code generates 32 map values at a time by setting *curr* = *curr* + *inc* and subtracts *rval* only for elements with values greater than r.

Listing 1 Permutation map generation with AVX512 instructions (the[h] actual names of the AVX512 instructions are replaced with upper-case macro names for clarity).

```
void gen_permutation_map (uint16_t map[R], uint16_t ell) {
  __m512i curr, inc, rval;
  uint32_t mask;
  // Initialization: compute the first 32 map elements
  for (int i = 0; i < 32; i++)
    map[i] = (i * ell) % R;
  curr = LOAD(map); // Load the 32 values into the register
  inc  = BCAST_U16((ell * 32) % R);
  rval = BCAST_U16(R);

  // Generate the rest of the map elements
  for (int i = 1; i < ceil(R / 32); i++) {
    curr = ADD_U16(curr, inc);
    mask = CMP_U16(curr, rval, CMP_GEQ);
    curr = SUB_U16(curr, rval, mask);
    STORE(&map[i * 32], curr);
  }
}
```

Vector masks were introduced in AVX512, thus we cannot use them for our AVX2 implementation. In fact, instead of generating a mask, the comparison instruction produces a vector *mask*, where $mask_i = -1$ when the comparison condition holds and 0 otherwise. Consequently, we need to perform the subtraction operation in two steps: *curr* = *curr* - (*mask* and *rval*).

Another complication is that the comparison instruction compares vector elements as signed integers (in contrast to unsigned integers in AVX512). Therefore, we cannot perform more than two additions without considering carry overflows of signed integers. To sidestep this limitation, we use the following trick: in the initialization phase we set *inc* = *inc* - *rval* and in the second phase we use a comparison to 0 instead of r. Subsequently, we perform addition of r if needed. The code is found in Appendix A.

The map generation implementations can be further optimized by processing two (or more) AVX registers at a time, e.g., if in Listing 1 we use several *curr* registers we may help the processor filling the execution pipeline and eliminating any latency that arrives from the sequential nature of the instructions in the for loop. Furthermore, vector processing units usually have two input and output ports which allows them to execute some pairs of instructions in parallel and achieve higher throughput (e.g., simple arithmetic instructions such as addition, and subtraction can be executed concurrently).

To conclude, SIMD implementations of the function that generates a permutation map are fairly efficient. In the AVX2 case we need an order of $r/16$ vector instructions to generate the whole map, while the performance of the AVX512 implementation is even better since the required number of instructions is an order of $r/32$.

4.2. Optimizing the permutation with SIMD instructions

Next, we optimize the second phase of Algorithm 4 (applying the permutation map). Here, the bottleneck is the dense representation of a polynomial in r_{size} bytes, where reading a specific memory bit requires 3 shifts, 2 ands, and 1 or instructions. In contrast, by using redundant representation (or byte representation) where every bit is encoded as a complete

byte (padded with zeros), we can avoid the above calculations. Indeed, these are all "light" operations, i.e., most processors perform them in a single cycle. However, they are executed for each of the r coefficients of the polynomial, and therefore, eliminating even some of them can significantly reduce the runtime of the algorithm. When using redundant representation the code also becomes much simpler: $out[i] = in[map[i]]$ for $i \in [0, r-1]$.

Unfortunately, Algorithm 2 performs polynomial multiplication between subsequent calls to the k-square function, which operates in *binary representation* and would be very inefficient when the operands are given in byte representation. Therefore, we cannot perform the entire inversion algorithm using byte represented polynomials and we are forced to move the polynomials between the different representations. Luckily, using SIMD instructions we can implement the conversions in a very efficient way.

AVX512 implementation

Listing 2 Conversion of a polynomial from *binary* to *byte* representation using `AVX512` instructions.

```
void convert_bin_byte (uint8_t out[R], uint8_t in[R_SIZE]) {
  // Consider the input as an array of 64-bit elements
  uint64_t *in64 = (uint64_t*)in;

  for (int i = 0; i < ceil(R / 64); i++) {
    // Convert 64 bits to byte representation
    __m512i t = _mm512_maskz_set1_epi8(in64[i], 1);
    STORE(&out[i * 64], t); // Store the resulting 64 bytes to the output
  }
}
```

To convert from *binary* to *byte* representation we use the instruction called `_mm512_maskz_set1_epi8` [25], as shown in Listing 2. This instruction accepts two parameters: a 64-bit *mask* and an 8-bit *val*, and produces a vector register that contains 64 elements each of size 8 bits by broadcasting *val* to all elements of the resulting vector using zeromask *mask*, i.e., element of the vector is zeroed out when the corresponding bit of the mask is not set. Therefore, we process the input polynomial 64 bits at a time, where we use the 64 bits as the *mask* and set $val = 1$. In this way, when a bit of *mask* is set (i.e., the polynomial coefficient is one) then the corresponding byte in the output vector is set to one, otherwise it is set to zero.

The conversion in the opposite direction, *byte* to *binary* representation, can be done with `_mm512_cmp_epi8_mask` instruction [25], as shown in Listing 3. Recall from the previous section that the comparison instructions in `AVX512` receive two vectors and produce the output mask by comparing the corresponding elements of the vector. More precisely, the specified instruction takes two vectors viewed as arrays of 64 bytes compares the bytes in the corresponding positions and if they are equal sets the corresponding bit in the output mask to one. To realize the conversion, we use this instruction and provided it with a vector register containing bytes of the polynomial (each byte is zero or one) and a register where we set all bytes to one.

Listing 3 Conversion of a polynomial from *byte* to *binary* representation using `AVX512` instructions.

```
void convert_byte_bin (uint8_t out[R_SIZE], uint8_t in[R]) {
  // Consider the output as an array of 64-bit elements
  uint64_t *out64 = (uint64_t*)out;
  for (int i = 0; i < ceil(R / 64); i++) {
    // Convert 64 bytes of the input
    // and store the resulting 64 bits to the output
    __m512i one = BCAST_U8(1);
    __m512i t = LOAD(&in[i * 64]);
    out64[i] = _mm512_cmp_epi8_mask(t, one, CMP_EQ);
  }
}
```

AVX2 implementation

When only `AVX2` is available, the conversion is slightly more complicated because the two instructions we used for the `AVX512` implementation are not available. The algorithm is depicted in Fig. 1. Let $val = a_3a_2a_1a_0$ be the 32-bit value (consisting of four bytes a_i) that we convert to *byte* representation. We start by broadcasting *val* to the eight elements of the vector register t. Ideally, we would then shuffle the byte-size elements in t such that the i-th element contains the byte of *val* which contains the i-th bit of *val*, e.g., elements of t at positions 0 to 7 are set to a_0, at positions 8 to 15 are set to a_1, etc. Once we have a_i's ordered like this we can obtain the result by appropriately shifting each element of t such that the desired bit is shifted to the most significant bit position of the element, e.g., shift i-th element of t to the left by

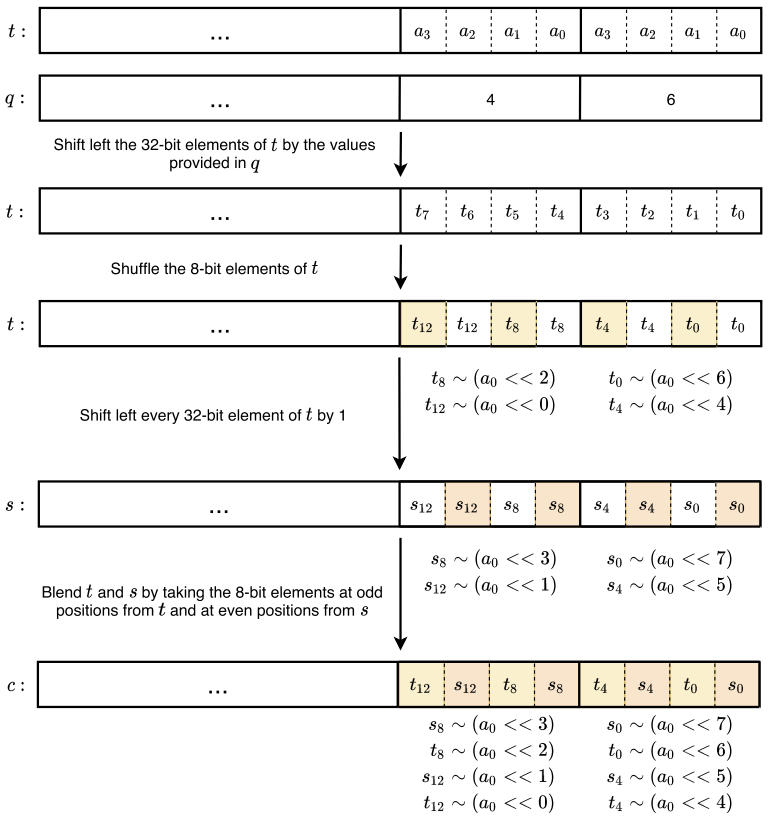

Fig. 1. Conversion of a 32-bit value $a_3a_2a_1a_0$ consisting of four bytes from *binary* to *byte* representation with AVX2 instructions.

$(7-i)$ mod 8 bit positions. Unfortunately, AVX2 does not support shift operations on byte-sized elements of a register, but only on 32-bit and 64-bit sized elements.

Remark 3. Note that, contrary to the AVX512 implementation where each byte in the *byte* representation holds the corresponding bit in the least significant bit position, here, we want the bit to be in the most significant position. The reason for this change is that it allows simple and efficient implementation of the conversion in the opposite direction, as we explain later in the text.

The above stated limitations of the AVX2 instruction set are overcome in the following way. The explanation follows the procedure in Fig. 1. Note that for clarity we depict only the conversion of a_0, the first 8 bits of *val*, however, the remaining part of *val* is simultaneously processed. First, we shift the 32-bit elements of t by the values provided in register q, e.g., the first two elements, $t[0]$ and $t[1]$ are shifted to the left by 6 and 4 places, respectively. Note that by shifting, for example, the 32-bit value $t[0] = a_3a_2a_1a_0$ to the left by 6 places, we obtain $t[0] = a'_3a'_2a'_1a'_0$ where $a'_i \neq a_i << 6$, but the most significant bit of a'_i is equal to the most significant bit of $a_i << 6$ (we denote this by the "~" sign in Fig. 1). Then we use the AVX2 shuffle instruction to reorder the byte-sized elements of t as shown in the figure. Thus, we obtain register t with the bytes in odd positions exactly as we need them for the output – they hold values with most significant bit set to the value of the corresponding bit in *val*, e.g., byte at position 1 holds the value $t_0 \sim (a_0 << 6)$. The bytes of t in even positions have to be shifted once more to the left by one place, to obtain register s that has even positioned elements filled with the right values. The resulting register is then simply generated by blending t and s. For this, we use the AVX2 blend instruction which we provide with a mask such that it copies elements at odd positions from t and at even positions from s.

Therefore, *binary* to *byte* conversion of a polynomial is performed by applying the described algorithm to every 32 consecutive bits of the polynomial, as shown in Listing 4.

Conversion from *byte* to *binary* representation is straightforward thanks to the fact that, as previously noted, the *byte* representation is such that each byte holds the corresponding bit in the most significant position. To convert 32 consecutive bytes we use `_mm256_movemask_epi8` instruction available in the AVX2 instruction set [25]. The instruction takes a vector register as input (consisting of 32 byte-sized elements) and creates a 32-bit mask from the most significant bit of each element of the register. Since this is the exact functionality that we need for the conversion, we simply iterate over the coefficients of the *byte* represented polynomial, 32 bytes at a time, and generate the desired 32 bits of output (the implementation is shown in Listing 5).

Listing 4 Conversion of a polynomial from *binary* to *byte* representation using AVX2 instructions.

```
void convert_bin_byte (uint8_t out[R], uint8_t in[R_SIZE]) {
  // shift values (q), shuffle mask (p), blend mask (w)
  __m256i p, q, w, t;
  q = SET_I32(0, 2, 4, 6, 0, 2, 4, 6);
  p = SET_I8(15, 15, 11, 11, 7, 7, 3, 3, 14, 14, 10, 10, 6, 6, 2, 2
             13, 13, 9, 9, 5, 5, 1, 1, 12, 12, 8, 8, 4, 4, 0, 0);
  w = BCAST_I16(0x00ff);

  // Consider the input as an array of 32-bit elements
  uint32_t *in32 = (uint32_t*)in;
  for (int i = 0; i < ceil(R / 32); i++) {
    // Convert 32 bits to byte representation
    t = BCAST_I32(in32[i]);
    t = _mm256_sllv_epi32(t, q); // shift left elements of t by vals in q
    t = _mm256_shuffle_epi8(t, p);
    s = _mm256_slli_epi32(t, 1); // shift left each element of t by 1
    t = _mm256_blendv_epi8(t, s, w); // blend t and s
    STORE(&out[i * 32], t); // Store the resulting 32 bytes to the output
  }
}
```

Listing 5 Conversion of a polynomial from *byte* to *binary* representation using AVX2 instructions.

```
void convert_byte_bin (uint8_t out[R_SIZE], uint8_t in[R]) {
  // Consider the output as an array of 32-bit elements
  uint32_t *out32 = (uint32_t*)out;
  for (int i = 0; i < ceil(R / 32); i++) {
    // Convert 32 bytes of the input
    // and store the resulting 32 bits to the output
    __m512i t = LOAD(&in[i * 32]);
    out32[i] = _mm256_movemask_epi8(t);
  }
}
```

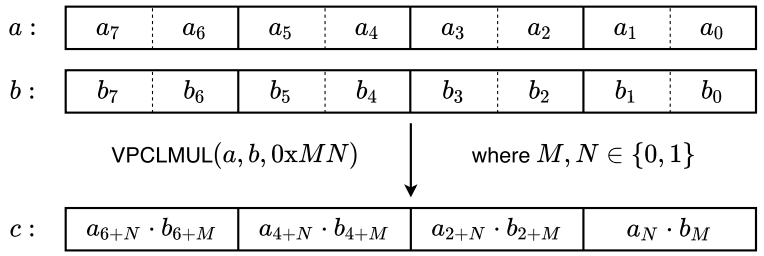

Fig. 2. VPCLMUL instruction.

4.3. Optimizing squaring and multiplication

Modern CPUs offer a fast carry-less multiplication instruction (PCLMUL) that can be used for multiplying two elements of a field with characteristic 2. We note that PCLMUL multiplies two 64-bit inputs and produces a 128-bit result. Since AVX512 and AVX2 offer many instructions that can operate on wider registers (512-bit and 256-bit, respectively), PCLMUL can be a bottleneck when polynomial multiplication is implemented with one of these SIMD instruction extension sets.

In the recent 10^{th} generation CPUs (codename "Ice Lake") Intel introduced a vectorized version of the PCLMUL instruction, namely VPCLMUL, which can multiply simultaneously four pairs of 64-bit inputs (in a SIMD manner). We leverage the new instruction to improve the performance of the existing polynomial multiplication in BIKE and to implement polynomial squaring required for the inversion. Fig. 2 shows how VPCLMUL instruction works. It receives two 512-bit registers *a* and *b*, each containing eight 64-bit elements, and a mask. The elements of *a* and *b* are grouped into four groups of two elements. The mask determines which elements (lower or higher) of the corresponding groups will be multiplied, as illustrated in the figure. Finally, the specified elements are multiplied and four 128-bit products are stored in the output register.

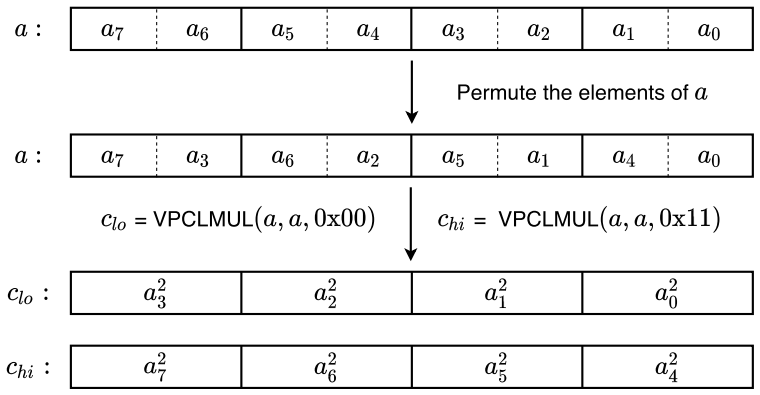

Fig. 3. Squaring eight consecutive 64-bit digits of a binary polynomial with `VPCLMUL` instruction.

Binary polynomial squaring with `VPCLMUL`

Squaring a polynomial with binary coefficients is particularly efficient. Let $a \in \mathcal{R}$ be the polynomial $a = \sum_{i=0}^{r-1} a_i x^i$. Then its square is $a^2 = \left(\sum_{i=0}^{r-1} a_i x^i\right)^2 = \sum_{i=0}^{r-1} \left(a_i x^i\right)^2$ since the coefficients of a are in $\mathbb{F}_2$. Therefore, squaring can be performed by squaring every term of the input polynomial. We note that this can be implemented with the non-vectorized `PCLMUL` instruction in a straightforward manner. Consider the polynomial as consisting of sub-polynomials with 64 terms, i.e., $a = \sum_{i=0}^{\lceil r-1/64 \rceil} a_i(x) x^{i \cdot 64}$ where $a_i(x) \in \mathbb{F}_2[x]$ with degree at most 63. Hereafter, we refer to polynomials $a_i(x)$ as 64-bit digits of a. Squaring is then implemented by iterating over the digits of a and squaring them with `PCLMUL`. The code that implements this functionality is given in Appendix B. In this section we focus on using `VPCLMUL` instruction for squaring.

In Fig. 3 we show how to square a binary polynomial of 512 bits length to obtain the 1024-bit result with `VPCLMUL` instruction. We consider the polynomial as consisting of eight 64-bit digits which occupy `AVX512` register a. To be able to get the digits in the resulting registers in correct order, we first need to permute the elements of a, as shown in the figure. Then we invoke `VPCLMUL` instruction twice, with mask `0x00` and `0x11`, to square the lower and the higher elements of the four 128-bits parts of a, respectively. In this way we obtain the result in two registers c_{lo} and c_{hi} with their elements appropriately ordered such that we can simply store them in memory. Squaring a polynomial of size r is done by iterating over it, 512 bits at a time, and applying the described algorithm (the source code of this function is given in Listing 8 in the appendix). After computing the square of a polynomial (or a product of two polynomials) we need to reduce the result modulo $x^r - 1$. The implementation of the reduction is not presented, but we note that since $x^r = 1$ the reduction can be done by shifting to the right by r places the higher part of the result (bits at positions $\geq r$) and adding it to the lower part of the result (bits at positions 0 to $r - 1$).

Binary polynomial multiplication with `VPCLMUL`

The "Additional implementation" of BIKE [22], submitted to the second round of the NIST Post-Quantum standardization project, implements polynomial multiplication with the recursive Karatsuba algorithm. The recursion splits the input into two equally sized parts, proceeds with multiplying the new parts individually in the same manner, and stops when inputs of size four 64-bit digits are encountered. Then, the *base case* multiplication is performed with a 4×4 64-bit digits schoolbook multiplication algorithm (using the `PCLMUL` instruction). Since the new `VPCLMUL` instruction operates on 512-bit registers, we replaced the existing *base case* multiplication with the code described in [26] that multiplies two binary polynomials of size eight 64-bit digits. This yields some improvements. However, we further optimize the code by implementing the *base case* as 16×16 digits multiplication using the Karatsuba algorithm with `AVX512` and `VPCLMUL` instructions.

We start by making a function that multiplies four digits of a with four digits of b:

$$c = a_3 a_2 a_1 a_0 \cdot b_3 b_2 b_1 b_0.$$

Recall that in Karatsuba's algorithm we would split the terms in half and compute the product as $c = x \cdot 2^{256} + y \cdot 2^{128} + z$ (first level of Karatsuba), with

$$x = a_3 a_2 \cdot b_3 b_2$$
$$y = (a_3 a_2 + a_1 a_0) \cdot (b_3 b_2 + b_1 b_0) + x + z$$
$$z = a_1 a_0 \cdot b_1 b_0,$$

where each of the three products (in x, y, z) would be computed in the same way (second level), i.e., by splitting the terms and computing three sub-products. Therefore, to compute c we need in total nine single digit multiplications, which if done with VPCLMUL instruction (which performs four single digit multiplications in parallel), we would need three calls to VPCLMUL. This was our initial approach. However, since we are using VPCLMUL three times it means that we can actually perform twelve single digit multiplications instead of nine at the same cost. We use this fact to our advantage to implement a hybrid between Karatsuba and schoolbook multiplication, which simplifies the algorithm and removes some additions (and register permutations). The idea is the following: replace the first level of Karatsuba by schoolbook multiplication and compute the sub-products by Karatsuba. Namely, we compute:

$$c = (a_3a_2 \cdot b_3b_2) \cdot 2^{256} + (a_3a_2 \cdot b_1b_0 + a_1a_0 \cdot b_3b_2) \cdot 2^{128} + a_1a_0 \cdot b_1b_0.$$

To compute the four sub-products with Karatsuba we need to obtain the following twelve products:

	$\underline{a_3a_2 \cdot b_3b_2:}$	$\underline{a_3a_2 \cdot b_1b_0:}$	$\underline{a_1a_0 \cdot b_3b_2:}$	$\underline{a_1a_0 \cdot b_1b_0:}$
(1)	$a_2 \cdot b_2$	$a_2 \cdot b_0$	$a_0 \cdot b_2$	$a_0 \cdot b_0$
(2)	$a_3 \cdot b_3$	$a_3 \cdot b_1$	$a_1 \cdot b_3$	$a_1 \cdot b_1$
(3)	$(a_2 + a_3)(b_2 + b_3)$	$(a_2 + a_3)(b_0 + b_1)$	$(a_0 + a_1)(b_2 + b_3)$	$(a_0 + a_1)(b_0 + b_1)$

After all the products are computed we have to perform several more additions. Firstly, since we are using Karatsuba's method to compute the four products $a_ia_j \cdot b_kb_l$, we need to add the first two computed terms to the third one, and then shift the third term (multiply by 2^{64}) and add it to the result. For example,

$$a_1a_0 \cdot b_1b_0 = a_1 \cdot b_1 \cdot 2^{128} + ((a_0 + a_1)(b_0 + b_1) + a_1 \cdot b_1 + a_0 \cdot b_0)2^{64} + a_0 \cdot b_0.$$

Finally, we need to sum the two middle $a_ia_j \cdot b_kb_l$ products, and again, shift appropriately and add to obtain the final result.

To illustrate how the whole procedure is done with AVX512 and VPCLMUL instructions we present Fig. 4. The plan is to compute the four products in each row, denoted by (1), (2), (3) in the equations above, in parallel. Let the AVX512 registers a and b hold the corresponding four 64-bit digits in the order as shown in the figure (this can be achieved with AVX512 permutation function). First, we obtain the sums that are required to compute the products in the third row. This is done by shuffling the elements of a and b to get them ordered as shown in sa and sb in the figure and then by simply adding the values of a and b to sa and sb, respectively.

Now we can use the VPCLMUL instruction to compute all the required products and store them in registers u, v, and w. Note that w holds the products of the third row which represent the middle term in Karatsuba's algorithm so we add both u and v to w. For example, the lowest 128 bits of w hold two digits $w_1w_0 = (a_0 + a_1)(b_0 + b_1) + a_1 \cdot b_1 + a_0 \cdot b_0$. Recall that to compute $(a_1a_0 \cdot b_1b_0)$ we need to add $w_1w_0 \cdot 2^{64}$ to the sum $(a_1 \cdot b_1 \cdot 2^{128} + a_0 \cdot b_0)$. Since $w_1w_0 \cdot 2^{64} = w_1 \cdot 2^{128} + w_0 \cdot 2^{64}$ this means that we can add $w_0 \cdot 2^{64}$ to the $a_0 \cdot b_0$ product (basically add w_0 to the higher digit of $a_0 \cdot b_0$) and add w_1 to the $a_1 \cdot b_1$ product. Products $a_0 \cdot b_0$ and $a_1 \cdot b_1$ are stored in u_1u_0 and v_1v_0, respectively. Therefore, we shuffle w to sw to obtain the elements of w ordered as shown in the figure and perform two additions – we add the elements of sw at odd positions to u, and the elements at even positions to v. With this the four Karatsuba multiplications are done. The only thing left to do is to permute the elements of u and v in the right order and store the result (this step is not shown in the figure).

The described algorithm to compute a 4×4 digits product is used as a function that is called inside the 8×8 digits Karatsuba multiplication. The 8×8 multiplication function takes care of providing correctly ordered input registers and also handles the output of the 4×4 multiplication. The source code which implements the 4×4 multiplication function is given in Listing 9 in Appendix C.

4.4. Side-channel protection considerations

The proposed polynomial inversion algorithm (Algorithm 2) is used during the key generation process in BIKE where a polynomial, which is a part of the secret key, has to be inverted. Because we are dealing with secret data the inversion has to be implemented securely. On the high level, the algorithm involves several polynomial multiplications and several k-squarings. We note that the number of these operations and the order in which they are performed depend only on the public parameter r (not on a given input). Therefore, the algorithm is inherently constant-time and if the required subroutines are implemented securely, then the algorithm itself is secure without any modification.

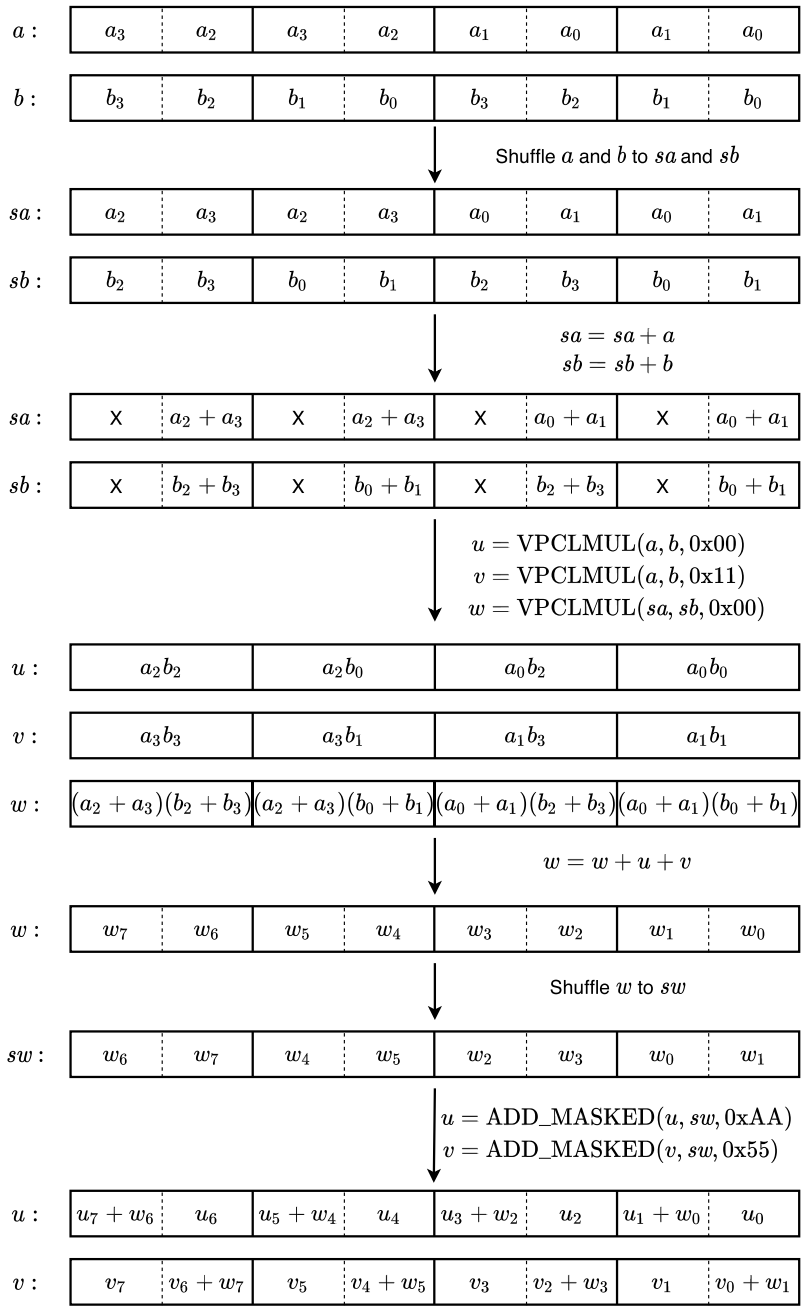

Fig. 4. Multiplying four 64-bit digits of two binary polynomials using `AVX512` and `VPCLMUL` instructions.

The subroutines used in the inversion algorithm are the following: multiplication, squaring, and k-squaring. The Additional code of BIKE [22] already implements multiplication in constant-time and the optimizations we introduce in this chapter follow the same secure implementation practices. Likewise, our implementation of polynomial squaring is constant-time and memory access. The k-squaring function is implemented as a permutation of the coefficients of the input polynomial. During the permutation we scan every bit of the input and update the appropriate bit in the output. Hence, k-squaring is constant-time. Memory locations that are accessed when copying the bits of input to the output are fully determined by the parameter k, which is itself derived solely from the public value r. Therefore, our implementation of k-squaring is also secure against side-channel attacks which exploit the knowledge of memory locations that are accessed by the program.

5. Results

In this section we provide performance results of different implementations of the inversion function. Namely, we implement and benchmark the following versions of the function:

1. PORTABLE – fully portable version of the code, implemented in C without any platform specific instructions.
2. PCLMUL – the same as PORTABLE with the exception that the `PCLMUL` instruction is used for polynomial multiplication and squaring.
3. AVX2 – implementation leveraging the instructions offered by the `AVX2` instruction set.
4. AVX512 – implementation leveraging the instructions offered by the basic `AVX512` instruction set, called `AVX512F`, which is supposed to be supported by any x86_64 platform with 512-bit wide SIMD capabilities.
5. VPCLMUL – the same as AVX512 with the exception that the new `VPCLMUL` instruction is used for polynomial multiplication and squaring.

For each of these implementation flavors we benchmark two variants of inversion based on the k-squaring implementation – generating permutation maps on the fly or using precomputed maps. Moreover, we measure and present the runtime of the BIKE-2 key generation algorithm that uses the described implementations of inversion.

The comparison baseline. The performance of our implementations is compared with two popular open-source libraries NTL (compiled with GF2X) [19,20] and OpenSSL [21]. We do not compare to [16,17,14,18] because they are all slower than NTL: a) the inversion algorithm of [17] is reported to be twice faster than [18] and 12 times faster than [14], but 1.7 times slower than NTL; b) the implementation in [16] is reported to be 3 times slower than NTL. Another reason for choosing NTL and OpenSSL for comparison is that the "Additional implementation" [22] of BIKE protocol submitted to the second round of the NIST standardization project uses the inversions from these two libraries.

Remark 4. We also measured the inversion function of the LEDAcrypt optimized code [13] that implements safegcd algorithm [15]. This code uses AVX2, so for fair comparison, we compile our code with AVX2 instructions only and compare the runtime. The performance of the LEDAcrypt inversion is: a) using gcc: 4.05 and 12.43 million cycles for Level-1 and 3, respectively; b) using clang: 3.29 and 10.30 million cycles for Level-1 and 3, respectively. The performance of our inversion on the same platform is: 0.57 and 2.08 million cycles for Level-1 and 3, respectively. The code of [13] runs in constant time and is faster than NTL. On the other hand, it is significantly 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 in either NTL [19] or OpenSSL [21] 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 $a^{-1} \bmod s$ 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 [27]). 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 internal 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 platform. We carried out performance measurements on a platform which supports all the required instructions for the five versions of the code specified above. The platform is a Dell XPS 13 7390 2-in-1 laptop. It has the latest, 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 `VPCLMUL` 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).

Measurements methodology. The performance reported hereafter is measured in processor cycles (per single core). 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 in C with intrinsic functions [25] for AVX functionality. The code is compiled with gcc (version 9.3.0), using the "-O3" optimization flag, and ran on a Linux OS (Ubuntu 20.04).

Table 2
Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r-1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([19,21]). The remaining columns represent our implementation: (a) with AVX2; (b) with AVX512; (c) with AVX512 and VPCLMUL; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles.

r	$wt(r-2)$	NTL	OSSL	(a)	(a)*	(b)	(b)*	(c)	(c)*
12323	4	6.75	49.19	0.59	0.56	0.54	0.52	0.43	0.41
11779	5	5.86	42.61	0.57	0.54	0.54	0.51	0.44	0.41
12347	6	6.52	48.67	0.64	0.63	0.60	0.58	0.47	0.45
11789	7	6.10	43.83	0.62	0.59	0.58	0.55	0.45	0.44
11821	8	5.99	44.98	0.66	0.62	0.61	0.59	0.48	0.46
11933	9	6.22	43.31	0.69	0.65	0.64	0.63	0.52	0.49
12149	10	6.37	46.60	0.75	0.71	0.70	0.67	0.55	0.52
12157	11	6.30	47.00	0.78	0.74	0.72	0.70	0.58	0.55
25603	4	9.00	213.84	1.75	1.72	1.65	1.61	1.28	1.24
24659	5	8.67	188.42	1.77	1.71	1.66	1.61	1.30	1.24
24677	6	8.61	193.27	1.88	1.83	1.74	1.71	1.35	1.32
24733	7	8.77	204.55	1.93	1.89	1.79	1.77	1.40	1.35
24821	8	9.07	185.17	2.08	2.02	1.92	1.87	1.51	1.49
25453	9	8.86	197.20	2.26	2.20	2.09	2.06	1.61	1.54
24547	10	8.32	182.11	2.13	2.08	1.99	1.95	1.61	1.53
24533	11	8.79	175.41	2.21	2.14	2.08	2.00	1.67	1.60
24509	12	8.47	181.95	2.27	2.20	2.13	2.07	1.66	1.61

Performance of inversion

The performance of the inversion algorithm depends on the Hamming weight of $r-2$ (recall that r defines the polynomial ring $\mathcal{R}$), as explained in Section 3. Therefore, we generate a set of r values, with different $wt(r-2)$, from the range of values relevant for BIKE. Then, we choose one representative for every value of $wt(r-2)$, and measure the runtime of the algorithm for the chosen parameters. Note that only r values for BIKE level 1 and 3 are considered since NIST announced that the highest level of security, Level-5, is not critical for standardization.

Tables that contain all the measurements that were performed and performance improvements over the NTL library, i.e., all the different implementation variants listed in the introduction of this section, are given in the appendix (Table E.6 and E.7). Here, we present only the most interesting and relevant data points. The AVX2 instruction set was introduced with the Haswell lineup of Intel processors in 2013. We assume that most of the CPUs in use today support at least AVX2, and therefore we present the performance numbers for AVX2 and AVX512 implementations. The third option, AVX512 plus VPCLMUL, is included to showcase the improvements that can be achieved on the latest generation of Intel CPUs and to get a glimpse of what can be expected from future processor architectures.

In Table 2 we show the runtime of the two baseline implementations, NTL and OpenSSL, together with our implementations. Firstly, we note that all the different variants of the algorithm that we implemented significantly outperform the baseline. While NTL is an order of magnitude faster than OpenSSL, our implementations are an order of magnitude faster than NTL.

It is interesting to note that for the GCD based inversion algorithms the runtime increases with the size of r, while this is not necessarily the case for our algorithm. For example, if we take $r_1 = 12323$ (first row) and $r_2 = 12157$ (last row), both NTL and OpenSSL are faster for the smaller r_2, while our implementations show a better performance for the larger r_1. This is due to the fact that $wt(r_1-2) < wt(r_2-2)$ and therefore, the algorithm proposed in this chapter performs fewer operations for r_1 than for r_2. Note that this does not hold in general for $r_1 > r_2$, especially when the corresponding weights $wt(r_1-2) < wt(r_2-2)$ are close, because even though with r_1 we perform a smaller number of operations, the operations themselves are more time consuming since the polynomial ring we work in is larger.

The use of pre-computed permutation maps for k-squaring provides an interesting trade-off. It improves the overall performance at a cost of occupying some memory space. The maps 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 in 2 bytes of memory). For example, BIKE-2 IND-CCA version (as proposed to the Round-2 NIST project) requires 450KB and 1.1MB of memory to store the maps for parameter sizes defined for Level-1 and Level-3 security, respectively. However, the performance improvements when using the maps are not very impressive – the difference in the runtime with and without precomputed maps is always around five percent. For example, the AVX2 implementations for $r = 11779$ invert a polynomial in 560K and 590K cycles with and without the precomputed maps, respectively, showing a difference of 30K cycles. The small contribution of the precomputed maps to the performance of the inversion can be attributed to the heavily optimized functions for generating permutation maps on the fly (described in Section 4.1).

It is also interesting to note the differences in the performance of the three SIMD implementations. For example, consider the columns (a), (b), and (c) of Table 2. The jump from AVX2, in (a), which operates on 256-bit wide registers to the AVX512 implementation, in (b), which works with registers of twice the size, does not improve the performance as much as we

Table 3
Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r - 1)$ compared to NTL with GF2X [19]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) `AVX2`; (b) `AVX512`; (c) `AVX512` and `VPCLMUL`; columns labeled with "*" denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c)*
12323	4	11.51	12.15	12.50	13.02	15.68	16.55
11779	5	10.26	10.80	10.85	11.45	13.32	14.37
12347	6	10.11	10.36	10.86	11.26	13.87	14.42
11789	7	9.85	10.37	10.44	11.03	13.44	13.96
11821	8	9.10	9.61	9.89	10.15	12.42	13.10
11933	9	8.97	9.55	9.67	9.93	12.03	12.70
12149	10	8.48	8.99	9.09	9.46	11.54	12.23
12157	11	8.10	8.48	8.72	9.04	10.91	11.46
25603	4	5.15	5.23	5.45	5.59	7.06	7.23
24659	5	4.89	5.06	5.22	5.40	6.66	6.98
24677	6	4.58	4.71	4.96	5.04	6.38	6.54
24733	7	4.54	4.65	4.91	4.97	6.25	6.48
24821	8	4.37	4.49	4.72	4.84	6.01	6.10
25453	9	3.92	4.03	4.23	4.31	5.51	5.74
24547	10	3.91	4.00	4.18	4.27	5.18	5.44
24533	11	3.97	4.11	4.23	4.39	5.28	5.49
24509	12	3.73	3.85	3.98	4.10	5.10	5.27

would expect. The `AVX512` is slightly faster than the `AVX2` implementation with the difference in performance around five percent. One possible reason for this is that the `AVX512` instructions that we use have higher latency compared to the used `AVX2` instructions. Another likely culprit for the unimpressive performance of `AVX512` is the platform used for the experiments which has a low-powered mobile processor designed for portable devices. Based on the previous generations of Intel CPUs, one of the ways that the power demand is lowered is by crippling the SIMD unit because it is one of the most power hungry parts of a processor. Unfortunately, these are the only 10^{th} generation IceLake Intel CPUs available on the market currently, but we expect to see higher performance improvements on the desktop and server versions of IceLake once they are released. Nevertheless, the contribution of the `VPCLMUL` instruction to the reduction in the runtime is more noticeable. For example, inversion time for $r = 11779$ drops by 100K cycles, from 540K to 440K, when `VPCLMUL` is used in addition to `AVX512`. The reason for the ~ 20 percent performance improvement here is that the bottleneck in the polynomial multiplication function when implemented with basic `AVX512` instruction set is the use of the (non-vectorized) `PCLMUL` instruction for multiplying two 64-bit digits, while the remaining part of the function is able to use the 512-bit vector registers offered by `AVX512`. The use of `VPCLMUL` does improve the situation, but it is difficult to leverage its full power when implementing the multiplication with Karatsuba's method (as explained in Section 4.3).

In Table 3 we show the relative speedups over the NTL inversion achieved by our various implementations. Depending on the specific implementation the measurements show an 8-fold to 16-fold speedup for Level-1 parameter sizes, while the improvements for Level-3 parameters are more modest, exhibiting 3-fold to 7-fold speedup over NTL. The proposed parameters in Round-2 of NIST project for CCA secure BIKE-2 are $r = 11779$ and $r = 24821$ for the first two security levels. Our most efficient implementation (`AVX512` with `VPCLMUL`) is able to invert a polynomial 14.37 times faster than NTL when $r = 11779$, and 6.1 times faster when $r = 24821$. It is interesting to note that the relative speedups for Level-1 parameter sizes are much higher than those for Level-3, meaning that NTL's implementation of the inversion scales better with the polynomial size than our implementation. This may be attributed to the fact that NTL's implementation is a GCD based inversion with linear complexity in r of the number of ring operations that are required, together with the fact that these ring operations are fairly efficient. On the other hand, our implementation requires $\lfloor \log(r-2) \rfloor + 1 + wt(r-2) - 1$ polynomial multiplications which themselves require an order of $(r/64)^{\log_2 3}$ processor instructions, and therefore might scale worse than NTL's implementation. However, we leave the investigation of this phenomenon for future research.

The speedups shown in the table highlight again the difference in the GCD based inversion algorithm of NTL and ITI based algorithm proposed in this chapter. Namely, the performance of the former one depends only on the size of the polynomials (determined by r), while the performance of the latter depends also on the value of $wt(r-2)$. This effect is embodied in the fact that the relative speedups of our implementations decrease as the corresponding weight of r increases.

Remark 5. We note that our implementations are two orders of magnitude faster than the inversion from the OpenSSL library. The exact numbers can be found in Tables E.6 and E.7 in the appendix.

Performance of BIKE key generation

In Table 4 we report the performance of BIKE key generation procedure that uses our implementation of inversion. The table contains only data for those implementations that during the inversion generate permutation maps, required for k-

Table 4
BIKE-2 key generation performance, for four relevant r values, when our implementation of the inversion algorithm is used (without precomputed maps). Columns 2-4 represent the following implementations: (a) `AVX2`; (b) `AVX512`; (c) `AVX512` and `VPCLMUL`. The runtime is measured in thousands of cycles.

r	(a)	(b)	(c)
11779	630	590	480
12323	644	587	473
24821	2222	2061	1607
24659	1913	1781	1408

squaring, on the fly, i.e., implementations without precomputed maps. For details about all our implementations for the full set of r values refer to Table E.8 in the appendix.

Table 4 present the numbers for $r = 11779$ and $r = 24821$ which are the parameters proposed for security Levels 1 and 3, respectively, in BIKE submission to the second round of the NIST PQ project. Additionally, we show the runtime of the inversion when $r = 12323$ and $r = 24659$. With these two r values we can achieve an improvement in performance of the key generation while maintaining the DFR at the level required for the corresponding security level.

It is evident from the data in Tables 2 and 4 that the cost of key generation is dominated by the cost of inversion. For example, the `AVX2` implementation for $r = 11779$ takes 570K cycles to invert the secret key polynomial, while the rest of the operations performed during the key generation take only 60K cycles, yielding in total 630K cycles to generate a key pair. This signifies the importance of having an efficient inversion in the implementation of BIKE-2 protocol.

6. Discussion

In this paper we proposed an algorithm for polynomial inversion in the context of code-based cryptographic schemes submitted to the NIST Post-Quantum Cryptography Standardization Project. The algorithm is based on the ITI algorithm [14], with some modifications that make it particularly efficient and applicable in the context of inverting elements of a polynomial ring $\mathbb{F}_2[x]/(x^r - 1)$ used for example in BIKE KEM. We also explain how this algorithm can be implemented, and indeed implement the algorithm such that it offers a very competitive performance. Moreover, our experiments show that it can substitute the NTL and OpenSSL inversion, which is used in BIKE Round-2 NIST submission, and achieve significant performance improvements.

The effect of different choices of r on BIKE 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 [28,29]. We propose an additional consideration, namely $wt(r-2)$ because the inversion Algorithm 2 is more efficient when $wt(r-2)$ is smaller. The currently recommended r for Level-1 is $r = 11779$ for which $wt(r-2) = 5$. Interestingly, a considerably larger $r = 12323$ has $wt(r-2) = 4$, and therefore offers faster key generation than $r = 11779$. Note that [29] shows that $\sim r = 12323$ is needed and sufficient in order to achieve a DFR of 2^{-128}.

BIKE-2 versus BIKE-1. Until this work, 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 desired. This left out BIKE-2's bandwidth advantage. BIKE Round-2 specification [3] addresses this difficulty by using a "batch inversion" approach that requires pre-computation of a batch of key pairs. Such solutions require that other protocols are adapted to using batched key pairs, and this introduces 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 480K/180K/1.2M cycles, and requires 1.4KB of data to be sent in each direction. By comparison, BIKE-1 (after using our latest multiplication implementation) has key generation/encapsulation/decapsulation at 67K/230K/1.3M cycles, with 2.8KB of data communicated in each direction.

As a result of our work the BIKE team decided to make BIKE-2 as their only proposed design (technically a variant of BIKE-2, per our additional independent contribution [30]), which is now called simply BIKE [1]. After evaluating the Round-2 proposals, NIST accepted BIKE as the only QC-MDPC KEM design and promoted BIKE to Round-3, as an alternative finalist.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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 thank Thorsten Kleinjung for his valuable comments on this work.

Appendix A. Generating permutation map with AVX2 instructions

In Listing 6 we show the AVX2 implementation of a function that generates the permutation map given the l parameter, as explained in Section 4.1.

Listing 6 Permutation map generation with AVX2 instructions (the actual names of the AVX2 instructions are replaced with upper-case macro names for clarity).

```
void gen_permutation_map (uint16_t map[R], uint16_t ell) {
  __m512i curr, inc, rval, zero;
  uint32_t mask;
  // Initialization: compute the first 16 map elements
  for (int i = 0; i < 16; i++)
    map[i] = (i * ell) % R;

  rval = BCAST_I16(R);
  zero = BCAST_I16(0);
  inc  = BCAST_I16((ell * 16) % R);
  inc  = SUB_I16(inc, rval);

  // Load the initial 16 values into the register
  curr = LOAD(map);

  // Generate the rest of the map elements
  for (int i = 0; i < ceil(R / 16); i++) {
    curr = ADD_I16(curr, inc);
    mask = CMP_I16(zero, curr, CMP_GT);
    curr = ADD_I16(curr, rval & mask);
    STORE(&map[i * 16], curr);
  }
}
```

Appendix B. Squaring using PCLMUL and VPCLMUL

As explained in Section 4.3, squaring of a binary polynomial a can be performed by squaring every digit of a. On platforms that offer the PCLMUL instruction, this instruction can be used to multiply two 64-bit digits, and therefore, can be used to square a digit. The PCLMUL instruction takes as an input two 128-bit values and an additional parameter denoting which 64-bit words of the input should be multiplied. For example, mask 0x00 instructs PCLMUL to multiply the two lower 64-bit words of the inputs, while the mask 0x01 requests multiplication of the lower word of the first input and the higher word of the second input. The code that implements polynomial squaring with PCLMUL is shown in Listing 7.

Listing 7 Squaring a polynomial in $\mathcal{R}$ with PCLMUL instruction.

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

When VPCLMUL instruction is available, the vectorized version of PCLMUL, we can execute four 64-bit multiplications in parallel. In Section 4.3, we explain how VPCLMUL works and moreover, how to apply it to compute a square of a binary polynomial. In Listing 8 we present the described implementation.

Listing 8 Squaring a polynomial in $\mathcal{R}$ with VPCLMUL instruction.

```
void gf2x_sqr(uint64_t *c, const uint64_t *a) {

  __m512i perm_mask = _mm512_set_epi64(7, 3, 6, 2, 5, 1, 4, 0);

  for (size_t i = 0; i < ceil(R / 512); i++) {
      __m512i va = LOAD(&a[i*8]);
      va = PERMUTE(va, perm_mask);
      STORE(&c[i*16],   VPCLMUL(va, va, 0x00));
      STORE(&c[i*16+8], VPCLMUL(va, va, 0x11));
  }
}
```

Appendix C. A 4 × 4 digits multiplication using VPCLMUL

Listing 9 shows the implementation of the function that multiplies four by four 64-bit digits of a binary polynomial. The algorithm is explained in details in Section 4.3 and Fig. 4.

Listing 9 Multiplying four 64-bit digits of two binary polynomials using AVX512 and VPCLMUL instructions as explained in Section 4.3 and Fig. 4.

```
void mul4x4(__m512i *h, __m512i *l, __m512i a, __m512i b) {

    __m512i sa = PERM64(a, _MM_SHUFFLE(2, 3, 0, 1));
    __m512i sb = PERM64(b, _MM_SHUFFLE(2, 3, 0, 1));

    sa = sa ^ a;
    sb = sb ^ b;

    __m512i u = VPCLMUL(a,  b,  0x00);
    __m512i v = VPCLMUL(a,  b,  0x11);
    __m512i w = VPCLMUL(sa, sb, 0x00);

    w = w ^ u ^ v;
    w = PERM64(w, _MM_SHUFFLE(2, 3, 0, 1));

    *l = XOR_MASKED(u, w, 0xaa);
    *h = XOR_MASKED(v, w, 0x55);
}
```

Appendix D. Example of *k*-square versus series of *k* squares

In Section 3 we explain that for values of $k < k_{thr}$ instead of performing k-squaring we perform k regular squares. The threshold k_{thr} depends on the implementation and the platform. For example, in Table D.5 we compare the performance of squaring and k-squaring in $\mathcal{R}$ using AVX512 and VPCLMUL instructions, and compute the thresholds.

Table D.5
Squaring and k-squaring in $\mathcal{R}$ using our code (AVX512 and VPCLMUL). Columns 2 and 3 count cycles. The threshold is computed by $k_{thr} = k$-square/square. The r values correspond to the IND-CCA variants of BIKE for Level-1/3.

r	k-square	square	k_{thr}
11779	16000	230	69
24821	35000	510	68

Appendix E. Performance results

Table E.6
Performance of our implementations of inversion in $\mathbb{F}_2[x]/(x^r - 1)$ for a set of r values with different $wt(r-2)$. The NTL and OSSL columns denote the runtime of the inversion from the corresponding libraries ([19,21]). The remaining columns represent our implementation: (a) with `AVX2`; (b) with `AVX512`; (c) with `AVX512` and `VPCLMUL`; (d) fully portable implementation, independent of any platform; (e) portable with `PCLMUL` instruction used for multiplication and squaring; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in millions of cycles.

r	$wt(r-2)$	NTL	OSSL	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	6.75	49.19	12.79	0.56	0.54	0.52	0.43	0.41	12.64	0.95	0.78	0.59
11779	5	5.86	42.61	11.81	0.54	0.54	0.51	0.44	0.41	11.42	1.15	0.79	0.57
12347	6	6.52	48.67	15.03	0.63	0.60	0.58	0.47	0.45	14.67	1.15	0.86	0.64
11789	7	6.10	43.83	12.95	0.59	0.58	0.55	0.45	0.44	12.74	1.05	0.84	0.62
11821	8	5.99	44.98	14.04	0.62	0.61	0.59	0.48	0.46	13.98	1.10	0.89	0.66
11933	9	6.22	43.31	14.50	0.65	0.64	0.63	0.52	0.49	14.28	1.18	0.94	0.69
12149	10	6.37	46.60	15.60	0.71	0.70	0.67	0.55	0.52	15.31	1.29	1.02	0.75
12157	11	6.30	47.00	16.57	0.74	0.72	0.70	0.58	0.55	16.23	1.33	1.06	0.78
25603	4	9.00	213.84	39.10	1.72	1.65	1.61	1.28	1.24	38.62	2.78	2.33	1.75
24659	5	8.67	188.42	41.75	1.71	1.66	1.61	1.30	1.24	40.94	3.10	2.35	1.77
24677	6	8.61	193.27	44.41	1.83	1.74	1.71	1.35	1.32	43.53	3.19	2.48	1.88
24733	7	8.77	204.55	46.47	1.89	1.79	1.77	1.40	1.35	45.65	3.30	2.56	1.93
24821	8	9.07	185.17	49.16	2.02	1.92	1.87	1.51	1.49	49.00	3.24	2.73	2.08
25453	9	8.86	197.20	51.42	2.20	2.09	2.06	1.61	1.54	50.86	3.93	2.97	2.26
24547	10	8.32	182.11	45.81	2.08	1.99	1.95	1.61	1.53	44.46	4.07	2.88	2.13
24533	11	8.79	175.41	47.10	2.14	2.08	2.00	1.67	1.60	46.14	4.11	3.00	2.21
24509	12	8.47	181.95	50.24	2.20	2.13	2.07	1.66	1.61	50.06	3.67	3.05	2.27

Table E.7
Speedup of our implementations of inversion in $\mathbb{F}_2[x]/(x^r - 1)$ compared to NTL with GF2X [19]. Columns 3-8 represent the speedup over NTL of the following implementation: (a) `AVX2`; (b) `AVX512`; (c) `AVX512` and `VPCLMUL`; (d) PORTABLE; (e) `PCLMUL`; columns labeled with "*" denote implementations with pre-computed permutation maps. The speedup is measured for a set of r values with different $wt(r-2)$.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	11.51	12.15	12.50	13.02	15.68	16.55	0.53	0.53	7.12	8.68
11779	5	10.26	10.80	10.85	11.45	13.32	14.37	0.50	0.51	5.11	7.46
12347	6	10.11	10.36	10.86	11.26	13.87	14.42	0.43	0.44	5.64	7.61
11789	7	9.85	10.37	10.44	11.03	13.44	13.96	0.47	0.48	5.79	7.26
11821	8	9.10	9.61	9.89	10.15	12.42	13.10	0.43	0.43	5.46	6.75
11933	9	8.97	9.55	9.67	9.93	12.03	12.70	0.43	0.44	5.29	6.59
12149	10	8.48	8.99	9.09	9.46	11.54	12.23	0.41	0.42	4.93	6.23
12157	11	8.10	8.48	8.72	9.04	10.91	11.46	0.38	0.39	4.75	5.94
25603	4	5.15	5.23	5.45	5.59	7.06	7.23	0.23	0.23	3.23	3.87
24659	5	4.89	5.06	5.22	5.40	6.66	6.98	0.21	0.21	2.80	3.69
24677	6	4.58	4.71	4.96	5.04	6.38	6.54	0.19	0.20	2.69	3.47
24733	7	4.54	4.65	4.91	4.97	6.25	6.48	0.19	0.19	2.66	3.43
24821	8	4.37	4.49	4.72	4.84	6.01	6.10	0.18	0.19	2.80	3.32
25453	9	3.92	4.03	4.23	4.31	5.51	5.74	0.17	0.17	2.25	2.98
24547	10	3.91	4.00	4.18	4.27	5.18	5.44	0.18	0.19	2.05	2.88
24533	11	3.97	4.11	4.23	4.39	5.28	5.49	0.19	0.19	2.14	2.93
24509	12	3.73	3.85	3.98	4.10	5.10	5.27	0.17	0.17	2.31	2.78

Table E.8
BIKE-2 key generation performance when our implementation of the inversion algorithm is used. Columns represent the following implementations: (a) `AVX2`; (b) `AVX512`; (c) `AVX512` and `VPCLMUL`; (d) PORTABLE; (e) `PCLMUL`; columns labeled with "*" denote implementations with pre-computed permutation maps. The runtime is measured in thousands of cycles.

r	$wt(r-2)$	(a)	(a)*	(b)	(b)*	(c)	(c*)	(d)	(d)*	(e)	(e)*
12323	4	642	644	581	587	470	473	13829	13574	1344	1121
11779	5	625	630	585	591	477	479	12651	12530	1454	1304
12347	6	709	708	642	650	512	516	15882	15773	1431	1332
11789	7	672	674	623	629	500	510	13959	13635	1516	1226
11821	8	709	715	656	658	520	529	15101	14786	1556	1272
11933	9	743	751	692	696	551	561	15552	15226	1659	1334
12149	10	806	806	743	748	594	593	16746	16291	1818	1451
12157	11	829	842	769	772	616	621	17659	17288	1865	1493
25603	4	1907	1906	1773	1762	1440	1391	42186	41515	3910	3241
24659	5	1944	1913	1777	1781	1406	1408	44485	44131	3881	3550
24677	6	2024	1994	1865	1892	1474	1454	47236	46788	3979	3659
24733	7	2126	2097	1918	1908	1509	1504	49141	48942	4107	3769
24821	8	2246	2222	2064	2061	1648	1607	52216	51642	4392	3697
25453	9	2420	2414	2241	2230	1732	1703	54726	53957	4786	4388
24547	10	2324	2299	2159	2172	1700	1706	48211	47725	5001	4493
24533	11	2367	2348	2213	2190	1763	1733	49638	49292	5037	4587
24509	12	2454	2432	2271	2239	1781	1765	53145	52372	5004	4115

References

[1] N. Aragon, P.S.L.M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, C.A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, G. Zémor, BIKE: bit flipping key encapsulation, https://bikesuite.org/files/v4.1/BIKE_Spec.2020.10.22.1.pdf, 2020.

[2] NIST, Post-quantum cryptography, https://csrc.nist.gov/projects/post-quantum-cryptography, 2019. (Accessed 20 August 2019).

[3] N. Aragon, P.S.L.M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C.A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, G. Zémor, BIKE: bit flipping key encapsulation, https://bikesuite.org/files/round2/spec/BIKE-Spec-2019.06.30.1.pdf, 2019.

[4] R.J. McEliece, A public-key cryptosystem based on algebraic coding theory, Deep Space Network Progress Report 44 (1978) 114–116, https://ui.adsabs.harvard.edu/abs/1978DSNPR..44..114M.

[5] H. Niederreiter, Knapsack-type cryptosystems and algebraic coding theory, Probl. Control Inf. Theory 15 (2) (1986) 157–166, https://ci.nii.ac.jp/naid/80003180051/en/.

[6] N. Drucker, S. Gueron, D. Kostic, Fast polynomial inversion for post quantum QC-MDPC cryptography, in: S. Dolev, V. Kolesnikov, S. Lodha, G. Weiss (Eds.), Cyber Security Cryptography and Machine Learning, Springer International Publishing, Cham, 2020, pp. 110–127.

[7] N. Drucker, S. Gueron, D. Kostic, Fast polynomial inversion for post quantum QC-MDPC cryptography, Cryptology ePrint Archive, Report 2019/298 (2020) https://eprint.iacr.org/2020/298.

[8] Open Quantum Safe Project, liboqs, https://github.com/open-quantum-safe/liboqs, 2020. (Accessed 16 February 2020).

[9] Amazon Web Services, s2n, https://github.com/awslabs/s2n, 2020. (Accessed 16 February 2020).

[10] R. Misoczki, BIKE - bit-flipping key encapsulation, https://csrc.nist.gov/CSRC/media/Presentations/bike-round-2-presentation/images-media/bike-misoczki.pdf, 2019. (Accessed 18 February 2020).

[11] C. Aguilar Melchor, N. Aragon, S. Bettaieb, B. Loïc, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, Hamming quasi-cyclic (HQC), https://pqc-hqc.org/doc/hqc-specification_2017-11-30.pdf, 2017.

[12] A. Hülsing, J. Rijneveld, J. Schanck, P. Schwabe, High-speed key encapsulation from NTRU, in: Wieland Fischer, Naofumi Homma (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2017, Springer International Publishing, Cham, 2017, pp. 232–252.

[13] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, P. Santini, LEDAcrypt, https://www.ledacrypt.org/, 2019.

[14] T. Itoh, S. Tsujii, A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases, Inf. Comput. 78 (3) (1988) 171–177, https://doi.org/10.1016/0890-5401(88)90024-7.

[15] D.J. Bernstein, B.-Y. Yang, Fast constant-time gcd computation and modular inversion, IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019 (3) (2019) 340–398, https://doi.org/10.13154/tches.v2019.i3.340-398.

[16] A. Guimarães, D.F. Aranha, E. Borin, Optimized implementation of QC-MDPC code-based cryptography, Concurr. Comput., Pract. Exp. 31 (18) (2019) e5089, https://doi.org/10.1002/cpe.5089.

[17] A. Guimar, E. Borin, D.F. Aranha, A. Guimarães, E. Borin, D.F. Aranha, Introducing arithmetic failures to accelerate QC-MDPC code-based cryptography, Code Based Cryptogr. 2 (2019) 44–68, https://doi.org/10.1007/978-3-030-25922-8.

[18] 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 Trans. Comput. 53 (3) (2004) 375–380, https://doi.org/10.1109/TC.2004.1261843.

[19] V. Shoup, Number theory c++ library (ntl) version 11.3.2, http://www.shoup.net/ntl, November 2018.

[20] P.Z. Pierrick Gaudry, Richard Brent, E. Thome, gf2x-1.2, https://gforge.inria.fr/projects/gf2x/, July 2017.

[21] The OpenSSL Project, OpenSSL 1.1.1: the open source toolkit for SSL/TLS, https://github.com/openssl/openssl.

[22] N. Drucker, S. Gueron, D. Kostic, Additional implementation of BIKE, https://bikesuite.org/additional.html, 2019.

[23] J.W. Bos, T. Kleinjung, R. Niederhagen, P. Schwabe, ECC2K-130 on cell CPUs, in: D.J. Bernstein, T. Lange (Eds.), Progress in Cryptology – AFRICACRYPT 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 225–242.
[24] N. Drucker, S. Gueron, D. Kostic, Additional implementation of BIKE, https://github.com/awslabs/bike-kem, 2020.
[25] N. Drucker, S. Gueron, D. Kostic, Intel®64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
[26] N. Drucker, S. Gueron, V. Krasnov, Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction, in: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), 2018, pp. 115–119.
[27] Shay Gueron, https://github.com/open-quantum-safe/openssl/issues/42%23issuecomment-433452096, October 2018.
[28] N. Sendrier, V. Vasseur, On the decoding failure rate of QC-MDPC bit-flipping decoders, in: J. Ding, R. Steinwandt (Eds.), Post-Quantum Cryptography, Vol. 2, Springer International Publishing, Cham, 2019, pp. 404–416.
[29] N. Drucker, S. Gueron, D. Kostic, QC-MDPC decoders with several shades of gray, Cryptology ePrint Archive, Report 2019/1423, Dec 2019, https://eprint.iacr.org/2019/1423.
[30] N. Drucker, S. Gueron, D. Kostic, E. Persichetti, On the Applicability of the Fujisaki-Okamoto Transformation to the BIKE KEM, Cryptology ePrint Archive, Report 2020/510, 2020, https://eprint.iacr.org/2020/510.