QC-MDPC Decoders with Several Shades of Gray

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

[1] University of Haifa, Haifa, Israel
drucker.nir@gmail.com
[2] Amazon, Seattle, USA
[3] EPFL, Lausanne, Switzerland

Abstract. QC-MDPC code-based KEMs rely on decoders that have a small or even negligible Decoding Failure Rate (DFR). These decoders should be efficient and implementable in constant-time. One example for a QC-MDPC KEM is the Round-2 candidate of the NIST PQC standardization project, "BIKE". We have recently shown that the Black-Gray decoder achieves the required properties. In this paper, we define several new variants of the Black-Gray decoder. One of them, called Black-Gray-Flip, needs only 7 steps to achieve a smaller DFR than Black-Gray with 9 steps, for the same block size. On current `AVX512` platforms, our BIKE-1 (Level-1) constant-time decapsulation is 1.9× faster than the previous decapsulation with Black-Gray. We also report an additional 1.25× decapsulating speedup using the new `AVX512-VBMI2` and vector-`PCLMULQDQ` instructions available on "Ice-Lake" micro-architecture.

Keywords: BIKE · QC-MDPC codes · Constant-time implementation · QC-MDPC decoders

1 Introduction

The Key Encapsulation Mechanism (KEM) called Bit Flipping Key Encapsulation (BIKE) [2] is based on Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes, and is one of the Round-2 candidates of the NIST PQC Standardization Project [15]. The submission includes several variants of the KEM and we focus here on BIKE-1-CCA Level-1 and Level-3.

The common QC-MDPC decoding algorithms are derived from the Bit-Flipping algorithm [12] and come in two main variants.

- "Step-by-Step": it recalculates the threshold every time that a bit is flipped. This is an enhancement of the "in-place" decoder described in [11].
- "Simple-Parallel": a parallel algorithm similar to that of [12]. It first calculates some thresholds for flipping bits and then flips the bits in all of the relevant positions, in parallel.

J. Ding and J.-P. Tillich (Eds.): PQCrypto 2020, LNCS 12100, pp. 35–50, 2020.
https://doi.org/10.1007/978-3-030-44223-1_3

BIKE uses a decoder for the decapsulation phase. The specific decoding algorithm is a choice shaped by the target DFR, security, and performance. The IND-CCA version of BIKE Round-2 [2] is specified with the "BackFlip" decoder, which is derived from Simple-Parallel. The IND-CPA version is specified with the "One-Round" decoder, which combines the Simple-Parallel and the Step-By-Step decoders. In the "additional implementation" [7] we chose to use the "Black-Gray" decoder (BG) [5,8], with the thresholds defined in [2]. This decoder (with different thresholds) appears in the BIKE pre-Round-1 submission "CAKE" and is due to N. Sendrier and R. Misoczki.

This paper explores a new family of decoders that combine the BG and the Bit-Flipping algorithms in different ways. Some combinations achieve the same or even better DFR compared to BG with the same block size, and at the same time also have better performance.

For better security we replace the mock-bits technique of the additional implementation [5] with a constant-time implementation that applies rotation and bit-slice-adder as proposed in [3] (and vectorized in [13]), and enhance it with further optimizations. We also report the first measurements of BIKE-1 on the new Intel "Ice-Lake" micro-architecture, leveraging the new `AVX512-VBMI2`, vector-`AESENC` and vector-`PCLMULQDQ` instructions [1] (see also [4,10]).

The paper is organized as follows. Section 2 defines notation and offers some background. The Bit-Flipping and the BG algorithms are given in Sect. 3. In Sect. 4 we define new decoders (BGF, B and BGB) and report our DFR per block size studies in Sect. 5. We discuss our new constant-time QC-MDPC implementation in Sect. 6. Section 7 reports the resulting performance. Section 8 concludes the paper.

2 Preliminaries and Notation

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 every element $v \in \mathcal{R}$ its Hamming weight is denoted by $wt(v)$, its bits length by $|v|$, and its support (i. e., the positions of its set bits) by $supp(v)$. Polynomials in $\mathcal{R}$ are viewed, interchangeably, also as square circulant matrices in $\mathbb{F}_2^{r\times r}$. For a matrix $H \in \mathbb{F}_2^{r\times r}$, let H_i denote its i-th column written as a row vector. We denote a failure by the symbol $\perp$. Uniform random sampling from a set W is denoted by $w \xleftarrow{\$} W$. For an algorithm A, we denote its output by $out = A()$ if A is deterministic, and by $out \leftarrow A()$ otherwise. Hereafter, we use the notation $x.ye{-}z$ to denote the number $(x + \frac{y}{10}) \cdot 10^{-z}$ (e. g., $1.2e{-}3 = 1.2 \cdot 10^{-3}$.

BIKE-1 IND-CCA. BIKE-1 (IND-CCA) flows are shown in Table 1. The computations are executed over $\mathcal{R}$, and the block size r is a parameter. The weights of the secret key $h = (h_0, h_1, \sigma_0, \sigma_1)$ and the errors vector $e = (e_0, e_1)$, are w and t, respectively, the public key, ciphertext, and shared secret are $f = (f_0, f_1)$, $c = (c_0, c_1)$, and k, respectively. $\mathbf{H}$, $\mathbf{K}$ denote hash functions (as in [2]). Currently, the parameters of BIKE-1 IND-CCA for NIST Level-1 are: $r = 11,779$, $|f| = |c| = 23,558$, $|k| = 256$, $w = 142$, $d = w/2 = 71$ and $t = 134$.

Table 1. BIKE-1-CCA

Key generation	• $h_0, h_1 \xleftarrow{\$} \mathcal{R}$ of odd weight $wt(h_0) = wt(h_1) = w/2$ • $\sigma_0, \sigma_1 \xleftarrow{\$} \mathcal{R}$ • $g \xleftarrow{\$} \mathcal{R}$ of odd weight (so $wt(g) \approx r/2$) • $(f_0, f_1) = (gh_1, gh_0)$
Encapsulation	• $m \xleftarrow{\$} \mathcal{R}$ • $(e_0, e_1) = \mathbf{H}(mf_0, mf_1)$ where $wt(e_0) + wt(e_1) = t$ • $(c_0, c_1) = (mf_0 + e_0, mf_1 + e_1)$ • $k = \mathbf{K}(mf_0, mf_1, c_0, c_1)$
Decapsulation	• Compute the syndrome $s = c_0h_0 + c_1h_1$ • $(e'_o, e'_1) \leftarrow \text{decode}(s, h_0, h_1)$ • If $wt((e'_0, e'_1)) \neq t$ or decoding failed then $k = \mathbf{K}(\sigma_0, \sigma_1, c)$ • else $k = \mathbf{K}(c_0 + e'_0, c_1 + e'_1, c_0, c_1)$

3 The Bit-Flipping and the Black-Gray Decoders

Algorithm 1 describes the Bit-Flipping decoder [12]. The `computeThreshold` step computes the relevant threshold according to the syndrome, the errors vector, or the Unsatisfied Parity-Check (UPC) values. The original definition of [12] takes the maximal UPC as its threshold.

Algorithm 1. e=Bit-Flipping(c, H)

Input: $H \in \mathbb{F}_2^{r \times n}$ (parity-check matrix), $c \in \mathbb{F}_2^n$ (ciphertext), X (Maximal number of iterations), u (Maximal syndrome weight)
Output: $e \in \mathbb{F}_2^n$ (errors vector)
Exception: A "decoding failure" returns $\perp$

1: **procedure** BIT-FLIPPING(c, H)
2: s = Hc^T, e = 0, upc[n-1:0] = 0^n
3: **for** $itr = 0 \dots X$ **do**
4: th = computeThreshold(s,e)
5: **for** i in $0 \dots n-1$ **do**
6: $upc[i] = H_i \cdot s$
7: **if** $upc[i] > th$ **then** $e[i] = e[i] \oplus 1$ ▷ Flip an error bit
8: s = $H(c^T + e^T)$ ▷ Update the syndrome
9: **if** $(wt(s) = u)$ **then return** e
10: **else return** $\perp$

Algorithm 2 describes the BG decoder. It is implemented in BIKE additional code package [7]. Every iteration of BG involves three main steps. Step I calls `BitFlipIter` to perform one Bit-Flipping iteration and sets the `black` and

`gray` arrays. Steps II and III call `BitFlipMaskedIter`. Here, another Bit-Flipping iteration is executed, but the errors vector e is updated according to the `black/gray` masks, respectively.

In Step I the decoder uses some threshold (th) to decide whether or not a certain bit is an error bit. The probability that the bit is indeed an error bit increases as a function of the gap ($upc[i]$ - th). The algorithm records bits with a small gap in the `black/gray` masks so that the subsequent Step II and Step III can use the masks in order to gain more confidence in the flipped bits. In this paper $\delta = 4$.

Algorithm 2. e=BG(c, H)

Input: $H \in \mathbb{F}_2^{r \times n}$ (parity-check matrix), $c \in \mathbb{F}_2^n$ (ciphertext), X_{BG} (maximal number of iterations)
Output: $e \in \mathbb{F}_2^n$ (errors vector)
Exception: A "decoding failure" returns $\perp$

procedure BitFlipIter(s, e, th, H)
  $black[n-1:0] = gray[n-1:0] = 0^n$
  for i in $0 \ldots n-1$ **do**
    $upc[i] = H_i \cdot s$
    if $upc[i] \geq th$ **then**
      $e[i] = e[i] \oplus 1$ ▷ Flip an error bit
      $black[i] = 1$ ▷ Update the Black set
    else if $upc_i >= th - \delta$ **then**
      $gray[i] = 1$ ▷ Update the Gray set
  $s = H(c^T + e^T)$ ▷ Update the syndrome
  return $(s, e, black, gray)$

procedure BitFlipMaskedIter($s, e, mask, th, H$)
  for i in $0 \ldots n-1$ **do**
    $upc[i] = H_i \cdot s$
    if $upc[i] \geq th$ **then**
      $e[i] = e[i] \oplus mask[i]$ ▷ Flip an error bit
  $s = H(c^T + e^T)$ ▷ Update the syndrome
  return (s, e)

procedure Black-Gray(c, H)
  $s = Hc^T$, $e[n-1:0] = 0^n$, $\delta = 4$
  for itr in $1 \ldots X_{BG}$ **do**
    th = computeThreshold(s)
    $(s, e, black, gray)$ = BitFlipIter(s, e, th, H) ▷ Step I
    (s, e) = BitFlipMaskedIter$(s, e, black, ((d+1)/2), H)$ ▷ Step II
    (s, e) = BitFlipMaskedIter$(s, e, gray, ((d+1)/2), H)$ ▷ Step III
  if $(wt(s) \neq 0)$ **then**
    return $\perp$
  else
    return e

4 New Decoders with Different Shades of Gray

In cases where Algorithm 2 can safely run without a constant-time implementation, Step II and Step III are fast. The reason is that the UPC values are calculated only for indices in $supp(black)/supp(gray)$, and the number of these indices is at most the number of bits that were flipped in Step I (certainly less than n). By contrast, if constant-time and constant memory-access are required, the implementation needs to access all of the n positions uniformly. In such case the performance of Step II and Step III is similar to the performance of Step I. Thus, the overall decoding time of the BG decoder with X_{BG} iterations, where each iteration is executing steps I, II, and III, is proportional to $3 \cdot X_{BG}$.

The decoders that are based on Bit-Flipping are not perfect - they can erroneously flip a bit that is not an error bit. The probability to erroneously flip a "non-error" bit is an increasing function of $wt(e)/n$ and also depends on the threshold (note that $wt(e)$ is changing during the execution). Step II and Step III of BG are designed to fix some erroneously flipped bits and therefore decrease $wt(e)$ compared to $wt(e)$ after one iteration of Simple-Parallel (without the `black`/`gray` masks). Apparently, when $wt(e)/n$ becomes sufficiently small the black/gray technique is no longer needed because erroneous flips have low probabilities. This observation leads us to propose several new variations of the BG decoder (see Appendix A for their pseudo-code).

1. A Black decoder (B): every iteration consists of only Steps I, II (i. e., there is no gray mask).
2. A Black-Gray-Flip decoder (BGF): it starts with one BG iteration and continues with several Bit-Flipping iterations.
3. A Black-Gray-Black decoder (BGB): it starts with one BG iteration and continues with several B-iterations.

Example 1 (Counting the number of steps). Consider BG with 3 iterations. Here, every iteration involves 3 steps (I, II, and III). The total number of practically identical steps is 9. Consider, BGF with 3 iterations. Here, the first iteration involves 3 steps (I, II, and III) and the rest of the iterations involve only one step. The total number of practically identical steps is $3 + 1 + 1 = 5$.

5 DFR Evaluations for Different Decoders

In this section we evaluate and compare the B, BG, BGB, and BGF decoders under two criteria.

1. The DFR for a given number of iterations and a given value of r.
2. The value of r that is required to achieve a target DFR with a given number of iterations.

In order to approximate the DFR we use the extrapolation method [16], and apply two forms of extrapolation: "best linear fit" [8] and "two large r's fit" (as in [8][Appendix C]). We point out that the extrapolation method relies on the

assumption that the dependence of the DFR on the block size r is a concave function in the relevant range of r. Table 2 summarizes our results. It shows the r-value required for achieving a DFR of $2^{-23}(\approx 10^{-8})$, 2^{-64}, and 2^{-128}. It also shows the approximated DFR for $r = 11,779$ (which is the value used for BIKE-1 Level-1 CCA). Appendix B provides the full information on the experiments and the extrapolation analysis.

Table 2. The DFR achieved by different decoders. Two extrapolation methods are shown: "best linear fit" (as in [8]), "two large r's fit" (as in [8][Appendix C]). The second column shows the number of iterations for each decoder. The third column shows the total number of (time-wise identical) executed steps.

			Best linear fit				Two large r's fit			
Decoder	#I	#S	DFR = 2^{-23}	2^{-64}	2^{-128}	DFR at 11,779	DFR = 2^{-23}	2^{-64}	2^{-128}	DFR at 11,779
BG	3	9	10,253	11,213	12,739	2^{-88}	10,253	11,171	12,619	2^{-90}
	4	12	10,163	11,003	12,347	2^{-100}	10,163	10,909	12,107	2^{-110}
	5	15	10,133	10,909	12,107	2^{-111}	10,133	10,853	11,987	2^{-116}
BGB	4	9	10,253	11,093	12,491	2^{-95}	10,253	11,083	12,491	2^{-96}
	5	11	10,163	10,973	12,227	2^{-105}	10,163	11,027	12,413	2^{-99}
	6	13	10,133	10,973	12,269	2^{-104}	10,133	10,949	12,197	2^{-107}
BGF	5	7	10,301	11,171	12,539	2^{-92}	10,301	11,131	12,491	2^{-95}
	6	8	10,253	11,027	12,277	2^{-102}	10,253	10,973	12,197	2^{-107}
	7	9	10,181	10,949	12,149	2^{-108}	10,181	10,949	12,107	2^{-112}
B	4	8	10,259	11,699	13,901	2^{-67}	10,301	11,813	14,221	2^{-63}
	5	10	10,133	11,437	13,229	2^{-79}	10,133	11,437	13,451	2^{-76}
	6	12	10,067	11,213	13,037	2^{-84}	10,067	11,437	13,397	2^{-78}

Interpreting the Results of Table 2. The conclusions from Table 2 indicate that it is possible to trade BG with 3 iterations for BGF with 6 iterations. This achieves a better DFR and also a $\frac{9}{8} = 1.125\times$ speedup. Moreover, if the required DFR is at most 2^{-64}, it suffices to use BGF with only 5 iterations (and get the same DFR as BG with 3 iterations). This achieves a factor of $\frac{9}{7} = 1.28\times$ speedup. The situation is similar for BG with 4 iterations compared to BGB with 5 iterations: this achieves a $\frac{12}{11} = 1.09\times$ speedup. If a DFR of 2^{-128} is required it is possible to trade BG with 4 iterations for BGF with 7 iterations and achieve a $\frac{12}{9} = 1.33\times$ speedup. Another interesting trade off is available if we are willing to slightly increase the value of r. Compare BG with 4 iterations (i. e., 12 steps) and BGF with 6 iterations (i. e., 8 steps). For a DFR of 2^{-64} we have $r_{BG} = 11,003$ and $r_{BGF} = 11,027$. A very small relative increase in the block size, namely $(r_{BGF} - r_{BG})/r_{BG} = 0.0022$, gives a $\frac{12}{8} = 1.5\times$ speedup.

Example 2 (BGF versus BG with 3 iterations). Fig. 1 shows a qualitative comparison (the precise details are provided in Appendix B). The left panel indicates

that BGF has a better DFR than BG for the same number of (9) steps when $r > 9,970$. Similarly, The right panel shows the same phenomenon even with a smaller number of BGF steps (7) when $r > 10,726$ (with the best linear fit method) and $r > 10,734$ (with the two large r's method) that correspond to a DFR of 2^{-43} and 2^{-45}, respectively. Both panels show that that crossover point appears for values of r below the range that is relevant for BIKE.

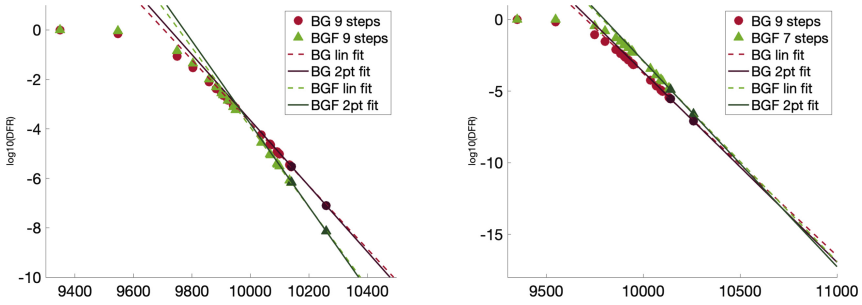

Fig. 1. DFR comparison of BG with 3 iterations (9 steps) to BGF with: (Left panel) 7 iterations (9 steps); (Right panel) 5 iterations (7 steps). See the text for details.

6 Constant-Time Implementation of the Decoders

The mock-bits technique was introduced in [5] for side-channel protection in order to obfuscate the (secret) $supp(h_0), supp(h_1)$. Let M_i denote the mock-bits used for obfuscating $supp(h_i)$ and let $\overline{M_i} = M_i \sqcup supp(h_i)$. For example, the implementation of BIKE-1 Level-1 used $|M_i| = 62$ mock-bits and thus $|\overline{M_i}| = 133$. The probability to correctly guess the secret 71 bits of h_i if the whole set $|\overline{M_i}|$ is given is $\binom{133}{71}^{-1} \approx 2^{-128}$. This technique was designed for ephemeral keys but may leak information on the private key if it is used multiple times (i. e., if most of $|\overline{M_i}|$ can be trapped). By knowing that $supp(h_i) \subset \overline{M_i}$, an adversary can learn that all the other $(r - |\overline{M_i}|)$ bits of h_i are zero. Subsequently, it can generate the following system of linear equations $(h_0, h_1)^T \cdot (f_0, f_1) = 0$, set the relevant variables to zero and solve it. To avoid this, $|\overline{M_i}|$ needs to be at least $r/2$ (probably more) so the system is sufficiently undetermined. However, using more than M_i mock-bits makes this method impractical (it was used as an optimization to begin with).

Therefore, to allow multiple usages of the private key we modify our implementation and use some of the optimizations suggested in [3] that were later vectorized in [13][1]. Specifically, we leverage the (array) rotation technique (which was also used in [14] for FPGAs). Here, the syndrome is rotated, d times, by $supp(h_i)$. The rotated syndrome is then accumulated in the `upc` array, using a bit-slice technique that implements a Carry Save Adder (CSA).

[1] The paper [13] does not point to publicly available code.

6.1 Optimizing the Rotation of an Array

Consider the rotation of the syndrome s (of r bits) by e. g., $1,100$ positions. It starts with "Barrel shifting" by the word size of the underlying architecture (e. g., for AVX512 the words size is 512-bits), here twice ($1,024$ positions). It then continues with internal shifting here by 76 positions. Reference [13] shows a code snippet (for the core functionality) for rotating by a number of positions that is less than the word size. Figure 2 presents our optimized and simplified snippet for the same functionality using the _mm512_permutex2var_epi64 instruction instead of the BLENDV and the VPALIGND.

```
__m512i previous, current, a0, a1, idx, idx1, num_full_qw, one;
uint64_t count64 = bitscount & 0x3f;

num_full_qw = _mm512_set1_epi8(bitscount >> 6);
one         = _mm512_set1_epi64(1);
previous    = _mm512_setzero_si512();
idx         = _mm512_setr_epi64(0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7);
idx         = _mm512_add_epi64(idx, num_full_qw);
idx1        = _mm512_add_epi64(idx, one);

for(int i = R_ZMM; i >= 0; i--)
{
  current = _mm512_loadu_si512(in[i]);
  a0 = _mm512_permutex2var_epi64(current, idx, previous);
  a1 = _mm512_permutex2var_epi64(current, idx1, previous);
  a0 = _mm512_srli_epi64(a0, count64);
  a1 = _mm512_slli_epi64(a1, 64 - count64);
  _mm512_storeu_si512(out[i], _mm512_or_si512(a0, a1));
  previous = current;
}
```

Fig. 2. Right rotate of 512-bit R_ZMM registers using AVX512 instructions.

The latest Intel micro-architecture "Ice-Lake" introduces a new instruction VPSHRDVQ as part of the new AVX512-VBMI2 set. This instruction receives two 512-bit (ZMM) registers (a, b) together with another 512-bit index register (c) and outputs in *dst* the following results:

```
For j = 0 to 7
     i = j*64
     dst[i+63:i] := concat(b[i+63:i], a[i+63:i]) >> (c[i+63:i] & 63)
```

Figure 3 shows how VPSHRDVQ can be used in order to replace the three instructions in lines 16–18 of Fig. 2.

Remark 1. Reference [13] remarks on using tables for some syndrome rotations but mentions that it does not yield significant speedup (and in some cases even shows a performance penalty). This is due to two bottlenecks in a constant-time implementation: (a) extensive memory access; (b) pressure on the execution port

that the shift operations are using. In our case, the bottleneck is (a) so using tables to reduce the number of shifts is not a remedy. For completeness, we describe a new table method that can be implemented using Ice-Lake CPUs. The new `VPERMI2B` (`_mm512_permutex2var_epi8`) instruction [1] allows to permute two `ZMMs` at a granularity of bytes, and therefore to perform the rotation in lines 16–18 of Fig. 2 at a granularity of 8 bits (instead of 64). To use tables for caching: (a) initialize a table with $i = 0, \ldots, 7$ right-shifts of the syndrome (only 8 rows); (b) modify lines 14–15 to use `VPERMI2B`; (c) load (in constant-time) the relevant row before calling the Barrel-shifter. As a result, lines 16–18 can be removed to avoid all the shift operations. As explained above, this technique does not improve the performance of the rotation.

```
__m512i count64 = _mm512_set1_epi64(bitscount & 0x3f);

for(int i = R_ZMM; i >= 0; i--)
{
  data = _mm512_loadu_si512(&in->qw[8 * i]);
  a0 = _mm512_permutex2var_epi64(current, idx, previous);
  a1 = _mm512_permutex2var_epi64(current, idx1, previous);
  a0 = _mm512_shrdv_epi64(a0, a1, count64);
  _mm512_storeu_si512(&out->qw[8 * i], a0);
  previous = current;
}
```

Fig. 3. Right rotate of 512-bit `R_ZMM` registers using `AVX512-VBMI2` instructions. The initialization in Fig. 2 (lines 1–10) is omitted.

6.2 Using Vector-`PCLMULQDQ` and vector-`AESENC`

The Ice-Lake processors support the new vectorized `PCLMULQDQ` and `AESENC` instructions [1]. We used the multiplication code presented in [9][Figure 2], and the CTR DRBG code of [6,10], in order to improve our BIKE implementation. We also used larger caching of random values (1,024 bytes instead of 16) to fully leverage the DRBG. The results are given in Sect. 7.

7 Performance Studies

We start with describing our experimentation platforms and measurements methodology. The experiments were carried out on two platforms, (Intel® Turbo Boost Technology was turned off on both):

- **EC2 Server:** An AWS EC2 `m5.metal` instance with the 6^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Sky Lake" [SKL]) Xeon®Platinum 8175M CPU 2.50 GHz. This platform has 384 GB RAM, 32K L1d and L1i cache, 1MiB L2 cache, and 32MiB L3 cache.
- **Ice-Lake:** Dell XPS 13 7390 2-in-1 with the 10^{th} Intel®Core^{TM} Generation (Micro architecture Codename "Ice Lake" [ICL]) Intel®Core^{TM} i7-1065G7 CPU 1.30 GHz. This platform has 16 GB RAM, 48K L1d and 32K L1i cache, 512K L2 cache, and 8MiB L3 cache.

The Code. The code is written in C and x86-64 assembly. The implementations use the (vector) `PCLMULQDQ`, `AES-NI`, `AVX2`, `AVX512` and `AVX512-VBMI2` instructions when available. The code was compiled with gcc (version 8.3.0) in 64-bit mode, using the "O3" Optimization level with the "-funroll-all-loops" flag, and run on a Linux (Ubuntu 18.04.2 LTS) OS.

Measurements Methodology. The performance measurements reported hereafter are measured in processor cycles (per single core), where lower count is better. All the results were obtained using the same measurement methodology, as follows. Each 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.

7.1 Decoding and Decapsulation: Performance Studies

Performance of BG. Table 3 shows the performance of our implementation which uses the rotation and bit-slice-adder techniques of [3,13], and compares the results to the additional implementation of BIKE [7]. The results show a speedup of 3.75×−6.03× for the portable (C code) of the decoder, 1.1× speedup for the `AVX512` implementations but a 0.66× slowdown for the `AVX2` implementation. The `AVX512` implementation leverages the masked store and load operations that do not exist in the `AVX2` architecture. Note that key generation is faster because generation of mock-bits is no longer needed.

Table 4 compares our implementations with different instruction sets (`AVX512F`, `AVX512-VBMI2`, vector-`PCLMULQDQ`, and vector-`AES`). The results for BIKE-1 Level-1 show speedups of 1.47×, 1.28×, and 1.26× for key generation, encapsulation, and decapsulation, respectively. Even better speedups are shown for BIKE-1 Level-3 of 1.58×, 1.39×, and 1.24×, respectively.

Consider the 6th column and the BIKE-1 Level-1 results. The ∼ 94K (93, 521) cycles of the key generation consists of 13K, 13K, 1K, 1K, 5.5K, 26K, 26K cycles for generating $h_0, h_1, \sigma_0, \sigma_1, g, f_0, f_1$, respectively (and some additional overheads). Compared to the 3rd column of this table (with only `AVX512F` implementation): 13.6K, 13.6K, 2K, 2K, 6K, 46K, 46K, respectively. Indeed, as reported in [9], the use of vector-`PCLMULQDQ` contributes a 2× speedup to the polynomial multiplication. Note that the vector-AES does not contribute much, because the bottleneck in generating h_0, h_1 is the constant-time rejection sampling check (if a bit is set) and not the AES calculations.

Table 5 compares our right-rotation method to the snippet shown in [13]. To accurately measure these "short" functionalities, we ported them into separate compilation units and compiled them separately using the "-c" flag. In addition, the number of repetitions was increased to 10,000. This small change improves the rotation significantly (by 2.3×) and contributes ∼ 2% to the overall decoding performance.

8 Discussion

Our study shows an unexpected shades-of-gray combination decoders: BGF offers the most favorable DFR-efficiency trade off. Indeed (see Table 2), it is possible to trade BG, which was our leading option so far, for another decoder and have the same or even better DFR for the same block size. The advantage

Table 3. The EC2 server performance of BIKE-1 Level-1 when using the BG decoder with 3 iterations. The cycles (in columns 4, 5) are counted in millions.

Implementation	Level	Op	Additional Implementation [7]	This paper	Speedup
C-portable stand-alone	Level-1	Keygen	1.67	1.37	1.22
		Decaps	60	15.99	3.75
	Level-3	Keygen	4.75	4.03	1.18
		Decaps	242.72	64.09	3.79
C-portable + OpenSSL	Level-1	Keygen	0.86	0.56	1.54
		Decaps	52.38	8.68	6.03
	Level-3	Keygen	2.71	1.98	1.37
		Decaps	218.42	39.82	5.48
AVX2	Level-1	Keygen	0.27	0.15	1.81
		Decaps	3.03	3.62	0.84
	Level-3	Keygen	0.62	0.38	1.64
		Decaps	10.46	15.84	0.66
AVX512	Level-1	Keygen	0.26	0.15	1.79
		Decaps	2.59	1.83	1.42
	Level-3	Keygen	0.57	0.37	1.57
		Decaps	8.97	8.14	1.10

Table 4. BIKE-1 Level-1 using the BG decoder with 3 iterations. Performance on Ice-Lake using various instruction sets.

Level	Op	`AVX512F`	`AVX512F` `AVX512-VBMI2` `VPCLMULQDQ`	Speedup	`AVX512F` `AVX512-VBMI2` `VPCLMULQDQ`, `VAES`	Speedup
Level-1	Keygen	137,095	95,068	1.44	93,521	1.47
	Encaps	192,123	150,860	1.27	150,612	1.28
	Decaps	2,192,433	1,711,127	1.28	1,737,912	1.26
Level-3	Keygen	375,604	240,350	1.56	238,198	1.58
	Encaps	432,577	310,908	1.39	310,533	1.39
	Decaps	9,019,103	7,201,222	1.25	7,277,357	1.24

Table 5. Rotation performance, comparison of our impl. and the snippet of [13].

Level	$\|R\|$	Platform	Snippet of [13]	Fig. 2	Fig. 3	AVX512 Speedup	AVX512-VBMI Speedup
L1	11,779	EC2 server	128	105	–	1.21	–
L1	11,779	Ice-Lake	149	120	63.97	1.24	2.33
L3	24,821	EC2 server	250	205	–	1.22	–
L3	24,821	Ice-Lake	296	236	121.72	1.25	2.43
L5	40,597	EC2 server	404	329	–	1.23	–
L5	40,597	Ice-Lake	475	382	194.46	1.24	2.44

is either in performance (e. g., BGF with 6 iterations is $\frac{12}{8} = 1.5\times$ faster than BG with 4 iterations) or in implementation simplicity (e. g., the B decoder that does not involve gray steps).

A Comment on the Backflip Decoder. In [8] we compared Backflip with BG and showed that it requires a few more steps to achieve the same DFR (in the relevant range of r). We note that a Backflip iteration is practically equivalent to Step I of BG plus the Time-To-Live (TTL) handling. It is possible to improve the constant-time TTL handling with the bit-slicing techniques and reduce this gap. However, this would not change the DFR-efficiency properties reported here.

Further Optimizations. The performance of BIKE's constant-time implementation is dominated by three primitives: (a) polynomial multiplication (it remains a significant portion of the computations even after using the vector-`PCLMULQDQ` instructions); (b) polynomial rotation (that requires extensive memory access); (c) the rejection sampling (approximately 25% of the key generation). This paper showed how some of the new Ice-Lake features can already be used for performance improvement. Further optimizations are an interesting challenge.

Parameter Choice Recommendations for BIKE. BIKE-1 Level-1 (IND-CCA) [2] uses $r = 11,779$ with a target DFR of 2^{-128}, and uses the Backflip decoder. Our paper [8] shows some problems with this decoder and therefore recommends to use BG instead. It also shows that even if DFR $= 2^{-128}$ there is still a gap to be addressed, in order to claim IND-CCA security (roughly speaking - a bound on the number of weak keys). We set aside this gap for now and consider a non-weak key. If we limit the number of usages of this key to Q and choose r such that $Q \cdot DFR < 2^{-\mu}$ (for some target margin μ), then the probability that an adversary with at most Q queries sees a decoding failure is at most $2^{-\mu}$. We suggest that KEMs should use ephemeral keys (i. e., $Q = 1$) for forward secrecy, and this usage does not mandate IND-CCA security (IND-CPA suffices). Here, from the practical view-point, we only need to target a sufficiently small DFR such that decapsulation failures would be a significant operability impediment. However, an important property that *is* desired, even with ephemeral keys, is some guarantee that an inadvertent $1 \leq \alpha$ times key reuse (where α is presumably not too large) would not crash the security. This

suggests the option for selecting r so that $\alpha \cdot DFR < 2^{-\mu}$. For example, taking $\mu = 32$ and $\alpha = 2^{32}$ (an extremely large number of "inadvertent" reuses), we can target a DFR of 2^{-64}. Using BGF with 5 iterations, we can use $r = 11,171$, which is smaller than $11,779$ that is currently used for BIKE.

Acknowledgments. We thank Ray Perlner from NIST for pointing out that the mock-bits technique is not sufficient for security when using static keys, which drove us to change our BIKE implementation. This research was partly supported by: The Israel Science Foundation (grant No. 3380/19); 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.

A Pseudo-Code for B, BG, BGB, BGF

A description of the B, BG, BGB, BGF decoders is given in Sect. 4. Algorithm 3 provides a formal definition of them.

Algorithm 3. e=decoder(D, c, H)

Input: D (decoder type one of {B, BG, BGB, BGF}), $H \in \mathbb{F}_2^{r \times n}$ (parity-check matrix), $c \in \mathbb{F}_2^n$ (ciphertext), X (maximal number of iterations)
Output: $e \in \mathbb{F}_2^n$ (errors vector)
Exception: A "decoding failure" returns $\perp$

procedure DECODER(D, c, H)
  $s = Hc^T$, $e[n-1:0] = 0^n$, $\delta = 3$
  for itr in $1 \ldots X$ **do**
    th = computeThreshold(s)
    $(s, e, black, gray)$ = BitFlipIter(s, e, th, H) ▷ Step I
    if $(D \in \{B, BG, BGB\})$ or $(D = BGF$ and $it = 1)$ **then**
      (s, e) = BitFlipMaskedIter$(s, e, black, ((d+1)/2), H)$ ▷ Step II
    if $(D \in \{BG, BGB, BGF\}$ and $itr = 1)$ **then**
      (s, e) = BitFlipMaskedIter$(s, e, gray, ((d+1)/2), H)$ ▷ Step III
  if $(wt(s) \neq 0)$ **then**
    return $\perp$
  else
    return e

B Additional Information on the Experiments and Results

The following values of r were used by the *best linear fit* extrapolation method:

– BIKE-1 Level-1: 9349, 9547, 9749, 9803, 9859, 9883, 9901, 9907, 9923, 9941, 9949, 10037, 10067, 10069, 10091, 10093, 10099, 10133, 10139.

Table 6. The *best linear* and the *two points* extrapolation equations, and the estimated r values for three target DFRs. Level is abbreviated to Lvl, the number of iterations is abbreviated to iter, linear is abbreviated to lin., equation is abbreviated to eq. The Lin. start column indicates the index of the first value of r where the linear fit starts. The 5 column (number of steps) is the indication for the overall performance of the decoder (lower is better).

KEM	Lvl	Decoder	Iter	Steps	Lin. start	Best lin. fit eq. s.t $\log_{10}\texttt{DFR} = ar + b =$	2^{-23}	2^{-64}	2^{-128}	Two points line eq. (a,b) $\log_{10}\texttt{DFR} = ar + b =$	2^{-23}	2^{-64}	2^{-128}
BIKE-1	1	BG	3	9	15	$(-1.27\text{e}{-2}, 124)$	10,253	11,213	12,739	$(-1.33\text{e}{-2}, 129)$	10,253	11,171	12,619
BIKE-1	1	BG	4	12	8	$(-1.45\text{e}{-2}, 140)$	10,163	11,003	12,347	$(-1.63\text{e}{-2}, 158)$	10,163	10,909	12,107
BIKE-1	1	BG	5	15	13	$(-1.61\text{e}{-2}, 156)$	10,133	10,909	12,107	$(-1.70\text{e}{-2}, 165)$	10,133	10,853	11,987
BIKE-1	1	BGB	4	9	13	$(-1.38\text{e}{-2}, 134)$	10,253	11,093	12,491	$(-1.40\text{e}{-2}, 136)$	10,253	11,083	12,491
BIKE-1	1	BGB	5	11	13	$(-1.52\text{e}{-2}, 147)$	10,163	10,973	12,227	$(-1.41\text{e}{-2}, 136)$	10,163	11,027	12,413
BIKE-1	1	BGB	6	13	7	$(-1.48\text{e}{-2}, 143)$	10,133	10,973	12,269	$(-1.54\text{e}{-2}, 149)$	10,133	10,949	12,197
BIKE-1	1	BGF	5	7	14	$(-1.40\text{e}{-2}, 137)$	10,301	11,171	12,539	$(-1.44\text{e}{-2}, 141)$	10,301	11,131	12,491
BIKE-1	1	BGF	6	8	13	$(-1.53\text{e}{-2}, 149)$	10,253	11,027	12,277	$(-1.61\text{e}{-2}, 157)$	10,253	10,973	12,197
BIKE-1	1	BGF	7	9	13	$(-1.61\text{e}{-2}, 157)$	10,181	10,949	12,149	$(-1.68\text{e}{-2}, 164)$	10,181	10,949	12,107
BIKE-1	1	B	4	8	15	$(-8.69\text{e}{-3}, 82.4)$	10,259	11,699	13,901	$(-8.05\text{e}{-3}, 75.8)$	10,301	11,813	14,221
BIKE-1	1	B	5	10	15	$(-1.02\text{e}{-2}, 96.3)$	10,133	11,437	13,229	$(-9.56\text{e}{-3}, 89.9)$	10,133	11,437	13,451
BIKE-1	1	B	6	12	14	$(-1.08\text{e}{-2}, 101)$	10,067	11,213	13,037	$(-9.52\text{e}{-3}, 88.8)$	10,067	11,437	13,397

For Level-1 studies the number of tests for every value of r is $3.84M$ for $r \in [9349, 9901]$ and $384M$ for (larger) $r \in [9907, 10139]$. For the *line through two large points* extrapolation method (see [8][Appendix C] and Level-1, we chose: $r = 10141$ running 384M tests, and $r = 10259$ running ~ 7.3 (technically 7.296) billion tests (Table 6).

References

1. Intel®64 and IA-32 architectures software developer's manual. Combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, November 2019. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
2. Aragon, N., et al.: BIKE: Bit Flipping Key Encapsulation (2017). https://bikesuite.org/files/round2/spec/BIKE-Spec-2019.06.30.1.pdf
3. Chou, T.: QcBits: constant-time small-key code-based cryptography. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2016, pp. 280–300. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53140-2_14
4. Drucker, N., Gueron, S.: Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), June 2018
5. Drucker, N., Gueron, S.: A toolbox for software optimization of QC-MDPC code-based cryptosystems. J. Cryptographic Eng. **9**, 1–17 (2019). https://doi.org/10.1007/s13389-018-00200-4
6. Drucker, N., Gueron, S.: Fast CTR DRBG for x86 platforms, March 2019. https://github.com/aws-samples/ctr-drbg-with-vector-aes-ni
7. Drucker, N., Gueron, S., Dusan, K.: Additional implementation of BIKE (2019). https://bikesuite.org/additional.html
8. Drucker, N., Gueron, S., Kostic, D.: On constant-time QC-MDPC decoding with negligible failure rate. Technical report 2019/1289, November 2019. https://eprint.iacr.org/2019/1289
9. 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, June 2018. https://doi.org/10.1109/ARITH.2018.8464777
10. Drucker, N., Gueron, S., Krasnov, V.: Making AES great again: the forthcoming vectorized AES instruction. In: Latifi, S. (ed.) 16th International Conference on Information Technology-New Generations. (ITNG 2019), pp. 37–41. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14070-0_6
11. Eaton, E., Lequesne, M., Parent, A., Sendrier, N.: QC-MDPC: a timing attack and a CCA2 KEM. In: Lange, T., Steinwandt, R. (eds.) PQCrypto 2018. LNCS, vol. 10786, pp. 47–76. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_3
12. Gallager, R.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962). https://doi.org/10.1109/TIT.1962.1057683
13. Guimarães, A., Aranha, D.F., Borin, E.: Optimized implementation of QC-MDPC code-based cryptography **31**(18), e5089 (2019). https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5089
14. Maurich, I.V., Oder, T., Güneysu, T.: Implementing QC-MDPC McEliece encryption. ACM Trans. Embed. Comput. Syst. **14**(3), 441–4427 (2015). https://doi.org/10.1145/2700102

15. NIST: Post-Quantum Cryptography (2019). https://csrc.nist.gov/projects/post-quantum-cryptography. Accessed 20 Aug 2019
16. Sendrier, N., Vasseur, V.: On the decoding failure rate of QC-MDPC bit-flipping decoders. In: Ding, J., Steinwandt, R. (eds.) PQCrypto 2019. LNCS, vol. 11505, pp. 404–416. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25510-7_22