Optimizing BIKE for the Intel Haswell and ARM Cortex-M4

Ming-Shing Chen[1], Tung Chou[2] and Markus Krausz[1]

[1] Ruhr University Bochum, Bochum, Germany
mschen@crypto.tw, markus.krausz@rub.de
[2] Academia Sinica, Taipei, Taiwan
blueprint@crypto.tw

Abstract. BIKE is a key encapsulation mechanism that entered the third round of the NIST post-quantum cryptography standardization process. This paper presents two constant-time implementations for BIKE, one tailored for the Intel Haswell and one tailored for the ARM Cortex-M4. Our Haswell implementation is much faster than the `avx2` implementation written by the BIKE team: for `bikel1`, the level-1 parameter set, we achieve a 1.39x speedup for decapsulation (which is the slowest operation) and a 1.33x speedup for the sum of all operations. For `bikel3`, the level-3 parameter set, we achieve a 1.5x speedup for decapsulation and a 1.46x speedup for the sum of all operations. Our M4 implementation is more than two times faster than the *non-constant-time* implementation `portable` written by the BIKE team. The speedups are achieved by both algorithm-level and instruction-level optimizations.

Keywords: constant-time implementations · NIST PQC standardization · Cortex-M4

1 Introduction

BIKE [4] is a key encapsulation mechanism (KEM) involved in the NIST post-quantum cryptography standardization process. It can be viewed as a variant of the code-based cryptosystem introduced in [24], which makes use of so-called "quasi-cyclic moderate-density parity-check" (QC-MDPC) codes. In July 2020, NIST announced that BIKE has been selected as one of 15 schemes that advanced to the third round of the process, making BIKE one of the candidates considered for standardization.

During the standardization process, BIKE holds a nice security record: unlike many other candidates, the claimed security levels of BIKE haven never been challenged. BIKE also features small public keys: the public key sizes are around 1.5KB, 3KB, and 5KB for the level-1, level-3, and level-5 parameter sets, respectively. These features make BIKE a competitive candidate in the process. In fact, NIST also commented that

> "NIST views BIKE as one of the most promising code-based candidates."

in the status report for the second round candidates [2].

Despite the advantages, there are two factors that arguably make BIKE look weaker than some other candidates. The first factor is that, although the BIKE team has identified the conditions on the decoder that would make the scheme CCA-secure, they are not able to prove that the decoder actually satisfies the conditions. On the other hand, BIKE can still be used in applications which require only CPA security. The other factor is the relatively slow speeds of the operations. The speed issue is what this paper aims to deal with.

Permanent ID of this document: `7ea7ddcc350bc2666b5fb0221161a420`. Date: 2021.04.18.

1.1 Previous Works

There have been many previous papers on improving the speed of the cryptosystem introduced in [24] and its variants. In particular, the QcBits paper [12] was the first one to present a fully constant-time software implementation. Later on, many papers tried to improve the speed of constant-time software implementations. For the purpose of this paper, instead of reviewing all related papers, we list some important optimization techniques adopted by the BIKE team. All the optimization techniques are related to $\mathcal{R}_z = \mathbb{Z}[y]/(y^r - 1)$ or $\mathcal{R} = \mathbb{F}_2[x]/(x^r - 1)$.

- As pointed out in [12], there are many operations that can be viewed as multiplications in $\mathcal{R}_z$ in the decoding algorithm. These multiplications are of the form $g(y)f(y)$, where each coefficient g_i and f_i is either 0 or 1, and $g(y)$ is a sparse polynomial. [12] proposed to consider $g(y)$ as $\sum_s y^{-s}$ and compute the sum of all $y^{-s}f(y)$ to obtain $g(y)f(y)$. Each $y^{-s}f(y)$ is computed using a circular shift on the coefficients of f by s positions. In order to make sure that the circular shift does not leak information through timing, [12] proposed to make use of the concept of a Barrel shifter to build a “Barrel rotator”.

- This idea of using a “Barrel rotator” would be very efficient if r was a multiple of the register size. However, as r is always an odd prime in BIKE, performing the circular shift directly will introduce some overhead. In order to reduce the overhead, [14] proposed to represent f in a “duplicated form” such that $y^{-s}f(y)$ can be obtained by performing a logical shift on the duplicated form. The logical shift is again implemented using the concept of a Barrel shifter.

- To compute the sum $\sum_s(y^{-s}f(y))$, [12] proposed to perform the computation in a bitsliced fashion to make use of the fact that each coefficient of $y^{-s}f(y)$ is either 0 or 1.

- There are many multiplications in $\mathcal{R}$ in key generation, encapsulation, and decapsulation. In order to compute the product of two elements $f = \sum_i f_i x^i$ and $g = \sum_j g_j x^j$ in $\mathcal{R}$, the `clmul` implementation of [12] represents f and g as $\sum_i F_i z^i$ and $\sum_j G_j z^j$, where $z = x^{64}$, and uses the `pclmulqdq` instruction to compute every $F_i G_j$. [14] proposed to use Karatsuba recursively, where the bottom level of recursions is handled by `pclmulqdq`.

- The public key is of the form $h_1 \cdot h_0^{-1}$ where each $h_i \in \mathcal{R}$. Computation of $h_0^{-1} \in \mathcal{R}$ can be computed using a multiplication chain as shown in [12]. Inside the multiplication chain, it is often required to compute the 2^kth power of an element $f \in \mathcal{R}$ for some constant k. [15] proposed to compute such an exponentiation by permuting the coefficients in f. When k is large, this is much faster than carrying out k squarings sequentially.

Unfortunately, even with so many optimizations, the latest software speed of BIKE is still not very satisfying: the eBACs website [8] reports that, on `titan0`, a machine with an Intel Haswell CPU, the level-1 parameter set `bikel1` takes more than 138000 cycles and 2650000 cycles to perform encapsulation and decapsulation, respectively. For comparison, `mceliece348864`, one of the level-1 parameter sets of a third-round code-based candidate Classic McEliece [3], takes only 44652 and 132360 cycles, respectively, on the same machine. Note that this means that the situation of code-based candidates is quite different from the situation of lattice-based candidates, as structured lattice-based schemes are typically much faster than non-structured ones.

The BIKE team has some optimized implementations for x86 machines, whose speeds rely heavily on the usage of instructions for carryless multiplications such as `pclmulqdq`.

This raises the question whether BIKE can still run with reasonable speed on embedded systems where such instructions are not available. To our knowledge, there has not been any BIKE implementation tailored for embedded systems yet.

1.2 Our Contribution

This paper presents two implementations of BIKE, one optimized for the Intel Haswell and one optimized for the Cortex-M4. Compared to the `avx2` implementation written by the BIKE team, for `bikel1`, we achieve a 1.39x speedup for decapsulation and a 1.33x speedup for the sum of all operations on Haswell. For `bikel3`, we achieve a 1.5x speedup for decapsulation and a 1.46x speedup for the sum of all operations on Haswell. Compared to the non-constant-time `portable` implementation written by the BIKE team, our implementation is more than two times faster for all the three operations on M4. Both of our implementations are constant-time: there is no memory access using secret indices, no branching with secret conditions, and no usage of variable-time instructions.

We believe that our implementations will make BIKE a more interesting option for standardization. Even though we did not implement the largest parameter set `bikel5`, which was introduced in the 3rd-round specification, we expect that our optimization techniques (see below) will be useful for optimizing `bikel5` as well. We also expect the implementation techniques used for Haswell will be useful for optimizing BIKE for newer microarchitectures, e.g., those supporting AVX512. In addition, our optimization techniques for polynomial multiplications in $\mathcal{R}$ can be useful for optimizing HQC [1], another third-round code-based candidate.

1.3 Optimization Techniques

The speed of our implementations is achieved by using the following optimization techniques.

a) When performing a logical shift on the duplicated form of $f \in \mathcal{R}_z$, we make use of matrix transpositions to change the representation of f, so that a Barrel shifter is no longer required. As far as we can tell, this idea is completely original. This optimization is only used in our Haswell implementation.

b) We implement the Barrel shifter for shifting the duplicated form of $f \in \mathcal{R}_z$ using conditional execution and powerful multiply-and-accumulate instructions supported by the M4. We consider this as a purely instruction-level optimization.

c) The sum of all $y^{-s}f$ is computed using a sequence of half adders in the BIKE team's implementation. We follow an algorithm explained by Boyar and Peralta [9] to use full adders whenever possible. The algorithm is more complex and requires more

Table 1: Optimization techniques and where they are used: • means that the technique is used for the corresponding operation on the corresponding platform, and ◦ means that the technique is not used.

	Haswell implementation			M4 implementation			
technique	keygen	encap.	decap.	keygen	encap.	decap.	section
a)	◦	◦	•	◦	◦	◦	3
b)	◦	◦	◦	◦	◦	•	4
c)	◦	◦	•	◦	◦	•	5
d)	•	•	•	◦	◦	◦	6
e)	◦	◦	◦	•	•	•	7

memory, but it reduces the number of logical operations required to compute the sum by a factor of more than 2.

d) The BIKE team's implementations use the Karatsuba algorithm to optimize multiplications in $\mathcal{R}$. The Karatsuba algorithm is used recursively. Instead, we use a five-way recursive algorithm proposed by Bernstein in [5] for the top level of recursion. This optimization is only applied in our Haswell implementation.

e) We use a "Frobenius additive FFT" (FAFFT) to optimize multiplications in $\mathcal{R}$ on the M4. We found that FAFFTs fit nicely with bitslicing, such that we are able to carry out multiplications in $\mathcal{R}$ efficiently even without instructions for carryless multiplications.

To summarize, a), b), and c) are used for multiplications in $\mathcal{R}_z$, while d) and e) are used for multiplications in $\mathcal{R}$. Table 1 summarizes where each of these techniques is used in our implementations.

1.4 Availability of Source Code

Our implementations are adapted from the BIKE team's implementation and hence will be distributed under the same Apache 2.0 license of the orignal implementations. We plan to submit our Haswell implementation to the eBACS project [8] so that the source code can be included in SUPERCOP. The source code of our M4 implementation has been included in the `pqm4` project [20] under the directories `crypto_kem/bikel1/m4f` and `crypto_kem/bikel3/m4f`. We also plan to submit both of our implementations as the artifact of our paper for CHES 2021.

1.5 Organization of the paper

Section 2 reviews the latest specification of BIKE and shows why we need to perform multiplications in $\mathcal{R}_z$ and multiplications in $\mathcal{R}$. Each of the remaining sections introduces one of our five optimization techniques; See Table 1. Section 8 shows our experimental results for multiplications in $\mathcal{R}_z$, multiplications in $\mathcal{R}$, and the three KEM operations.

2 BIKE

This section reviews the third-round specification of BIKE [4]. For the sake of completeness, we repeat some contents from the specification in this section.

2.1 System Parameters.

BIKE uses system parameters r, w, t, and ℓ. The following table shows the values of the system parameters and the security level for each of the three parameter sets of BIKE, which we denote as `bikel1`, `bikel3`, and `bikel5`.

	r	w	t	ℓ	level
`bikel1`	12323	142	134	256	1
`bikel3`	24659	206	199	256	3
`bikel5`	40973	274	264	256	5

KeyGen: $() \mapsto (h_0, h_1, \sigma), h$ Output: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M}, h \in \mathcal{R}$ 1: $h_0, h_1 \xleftarrow{\$} \mathcal{H}_w$ 2: $h = h_1 \cdot h_0^{-1}$ 3: $\sigma \xleftarrow{\$} \mathcal{M}$	**Encaps**: $h \mapsto K, c$ Input: $h \in \mathcal{R}$ Output: $K \in \mathcal{K}, c \in \mathcal{R} \times \mathcal{M}$ 1: $m \xleftarrow{\$} \mathcal{M}$ 2: $e_0, e_1 \leftarrow \mathbf{H}(m)$ 3: $c = (e_0 + e_1 \cdot h, m \oplus \mathbf{L}(e_0, e_1))$ 4: $K \leftarrow \mathbf{K}(m, c)$
Decaps: $(h_0, h_1, \sigma), c \mapsto K$ Input: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}, c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$ Output: $K \in \mathcal{K}$ 1: $e' \leftarrow \text{decoder}(c_0 h_0, h_0, h_1)$ ▷ $e' \in \mathcal{R}^2 \cup \{\perp\}$ 2: $m' = c_1 \oplus \mathbf{L}(e')$ ▷ with the convention $\perp = (0, 0)$ 3: **if** $e' = \mathbf{H}(m')$ **then** $K \leftarrow \mathbf{K}(m', c)$, **else** $K \leftarrow \mathbf{K}(\sigma, c)$	

Figure 1: BIKE's key generation, encapsulation, and decapsulation.

2.2 Hash Functions.

The BIKE team defines the hash functions **H**, **K**, **L** to have the following domains and ranges.

- **H**: $\mathcal{M} \to \mathcal{E}_t$,
- **K**: $\mathcal{M} \times \mathcal{R} \times \mathcal{M} \to \mathcal{K}$,
- **L**: $\mathcal{R}^2 \to \mathcal{M}$,

where the spaces $\mathcal{M}$, $\mathcal{K}$, and $\mathcal{E}_t$ are defined as follows.

- $\mathcal{M} = \{0, 1\}^\ell$.
- $\mathcal{K} = \{0, 1\}^\ell$.
- $\mathcal{E}_t = \{(e_0, e_1) \in \mathcal{R}^2 \mid |e_0| + |e_1| = t\}$, where $|e_i|$ means the number of nonzero coefficients in e_i.

The concrete instantiation of **H**, **K**, **L** can be found in the specification and is beyond the scope of this paper.

2.3 Key Generation, Encapsulation, and Decapsulation.

BIKE's key generation, encapsulation, and decapsulation algorithms are depicted in Figure 1. As shown in the figure, the key generation algorithm generates a secret key (h_0, h_1, σ) and a public key h. (h_0, h_1) is in $\mathcal{H}_w$, which is defined as

$$\{(h_0, h_1) \in \mathcal{R}^2 \mid |h_0| = |h_1| = w/2\}.$$

The encapsulation algorithm, on input a public key h, generates a session key K and a ciphertext c encapsulating the session key. The decapsulation algorithm, on input a secret key (h_0, h_1, σ) and a ciphertext c, outputs a session key K or $\perp$. There are many multiplications in $\mathcal{R}$ as one can see from the KEM operations and the decoding algorithm `decoder` defined in the next subsection,

2.4 The Black-Gray Decoder

The decoding algorithm specified in the third-round specification is the black-gray decoder, which was introduced in [16]. The algorithm is shown in Algorithm 1 of the specification.

The BIKE team described it as an algorithm that can be used for any code with a low-weight parity-check matrix. However, as shown in Figure 1, BIKE is a scheme with a ring structure. To facilitate our discussion in the following sections, we rephrased the algorithm so that operations in rings are explicitly shown. We emphasize that "rephrased" means that we do not create a new algorithm but merely write the original algorithm in a different way. Our version of the black-gray decoder uses the following notations.

- $\mathrm{Lift}(\cdot)$, which lifts an element from $\mathcal{R}$ to $\mathcal{R}_z$.
- $\mathrm{Tr}(\cdot)$, which converts an element $\sum_{i=0}^{r-1} f_i x^i \in \mathcal{R}$, into $\mathrm{Lift}(f_0 + \sum_{i=1}^{r-1} f_{r-i} x^i)$.

The black-gray decoder is summarized in Algorithm 1. It should be easy to see where the operations in $\mathcal{R}$ are from with respect to the specification. Note that the multiplications of the form $\mathrm{Tr}(h_i) \cdot \mathrm{Lift}(s)$ are in $\mathcal{R}_z$, and $|\mathrm{Tr}(h_i)|$ is always $w/2$. These multiplications are used to compute the numbers of unsatified parity-check equations. It has been explained in [12] why computation of the numbers of unsatified parity-check equations can be viewed as multiplications in $\mathcal{R}_z$.

For completeness, we note that the black-gray decoder uses (in addition to r, w, t) two integers $\texttt{NbIter}, \tau$ and a function `threshold` as its parameters. The third-round specification defines $\texttt{NbIter} = 5$ for all parameter sets. How τ and `threshold` are defined is irrelevant to the purpose of this paper.

3 Circular Shifts with Matrix Transposition

As mentioned in the Introduction, in order to compute $g \cdot f$ where $g, f \in \mathcal{R}_z$, $g_i, f_i \in \{0, 1\}$ for all i, we may write g as $\sum_s y^{-s}$ with $0 \le s < r$ and compute gf as $\sum_s (y^{-s} f)$. For each s, $y^{-s} f$ can be computed by carrying out a circular shift on $(f_0, \ldots, f_{r-1})$ by s positions due to the structure of $\mathcal{R}_z$. Below we review how such a circular shift can be implemented using a Barrel shifter and propose a way to avoid the Barrel shifter.

3.1 The Duplicated Form and the Barrel Shifter

In order to compute $y^{-s} f$, the BIKE team follows [12] to use a Barrel shifter and follows [14] to use a duplicated form for f. The idea is to

1. lift f to $\mathbb{Z}[y]$ and represent it as $f' = f + (f \bmod y^{r-1}) y^r$ [1] and
2. compute $y^{-s} f$ as $((f' - (f' \bmod y^s))/y^s) \bmod y^r$.

The second step might look complicated, but computationally it simply means performing a logical shift (toward the direction of f'_0) of s positions on the vector of $2r - 1$ coefficients of f' and taking the first r coefficients. Note that making f' a longer polynomial will lead to the same result, as long as $f'_i = f_{(i \bmod r)}$ for $i \in \{0, \ldots, 2r - 2\}$.

Let b be a power of 2. To implement this idea, f' is stored as an array of b-bit words, where the values of the words are $\sum_{i=0}^{b-1} f'_i 2^i$, $\sum_{i=0}^{b-1} f'_{i+b} 2^i$, and so on. Then, f' is first shifted by $s^{(1)} = s - (s \bmod b)$ positions and then by $s^{(0)} = s \bmod b$ positions. As $s^{(1)}$ is a multiple of b, the shift by $s^{(1)} = \sum_j s_j^{(1)} 2^j$ positions is carried out as a sequence of conditional moves from word $i + 2^j/b$ to word i for each j. The shift by $s^{(0)}$ positions can be carried out by a sequence of logical shifts and ORs on the words, assuming that corresponding instructions exist.

The `avx2` implementation follows the strategy above, where b is set to 256 to fit the size of YMM registers. Note that there is no general shift instruction for YMM registers, so the `avx2` implementation follows [19] to use AVX intrinsics to carry out the shift by $s^{(0)}$ positions.

[1] [14] actually uses $(1 + y^r) f$, which has one more coefficient than $f + (f \bmod y^{r-1}) y^r$.

Algorithm 1 The Black-Gray Decoder (rephrased)

Parameters: $r, w, t, d = w/2$, NbIter, τ, threshold

```
1:  procedure DECODER(s, h_0, h_1)
2:      (e_0, e_1) ← (0, 0) ∈ R^2
3:      for i = 1, ..., NbIter do
4:          T ← threshold(|s + e_0 h_0 + e_1 h_1|, i)
5:          e_0, e_1, b_0, b_1, g_0, g_1 ← BFIter(s + e_0 h_0 + e_1 h_1, e_0, e_1, T, h_0, h_1)
6:          if i = 1 then
7:              e_0, e_1 ← BFMaskedIter(s + e_0 h_0 + e_1 h_1, e_0, e_1, b_0, b_1, (d + 1)/2 + 1, h_0, h_1)
8:              e_0, e_1 ← BFMaskedIter(s + e_0 h_0 + e_1 h_1, e_0, e_1, g_0, g_1, (d + 1)/2 + 1, h_0, h_1)
9:          end if
10:     end for
11:     if s = e_0 h_0 + e_1 h_1 then
12:         return (e_0, e_1)
13:     else
14:         return ⊥
15:     end if
16: end procedure

17: procedure BFITER(s, e, T, h_0, h_1)
18:     (c_0, c_1) ← (Tr(h_0) · Lift(s), Tr(h_1) · Lift(s))
19:     (b_0, b_1) ← (0, 0) ∈ R^2
20:     (g_0, g_1) ← (0, 0) ∈ R^2
21:     for i = 0, 1 do
22:         for j = 0, ..., r − 1 do
23:             if c_ij ≥ T then
24:                 e_ij ← e_ij + 1
25:                 b_ij ← 1
26:             else if c_ij ≥ τ then
27:                 g_ij ← 1
28:             end if
29:         end for
30:     end for
31:     return e_0, e_1, b_0, b_1, g_0, g_1
32: end procedure

33: procedure BFMASKEDITER(s, e, m_0, m_1, T, h_0, h_1)
34:     (c_0, c_1) ← (Tr(h_0) · Lift(s), Tr(h_1) · Lift(s))
35:     for i = 0, 1 do
36:         for j = 0, ..., r − 1 do
37:             if c_ij ≥ T then
38:                 e_ij ← e_ij + m_ij
39:             end if
40:         end for
41:     end for
42:     return e_0, e_1
43: end procedure
```

3.2 Avoiding the Barrel Shifter with Matrix Transposition

We do better than the `avx` implementation by making use of a simple observation. In short, we observed that shifting f' by $s^{(1)}$ positions can be carried out by shifting a set of

```
1  _INLINE_ void
2  rotate256_big(OUT syndrome_t *out, IN __m256i *in, IN size_t ymm_num)
3  {
4    int i;
5
6    for (i = 0; i < 64; i++)
7      STORE(&out->qw[4*i], SRLI_I64(in[i], ymm_num) |
8                           SLLI_I64(in[i+64], 64-ymm_num));
9
10   transpose_64x256_sp_asm(out->qw);
11 }
```

Figure 2: The source code for computing $M^{(1)}$ from $N^{(0)}$ for `bikel1`.

b vectors by $s^{(1)}/b$ positions, if f' is represented in a different format. The observation is explained via the following example.

Let $r = b^2/2$. For a concrete number, one can imagine that $b = 64$. We choose f' to have a length of $2r = b^2$ instead of $2r - 1$. Consider f' as a $b \times b$ matrix $M^{(0)}$, where the entry at row i and column j contains the bit $f'_{b \cdot i+j}$. The matrix is illustrated below.

$$M^{(0)} = \begin{pmatrix} f'_0 & f'_1 & \cdots & f'_{b-1} \\ f'_b & f'_{b+1} & \cdots & f'_{2b-1} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}.$$

Our main observation is that shifting f' by $s^{(1)}$ positions, in the matrix view, simply means updating row i by row $i + s^{(1)}/b$ (and if row $i + s^{(1)}/b$ is undefined, row i is set to 0). If we can somehow make each column of $M^{(0)}$ lie in one b-bit word, this can be carried out easily via b shift operations by $s^{(1)}/b$ positions on the columns.

In order to make use of this observation, given f and s, we perform the following steps to compute $y^{-s}f$.

1. Compute f', which can be viewed as the matrix $M^{(0)}$, from f.
2. Compute $N^{(0)} = (M^{(0)})^T$. How to perform such a matrix transposition efficiently given $b = 64$ has been explained in, say, [13].
3. Perform a shift of $s^{(1)}/b$ positions on each row of $N^{(0)}$ to obtain $N^{(1)}$.
4. Compute $M^{(1)} = (N^{(1)})^T$. Note that we only need $\lceil (r + b - 1)/b \rceil$ rows of $M^{(1)}$ for the next step.
5. Shift the vector of length $r + b - 1$ represented by the first $\lceil (r + b - 1)/b \rceil$ rows of $M^{(1)}$, by $s^{(0)}$ positions. Then, the first r entries will be $y^{-s}f$.

Recall that what we actually want to compute is $\sum_s y^{-s}f$. In fact, the first two steps above do not depend on s, so $N^{(0)}$ is the same for any s. Therefore, one can save operations by computing $N^{(0)}$ with the first two steps, and then for each s perform the last three steps.

3.3 The Haswell Implementation for `bikel1`

For ease of discussion, we define the following operators that can be applied to any matrix.

- $\Leftarrow_j (\cdot)$, meaning replacing column i by column $i + j$ for all i.
- $\Rightarrow_j (\cdot)$, meaning replacing column i by column $i - j$ for all i.

- $\Uparrow_j(\cdot)$, meaning replacing row i by row $i+j$ for all i.
- $\Downarrow_j(\cdot)$, meaning replacing row i by row $i-j$ for all i.

For each operator, the destination row or column is set to 0 if the source is not well-defined. The parentheses will be omitted whenever it is convenient.

Our Haswell implementation for `bikel1` uses $b = 256$ because it mainly works on YMM registers. `bikel1` has $r = 12323$, so f' consists of at least $\lceil(2r-1)/256\rceil = 97$ words and $s^{(1)}/b \le 48$. For ease of implementation, we let f' be of length 32768, so $M^{(0)}$ is a matrix with 128 rows:

$$M^{(0)} = \begin{pmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \end{pmatrix} \in \mathbb{F}_2^{128\times 256},$$

where each M_{ij} is a 64×64 matrix. Following the discussion in Section 3.2, we only need the first $\lceil(12323+255)/256\rceil = 50$ rows of $\Uparrow_{s^{(1)}/b}(M^{(0)})$ to perform the final shift of $s^{(0)}$ positions. Let $s' = s^{(1)}/b$, for ease of implementation, we compute the first 64 rows of $\Uparrow_{s'}(M^{(0)})$, which is simply

$$M^{(1)} = \begin{pmatrix} \Uparrow_{s'} M_{00} & \Uparrow_{s'} M_{01} & \Uparrow_{s'} M_{02} & \Uparrow_{s'} M_{03} \\ + & + & + & + \\ \Downarrow_{64-s'} M_{10} & \Downarrow_{64-s'} M_{11} & \Downarrow_{64-s'} M_{12} & \Downarrow_{64-s'} M_{13} \end{pmatrix} \in \mathbb{F}_2^{64\times 256}.$$

Let $N_{ij} = M_{ij}^T$. Our implementation stores f' in a way that the corresponding matrix is

$$N^{(0)} = \begin{pmatrix} N_{00} & N_{01} & N_{02} & N_{03} \\ N_{10} & N_{11} & N_{12} & N_{13} \end{pmatrix} \in \mathbb{F}_2^{128\times 256}$$

instead of $M^{(0)}$. And then for each s, we compute

$$N^{(1)} = \begin{pmatrix} \Leftarrow_{s'} N_{00} & \Leftarrow_{s'} N_{01} & \Leftarrow_{s'} N_{02} & \Leftarrow_{s'} N_{03} \\ + & + & + & + \\ \Rightarrow_{64-s'} N_{10} & \Rightarrow_{64-s'} N_{11} & \Rightarrow_{64-s'} N_{12} & \Rightarrow_{64-s'} N_{13} \end{pmatrix} \in \mathbb{F}_2^{64\times 256}$$

and transpose each of the 4 64×64 submatrices in $N^{(1)}$ to obtain $M^{(1)}$.

We chose the implementation strategy above, because it can be implemented easily using AVX2 intrinsics/instructions. Our source code for computing $M^{(1)}$ from $N^{(0)}$ is given in Figure 2. The variable `in` is a pointer to $N^{(0)}$. The variable `ymm_num` is set to s'. `SRLI_I64` is defined (by the BIKE team) as `_mm256_srli_epi64`, and we use it to perform $\Leftarrow_{s'}$. `SLLI_I64` is defined (by the BIKE team) as `_mm256_slli_epi64`, and we use it to perform $\Rightarrow_{64-s'}$. The discussion above suggests that we need to compute XOR of the results of `SRLI_I64` and `SLLI_I64`, but the source code uses OR because it has the same effect. The function `transpose_64x256_sp_asm` is an assembly function that computes the "64×64-block-wise" matrix transposition for obtaining $M^{(1)}$ from $N^{(1)}$. The assembly function performs a vectorized version of the 64×64 matrix transposition mentioned in Section 3.2. We note that the assembly function is included in the source code of Classic McEliece. The source code of Classic McEliece, including the function, is in the public domain. In fact, we also compute $N^{(0)}$ from $M^{(0)}$ by calling the assembly function twice.

3.4 The Haswell Implementation for `bikel3`

`bikel3` has $r = 24659$, so f' consists of at least $\lceil(2r-1)/256\rceil = 193$ words and $s' = s^{(1)}/b \le 96$. For ease of implementation, we let f' be of lenght 65536, which means $M^{(0)}$ is a matrix with 256 rows:

$$M^{(0)} = \begin{pmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{pmatrix} \in \mathbb{F}_2^{256\times 256},$$

where each M_{ij} is a 64×64 matrix. Following the dicussion in Section 3.2, we only need the first $\lceil (24659+255)/256 \rceil = 98$ rows of $\Uparrow_{s'} (M^{(0)})$ to perform the final shift of $s^{(0)}$ positions. For ease of implementation, we instead compute 128 rows of $\Uparrow_{s'} (M^{(0)})$. Let $M^{(0k)} \in \mathbb{F}_2^{128\times 256}$ be the first 128 rows of $\Uparrow_{64k} M^{(0)}$. We observed that there are two cases for the first 128 rows of $\Uparrow_{s'} (M^{(0)})$, which we denote as $M^{(1)}$:

$$M^{(1)} = \begin{cases} \Uparrow_{s'} (M^{(00)}) + \Downarrow_{64-s'} (M^{(01)}) & \text{if } 0 \le s' < 64, \\ \Uparrow_{s'-64} (M^{(01)}) + \Downarrow_{128-s'} (M^{(02)}) & \text{if } 64 \le s' < 128. \end{cases}$$

Let $N_{ij} = M_{ij}^T$. Our implementation stores f' in a way that the corresponding matrix is

$$N^{(0)} = \begin{pmatrix} N_{00} & N_{01} & N_{02} & N_{03} \\ N_{10} & N_{11} & N_{12} & N_{13} \\ N_{20} & N_{21} & N_{22} & N_{23} \\ N_{30} & N_{31} & N_{32} & N_{33} \end{pmatrix} \in \mathbb{F}_2^{256\times 256}$$

If $0 \le s' < 64$, we compute

$$N^{(1)} = \begin{pmatrix} \Leftarrow_{s'} \begin{pmatrix} N_{00} \\ N_{10} \end{pmatrix} & \Leftarrow_{s'} \begin{pmatrix} N_{01} \\ N_{11} \end{pmatrix} & \Leftarrow_{s'} \begin{pmatrix} N_{02} \\ N_{12} \end{pmatrix} & \Leftarrow_{s'} \begin{pmatrix} N_{03} \\ N_{13} \end{pmatrix} \\ + & + & + & + \\ \Rightarrow_{64-s'} \begin{pmatrix} N_{10} \\ N_{20} \end{pmatrix} & \Rightarrow_{64-s'} \begin{pmatrix} N_{11} \\ N_{21} \end{pmatrix} & \Rightarrow_{64-s'} \begin{pmatrix} N_{12} \\ N_{22} \end{pmatrix} & \Rightarrow_{64-s'} \begin{pmatrix} N_{13} \\ N_{23} \end{pmatrix} \end{pmatrix} \in \mathbb{F}_2^{128\times 256}$$

Otherwise, we compute

$$N^{(1)} = \begin{pmatrix} \Leftarrow_{s'-64} \begin{pmatrix} N_{10} \\ N_{20} \end{pmatrix} & \Leftarrow_{s'-64} \begin{pmatrix} N_{11} \\ N_{21} \end{pmatrix} & \Leftarrow_{s'-64} \begin{pmatrix} N_{12} \\ N_{22} \end{pmatrix} & \Leftarrow_{s'-64} \begin{pmatrix} N_{13} \\ N_{23} \end{pmatrix} \\ + & + & + & + \\ \Rightarrow_{128-s'} \begin{pmatrix} N_{20} \\ N_{30} \end{pmatrix} & \Rightarrow_{128-s'} \begin{pmatrix} N_{21} \\ N_{31} \end{pmatrix} & \Rightarrow_{128-s'} \begin{pmatrix} N_{22} \\ N_{32} \end{pmatrix} & \Rightarrow_{128-s'} \begin{pmatrix} N_{23} \\ N_{33} \end{pmatrix} \end{pmatrix} \in \mathbb{F}_2^{128\times 256}$$

In either case, each of the 8 64×64 submatrices in $N^{(1)}$ is transposed to obtain $M^{(1)}$.

It might seem that the implementation strategy above will result in branching on s', and branching on s' is not allowed for constant-time implementations. However, this can be avoided with the ability of AVX intrinsics. Our source code for computing $M^{(1)}$ from $N^{(0)}$ is given in Figure 3. The variable `in`, again, is a pointer for $N^{(0)}$. The variable `ymm_num`, again, is set to s'. When $0 \le s' < 64$, the `SRLI_I64` and `SLLI_I64` in line 12 and 13 will give all-zero results because their shift amounts are out-of-range. Similarly, when $64 \le s' < 128$, the `SRLI_I64` and `SLLI_I64` in line 10 and 11 will give all-zero results because their shift amounts are out-of-range. In this way, we are able to avoid branching on s'. Finally, we compute $M^{(1)}$ from $N^{(1)}$. by calling the assembly function `transpose_64x256_sp_asm` twice. We also compute $N^{(0)}$ from $M^{(0)}$ by calling the assembly function four times.

4 Circular Shifts with Conditional Moves

Following the discussion in Section 3.1, our M4 implementation uses a Barrel shifter to shift the duplicated form of f by s bits. In the setting of M4, we have $s = s^{(1)} + s^{(0)}$ where $s^{(0)} = s \bmod 32$, and the duplicated form of f is stored as an array of 32-bit words. Our M4 implementation first performs the shift by $s^{(1)}$ bits and then the shift by $s^{(0)}$ bits.

```
_INLINE_ void
rotate256_big(OUT syndrome_t *out, IN __m256i *in, IN size_t ymm_num)
{
  int j;

  __m256i tmp;

  for (j = 0; j < 128; j++)
  {
    tmp = SRLI_I64(in[j], ymm_num)
        | SLLI_I64(in[j+64], 64 - ymm_num)
        | SRLI_I64(in[j+64], ymm_num - 64)
        | SLLI_I64(in[j+128], 128 - ymm_num);

    STORE(&out->qw[4*j], tmp);
  }

  transpose_64x256_sp_asm(&out->qw[0]);
  transpose_64x256_sp_asm(&out->qw[64*4]);
}
```

Figure 3: The function for computing $M^{(1)}$ from $N^{(0)}$ for `bikel3`.

4.1 The Shift by $s^{(1)}$ Bits

In the first phase of the Barrel shifter, each word i is conditionally replaced by word $i+2^j/32$ for several j's with $j \geq 5$ to carry out the shift by $s^{(1)}$ bits. Note that $j \in \{13, 12, \ldots, 5\}$ for `bikel1` and $j \in \{14, 13, \ldots, 5\}$ for `bikel3`. To complete the computation for each j, we can first create a 32-bit `mask` which is set to `0xFFFFFFFF` if the shift of $2^j/32$ words needs to be carried out or `0x0` otherwise. Then, for $\mathtt{i} = 0, 1, 2, \ldots$, the `portable` implementation does

```
w[i] = (w[i] & ~mask) | (w[i+idx] & mask),
```

where `idx` holds the value of $2^j/32$. This is essentially the code of the `portable` implementation, except that it actually works on 64-bit words.

In our implementation, for each k in the first phase, the words are partitioned into $2^j/32$ sets: each word i with $k = i \bmod 2^j/32$ is in the kth set. For each set, we use two general-purpose registers `Rx` and `Ry`, where `Rx` is initialized with word k. Then, we set i to k and perform the following loop.

1. Load word $i + 2^j/32$ to `Ry`.
2. Conditionally move `Ry` to `Rx`.
3. Store `Rx` to word i.
4. If $i < \min(\lceil (r + 2^j - 1)/32 \rceil$ and $i + 2^j/32 < \lceil (2r - 1)/32 \rceil$, increase i by $2^j/32$. Otherwise, break the loop.
5. Load word $i + 2^j/32$ to `Rx`.
6. Conditionally move `Rx` to `Ry`.
7. Store `Ry` to word i.
8. If $i < \min(\lceil (r + 2^j - 1)/32 \rceil$ and $i + 2^j/32 < \lceil (2r - 1)/32 \rceil$, increase i by $2^j/32$ and go back to Step 1. Otherwise, break the loop.

In Step 4 and 8, we check if $i < \min(\lceil (r + 2^j - 1)/32 \rceil$ because we only need

$$\lceil (r + \sum_{i=0}^{j-1} 2^i)/32 \rceil = \lceil (r + 2^j - 1)/32 \rceil$$

words for the remaining conditional shifts of $2^{j-1}, 2^{j-2}, \ldots, 32, s^{(0)}$ bits. We check if $i + 2^j/32 < \lceil (2r-1)/32 \rceil$ because there are only $\lceil (2r-1)/32 \rceil$ words. In our implementation, the if statements are simplified to "If $i < i_{\max}$, ...", where $i_{\max}$ is a precomputed value that depends on k and j.

To carry out the conditional moves, our M4 implementation uses the instruction `SEL`. As suggested by the name, `SEL` can move one of its two source registers into its destination register. Which source register is chosen is determined by the "GE" flags [2]. We note that the `SEL` instruction always takes 1 cycle.

In order to reduce the average cost of each conditional move, our implementation actually processes $n = \min(2^j/32, 4)$ words in one iteration of the loop: we first load n words to `Ry1`, ..., `Ryn`, use the `SEL` instruction n times to conditionally move `Ry1`, ..., `Ryn` to the corresponding `Rx1`, ..., `Rxn`, store `Ry1`, ..., `Ryn` to the memory of n words. We also partially unroll the loop so that Step 4 is omitted. A piece of code for a loop of $j = 12$ is given in Appendix A.

4.2 The Shift by $s^{(0)}$ Bits

In the second phase, the Barret shifter performs the shift of f by $s^{(0)} < 32$ bits. Suppose f is stored as an array of 32-bits words `w[n]`. In the `portable` implementation of BIKE team, the shift is carried out by doing

```
w[i] = (w[i] ≫ s0) | (w[i+1] ≪ (32-s0));
```

for $i = 1, 2, \ldots$, and so on. This approach takes 2 logic shits and 1 logic `OR` operations for each 32-bits word. We note that `portable` actually works on 64-bit words, but the idea is the same. In fact, we can also deal with the 32-bit words in the reversed order:

```
v = 0;
for (i = n-1; i >= 0; i--)
{
  tmp = w[i] << (32-s0);
  w[i] = (w[i] >> s0) | v;
  v = tmp;
}
```

Note that the logical OR can be replaced by an addition.

Now consider the following C function.

```
void F(uint32* w, uint32* v, s0)
{
  uint32_t tmp = *w << (32-s0);
  *w = (*w >> s0) + v;
  *v = tmp;
}
```

The shift by $s^{(0)}$ bits can then be carried out by doing simply

[2] Actually the selection is done on a byte per byte basis. For example, with the right values in the GE flags, one can write the lower 8 bits of the first source register to the corresponding bits of the destination register, and write the upper 24 bits of the second source register to corresponding bits of the destination register.

```
v = 0;
for (i = n-1; i >= 0; i--)
  F(w[i], &v, s0);
```

The interesting thing is that `F` can be replaced by a single instruction `UMLAL`. The instruction operates as

`UMLAL Rdlo, Rdhi, Rm, Rn`

where `Rdlo` and `Rdhi` are the 2 destination registers and `Rm` and `Rn` are the 2 source registers. The instruction multiplies `Rm` and `Rn` and adds the 64-bit result into the 64-bit integer represented by `Rdhi` (the top 32 bits) and `Rdlo` (the bottom 32 bits). To carry out `F(w[i], &v, s0)` with the `UMLAL`, the 4 registers `Rdlo`, `Rdhi`, `Rm`, and `Rn` are set to `w[i]`, `v`, `(1 « s0)-1`, and `w[i]` respectively. Note that we set `Rdlo` to `w[i]` and `Rm` to `(1 « (32-s0))-1` instead of setting `Rdlo` to 0 and `Rm` to `1 « (32-s0)` to avoid the overflow of `Rm` when $s^{(0)} = 0$. In this way, we are able to replace the 2 logical shifts and 1 logical OR with only 1 `UMAAL`.

A previous version of our M4 implementation actually used `IT` blocks instead of `SEL` to carry out conditional moves and use the same way as the `portable` implementation to perform the shift by $s^{(0)}$ bits. The ideas of using `SEL` and replacing shift instructions by multiplication instructions was suggested by an anonymous reviewer of this paper. In the current version of our M4 implementation, which uses the ideas, one circular shift is about 5% faster than in the previous version.

5 Hamming Weight Computation with Full Adders

The implementations of the BIKE team follows [12] to compute the sum $\sum_s (y^{-s} f)$ in a bitsliced fashion to obtain gf. As the result, a sequence of bitwise logical operations are performed to obtain the sum. Let n be the cardinality for the set of s, and let b_1 be the constant term of the first $y^{-s} f$, b_2 be the constant term of the second $y^{-s} f$, and so on. It is easy to see that the problem of computing $\sum_s (y^{-s} f)$ boils down to computing the Hamming weight of $(b_1, b_2, \ldots, b_n) \in \{0, 1\}^n$ using a sequence of gates. This sections shows how previous implementations and our implementation compute the Hamming weight.

5.1 Simple Algorithms for Computing the Hamming Weight

The `avx2` implementation of BIKE, following [12], first prepares a counter of $\lfloor \log_2 n \rfloor + 1$ zero bits. The counter is used to store the Hamming weight. Then, for each $i \in \{1, \ldots, n\}$, b_i is added to the counter. For each addition, if n is a power of 2, each of the least significant $\lfloor \log_2 n \rfloor$ bits can be updated using one half adder, and the most significant bit can be set to the carry-out bit of the last half adder. If n is not a power of 2, each of the least significant $\lfloor \log_2 n \rfloor$ bits can be updated using one half adder, and the most significant bit can be updated using 1 `XOR`. Let $T(n)$ be the number of bit operations it takes to compute the Hamming weight of n bits. We have

$$T(n) = \begin{cases} (2\lfloor \log_2 n \rfloor + 1)n, & \text{if } n \text{ is not a power of 2.} \\ 2\lfloor \log_2 n \rfloor n, & \text{if } n \text{ is a power of 2.} \end{cases}$$

In [19], the authors suggested a simple way to improve the approach above. The idea is that, since adding b_i only affects the first $\lfloor \log_2 i \rfloor + 1$ bits of the counter, there is no need to update the remaining bits. In fact, adding b_i takes $2\lfloor \log_2 i \rfloor + 1$ bit operations if i is not a power of 2, and adding b_i takes $2\lfloor \log_2 i \rfloor$ bit operations if i is a power of 2. Therefore, we have

$$T(n) = \sum_{i=1}^{n} C(i), \quad C(i) = \begin{cases} (2\lfloor \log_2 i \rfloor + 1), & \text{if } i \text{ is not a power of 2.} \\ 2\lfloor \log_2 i \rfloor, & \text{if } i \text{ is a power of 2.} \end{cases}$$

5.2 The Boyar–Peralta Algorithm

Our implementation uses a much faster algorithm which makes use of full adders. The algorithm was explained by Boyar and Peralta in [9]. We note that the technique is probably much older: [17] presents an algorithm that seems equivalent. However, as we are not able to tell who was the first to propose the algorithm, we simply call the algorithm the Boyar–Peralta Algorithm. Boyar and Peralta focused on proving that the algorithm gives the minimal number of `AND`s required to compute the Hamming weight. In the following discussion, we consider the number of `XOR`s as well as the space requirement of the algorithm.

To explain the idea, consider the case when $n = 2^k - 1$. Boyar and Peralta described the following recursive algorithm: apply the algorithm recursively to compute the Hamming weight x_1 of the first $2^{k-1} - 1$ bits, apply the algorithm recursively to compute the Hamming weight x_2 of the next $2^{k-1} - 1$ bits, and use a standard adder to compute the sum $x_1 + x_2 + b_n$, where b_n serves as the carry-in bit. Then the sum $x_1 + x_2 + b_{n-1}$ can be computed using $k - 1$ full adders, resulting in $5(k - 1)$ bit operations. Therefore, we have

$$T(2^k - 1) = 5(k - 1) + 2T(2^{k-1} - 1)$$

for $k > 1$, and obviously $T(0) = T(1) = 0$.

In order to handle general n, the Boyar–Peralta algorithm computes k, n_1, n_2 such that $n = 2^{k-1} + n_2 = n_1 + n_2 + 1$, where $0 \leq n_2 < 2^{k-1}$. Then, the algorithm is applied recursively to compute the Hamming weight x_1 of the first n_1 bits, and the Hamming weight x_2 of the next n_2 bits, and finally a standard adder is used to compute the sum $x_1 + x_2 + b_{n-1}$. Let $k' = \lfloor \log_2 n_2 \rfloor + 1$ if $n_2 > 0$, i.e., k' is the number of bits required to represent n_2. k' is set to 0 if $n_2 = 0$. Then, the sum $x_1 + x_2 + b_{n-1}$ can be computed using k' full adders and $k - 1 - k'$ half adders, resulting in $5k' + 2(k - 1 - k')$ bit operations. Therefore, we have

$$T(n) = 5k' + 2(k - 1 - k') + T(2^{k-1} - 1) + T(n_2)$$

for $n \geq 2$, and $T(0) = T(1) = 0$.

The reason why we need to perform multiplications of the form $gf = \sum_s (y^{-s} f)$ in the first place is because there are multiplications between $\mathrm{Tr}(h_i)$ and $\mathrm{Lift}(s)$ in the decoder (see Section 2.4). As each $\mathrm{Tr}(h_i)$ is always of weight $w/2$, our goal is actually to compute the Hamming weight of $w/2$ bits. Now we can easily compute $T(w/2)$ for the Boyar–Peralta algorithm and the simple algorithms in the previous subsection. The numbers are summarized in Table 2.

5.3 Code Generation for the Boyar–Peralta Algorithm

To implement the Boyar–Peralta algorithm, we wrote a code generator to generate the sequence of additions in the Boyar–Peralta algorithm. The code generator is included in Appendix B. It assumes that there is a buffer consisting of infinitely many bits that can be used for storing the results of the additions and the Hamming weight.

Our code generator is written as a Python function named `bp`. One can simply call `bp(n, 0)` to to generate operations for n input bits. Its the second argument, the variable `off` in its signature, indicates the index of the first bit in the buffer of the computation. The function, on input $n > 1$, computes k, n_1, n_2 such that $n = 2^k + n_2 = n_1 + n_2 + 1$. Then, the function first recursively calls

```
bp(n1, off);
bp(n2, off + nbits(n1));
```

to generate operations for computing x_1 and x_2, where `nbits(n1)` means the number of bits required to represent n_1. Note x_1 will be stored in the first `nbits(n1)` bits starting

Table 2: The numbers of gates required to compute the Hamming weight for the three methods.

	$n = w/2$	[12]	[19]	[9]
bikel1	71	923	676	326
bikel3	103	1339	1092	484
bikel5	137	2055	1553	655

from the buffer bit of index `off` after the operations generated by the first recursive call are carried out, and x_2 will be stored in the next `nbits(n2)` bits after the operations generated by the second recursive call are arried out.

After the recursive calls, the function first outputs

```
"rotate_right(bn, f, s[{0}]);".format(count)
```

which means a new s is used to compute $y^{-s}f$, so that the bit `bn` is generated. Here `count` is a global variable which is always increased by 1 after printing the line of `rotate_right`. Then, the function outputs

```
"adder_size_eq(buf, bn, {0}, {1});".format(off, nbits(n1))
```

if `nbits(n1)` is equal to `nbits(n2)` or

```
"adder_size_neq(buf, bn, {0}, {1}, {2});".format(off, nbits(n1), nbits(n2))
```

if they are different. In either case, this means that a standard adder is used to compute $x_1 + x_2 + b_n$. The variable `buf` is a pointer to the buffer, and the offset of x_1 is indicated by `off`. The difference between the two cases is that `adder_size_eq` only requires one argument to indicate the bit length of x_1 or x_2, while `adder_size_neq` requires two arguments. If $n = 1$, the function outputs the line of `rotate_right`, increases `count` by 1, and outputs

```
"adder_size_eq(buf, bn, {0}, 0);".format(off)
```

where the last argument 0 indicates that b_n is simply copied into the buffer bit of index `off`. If $n = 0$, the function returns immediately. The only remaining task is to implement the C functions `adder_size_eq` and `adder_size_neq`.

By analysing the output of the function, we found that a buffer of 16, 17, and 22 bits are required for $n = 71$, $n = 103$, and $n = 137$. Note that the simple algorithms requires a buffer of only 7, 7, and 8 bits respectively. In other words, using the Boyar–Peralta algorithm to compute the sum $\sum_s (y^{-s}f)$ in a bitsliced fashion takes at least $\lceil 16 \cdot 12323/8 \rceil = 24646$, $\lceil 17 \cdot 24659/8 \rceil = 52401$, and $\lceil 22 \cdot 40973/8 \rceil = 112676$ bytes for `bikel1`, `bikel3`, and `bikel5`, respectively.

6 Multiplications in $\mathbb{F}_2[x]$ with Bernstein's 5-way Recursive Algorithm

The implementations of the BIKE team use Karatsuba for multiplying polynomials in $\mathbb{F}_2[x]$: given two polynomials $F = F_0 + F_1 x^n$ and $G = G_0 + G_1 x^n$ where F_0, F_1, G_0, G_1 are of degree smaller than n, the algorithm computes FG as

$$F_0 G_0 + F_1 G_1 x^{2n} + ((F_0 + F_1)(G_0 + G_1) - F_0 G_0 - F_1 G_1) x^n.$$

This means that Karatsuba is a 3-way recursive algorithm: it reduces the multiplication into 3 smaller ones, where the smaller multiplications can be handled recursively. The complexity of Karatsuba is $O(n^{\log_2 3})$.

Recall that bikel1 has $r = 12323$ and bikel3 has $r = 24659$. The BIKE team always picks n to be the smallest possible power of 2, which means at the top level of recursion, we have $n = 8192$ for bikel1 and $n = 16384$ for bikel3. They then apply Karatsuba recursively. At the bottom level of recursion, the avx2 implementation uses pclmulqdq, and the portable implementation uses a non-constant-time function to handle multiplications of polynomials of degree smaller than 64.

As r seems to be large enough to make use of algorithms with lower complexities, our Haswell implementation uses a 5-way recursive algorithm by Bernstein.

6.1 Bernstein's 5-way Recursive Algorithm

In [5], Bernstein proposed a 5-way recursive algorithm for multiplying two polynomials in $\mathbb{F}_2[x]$ with five multiplications. The complexity of Bernstein's algorithm is $O(n^{\log_3 5})$. Given two polynomials $F_0 + F_1 z + F_2 z^2$ and $G_0 + G_1 z + G_2 z^2$ in $\mathbb{F}_2[x]$, where F_i's and G_i's are polynomials of degree smaller than n and $z = x^n$. The algorithm reduces the multiplication into five polynomial multiplications of polynomials of length about n:

$$\begin{aligned}
H(0) &= F_0 \cdot G_0, \\
H(1) &= (F_0 + F_1 + F_2) \cdot (G_0 + G_1 + G2), \\
H(x) &= (F_0 + F_1 x + F_2 x^2) \cdot (G_0 + G_1 x + G_2 x^2), \\
H(x+1) &= ((F_0 + F_1 + F_2) + F_1 x + F_2 x^2) \cdot ((G_0 + G_1 + G_2) + G_1 x + G_2 x^2), \\
H(\infty) &= F_2 \cdot G_2.
\end{aligned}$$

Bernstein showed that $H = F \cdot G$ can be constructed by the following formula

$$H = U + H(\infty)(z^4 + z) + \frac{U + V + H(\infty)(x^4 + x)}{x^2 + x}(z^2 + z) \tag{1}$$

where

$$U = H(0) + (H(0) + H(1))z \text{ and } V = H(x) + (H(x) + H(x+1))(x + z) \ .$$

6.2 Implementing Bernstein's algorithm

Our Haswell implementation uses Bernstein's algorithm at the top level of the recursion and Karatsuba for other levels. At the top level, we pick $n = 4096$ for bikel1 and $n = 8192$ for bikel3. Note that $12323 - 4096 \cdot 3 = 35$ and $24659 - 8192 \cdot 3 = 83$. These "leftover bits" are handle separately, and the costs of handling them is very small. In order to make the operands of the 5 multiplications fit into $n = 4096$ or $n = 8192$ bits, we also handle the top 2 bits of F_2 and G_2 and the top bit of F_1 and G_1 separately, together with the "leftover bits". At the bottom level of recursion, we also use pclmulqdq to handle multiplications for polynomials of degree smaller than 64.

Let $h(x) = \sum_{i=0}^{\ell-1} h_i x^i = (U + V + H(\infty)(x^4 + x))$. While most of the operations in Bernstein's algorithm are either multiplications or additions, there is one division between $h(x)$ and $(x^2 + x)$ in Eq. 1. We carry out the division by computing

$$h_{\ell-1}x^{\ell-3} + (h_{\ell-1} + h_{\ell-2})x^{\ell-4} + \ldots + (\Sigma_{i=2}^{\ell-1} h_i).$$

Note that instead of computing the coefficients sequentially, the computation can be easily parallelized: let h' be $(h - (h \bmod x^2))/x^2$, we update h' to $h' + (h' - (h' \bmod x))/x$, update h' to $h' + (h' - (h' \bmod x^2))/x^2$, update h' to $h' + (h' - (h' \bmod x^4))/x^4$, and so on. This results in a much faster implementation for the division.

7 Polynomial Multiplications in $\mathbb{F}_2[x]$ with Frobenius Additive FFTs

This section presents how we carry out the polynomial multiplications on the Cortex-M4. Section 7.1 reviews how the Gao-Mateer additive FFT works. Section 7.2 reviews how to perform multiplications in $\mathbb{F}_2[x]$ with the additive FFT in [22, 11]. Section 7.3 shows how we implement the algorithm for the Cortex-M4.

7.1 The Gao–Mateer Additive FFT

Below we introduce how the Gao–Mateer additive FFT [18], which we denote as AFFT, can be used for polynomials over $\mathbb{F}_{2^m}$, where m is a power of 2.

Cantor Basis An $\mathbb{F}_2$-linear basis $\{v_0, v_1, \ldots, v_{m-1}\}$ of $\mathbb{F}_{2^m}$, where m is a power of 2, is called a Cantor basis [10] if $v_0 = 1$ and $v_{i-1} = v_i^2 + v_i$ for all $i > 0$. We define $W_\ell \subseteq \mathbb{F}_{2^m}$ as the span of $\{v_0, \ldots, v_{\ell-1}\}$ for any $1 \le \ell \le m$ and W_0 as $\{0\}$. When ℓ is a power of 2, W_ℓ is the subfield of size 2^ℓ. In [10], Cantor defined a series of polynomials that are closely related to v_i's and W_ℓ's:

$$s_0(x) = x, \text{ and } s_i(x) = (s_{i-1}(x))^2 + s_{i-1}(x) \text{ for } i \ge 1.$$

In fact, each s_i satisfies the following properties.

- $s_i(x)$ is linear in the sense that $s_i(\alpha + \beta) = s_i(\alpha) + s_i(\beta)$.
- $s_i(v_{k+i}) = v_k$ for all $i \ge 0$.

Divide and Conquer Given $f(x) \in \mathbb{F}_{2^m}[x]$ of degree smaller than n, where $n = 2^\ell$, $v \in \mathbb{F}_{2^m}$, and $\ell \le m$, the AFFT computes $S = \{f(\alpha) \mid \alpha \in v + W_\ell\}$. When $\ell = 0$, the algorithm simply returns $f(x)$ itself. When $\ell > 0$, the algorithm first writes $f(x)$ as

$$f^{(0)}(s_1(x)) + x \cdot f^{(1)}(s_1(x)).$$

This is called "radix conversion" in [7]: the radix is changed from x to $s_1(x) = x^2 + x$. Note that the computation $f^{(0)}$ and $f^{(1)}$ from f requires only field additions, which are much cheaper than field multiplications, and each $f^{(i)}$ is of degree smaller than $2^{\ell-1}$. If we have $S^{(0)} = \{f^{(0)}(s_1(\alpha)) \mid \alpha \in v + W_\ell\}$ and $S^{(1)} = \{f^{(1)}(s_1(\alpha)) \mid \alpha \in v + W_\ell\}$, then every $f(\alpha)$ can be computed as $f(\alpha) = f^{(0)}(\alpha) + \alpha f^{(1)}(\alpha)$.

Observe that $s_1(v + W_\ell) = s_1(v) + W_{\ell-1}$, which is of size $2^{\ell-1}$. Gao and Mateer make use of this observation to compute $S^{(0)}$ and $S^{(1)}$ by recursively evaluating $f^{(0)}$ and $f^{(1)}$ at all elements of $s_1(v) + W_{\ell-1}$. In addition, as

$$v + W_\ell = (v + \operatorname{span}\{v_{\ell-1}, \ldots, v_1\}) \cup (v + \operatorname{span}\{v_{\ell-1}, \ldots, v_1\} + 1),$$

Gao and Mateer suggest to compute S by computing

$$f(\beta) = f^{(0)}(s_1(\beta)) + \beta f^{(1)}(s_1(\beta)), \quad f(\beta + 1) = f(\beta) + f^{(1)}(s_1(\beta)),$$

for all $\beta \in v + \operatorname{span}\{v_\ell - 1, \ldots, v_1\}$. Computation of $f(\beta)$ and $f(\beta + 1)$ is depicted by the following figure.

$f^{(0)}(s_1(\beta)) \longrightarrow \oplus \longrightarrow f(\beta)$

β

$f^{(1)}(s_1(\beta)) \longrightarrow \bullet \longrightarrow \oplus \longrightarrow f(\beta + 1)$

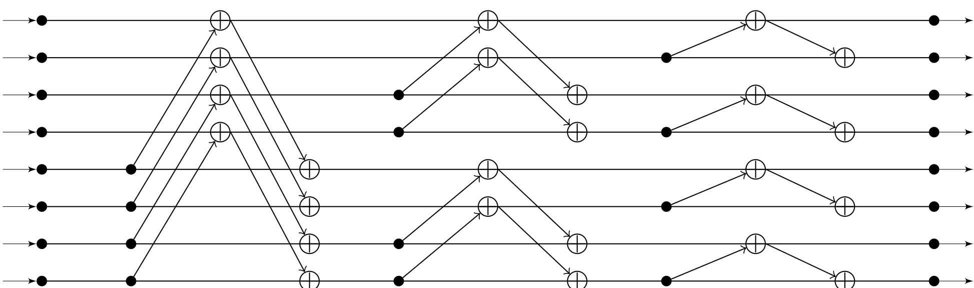

Figure 4: The three stages of butterflies in a size-8 AFFT. Each arrow going upwards is associated with a constant which is not shown, so it actually means "adding the product of the source and the constant to the destination" instead of "adding the source to the destination".

This is one "butterfly" in an AFFT.

To have a clearer picture regarding the pattern of computation in an AFFT, one can "unroll" the recursion. In this way, it can be seen that an AFFT always consists of two phases. The first phase consists of a bunch of field additions from radix conversions in all levels of recursion. The second phase consists of several stages of butterflies, where the last stage consists of the butterflies in level 0 (i.e., the top level) of the recursion, the second last stage consists of the butterflies in level 1 of the recursion, and so on. Figure 4 shows the three stages in the second phase of a size-8 ($n = 8$) AFFT.

7.2 Multiplication in $\mathbb{F}_2[x]$ with FAFFT

The AFFT allows us to perform multiplications in $\mathbb{F}_2[x]$: given polynomials $f, g \in \mathbb{F}_2[x]$ such that fg is of degree smaller than $n = 2^\ell$, we can lift f and g to $\mathbb{F}_{2^m}[x]$ such that m is a power of 2 and $\ell \le m$, perform an AFFT for each of f and g to evaluate them at W_ℓ, perform the pointwise multiplication, and perform the inverse AFFT to obtain fg. Such an AFFT-based polynomial multiplication always works, but one can reduce the cost of the AFFT by carefully choosing the evaluation points for f and g.

Reducing the Number of Evaluation Points In [22], Li et al. suggested to pick the set of evaluation points to be

$$\Sigma = W_{\ell'} + v_{\ell'+m/2}$$

with

$$\ell' + m/2 < m, \;\; n' = |\Sigma| = \frac{n}{m}, \;\; \ell' = \log_2^{n'} < \frac{m}{2}. \tag{2}$$

In this way, the function that maps f to $f(\Sigma)$ will be invertible, which means we can use the AFFT and the inverse AFFT to carry out multiplications in $\mathbb{F}_2[x]$. As explained by Li et al., it is the property that Σ can reach n elements in $\mathbb{F}_{2^m}$ via the Frobenius map which guarantees that the function is invertible. This is why Li et al. call such an AFFT a Frobenius AFFT (FAFFT). With the FAFFT, the number of evaluation points is reduced by a factor of m.

FAFFT by Truncating A Size-n AFFT Consider $\hat{\Sigma} = W_{\ell'+\log_2 m} + v_{\ell'+m/2}$, such that $\Sigma \subset \hat{\Sigma}$ and $|\hat{\Sigma}| = n$. The way Li et al. evaluate f as Σ is to perform a size-n AFFT to evaluate f at $\hat{\Sigma}$ but skip all the redundant operations. In other words, the approach of Li et al. is to carry out a truncated AFFT. The truncated AFFT starts with a usual radix conversion phase for a size-n AFFT. The second phase consists of the first $\log_2 m$

Algorithm 2 FAFFT-based Polynomial Multiplication

Parameters: n: maximum length of output polynomial

1: **procedure** FAFFT($f(x) \in \mathbb{F}_2[x]_{<n}, \Sigma$)
2: $(f_0, f_1, \ldots, f_{n-1}) \in \mathbb{F}_2^n \leftarrow$ `RadixConversions`$(f(x))$
3: $(f'_0, \ldots, f'_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow$ `Encode`$((f_0, f_1, \ldots, f_{n-1}), \Sigma)$
4: $(\hat{f}_0, \ldots, \hat{f}_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow$ `Butterflies`$((f'_0, \ldots, f'_{n'-1}), \Sigma)$
5: **return** $(\hat{f}_0, \ldots, \hat{f}_{n'-1})$
6: **end procedure**
7: **procedure** FAFFT^{-1}($(\hat{f}_0, \ldots, \hat{f}_{n'-1}) \in \mathbb{F}_{2^m}^{n'}, \Sigma$)
8: $(f'_0, \ldots, f'_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow$ `Butterflies`$^{-1}((\hat{f}_0, \ldots, \hat{f}_{n'-1}), \Sigma)$
9: $(f_0, \ldots, f_{n'-1}) \in \mathbb{F}_2^n \leftarrow$ `Encode`$^{-1}((f'_0, f'_1, \ldots, f'_{n-1}), \Sigma)$
10: $f(x) \in \mathbb{F}_2[x]_{<n} \leftarrow$ `RadixConversions`$^{-1}(f_0, f_1, \ldots, f_{n-1})$
11: **return** $f(x)$
12: **end procedure**
13: **procedure** BitPolyMul($a(x) \in \mathbb{F}_2[x]_{<\frac{n}{2}}, b(x) \in \mathbb{F}_2[x]_{<\frac{n}{2}}$)
14: $(\hat{a}_0, \ldots, \hat{a}_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow$ `FAFFT`$(a(x), \Sigma)$
15: $(\hat{b}_0, \ldots, \hat{b}_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow$ `FAFFT`$(b(x), \Sigma)$
16: $(\hat{c}_0, \ldots, \hat{c}_{n'-1}) \in \mathbb{F}_{2^m}^{n'} \leftarrow (\hat{a}_0 \cdot \hat{b}_0, \ldots, \hat{a}_{n'-1} \cdot \hat{b}_{n'-1})$ ▷ pointwise multiplication
17: $c(x) \in \mathbb{F}_2[x]_{<n} \leftarrow$ `FAFFT`$^{-1}((\hat{c}_0, \ldots, \hat{c}_{n'-1}), \Sigma)$
18: **return** $c(x)$
19: **end procedure**

levels of butterflies in the truncated AFFT. This phase is called encoding, in which the operations is defined below. The third phase consists of the last ℓ' stages of butterflies in the truncated AFFT. The last stage is identical to the butterfly phase in a usual size-n' AFFT for Σ.

As each of the three phase is invertible, one can carry out the inverse FAFFT by performing the inverses of the three phases in the reverse order. The FAFFT-based polynomial multiplication algorithm is shown in Algorithm 2.

Encoding Encoding performs the first $\log_2 m$ levels of butterflies in the truncated AFFT. With respect to $\hat{\Sigma}$, it is an invertible linear map which maps radix converted coefficients of $f(x)$, e.g., $(f_0, f_1, \ldots, f_{n-1}) \in \mathbb{F}_2^n$ to $(f'_0, \ldots, f'_{n'-1}) \in \mathbb{F}_{2^m}^{n'}$ where the f'_i is defined as

$$f'_i = \sum_{j=0}^{m-1} r_j \cdot f_{j \cdot n_p + i}, \quad r_j = \prod_{k=0}^{\log_2 m - 1} (v_{m/2-k})^{j_k}, \tag{3}$$

where $(j_0, j_1, \ldots, j_{\log_2 m - 1}) \in \{0,1\}^{\log_2 m}$ is the binary representation of the integer j, i.e., $j = \sum_k j_k \cdot 2^k$.

7.3 Implementing the FAFFT-based Polynomial Multiplication

Below we show how we implemented the FAFFT-based multiplication for the M4.

Parameter Selection The following table shows our parameters for the algorithm 2. We pick the parameters to minimize m satisfying Eq. (2). The field $\mathbb{F}_{2^{32}}$ is constructed as

	r	n	m	n'	ℓ'	Σ
BIKE-1	12323	32768	32	1024	10	$v_{26} + W_{10}$
BIKE-3	24659	65536	32	2048	11	$v_{27} + W_{11}$

a tower of extension fields of $\mathbb{F}_2$ as in [7, 22]:

$$\begin{aligned}
\mathbb{F}_{2^2} &= \mathbb{F}_2[x_1]/(x_1^2 + x_1 + 1),\\
\mathbb{F}_{2^4} &= \mathbb{F}_{2^2}[x_2]/(x_2^2 + x_2 + x_1),\\
\mathbb{F}_{2^8} &= \mathbb{F}_{2^4}[x_4]/(x_4^2 + x_4 + x_2x_1),\\
\mathbb{F}_{2^{16}} &= \mathbb{F}_{2^8}[x_8]/(x_8^2 + x_8 + x_4x_2x_1),\\
\mathbb{F}_{2^{32}} &= \mathbb{F}_{2^{16}}[x_{16}]/(x_{16}^2 + x_{16} + x_8x_4x_2x_1).
\end{aligned}$$

For completeness, we note that our Cantor basis is defined by

$$\begin{aligned}
v_{31} = &1 + x_1x_2 + x_1x_2x_4 + x_8 + x_1x_2x_8 + x_2x_4x_8 + x_1x_2x_4x_8\\
&+ x_{16} + x_1x_2x_{16} + x_4x_{16} + x_1x_4x_{16} + x_2x_4x_{16} + x_1x_2x_4x_{16}\\
&+ x_1x_8x_{16} + x_4x_8x_{16} + x_1x_2x_4x_8x_{16}.
\end{aligned}$$

The Pointwise Multiplication The pointwise multiplication consists of 1024 or 2048 independent multiplications in $\mathbb{F}_{2^{32}}$. To carry out these multiplications, we follow [7, 13] to represent field elements in a bitsliced format. As the M4 is a 32-bit platform, bitslicing means that we always perform 32 multiplications in parallel.

To multiply $a_0 + a_1x_{16} \in \mathbb{F}_{2^{32}}$ and $b_0 + b_1x_{16} \in \mathbb{F}_{2^{32}}$ (in parallel with 31 other multiplications), we use the Karatsuba algorithm to reduce the task into 3 multiplications $a_0b_0, a_1b_1, (a_0 + b_0)(a_1 + b_1)$ in $\mathbb{F}_{2^{16}}$ and apply the Karatsuba recursively as [7]. We note that the amount of data for multiplication in $\mathbb{F}_{2^4}$ fits into the general-purpose registers, so there is no register spilling in the case. For larger fields, we use floating-point registers as a pool for register spilling.

The Butterflies Section 7.1 shows that, at level i of recursion (level 0 represents the top level), multiplications in the butterflies are always of the form $\beta \cdot f^{(1)}(s_1(\beta))$, where

Table 3: Cost of multiplying an element in $\mathbb{F}_{2^{32}}$ by various field elements. Note that the cycle counts are for carrying out 32 such multiplications. For the environment of benchmarking, see Section 8.

mult. by	ANDs	XORs	cycles
$\mathbb{F}_{2^4}$	$12 \cdot 8$	$18 \cdot 8$	465
$\mathbb{F}_{2^8}$	$36 \cdot 4$	$74 \cdot 4$	871
$\mathbb{F}_{2^{16}}$	$108 \cdot 2$	$270 \cdot 2$	1652
$\mathbb{F}_{2^{32}}$	324	930	2849
v_{17}	0	210	760
v_{18}	0	211	742
v_{19}	0	231	800
v_{20}	0	226	798
v_{21}	0	231	795
v_{22}	0	222	795
v_{23}	0	239	836
v_{24}	0	249	854
v_{25}	0	251	881
v_{26}	0	248	861
v_{27}	0	243	847

$\beta \in v_{\ell'-i+m/2} + W_{\ell'-i}$. Following [7, 13], we pre-compute all β's and store them (in the bitsliced format) in a static table. The multiplications can of course be carried out by our function for 32 multiplications in $\mathbb{F}_{2^{32}}$. However, our implementation does better by exploiting the fact that $\beta = v + w$, where $v \in v_{\ell-i+m/2}$ and $w \in W_{\ell'-i}$. To carry out a multiplication between β and γ, our implementation multiplies γ by v, multiplies γ by w, and adds the two results to obtain $\beta \cdot \gamma$.

Observe that w is always in a proper subfield of size $2^{2^{\lceil \log_2(\ell'-i) \rceil}}$. Due to the tower field structure, we carry out the multiplication by w as $32/2^{\lceil \log_2(\ell'-i) \rceil}$ multiplications in the subfield. To carry out the multiplication by v, we consider the operation as an invertible linear map and use the circuit generator introduced in the ePrint version of [7] to generate a sequence of `XOR`s for the multiplication. We note that $m = 32$ is actually too large for the circuit generator to handle: it stops making progress usually after several tens of iterations. When the circuit generator stops to make progress, we generate another sequence of `XOR`s reaching the state in a naive way. Then, we combine the 2 sequences of `XOR`s to complete the whole invertible linear operation. We've considered Paar's circuit generator [25], but it hard to fit everything into available registers since it does not generate in-place code. We also tried Bernstein's circuit generator [6], but the results (in terms of numbers of `XOR`s) are worse than the one from [7].

We show the costs of various multiplications in Table 3 to evaluate our strategy of performing multiplication by $\beta = v + w$. As shown in the table, carrying out 32 multiplications by v takes at most 881 cycles, and carrying out 32 multiplications by 32 w's takes at most 1652 cycles. Carrying out 32 multiplications in $\mathbb{F}_{2^{32}}$, however, takes $2849 > 881 + 1652$ cycles.

Figure 4 shows the stride of the butterflies decreases by a factor of 2 in each stage. For the first few stages, the strides are always greater than 32, so 32 butterflies can be carried out in parallel with bitslicing. However, in the last 5 stages, there are stride-$16, 8, 4, 2, 1$ butterflies, which make it hard to carry out 32 butterflies in parallel. The same issue has appeared in [13] regarding Beneš networks as Beneš networks have a similar structure as the butterflies. The author of [13] solve the issue by carrying out matrix transpositions on the data. We make use of the same idea: one can imagine that our implementation performs a matrix transposition on the data right before the last 5 stages. The matrix transposition turns stride-$16, 8, 4, 2, 1$ butterflies into butterflies with strides greater than or equal to 32, so that we are able to carry out 32 butterflies in parallel. At the end of the forward FAFFTs, the order of the field elements is changed because of the matrix transposition, but this does not affect the pointwise multiplication since the multiplication can be carried out in any order. Similarly, in the inverse FAFFT, one can imagine that our implementation performs a matrix transposition on the data right after the first 5 stages.

We note that instead of carrying out the whole matrix transpositions, we actually carry out only a part of the matrix transposition in each of the 5 stages. The part of the matrix transposition we carry out in a stage is just enough to makes it possible to carry out 32 butterflies in parallel in that stage. This is a different way of implementing the idea in [13].

Encoding Consider the input f' as a matrix $M \in \mathbb{F}_2^{32 \times n'}$ where the $M_{j,i} = c_{j \cdot n' + i}$. One way to understand the encoding is to consider it as multiplying M by a 32×32 invertible matrix (from the left), where the invertible matrix is defined by $v_{m/2}, \dots, v_{m/2-4}$. As multiplying by the invertible matrix is a linear map, we again make use of the circuit generator in [13] to optimize it. We also use the circuit generator to optimize the inverse of encoding, which is used in the inverse FAFFT.

Radix Conversions Our implementation follows the algorithm presented in [23], which seems to be implicitly shown in [18, Section IV]. As the implementation is quite straightforward, we do not describe how we implemented the algorithm in detail.

8 Experiment Results and Discussions

We chose the STM32F407 Discovery board to be the platform for benchmarking. It is a low-cost development board by STMicroelectronics featuring the STM32F407VG microcontroller that can be clocked with up to 168 MHz, and it has 192 KB of SRAM and 1 MB of flash memory. All implementations for M4 are benchmarked in the `pqm4` [21, 20] benchmarking framework on the development board with the compiler `arm-none-eabi-gcc-10.2.1`. Note that `pqm4` always sets the frequency to 24 MHz so that the CPU can be set to have zero wait states when fetching instructions from the flash memory.

All the Haswell cycle counts in this section, unless specified otherwise, are measured on one core of an Intel Xeon E3-1275 v3 CPU, with Turbo Boost and hyper-threading disabled.

8.1 Multiplications in $\mathcal{R}_z$

The following table shows the cycle counts for carrying out one circular shift in our Haswell and M4 implementations with the techniques in Section 3 and 4.

	Haswell	M4
`bikel1`	1112	24856
`bikel3`	2547	52722

The following table shows the numbers of cycles on the Haswell core to carry out a multiplication $\mathrm{Tr}(h_i) \cdot \mathrm{Lift}(s)$ in $\mathcal{R}_z$. Our code uses the techniques in Section 3 and 5.

	our code	`avx2`
`bikel1`	113822	202176
`bikel3`	369782	629952

The way the `avx2` implementation carries out circular shifts is similar to the approach in Section 4, because it also makes use of conditional moves. The conditional moves are realized by using the intrinsic `_mm256_blendv_epi8`.

The following table shows the numbers of cycles on the Cortex-M4 to carry out a multiplication $\mathrm{Tr}(h_i) \cdot \mathrm{Lift}(s)$ in $\mathcal{R}_z$. Our code uses the techniques in Section 4 and 5.

	our code	`portable`
`bikel1`	2195104	4346484
`bikel3`	6746105	13981293

We actually tried to use the techniques in Section 3 for our M4 implementation but found that the approach in Section 4 is still faster. We believe this is due to the fact that the size of YMM registers on Haswell is much larger than the size of general-purpose registers on the Cortex-M4, and that there are powerful intrinsics/instructions (as shown in [19]) that can be used to manipulate YMM registers so that even a shift of $s^{(0)} \in \{0, \ldots, 255\}$ bits can be carried out efficiently.

8.2 Multiplications in $\mathcal{R}$

The following table shows the Haswell cycles for a multiplication between two elements in $\mathcal{R}$. Our code uses techniques in Section 6. We actually tried to implement the FAFFT-based multiplication described in Section 7. Instead of bitslicing, our FAFFT-based implementation uses `pclmulqdq` for field multiplications. However, as shown in the last column of the table, this leads to a much slower implementation.

	our code	`avx2`	FAFFT
`bikel1`	22856	29169	66265
`bikel3`	68912	91641	136194

The following table shows the M4 cycles for a multiplication between two elements in $\mathcal{R}$ with respect to methods in Section 7 and from the BIKE team.

	our code	`portable`
`bikel1`	1320940	2897887
`bikel3`	2929293	9606051

We can further reduce the cost of FAFFT multiplication by reusing the intermediate data while multiplying same elements. As shown in Algorithm 1, there are many multiplications in $\mathcal{R}$ of the form $e_i h_i$ in the decoder. Since h_0 and h_1 do not change during decapsulation, our implementation computes two forward FAFFTs, one for h_0 and one for h_1, at the beginning of decapsulation. Then, each time we need to compute $e_i h_i$, one forward FFT can be saved. In this way, about 30% of the 1320940 cycles and 2929293 cycles can be saved for each $e_i h_i$.

It is mentioned in [12, Section 5.1] that one can use circular shifts to carry out each multiplication of $e_i h_i$: write h_i as $\sum_s x^{-s}$ and compute $e_i h_i$ as $\sum_s (x^{-s} e_i)$. The numbers in the first table of Section 8.1 show that the resulting cycle counts are expected to be

- $1112 \cdot 71 = 78952$ cycles for `bikel1` on Haswell,
- $2547 \cdot 103 = 262341$ cycles for `bikel3` on Haswell,
- $24856 \cdot 71 = 1764776$ cycles for `bikel1` on M4,
- $52722 \cdot 103 = 5430366$ cycle for `bikel3` on M4.

8.3 Key Generation, Encapsulation, and Decapsulation

The Haswell cycles for key generation, encapsulation, and decapsulation are shown in the following table. The cycle counts of the `avx` implementation are from [8], and they are measured on a machine `titan0` with a Haswell CPU.

	key gen.	encap.	decap.	imple.
`bikel1`	833968	131276	2636108	`avx2`
	688372	124892	1900908	our code
`bikel3`	2515016	313976	9040604	`avx2`
	1857256	283932	6010004	our code

The M4 cycles for key generation, encapsulation, and decapsulation are shown in the following table.

	key gen.	encap.	decap.	imple.
`bikel1`	65414337	4824059	114592442	`portable`
	24935033	3253379	49911673	our code
`bikel3`	212999628	15041356	374777003	`portable`
	59820502	8376212	139234176	our code

Acknowledgements

The work of Ming-Shing Chen was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. The work of Tung Chou was supported by Taiwan Ministry of Science and Technology (MOST) Grant 109-2222-E-001-001-MY3. The work of Markus Krausz was funded by the German Federal Ministry of Education and Research (BMBF) under the project "QuantumRISC" (ID 16KIS1038) [26] and project "PQC4MED" (ID 16KIS1044).

References

[1] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic (HQC), 2017. https://pqc-hqc.org/.

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the NIST post-quantum cryptography standardization process, 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf.

[3] Martin Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece, 2017. https://classic.mceliece.org/.

[4] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE – bit flipping key encapsulation, 2017. https://bikesuite.org/.

[5] Daniel J. Bernstein. Batch binary edwards. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009. http://binary.cr.yp.to/bbe-20090604.pdf.

[6] Daniel J. Bernstein. Optimizing linear maps modulo 2, 2009. https://binary.cr.yp.to/linearmod2-20091005.pdf.

[7] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field MACs. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014. https://doi.org/10.1007/978-3-319-13051-4_6.

[8] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems. Accessed Dec. 23, 2020. https://bench.cr.yp.to.

[9] Joan Boyar and René Peralta. The exact multiplicative complexity of the hamming weight function. In *Electronic Colloquium on Computational Complexity (ECCC'05),(049)*, 2005. https://cpsc.yale.edu/sites/default/files/files/tr1260.pdf.

[10] David G. Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, March 1989. http://dx.doi.org/10.1016/0097-3165(89)90020-4.

[11] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean polynomials with Frobenius partitions in additive fast Fourier transform. http://arxiv.org/abs/1803.11301.

[12] Tung Chou. QcBits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer, 2016. https://eprint.iacr.org/2019/150.

[13] Tung Chou. Mcbits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017. https://doi.org/10.1007/978-3-319-66787-4_11.

[14] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *Journal of Cryptographic Engineering*, 9(4):341–357, 2019. https://eprint.iacr.org/2017/1251.pdf.

[15] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum QC-MDPC cryptography. *IACR Cryptology ePrint Archive*, 2020:298, 2020. https://eprint.iacr.org/2020/298.pdf.

[16] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In *International Conference on Post-Quantum Cryptography*, pages 35–50. Springer, 2020. https://eprint.iacr.org/2019/1423.

[17] Caxton C. Foster and Fred D. Stockton. Counting responders in an associative memory. *IEEE Transactions on Computers*, 100(12):1580–1583, 1971.

[18] Shuhong Gao and Todd Mateer. Additive fast Fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, December 2010. http://dx.doi.org/10.1109/TIT.2010.2079016.

[19] Antonio Guimarães, Diego F Aranha, and Edson Borin. Optimized implementation of QC-MDPC code-based cryptography. *Concurrency and Computation: Practice and Experience*, 31(18):e5089, 2019.

[20] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[21] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. *IACR Cryptology ePrint Archive*, 2019:844, 2019. https://eprint.iacr.org/2019/844.

[22] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast fourier transform. In Manuel Kauers, Alexey Ovchinnikov, and Éric Schost, editors, *Proceedings of the 2018 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2018, New York, NY, USA, July 16-19, 2018*, pages 263–270. ACM, 2018. https://doi.org/10.1145/3208976.3208998.

[23] Sian-Jheng Lin, Tareq Y. Al-Naffouri, and Yunghsiang S. Han. FFT algorithm for binary extension finite fields and its application to Reed–Solomon codes. *IEEE Transactions on Information Theory*, 62(10):5343–5358, October 2016. https://doi.org/10.1109/TIT.2016.2600417.

[24] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto. MDPC-McEliece: new McEliece variants from moderate density parity-check codes. In *2013 IEEE international symposium on information theory*, pages 2069–2073. IEEE, 2013. https://eprint.iacr.org/2012/409.pdf.

[25] Christof Paar. Optimized arithmetic for reed-solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*, page 250. IEEE, 1997. https://www.ei.ruhr-uni-bochum.de/media/crypto/veroeffentlichungen/2011/01/19/cnst.ps.

[26] QuantumRISC. Quantumrisc — next generation cryptography for embedded systems, 2020.

A A Loop with Conditional Shift in Our M4 Implementation

Note that the selection instruction `SEL` depends on the `GE` flag, which is not affected by the `TEQ` instruction that is used to control the loop.

```
lsr Ry0, s, #13
and Ry0, #1
mov Rx0, #983040
mul Ry0, Rx0
msr APSR_g, Ry0
add ptr, ptr_a, #-16
add ptr_end, ptr, #1024
loop256_a:
  ldr Rx0, [ptr, #16]!
  ldr Rx1, [ptr, #4]
  ldr Rx2, [ptr, #8]
  ldr Rx3, [ptr, #12]
    ldr Ry0, [ptr, #1024]
    ldr Ry1, [ptr, #1028]
    ldr Ry2, [ptr, #1032]
    ldr Ry3, [ptr, #1036]
```

```
      sel Rx0, Ry0, Rx0
      sel Rx1, Ry1, Rx1
      sel Rx2, Ry2, Rx2
      sel Rx3, Ry3, Rx3
      str Rx0, [ptr, #0]
      str Rx1, [ptr, #4]
      str Rx2, [ptr, #8]
      str Rx3, [ptr, #12]
      ldr Rx0, [ptr, #2048]
      ldr Rx1, [ptr, #2052]
      ldr Rx2, [ptr, #2056]
      ldr Rx3, [ptr, #2060]
      sel Ry0, Rx0, Ry0
      sel Ry1, Rx1, Ry1
      sel Ry2, Rx2, Ry2
      sel Ry3, Rx3, Ry3
      str Ry0, [ptr, #1024]
      str Ry1, [ptr, #1028]
      str Ry2, [ptr, #1032]
      str Ry3, [ptr, #1036]
    teq ptr_end, ptr
    bne loop256_a
```

B Python Code for the Boyar–Peralta Algorithm

```
count = 0
index = []

def nbits(n):

  ret = 0;
  while n > 0:
    ret += 1
    n >>= 1

  return ret

def find(n):

  exp = 0
  while True:
    if 2**exp-1 > n:
      break

    exp += 1

  return 2**(exp-1)-1

def bp(n, off):

  global count
  global index

  if n == 1:
```

```
    print ("rotate_right(&bn, f, s[{0}]);".format(count))
    count += 1
    print ("adder_size_eq(buf, bn, {0}, 0);".format(off))
    index.append(off + 0)
    return

  n1 = find(n-1)
  n2 = n-1-n1

  if n1 >= 1: bp(n1, off)
  if n2 >= 1: bp(n2, off + nbits(n1))

  print ("rotate_right(&bn, f, s[{0}]);".format(count))
  count += 1
  if n1 == n2:
    print ("adder_size_eq(buf, bn, {0}, {1});".format(off, nbits(n1)))
    index.append(off + 2*nbits(n1) - 1);
  else:
    print ("adder_size_neq(buf, bn, {0}, {1}, {2});".format(off, nbits(n1), nbits(n2)))
    index.append(off + nbits(n1) + nbits(n2) - 1);
```