Kyber on ARM64: Compact Implementations of Kyber on 64-bit ARM Cortex-A Processors

Pakize Sanal[1], Emrah Karagoz[1], Hwajeong Seo[2], Reza Azarderakhsh[1] and Mehran Mozaffari-Kermani[3]

[1] Florida Atlantic University, Boca Raton, USA, {psanal2018,ekaragoz2017,razarderakhsh}@fau.edu
[2] Hansung University, Seoul, South Korea, hwajeong84@gmail.com
[3] University of South Florida, Tampa, USA, mehran2@usf.edu

Abstract. Public-key cryptography based on the lattice problem is efficient and believed to be secure in a post-quantum era. In this paper, we introduce carefully-optimized implementations of Kyber encryption schemes for 64-bit ARM Cortex-A processors. Our research contribution includes several optimizations for Number Theoretic Transform (NTT), noise sampling, and AES accelerator based symmetric function implementations. The proposed Kyber512 implementation on ARM64 improved previous works by 1.72×, 1.88×, and 2.29× for key generation, encapsulation, and decapsulation, respectively. Moreover, by using AES accelerator in the proposed Kyber512-90s implementation, it is improved by 8.57×, 6.94×, and 8.26× for key generation, encapsulation, and decapsulation, respectively. These results set new speed records for Kyber encryption on 64-bit ARM Cortex-A processors.

Keywords: Post-quantum Cryptography · Kyber · ARM64 · Vectorized Implementation

1 Introduction

Conventional Public Key Cryptography (PKC) algorithms, such as RSA and Elliptic Curve Cryptography (ECC), are based on integer factorization and discrete logarithm problems. These hard problems have been believed to be secure against even high-end super computers. However, these hard problems can be easily solved by using Shor's algorithm on a quantum computer [Sho94]. For this reason, there are a demand for quantum-resistant algorithms to prepare the upcoming quantum era.

In 2016, NIST has initiated the post-quantum cryptography standardization process. In 2020, NIST announced third round finalists. Among them, lattice-based cryptography algorithms, such as Kyber, Dilithium, Falcon, and NTRU, were selected. Lattice based cryptography is often considered as a promising candidate since its security relied on worst-case computational assumptions in lattices that will remain hard even for quantum computers. Among finalists, Kyber algorithm based on Learning With Errors (LWE) is fast due to the small parameter size and the algorithm is relatively easier to implement than other complex problems. Recently, many works devoted to improve the performance of Kyber on microcontrollers. In Africacrypt'19, the memory efficient implementation of Kyber on Cortex-M4 was presented [BKS19]. They presented novel optimization techniques for the Number-Theoretic Transform (NTT) inside Kyber, which utilized the "vector" DSP instructions of ARM Cortex-M4 microcontrollers. In CHES'20, novel NTT implementation for Kyber scheme was presented [ABCG20]. The performance enhancement comes from efficient modular reductions, small polynomial multiplications, and more aggressive layer merging in the NTT.

However, the efficient implementation of Kyber on high-end ARM processor (i.e. ARMv8 Cortex-A) was not conducted. Since the high-end ARM is widely used in smartphone, smartwatch, and laptop computer, the efficient implementation should be highly considered. In this paper, we propose an optimized implementation of Kyber on 64-bit ARMv8 processors. Detailed contributions are as follows:

- **Optimized vectorized implementation of primitive operations of Kyber:** Primitive operations of Kyber are fully vectorized in ASIMD instructions of 64-bit ARMv8 processors. The proposed NTT implementation improved by $3.0 \sim 5.0\times$ and $4.0 \sim 6.0\times$ than previous works for reduction and NTT, respectively.
- **First optimized implementation of Kyber on 64-bit ARM processors:** With optimized primitive operations of Kyber, we implemented full parameters for Kyber schemes. The result shows that the Kyber512 implementation outperforms previous state-of-the-art by $1.72\times$, $1.88\times$, and $2.29\times$, for key generation, encapsulation, and decapsulation, respectively.
- **Acceleration of symmetric functions through cryptography extension:** Symmetric functions are core operations of Kyber scheme. The operation is accelerated with the cryptography extension of 64-bit ARMv8 processors. Results show that Kyber512-90s w/ accelerator is faster than w/o accelerator by $8.57\times$, $6.94\times$, and $8.26\times$, for key generation, encapsulation, and decapsulation, respectively.

The remainder of this paper is organized as follows. Section 2 presents an overview of the Kyber algorithm. In Section 3, we introduce the ARMv8-A architecture. In Section 4, proposed implementations of Kyber on 64-bit ARM Cortex-A processors are presented. In Section 5, the performance evaluation of proposed implementations is described. Finally, the conclusion is given in Section 6.

2 Kyber

Kyber is an IND-CCA2-secure key-encapsulation mechanism (KEM). It is a lattice-based algorithm and its security is based on the hardness of solving the learning-with-errors problem in module lattices (MLWE problem). It was first described in the paper "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM" in 2018 [BDK+18]. It is now a third round candidate in ongoing NIST competition. The details of the algorithm is given in its specification document [SAB+20].

2.1 Mathematical Background

The ring used in Kyber is $\mathbb{Z}_q[X]/(X^n+1)$, denoted by R_q, with $n = 256$ and $q = 3329$ in all variants of Kyber. The computations are performed by using Number-Theoretic Transform (NTT): for a polynomial $f = \sum_{i=0}^{255} f_i X^i \in R_q$ it is defined as

$$\mathsf{NTT}(f) = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

where

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(2i+1)j} \quad \text{and} \quad \hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2i+1)j}$$

with $\zeta = 17$ being the 256-th root of unity.

Two polynomials f and g in R_q can be efficiently multiplied by using NTT:

$$\mathsf{NTT}(f) \circ \mathsf{NTT}(g) = \hat{f} \circ \hat{g} = \hat{h}$$

where $\circ$ is the component-wise multiplication of linear polynomials, that is,

$$\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \mod (X^2 - \zeta^{2i+1})$$

for $i = 0, 1, \ldots, 127$. Then, the product of f and g is

$$fg = \mathsf{NTT}^{-1}(\hat{h}) = \mathsf{NTT}^{-1}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g)).$$

2.2 Compression and Encoding

An element $x \in \mathbb{Z}_q$ is converted to an d-bit integer by `Compress`$_q(x, d)$. An d-bit integer x is converted to a $\mathbb{Z}_q$ element by `Decompress`$_q(x, d)$. They are defined as follows:

$$\begin{aligned} \mathsf{Compress}_q(x, d) &= \lceil (2^d/q) \cdot x \rfloor \mod 2^d, \\ \mathsf{Decompress}_q(x, d) &= \lceil (q/2^d) \cdot x \rfloor \end{aligned}$$

where $\lceil a \rfloor$ is the closest integer to a. When each function is applied to a polynomial (or a vector/matrix of polynomials), it is applied to each coefficient individually.

Moreover, a polynomial (or a vector/matrix of polynomials) is serialized to byte arrays by using `Encode`$_\ell()$ function, where ℓ is the bit-length of each coefficient. On the other hand, `Decode`$_\ell()$ is the inverse of `Encode`$_\ell()$, and it deserializes the byte arrays to polynomials.

Lastly, `Parse()` converts a byte stream to the `NTT` representation of a polynomial in R_q.

2.3 Sampling

The noise is sampled from a centered binomial distribution B_η for $\eta = 2$ or $\eta = 3$. For a sample $(a_1, \ldots, a_\eta, b_1, \ldots, b_\eta) \leftarrow \{0, 1\}^{2\eta}$, the output is computed as

$$\sum_{i=1}^{\eta} (a_i - b_i).$$

The possible outputs are $\{-2, -1, 0, 1, 2\}$ when $\eta = 2$, and $\{-3, -2, -1, 0, 1, 2, 3\}$ when $\eta = 3$, respectively.

Using B_η, a polynomial $f = \sum_{i=0}^{255} f_i X^i$ in R_q can be sampled by sampling each coefficient f_i deterministically from 512η-bit output $(\beta_0, \ldots, \beta_{512\eta-1})$ of a pseudo-random function:

$$f_i = \sum_{j=0}^{\eta-1} (\beta_{2i\eta+j} - \beta_{2i\eta+j+\eta}) \quad i = 0, 1, \ldots, 255.$$

For this purpose, Kyber uses a function namely `CBD`$_\eta$, which takes 512η-bit input and outputs the corresponding polynomial.

2.4 Parameters

The fixed parameters are $n = 256$ and $q = 3329$. The parameter k represents the dimension of the matrix of polynomials in R_q. The parameter pair (η_1, η_2) are used in `CBD`$_\eta$ function for sampling. The parameter pair (d_u, d_v) is used in `Compress` and `Decompress` functions. The list of parameters are given in Table 1.

Table 1: Kyber parameters.

Algorithm	NIST-Level	n	q	k	(η_1, η_2)	(d_u, d_v)
KYBER512	1 (AES-128)	256	3329	2	(3,2)	(10,3)
KYBER768	3 (AES-192)	256	3329	3	(2,2)	(10,4)
KYBER1024	5 (AES-256)	256	3329	4	(2,2)	(11,5)

2.5 Symmetric Functions

Kyber makes a use of a pseudo-random function (PRF), an extendable output function (XOF), two hash functions H, and G, and a key-derivation function (KDF). These functions are specified in Table 2. At this point, Kyber has an alternative version Kyber-90s which uses SHA-2 hash functions and AES, while Kyber uses SHA-3 hash functions.

Table 2: Symmetric primitives in Kyber.

Symmetric Primitive	Kyber	Kyber-90s
XOF	SHAKE-128	AES-256 in CTR mode
H and G	SHA3-256 and SHA3-512	SHA-256 and SHA-512
PRF (s, b)	SHAKE-256($s\|\|b$)	AES-256 in CTR mode (key=s and nonce=b)
KDF	SHAKE-256	SHAKE-256

2.6 Kyber-PKE and Kyber-KEM

Kyber-PKE is an IND-CPA-secure public-key encryption scheme. It encrypts messages of a fixed length of 32 bytes. It contains three algorithms: Key Generation, Encryption, and Decryption.

In Kyber-PKE Key Generation, the polynomial matrix $\mathbf{A}$ is randomly generated, and the polynomial vectors $\mathbf{s}$ and $\mathbf{e}$ are sampled according to B_{η_1}. Then, normally, the secret key is $\mathbf{s}$ and the public key is $\mathbf{As} + \mathbf{e}$. However, for efficient implementation purposes, the multiplication $\mathbf{As}$ is performed in NTT domain by generating $\mathbf{A}$ in NTT domain (i.e. $\hat{\mathbf{A}}$) and transforming $\mathbf{s}$ to $\hat{\mathbf{s}} = \mathsf{NTT}(s)$. To avoid NTT^{-1} operation, $\mathbf{e}$ is also transformed to $\hat{\mathbf{e}}$ and added to $\hat{\mathbf{A}} \circ \hat{\mathbf{s}}$. Therefore, the values of secret and public keys are left in NTT domain and encoded to sk and pk, respectively. In addition, the seed for randomness is appended to the public key for letting the recipient generate the matrix $\mathbf{A}$.

In Kyber-PKE Encryption, the message m is encrypted to the ciphertext $c = (c_1, c_2)$ by using the public key pk and random coins r. The polynomial vector $\mathbf{t}$ and the matrix $\mathbf{A}$ are obtained using the public key. The polynomial vector $\mathbf{r}$ is sampled according to B_{η_1} using r. The polynomial vector $\mathbf{e}_1$ and the polynomial e_2 are sampled according to B_{η_2} using r. Then, normally, the ciphertext $c = (c_1, c_2)$ is $(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1, \mathbf{t}^T r + e_2 + m)$. However, multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}. Moreover, the ciphertext is compressed and encoded.

In Kyber-PKE Decryption, the polynomial vector $\mathbf{u}$ and the polynomial v are obtained from the ciphertext by decoding and decompressing. The vector $\mathbf{s}$ is obtained from the secret key. Then, the message m is $v - \mathbf{s}^T\mathbf{u}$. Again, the multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}.

Nonce values (which are 0 in the beginning of the algorithms) are incremental in each computation. Algorithms are given in Algorithm 1, 2, and 3.

On the other hand, Kyber-KEM is an IND-CCA2-secure KEM and it is constructed from Kyber-PKE using (a slightly tweaked) Fujisaki-Okamoto transform. It contains three

Algorithm 1: Kyber-PKE Key Generation

```
Output : secret key and public key pair $(pk,\ sk)$
1: $d \xleftarrow{\$} \{0,1\}^{256}$
2: $\rho, \sigma \leftarrow G(d)$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho, \texttt{nonce}))$
4: $\mathbf{s} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, \texttt{nonce}))$
5: $\mathbf{e} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, \texttt{nonce}))$
6: $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
7: $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
8: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
9: $pk \leftarrow \mathsf{Encode}_{12}(\hat{\mathbf{t}}) \| \rho$
10: $sk \leftarrow \mathsf{Encode}_{12}(\hat{\mathbf{s}})$
11: return $= (pk, sk)$
```

Algorithm 2: Kyber-PKE Encryption

```
Input  : public key $pk$, message $m$, random coins $r \in \{0,1\}^{256}$
Output : ciphertext $c = (c_1, c_2)$
1: $\hat{\mathbf{t}} \leftarrow \mathsf{Decode}_{12}(pk)$
2: $\rho \leftarrow pk$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho, \texttt{nonce}))$
4: $\mathbf{r} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, \texttt{nonce}))$
5: $\mathbf{e}_1 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, \texttt{nonce}))$
6: $e_2 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, \texttt{nonce}))$
7: $\hat{\mathbf{r}} \leftarrow \mathsf{NTT}(\mathbf{r})$
8: $\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
9: $v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$
10: $c_1 \leftarrow \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
11: $c_2 \leftarrow \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
12: return $c = (c_1, c_2)$
```

Algorithm 3: Kyber-PKE Decryption

```
Input  : secret key $sk$, ciphertext $c = (c_1, c_2)$
Output : message $m$
1: $\mathbf{u} \leftarrow \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c_1), d_u)$
2: $v \leftarrow \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c_2), d_v)$
3: $\hat{\mathbf{s}} \leftarrow \mathsf{Decode}_{12}(sk)$
4: $m \leftarrow \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1)$
5: return $m$
```

steps: Key Generation, Encapsulation, and Decapsulation. In the first step, Alice generates the public and secret keys by using Kyber-PKE Key Generation algorithm, and shares her public key with Bob. In the second step, Bob encrypts the message to the ciphertext by using Kyber-PKE Encryption algorithm, and sends the ciphertext to Alice. He also computes the shared secret by using the message, Alice's public key, and the ciphertext. In the last step, Alice decrypts the ciphertext to the message by using Kyber-PKE Decryption algorithm, and then verifies whether it can be encrypted to the same ciphertext (sent by Bob) by following similar steps as Bob did by using Kyber-PKE Encryption algorithm. If ciphertexts match, Alice computes the shared secret by using the message, her public key, and the ciphertext. Otherwise, she computes the shared secret by using a random value and the ciphertext. Details of Kyber-KEM are illustrated in Figure 1.

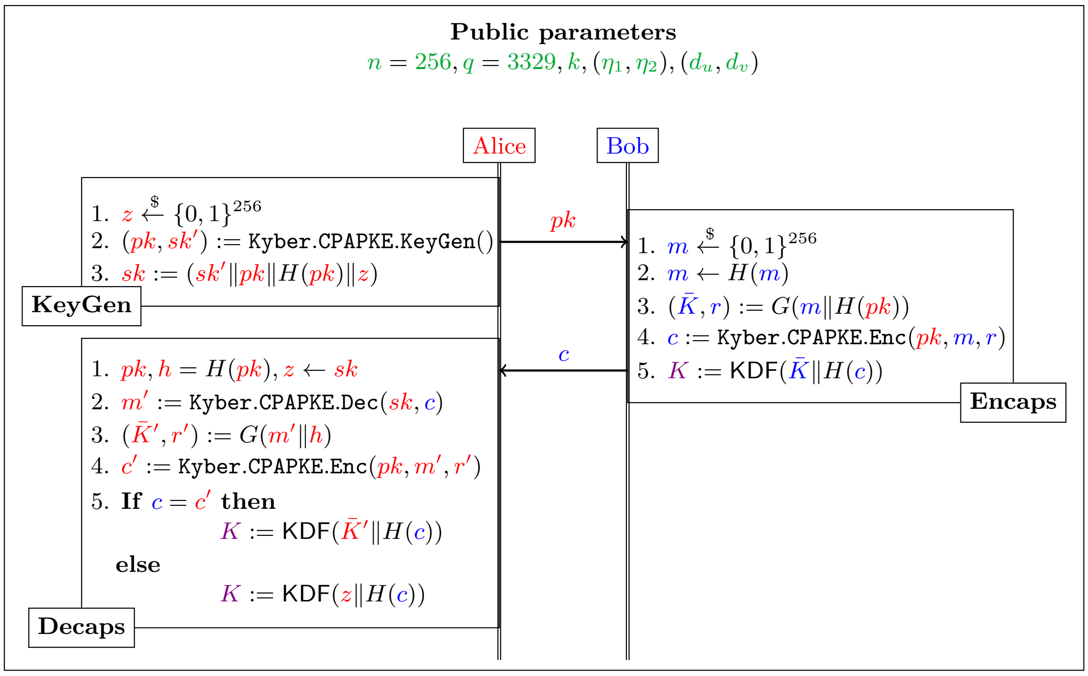
Public parameters
$n = 256, q = 3329, k, (\eta_1, \eta_2), (d_u, d_v)$
Alice
Bob
1. $z \xleftarrow{\$} \{0,1\}^{256}$
2. $(pk, sk') :=$ Kyber.CPAPKE.KeyGen()
3. $sk := (sk' \| pk \| H(pk) \| z)$
KeyGen
pk
1. $m \xleftarrow{\$} \{0,1\}^{256}$
2. $m \leftarrow H(m)$
3. $(\bar{K}, r) := G(m \| H(pk))$
4. $c :=$ Kyber.CPAPKE.Enc(pk, m, r)
5. $K :=$ KDF$(\bar{K} \| H(c))$
Encaps
c
1. $pk, h = H(pk), z \leftarrow sk$
2. $m' :=$ Kyber.CPAPKE.Dec(sk, c)
3. $(\bar{K}', r') := G(m' \| h)$
4. $c' :=$ Kyber.CPAPKE.Enc(pk, m', r')
5. If $c = c'$ then
$K :=$ KDF$(\bar{K}' \| H(c))$
else
$K :=$ KDF$(z \| H(c))$
Decaps

Figure 1: Kyber-CCA-KEM.

3 ARMv8-A Architecture

ARMv8-A is a 64-bit architecture. It provides 31 general purpose registers which can hold 32-bit values in registers `w0-w30` or 64-bit values in registers `x0-x30`. It provides SIMD (Single Instruction Multiple Data) instruction set, which can process 128 bit data per instruction on average. The SIMD vectorization is possible for same data per vector registers and it does not allow carry handling. There are 32 128-bit registers (`v0-v31`), which can be divided into lanes which are 8, 16, 32, or 64 bits wide. They are defined via operand suffix `b`, indicated byte, `h` indicates half-word, `s` indicates word, `d` indicated double-word. For instance, `v0.8h` create a vector with eight 16-bit elements. Single element of a vector can be accessed via square brackets (e.g. `v0.4s[0]` is the first 32-bit element of the vector `v0`, and `v1.8h[2]` is the third 16-bit element of the vector `v1`).

The ARMv8-A has a various SIMD instructions. Load and store operations are performed by using `LD` and `ST` operations. Each has 4 types according to the degree of interleaving: `LD1/ST1`, `LD2/ST2`, `LD3/ST3` and `LD4/ST4`. For example, `LD1` fills the vector `va` first, and continues to fill the vector `vb` later. However, `LD2` fills the vectors `va` and `vb` simultaneously, that is, one element for `va` and the next element for `vb`, another element for `va` again and so on. `LD3` follows a similar order for the vectors `va`, `vb` and `vc`. `ZIP1/ZIP2` zip two vectors into a single vector according to even/odd indices. `UZP1/UZP2` concatenate even or odd elements from two vectors. The `SXTL/SXTL2` instructions widens the lower/upper halfs of the source register (e.g. widens 8-bit elements to 16-bit elements). `TBL` is an instruction used for permutation according the indices given in a look-up table. `SSHR/USHR` performs vectorized signed/unsigned right shift operations. `AND/ORR` are bitwise and/or operations. `ADD/SUB` performs vectorized addition/subtraction. `MUL` performs vectorized multiplication restricted to the vector element size, however, `SMULL/SMULL2` perform the actual multiplication and widens the vector element. All of `MUL/SMULL/SMULL2` also support multiplication by a scalar, that is, all the vector elements are multiplied with a single scalar element. The details of ARMv8-A architecture can be found in [ARM].

4 Implementation Details

As mentioned in Section 2.1, Kyber performs its mathematical operations over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ and the elements are represented as polynomials over $\mathbb{Z}_q$ or vectors in NTT domain. However, both representations can be serialized as follows:

$$f = \sum_{i=0}^{255} f_i X^i \to (f_0, f_1, \dots, f_{255})$$

and

$$\hat{f} = \mathsf{NTT}(f) = (\hat{f}_0 + \hat{f}_1 X, \dots, \hat{f}_{254} + \hat{f}_{255} X) \to (\hat{f}_0, \dots, \hat{f}_{255}).$$

Considering both serialized vector representations of two polynomials f and g, addition and subtraction can be performed component-wise: $f_i \pm g_i$ or $\hat{f}_i \pm \hat{g}_i$. However, the multiplication is only performed in NTT domain (for efficiency) by multiplying component-wise pairs: $(\hat{f}_{2i}, \hat{f}_{2i+1}) \circ (\hat{g}_{2i}, \hat{g}_{2i+1})$. Modular reductions can also be performed component-wise: $f_i \mod q$, or $\hat{f}_i \mod q$.

In our implementation, the basic goal is to vectorize the input and take the advantage of SIMD operations on ARM. As $q = 3329$ is a 12-bit integer, the input values can be stored in 16-bit (or multiples of 16-bit). Later, 16-bit values are vectorized in `vx.8h` registers. In addition, 32-bit values are vectorized in `vx.4s` registers, if needed. Here, `x` is the vector index in $\{0, 1, \dots, 31\}$.

4.1 Reduction

We use the vectorized form of Barrett reduction as given in Listing 1. For a given 16-bit integer a, Barrett reduction computes the centered representative congruent to $a \bmod q$, that is, the unique integer x in the interval $\left[-\frac{q-1}{2}, ..., \frac{q-1}{2}\right]$ such that $x = a \bmod q$. Barrett reduction is used in `poly_reduce` function to compute modular reduction of polynomial coefficients in $\mathbb{Z}_q$. It uses a special constant value $r = \lfloor (2^{26} + \lfloor q/2 \rfloor)/q \rfloor)$, which is 20159 as $q = 3329$.

On the other hand, for a given 32-bit integer a, Montgomery reduction computes 16-bit integer congruent to $aR^{-1} \bmod q$, where $R = 2^{16}$, in the interval $[-q+1, \dots, q-1]$. We use the vectorized form of Montgomery reduction inplaced in `tomont` (see Listing 2) and `fqmul` (see Listing 3) functions. The `tomont` function performs the conversion of polynomial coefficients from normal domain to Montgomery domain by multiplying them with $t = 2^{32} \bmod q$ first and by applying the Montgomery reduction later. As $q = 3329$, the constant values in `tomont` are $q' = 62209 = q^{-1} \bmod 2^{16}$ and $t = 1353$. Moreover, the `fqmul` function performs the multiplication of two $\mathbb{Z}_q$-elements and then apply the Montgomery reduction. It uses the constant value q' as defined before.

In the comments of the listings, `MSB16` and `LSB16` refer to the most significant and the least significant 16-bit of a 32-bit integer, respectively.

Listing 1: `BARR`: Vectorized Barrett Reduction
$(r = \lfloor (2^{26} + \lfloor q/2 \rfloor)/q \rfloor)$

```
Input  : va.8h = [a_0, a_1, ..., a_7]                  (a_0, a_1, ..., a_7)
         vq.8h = [q, r, ...]
         (vb, vc, vd are intermediate vectors)
Output : va.8h = [a_0, a_1, ..., a_7]                  (a_0, a_1, ..., a_7)

SMULL   vb.4s, va.4h, vq.h[1]
SMULL2  vc.4s, va.8h, vq.h[1]
UZP2    vd.8h, vb.8h, vc.8h          ▷ $t \leftarrow$ (MSB16)$(r \cdot a)$
SSHR    vd.8h, vd.8h, 10             ▷ $t \leftarrow t \gg 10$
MLS     va.8h, vd.8h, vq.h[0]        ▷ $a \leftarrow a - q \cdot t$
```

Listing 2: `TOMONT`: Vectorized conversion of polynomial coefficients from normal domain to Montgomery domain ($q' = q^{-1} \bmod 2^{16}$ and $t = 2^{32} \bmod q$)

```
Input  : va.8h = [a_0, a_1, ..., a_7]                  (a_0, a_1, ..., a_7)
         vq.8h = [q, q', t, ...]
         (vb, vc, vd, ve, vf are intermediate vectors)
Output : va.8h = [a_0, a_1, ..., a_7]

SMULL   vb.4s, va.4h, vq.h[2]
SMULL2  vc.4s, va.8h, vq.h[2]
UZP1    vd.8h, vb.8h, vc.8h          ▷ $d \leftarrow$ (LSB16) $a \cdot t$
UZP2    ve.8h, vb.8h, vc.8h          ▷ $e \leftarrow$ (MSB16) $a \cdot t$
MUL     vf.8h, vd.8h, vq.h[1]        ▷ $f \leftarrow$ (LSB16) $q^{-1} \cdot d$
SMULL   vb.4s, vf.4h, vq.h[0]
SMULL2  vc.4s, vf.8h, vq.h[0]
UZP2    va.8h, vb.8h, vc.8h          ▷ $a \leftarrow$ (MSB16) $f \cdot q$
SUB     va.8h, ve.8h, va.8h          ▷ $a \leftarrow e - a$
```

4.2 NTT Operations

In NTT, the state-of-art computation is performed using Butterfly operations. As $n = 256 = 2^8$ in Kyber, the Butterfly operations are performed in 7 levels. In each level, the serialized representation $(f_0, f_1, \ldots, f_{255})$ are filled into the 8×16-bit vectors, and each two vectors (according to some distance in each level) are updated using Butterfly operation in NTT (see Listing 4). In the end, the its NTT representation (i.e. $(\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{255})$) is obtained. Vice versa, a vector in NTT domain can also be transformed to the normal domain by using the Butterfly operation in NTT^{-1} (see Listing 5).

4.3 Polynomial Multiplication

As mentioned in Section 2.1, two linear polynomials $a_0 + a_1X$ and $b_0 + b_1X$ are multiplied to compute their product $c_0 + c_1X$ in modulo $X^2 - \zeta_k$. For this purpose, we use the

Listing 3: `FQMUL`: Multiplication followed by Montgomery Reduction

```
Input  : va.8h = [a_0, a_1, ..., a_7]
         vb.8h = [b_0, b_1, ..., b_7]
         vq.8h = [q, q', ...]
         (vd, ve, vf, vg, vh are intermediate vectors)
Output : vc.8h = [c_0, c_1, ..., c_7]

SMULL   vd.4s, va.4h, vb.4h          ▷ (LO) $a \cdot b$
SMULL2  ve.4s, va.8h, vb.8h          ▷ (HI) $a \cdot b$
UZP1    vf.8h, vd.8h, ve.8h          ▷ $f$
UZP2    vg.8h, vd.8h, ve.8h          ▷ $g$
MUL     vh.8h, vf.8h, vq.h[1]        ▷ $u = q^{-1} \cdot e$
SMULL   vd.4s, vh.4h, vq.h[0]        ▷ (LO) $q \cdot g$
SMULL2  ve.4s, vh.8h, vq.h[0]        ▷ (HI) $q \cdot g$
UZP2    vc.8h, vd.8h, ve.8h          ▷ $t \leftarrow$ (MSB16) $q \cdot g$
SUB     vc.8h, vg.8h, vc.8h          ▷ $a \leftarrow a - t$
```

Listing 4: Butterfly operation in NTT

```
Input  : va.8h = [a_0, a_1, ..., a_7]      $(a_0, a_1, \ldots, a_7)$
         vb.8h = [b_0, b_1, ..., b_7]      $(b_0, b_1, \ldots, b_7)$
         vz.8h = [z_0, z_1, ..., z_7]      $(\zeta_0, \zeta_1, \ldots, \zeta_7)$
         (vc is an intermediate vector)
Output : va.8h = [a_0, a_1, ..., a_7]      $(a_0, a_1, \ldots, a_7)$
         vb.8h = [b_0, b_1, ..., b_7]      $(b_0, b_1, \ldots, b_7)$

FQMUL   vc.8h, vz.8h, vb.8h
SUB     vb.8h, va.8h, vc.8h          ▷ $b \leftarrow a - b \cdot \zeta$
ADD     va.8h, va.8h, vc.8h          ▷ $a \leftarrow a + b \cdot \zeta$
```

Listing 5: Butterfly operation in NTT^{-1}

```
Input  : va.8h = [a_0, a_1, ..., a_7]      $(a_0, a_1, \ldots, a_7)$
         vb.8h = [b_0, b_1, ..., b_7]      $(b_0, b_1, \ldots, b_7)$
         vz.8h = [z_0, z_1, ..., z_7]      $(\zeta_0, \zeta_1, \ldots, \zeta_7)$
         (vc is an intermediate vector)
Output : va.8h = [a_0, a_1, ..., a_7]      $(a_0, a_1, \ldots, a_7)$
         vb.8h = [b_0, b_1, ..., b_7]      $(b_0, b_1, \ldots, b_7)$

MOV     vc.16b, va.16b               ▷ $c \leftarrow a$
ADD     va.8h, vc.8h, vb.8h          ▷ $a \leftarrow b + c$
BARR    va.8h                        ▷ $a \leftarrow$ BarrettRed($a$)
SUB     vb.8h, vc.8h, vb.8h          ▷ $b \leftarrow b - c$
FQMUL   vb.8h, vz.8h, vb.8h          ▷ $b \leftarrow b \cdot \zeta$
```

BASEMUL function (see Listing 6) as the vectorized multiplication of two linear polynomials.

Listing 6: BASEMUL: Vectorized multiplication of two linear polynomials

```
Input  : va0.8h = [a0_0, a0_1, ..., a0_7]
         va1.8h = [a1_0, a1_1, ..., a1_7]                    $a_0 + a_1X$
         vb0.8h = [b0_0, b0_1, ..., b0_7]
         vb1.8h = [b1_0, b1_1, ..., b1_7]                    $b_0 + b_1X$
         vz.8h = [z_0, −z_0, ..., z_3, −z_3]                  $\zeta$ values
         (vd is an intermediate vector)
Output : vc0.8h = [c0_0, c0_1, ..., c0_7]
         vc1.8h = [c1_0, c1_1, ..., c1_7]                    $c_0 + c_1X$

1: FQMUL  vc0.8h, va1.8h, vb1.8h
2: FQMUL  vc0.8h, vc0.8h, vz.8h           ▷ $c_0 \leftarrow a_1 \cdot b_1 \cdot \zeta$
3: FQMUL  vd.8h, va0.8h, vb0.8h
4: ADD    vc0.8h, vc0.8h, vd.8h           ▷ $c_0 \leftarrow c_0 + a_0b_0$
5: FQMUL  vc1.8h, va0.8h, vb1.8h
6: FQMUL  vd.8h, va1.8h, vb0.8h
7: ADD    vc1.8h, vc1.8h, vd.8h           ▷ $c_1 \leftarrow a_0 \cdot b_1 + a_1 \cdot b_0$
```

4.4 Polynomial Addition and Subtraction

Vectors are simply added or subtracted using ADD or SUB as many times as needed.

4.5 Noise Sampling

Vectors are sampled according to B_2 or B_3 since $\eta \in \{2, 3\}$. We use CBD2 (see Listing 7) and CBD3 (see Listing 8) when $\eta = 2$ and $\eta = 3$, respectively. We initialize some vector registers (as many as needed) in the beginning: the vectors vmi are used for masking and the vector vs is used for shuffling. As mentioned in Section 2.3, every 4 bits (resp. 6 bits) produce an output when $\eta = 2$ (resp. $\eta = 3$). Therefore, CBD2 takes a 128-bit input and produces 32 output values (which are stored in vc0.8h, vc1.8h, vc2.8h and vc3.8h). Similarly, CBD3 takes a 96-bit input and produces 16 output values (which are stored in vc0.8h and vc1.8h). For these functions, we mainly followed the steps in Kyber's AVX implementation given in [BDK+].

4.6 Symmetric Functions

As described in Table 2, symmetric functions, including SHAKE-128/256, SHA-256/512, SHA3-256/512, and AES-256, are required for symmetric primitives, such as XOF, H, G, PRF, and KDF. For hash functions, we utilized the implementation provided by PQClean [KRS+]. For the AES implementation, we utilized the AES accelerator in the target board. If the board does not support the AES accelerator, we utilized PQClean AES implementations.

5 Performance Results

Benchmark results were measured both ARM and Apple chips. The ARM board is on Google Pixel 3 Android smartphone. The processor (Snapdragon 845) on it has 8 cores including 4 of ARM Cortex-A53 (@1.77 GHz) and 4 of ARM Cortex-A75 (@2.8 GHz) based. Performance results are taken by using Cortex-A75 processor and on debug mode

Listing 7: `CBD2`: Vectorized noise sampling for $\eta = 2$

```
Input  : va.16b = [a_0, a_1, ..., a_15],              (input values)
         vm0.16b = [0x55, ..., 0x55],
         vm1.16b = [0x33, ..., 0x33],
         vm2.16b = [0x03, ..., 0x03],
         vm3.16b = [0x0F, ..., 0x0F]                  (masking)
         (vd, ve, vf are intermediate vectors)
Output : vc0.8h = [c0_0, c0_1, ..., c0_7]
         vc1.8h = [c1_0, c1_1, ..., c1_7]
         vc2.8h = [c2_0, c2_1, ..., c2_7]
         vc3.8h = [c3_0, c3_1, ..., c3_7]

USHR   vd.8h, va.8h, 1
AND    va.16b, va.16b, vm0.16b
AND    vd.16b, vd.16b, vm0.16b
ADD    va.16b, va.16b, vd.16b
USHR   vd.8h, va.8h, 2
AND    va.16b, va.16b, vm1.16b
AND    vd.16b, vd.16b, vm1.16b
ADD    va.16b, va.16b, vm1.16b
SUB    va.16b, va.16b, vd.16b
USHR   vd.8h, va.8h, 4
AND    va.16b, va.16b, vm3.16b
AND    vd.16b, vd.16b, vm3.16b
SUB    va.16b, va.16b, vm2.16b
SUB    vd.16b, vd.16b, vm2.16b
ZIP1   ve.16b, va.16b, vd.16b
ZIP2   vf.16b, va.16b, vd.16b
SXTL   vc0.8h, ve.8b
SXTL2  vc1.8h, ve.16b
SXTL   vc2.8h, vf.8b
SXTL2  vc3.8h, vf.16b
```

Listing 8: CBD3: Vectorized noise sampling for $\eta = 3$

```
Input  : va.16b = [a_0, a_1, ..., a_7]
         vs.16b = [-1, 11, 10, 9, -1, 8, 7, 6,
                   -1, 5, 4, 3, -1, 2, 1, 0],                    (shuffle)
         vm0.4s = [0x00249249, ...],
         vm1.4s = [0x006DB6DB, ...]
         vm2.16b = [0x00000007, ...],
         vm3.16b = [0x00070000, ...],
         vm4.16b = [0x00030003, ...],                            (masking)
         (vd is an intermediate vectors)
Output : vc0.4s = [c0_0, c0_1, c0_2, c0_3]
         vc1.4s = [c1_0, c1_1, c1_2, c1_3]

TBL    va.16b, va.16b, vs.16b
USHR   vd.4s, va.4s, 1
USHR   vc0.4s, va.4s, 2
AND    va.16b, va.16b, vm0.16b
AND    vd.16b, vd.16b, vm0.16b
AND    vc0.16b, vc0.16b, vm0.16b
ADD    va.4s, va.4s, vd.4s
ADD    va.4s, va.4s, vc0.4s
USHR   vd.4s, va.4s, 3
ADD    va.4s, va.4s, vm1.4s
SUB    va.4s, va.4s, vd.4s
SHL    vd.4s, va.4s, 10
USHR   vc0.4s, va.4s, 12
USHR   vc1.4s, va.4s, 2
AND    va.16b, va.16b, vm2.16b
AND    vd.16b, vd.16b, vm3.16b
AND    vc0.16b, vc0.16b, vm2.16b
AND    vc1.16b, vc1.16b, vm3.16b
ADD    va.8h, va.8h, vd.8h
ADD    vd.8h, vc0.8h, vc1.8h
SUB    va.8h, va.8h, vm4.8h
SUB    vd.8h, vd.8h, vm4.8h
ZIP1   vc0.4s, va.4s, vd.4s
ZIP2   vc1.4s, va.4s, vd.4s
```

to get more accurate results. The executable is `aarch64` cross-compiled on Linux operating system (Ubuntu 20.04) with `gcc-9`.

The Apple board is on iPad mini 5-th generation. The processor (A12 Bionic) on it has 6 cores including 2 of Vortex (@2.49 GHz) and 4 of Tempest (@1.54 GHz) based. Performance results are taken by using Vortex processor on Apple operating system (iPadOS 14.3).

The code is originally taken from Kyber Round 3 submission [BDK+]. Then, cycle count function is changed as how is written in Microsoft's SIDH code [Mic]. The clock is set as `CLOCK_MONOTONIC` which gives more accurate results than `CLOCK_REALTIME`. Results shown in the Tables 3, 4, and 5 are median values for 1,000 tests.

The Table 3 shows reference and optimized implementation performance results for the arithmetic functions in Kyber. Notice that these results are same for all Kyber variants, because each Kyber variant has the same number of polynomial coefficients (e.g. $n = 256$). The overall performance results of key generation (K), encapsulation (E) and decapsulation (D) for all Kyber variants (including Kyber-90s) are presented in Tables 4 and 5. They show that the optimized implementation is ~2x faster than reference implementation even though the arithmetic functions are optimized ~5x faster. The main reason here is that the hashing operations mainly in the matrix generation part and in other various sums up to a big portion of the timing results as it is also indicated in the paper [ABCG20]. Detailed percentages of these functions are illustrated in the Figure 2.

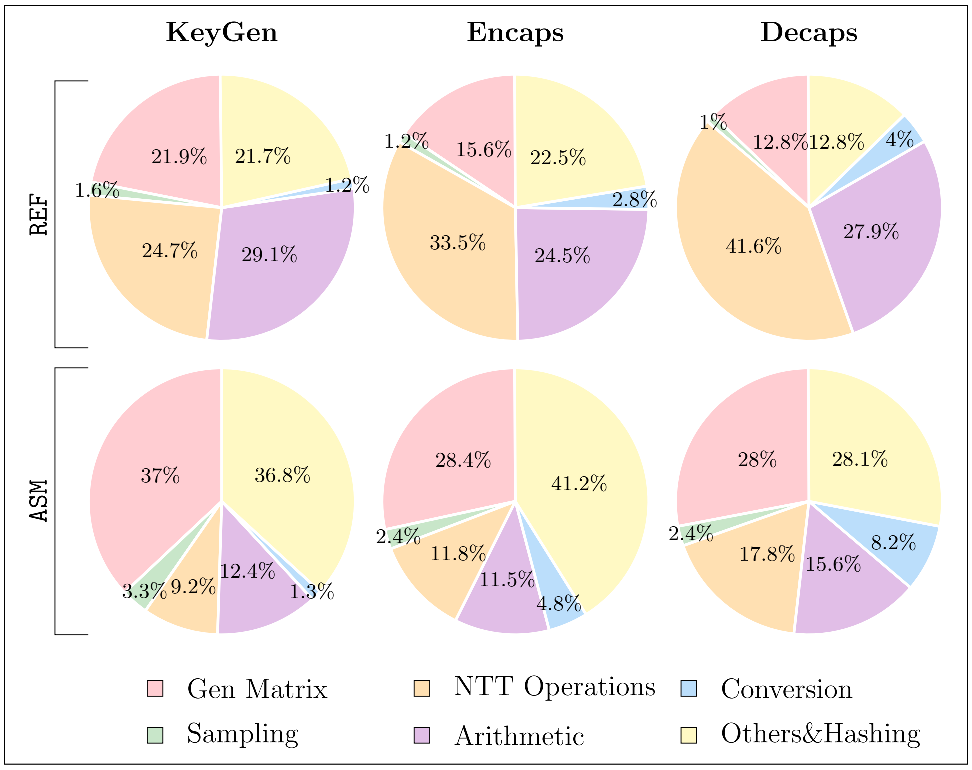

Figure 2: Percentages of used functions in Keypair, Encapsulation and Decapsulation

Table 3: Comparison of clock cycles for functions of Kyber schemes on 64-bit ARM Cortex-A75@2.8 GHz.

Functions	Timing [cc]		Ref [KRS+]/Opt
	Ref [KRS+]	Opt	
Reduction			
poly_tomont (Montgomery Red)	1,896	582	3.26
poly_reduce (Barrett Red)	2,187	437	5.00
NTT			
poly_ntt (NTT+Barrett Red)	11,228	2,332	4.81
poly_invntt_tomont (InvNTT)	17,500	3,209	5.45
poly_basemul_montgomery	5,396	1,313	4.11

Table 4: Comparison of clock cycles for Kyber schemes on 64-bit ARM Cortex-A75@2.8 GHz w/o AES accelerator.

Schemes		Timing [cc]		Ref [KRS+]/Opt
		Ref [KRS+]	Opt	
Kyber512	K	145,687	84,728	1.72
	E	206,209	109,668	1.88
	D	249,082	108,646	2.29
Kyber768	K	247,769	143,791	1.72
	E	326,810	180,687	1.81
	D	381,794	179,085	2.13
Kyber1024	K	385,148	228,082	1.69
	E	475,563	272,418	1.75
	D	545,126	270,668	2.01
Kyber512-90s	K	270,522	208,541	1.30
	E	334,541	238,148	1.40
	D	375,082	234,354	1.60
Kyber768-90s	K	491,750	384,272	1.28
	E	581,437	432,541	1.34
	D	632,918	426,269	1.48
Kyber1024-90s	K	790,269	633,794	1.25
	E	897,313	687,898	1.30
	D	959,582	680,313	1.41

Table 5: Comparison of clock cycles for Kyber schemes on 64-bit Apple A12@2.49 GHz w/ AES accelerator.

Schemes		Timing [cc] Ref [KRS+]	Timing [cc] Opt	Ref [KRS+]/Opt
Kyber512	K	60,370	34,932	1.78
	E	77,684	37,673	2.06
	D	94,623	37,259	2.53
Kyber768	K	106,028	62,206	1.70
	E	131,897	60,844	2.16
	D	146,694	59,996	2.44
Kyber1024	K	171,238	95,296	1.79
	E	182,220	93,004	1.95
	D	209,122	91,025	2.29
Kyber512-90s	K	279,751	32,640	8.57
	E	292,742	42,158	6.94
	D	305,511	36,982	8.26
Kyber768-90s	K	554,264	56,425	9.82
	E	576,012	64,560	8.92
	D	590,746	57,033	10.35
Kyber1024-90s	K	941,916	87,146	10.80
	E	964,815	93,820	10.28
	D	983,081	83,568	11.76

5.1 Cryptography Extension for Kyber–90s

64-bit ARMv8 Cortex-A processor supports cryptography extension, which accelerates AES encryption, SHA-1, SHA-224, and SHA-256[1]. In CT-RSA'15, compact implementations of AES-GCM were presented [GL15]. They utilized new cryptography instructions including 64-bit polynomial multiplication (e.g. `PMULL` and `PMULL2`) and AES operations (e.g. `AESE` (AddRoundKey, SubBytes, and ShiftRows) and `AESMC` (MixColumns)) for high-performance. In PQCrypto'18, SPHINCS with different cryptographic hash functions on ARMv8-A platform was presented [Köl18]. The implementation of SHA256 is optimized with cryptography extension (`SHA256H`, `SHA256H2`, `SHA256SU0`, and `SHA256SU1`). HARAKA implementation is optimized with AES extension. These dedicated instruction sets are also beneficial for a variant of Kyber, namely Kyber-90s, suggested by Kyber team. This new scheme utilizes AES-256 in counter mode and SHA2 instead of SHAKE. Kyber512-90s can be further optimized with AES-256 accelerator. We evaluated Kyber512-90s on 64-bit Apple A12 processors@2.49 GHz. Reference implementations require 279,751, 292,742, and 305,511 clock cycles for key generation, encryption, and decryption while optimized implementations with ARM64 assembly and AES-256 accelerator require 32,640, 42,158, and 36,982 clock cycles for key generation, encryption, and decryption, respectively. The implementation with accelerator shows 8.57×, 6.94×, and 8.26× faster than the implementation without the AES accelerator.

6 Conclusion

This paper presented several optimization techniques to efficiently implement Kyber-KEM on 64-bit ARM processors. We proposed optimizations for primitive operations of Kyber and symmetric functions to accelerate the execution time. A combination of these optimizations achieved 1.72×, 1.88×, and 2.29× faster than previous Kyber512

[1]Recent ARM architecture even supports SHA-3, SHA-512, SM3, and SM4 functions.

implementations for key generation, encapsulation, and decapsulation, which set new speed records for Kyber-KEM on an 64-bit ARM processor.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Cryptol. ePrint Arch.*, 2020:12, 2020.

[ARM] ARM. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/fc/. Accessed: 2021-01-15.

[BDK+] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Kyber project. https://github.com/pq-crystals/kyber. Accessed: 2020-12-12.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BKS19] Leon Botros, Matthias J Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *International Conference on Cryptology in Africa*, pages 209–228. Springer, 2019.

[GL15] Conrado PL Gouvêa and Julio López. Implementing GCM on ARMv8. In *Cryptographers' Track at the RSA Conference*, pages 167–180. Springer, 2015.

[Köl18] Stefan Kölbl. Putting wings on SPHINCS. In *International Conference on Post-Quantum Cryptography*, pages 205–226. Springer, 2018.

[KRS+] MJ Kannwischer, J Rijneveld, P Schwabe, D Stebila, and T Wiggers. The PQClean project. https://github.com/PQClean/PQClean. Accessed: 2020-12-10.

[Mic] Microsoft. PQCrypto-SIDH project. https://github.com/microsoft/PQCrypto-SIDH. Accessed: 2020-12-13.

[SAB+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE, 1994.