Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4

Leon Botros, Matthias J. Kannwischer, and Peter Schwabe*

Radboud University, Nijmegen, The Netherlands
l.botros@student.ru.nl, matthias@kannwischer.eu, peter@cryptojedi.org

Abstract. This paper presents an optimized software implementation of the module-lattice-based key-encapsulation mechanism Kyber for the ARM Cortex-M4 microcontroller. Kyber is one of the round-2 candidates in the NIST post-quantum project. In the center of our work are novel optimization techniques for the number-theoretic transform (NTT) inside Kyber, which make very efficient use of the computational power offered by the "vector" DSP instructions of the target architecture. We also present results for the recently updated parameter sets of Kyber which equally benefit from our optimizations.
As a result of our efforts we present software that is 18% faster than an earlier implementation of Kyber optimized for the Cortex-M4 by the Kyber submitters. Our NTT is more than twice as fast as the NTT in that software. Our software runs at about the same speed as the latest *speed-optimized* implementation of the other module-lattice based round-2 NIST PQC candidate Saber. However, for our Kyber software, this performance is achieved with a much smaller RAM footprint. Kyber needs less than half of the RAM of what the considerably slower *RAM-optimized* version of Saber uses. Our software does not make use of any secret-dependent branches or memory access and thus offers state-of-the-art protection against timing attacks.

Keywords: ARM Cortex-M4, number-theoretic transform, lattice-based cryptography, Kyber

1 Introduction

In 2016, NIST issued a call for proposals of new post-quantum cryptographic schemes including digital signatures and key encapsulation schemes (KEM) for future standardization [26]. In late 2017, 69 different proposals were accepted for a first round of evaluation. On January 30, 2019, NIST announced the second-round candidates which include 17 KEMs and 9 signature schemes. The report accompanying NIST's decision [1] states that the main criteria of selection were cryptanalytic attacks and message sizes. Implementation characteristics such as speed, memory consumption, or code size on various platforms was

* This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE). Date: May 13, 2019

not the main reason for not selecting any of the schemes to the second round evaluation. However, NIST stated that *"performance will play a larger role in the second round"* which is estimated to last for at least one year. Since only minor tweaks to submitted schemes are allowed it is likely that, unless there are major cryptanalytic advances, implementation performance will be a main criterion for schemes being considered beyond round two.

While many first-round submissions already include an implementation optimized for large Intel processors, most do not come with optimized implementations for other platforms. Yet, some of the schemes have been optimized for ARM Cortex-M microcontrollers and also FPGAs. One particularly important target platform is the ARM Cortex-M4 since a variety of schemes have been optimized for it and NIST recommended it to submission teams. Implementations of NIST candidates optimized for the Cortex-M4 are collected in `pqm4` [21] which also provides a testing and benchmarking framework for fair comparison.

7 out of 17 round-two candidates[1] for key encapsulation are based on structured lattices and as such heavily rely on arithmetic in polynomial rings. Recent work [20] optimized multiplication on Cortex-M4 in the polynomial ring $\mathbb{Z}_{2^k}[X]/(f(X))$ using the decomposition algorithms of Karatsuba [22] and Toom-Cook [12,32]. Having fast arithmetic in $\mathbb{Z}_{2^k}[X]/(f(X))$ allows to speed up the two second-round candidates Saber and NTRU[2].

Even though multiplication in $\mathbb{Z}_{2^k}[X]/(f(X))$ can be fast for practical values of n and k, it comes at a major cost: Toom and Karatsuba require additional memory to store intermediate results. For the NTRU-HRSS-KEM parameters $n = 701$ and $k = 13$, [20] achieve the fasted multiplication using Toom-4 and 4 layers of Karatsuba, which requires 11 208 bytes of additional stack space. Even for the smaller polynomials with $n = 256$ in Saber, the fastest multiplication routine described in [20] requires 3800 bytes of RAM. In case this memory is not available, one has to fall back to considerably slower multiplication algorithms.

The situation is very different for Kyber (and also the round-2 NIST candidate NewHope [2,3]), which are designed to support very efficient multiplication in the underlying polynomial ring *without* additional memory. The idea is to use fast number-theoretic transforms (NTTs), which are even part of the specification of these two schemes. The use of fast NTT-based multiplication is not new in those schemes and there exists a large body of work on optimizing this operation on a variety of platforms. The most recent works on optimizing the NTT on large Intel and AMD processors are by Seiler [30] and by Lyubashevsky and Seiler [24]. The fastest implementation so far on our target architecture, the ARM Cortex-M4, is presented by Alkım, Jakubeit, and Schwabe in [4]. Earlier works on the same architecture include [11] and [27].

Contribution. The main contribution of this paper is to present improved optimization techniques for the NTTs in Kyber. In comparison to the performance presented in [4], our NTT is more than a factor of 1.8 faster (when applying the same scaling to accommodate for the different dimension that was also used

[1] see https://www.safecrypto.eu/pqclounge/round-2-candidates/

[2] the second round merger of NTRU-HRSS-KEM [19] and NTRUEncrypt [33]

in [4]). Most of the techniques we present also apply to the NewHope parameters targeted in [4], but some of the speedup we achieve is specific to the smaller value of $q = 7681$ (NewHope uses $q = 12289$). We also optimize the other performance-critical routines in Kyber and describe how to reduce RAM usage in Kyber without significantly sacrificing performance. As a result we present the software, that at the same time has the smallest RAM footprint across all NIST PQC KEM candidates that have been optimized for the Cortex-M4, and has the lowest cycle count for the sum of key generation, encapsulation and decapsulation.

Kyber v2. While this paper was in submission, the Kyber team published various round-2 tweaks including the change of q from 7681 to 3329 which requires changing the NTT. All the optimizations presented in this paper still apply to Kyber v2. We have updated our software to support the new parameter sets and present the performance results for both versions.

Availability of software. We place all the software described in this paper into the public domain. It is available at `https://github.com/mupq/nttm4`. The implementations using the round-2 parameter sets have also been merged into `pqm4` [21].

Organization of this paper. Section 2 gives the necessary background on the key encapsulation scheme Kyber and the NTTs used within Kyber. Section 3 presents the speed optimizations we applied to the NTT which yields a significantly faster implementation of Kyber. Section 4 describes how the fast implementation of Kyber can be gradually modified to use less stack space with minor and moderate computational overhead. Finally, Section 5 presents the performance results for our implementations and compares them to previous implementations of Kyber and other second round candidates in the NIST post-quantum competition.

2 Preliminaries

In this section we establish notation, briefly recall Kyber and the NTT used within Kyber, and then proceed to describe our target platform, the ARM Cortex-M4.

Notation. We refer to polynomials by regular font lower-case letters (a), vectors of polynomials by bold lower-case letters ($\mathbf{a}$) and matrices of polynomials by bold upper-case letters ($\mathbf{A}$). For a polynomial a we use $\hat{a}$ to denote the representation of a in NTT-domain and similarly $\hat{\mathbf{a}}$ and $\hat{\mathbf{A}}$ are the results of element-wise application of the NTT to the entries of $\mathbf{a}$ and $\mathbf{A}$. (Random) bitstrings are referred to by the lower-case Greek letters ρ, σ, and μ. We abstract away from seed expansion to polynomials following a uniform or centered binomial distribution by just calling `SampleUniform` or `SampleCBD`. Let q be prime and let $\mathbb{Z}_q$ denote the field $\mathbb{Z}/q\mathbb{Z}$. We define polynomial rings of the form $\mathcal{R}_q = \mathbb{Z}_q/(X^n + 1)$ over this field where n is a power of two. We denote by $\circ$ the coefficient-wise multiplication of two polynomials in NTT domain with the natural extension to vectors and matrices. Similarly, let $c \in \mathcal{R}_q = \mathbf{a} \circ \mathbf{b}$ be the inner product of $\mathbf{a} \in \mathcal{R}_q^k$ and $\mathbf{b} \in \mathcal{R}_q^k$.

Algorithm 1 CPA KeyGen (v1)

Output: public key $pk = (\rho, \mathbf{t}')$
Output: secret key $sk = \hat{\mathbf{s}}$
1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$
2: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
3: $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\sigma)$
4: $\hat{\mathbf{s}} \leftarrow \texttt{NTT}(\mathbf{s})$
5: $\mathbf{t} \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{s}}) + \mathbf{e}$
6: **return** $pk = (\rho, \texttt{Compress}(\mathbf{t})), sk = \hat{\mathbf{s}}$

Algorithm 2 CPA Encryption (v1)

Input: public key $pk = (\rho, \mathbf{t}')$
Input: message $m \in \mathcal{R}_q$
Input: randomness $\mu \in \{0,1\}^{256}$
Output: ciphertext $(\mathbf{u}', v')$
1: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
2: $\mathbf{r}, \mathbf{e_1} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\mu)$
3: $e_2 \in \mathcal{R}_q \leftarrow \texttt{SampleCBD}(\mu)$
4: $\hat{\mathbf{r}} \leftarrow \texttt{NTT}(\mathbf{r})$
5: $\mathbf{u} \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
6: $\mathbf{t} \leftarrow \texttt{Decompress}(\mathbf{t}')$
7: $v \leftarrow \texttt{NTT}^{-1}(\texttt{NTT}(\mathbf{t})^T \circ \hat{\mathbf{r}}) + e_2 + m$
8: **return** $(\texttt{Compress}(\mathbf{u}), \texttt{Compress}(v))$

Algorithm 3 CPA Decryption (v1)

Input: secret key $sk = \hat{\mathbf{s}}$
Input: compressed ciphertext $(\mathbf{u}', v')$
Output: message $m \in \mathcal{R}_q$
$\mathbf{u} \leftarrow \texttt{Decompress}(\mathbf{u}')$
$v \leftarrow \texttt{Decompress}(v')$
return $m \leftarrow v - \texttt{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \texttt{NTT}(\mathbf{u}))$

2.1 Kyber v1

Kyber [6, 9], which is part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), is built on the hardness of the Module-LWE (MLWE) problem. Different from Ring-LWE, MLWE uses a matrix of polynomials in $\mathcal{R}_q$ as the public information $\hat{\mathbf{A}}$, whereas $\mathbf{s}$ and $\mathbf{e}$ become vectors of polynomials. For Kyber $\hat{\mathbf{A}}$ is a square $k \times k$ matrix and $\mathbf{s}$ and $\mathbf{e}$ are k-dimensional vectors. MLWE therefore presents a generalization of the Ring-LWE and the standard LWE problem. While this might have benefits in terms of security [9], it is also an advantage for implementations: One can change the security level by changing the dimension of the matrix, i.e., by changing k. Kyber uses the prime $q = 7681 = 2^{13} - 2^9 + 1$ and $\mathcal{R}_q = \mathbb{Z}_{7681}/(X^{256} + 1)$ for all security levels. Since $\mathcal{R}_q$ remains the same for all security levels it is possible to optimize all security levels of Kyber by optimizing arithmetic in $\mathcal{R}_q$. Kyber specifies three security levels: Kyber-512, Kyber-768, and Kyber-1024 which use $k = 2, 3, 4$, respectively. Besides k, the security levels only differ in the centered binomial distribution of the secret and error polynomials which is $\eta = 5, 4, 3$ respectively.

Kyber uses a two stage-construction to obtain a CCA-secure KEM: First, build an IND-CPA secure encryption scheme, which is called Kyber.CPA and then use a variant of the Fujisaki-Okamoto transform [15] to build the CCA-secure KEM. Algorithms 1, 2, and 3 illustrate key-generation, encryption, and decryption of the CPA-secure encryption scheme. For the details of the CCA transform, we refer the reader to [6, Alg. 7–9] for the pseudocode description. Since the public matrix $\mathbf{A}$ is sampled from a uniform distribution and since the number-theoretic transform of uniform randomness is again uniformly distributed, the NTT of $\mathbf{A}$

is omitted and $\hat{\mathbf{A}}$ is instead sampled directly in NTT domain. However, this is not possible for the secrets and errors, since those need to be small in normal domain.

Aside from symmetric cryptography used for randomness generation and hashing (in particular in the CCA transform), the main cost in Kyber is arithmetic in $\mathcal{R}_q$ and even more specifically multiplications. The main cost of these multiplications are the (forward and inverse) NTT. The number of NTT operations depends on the parameter k and is $2k, 3k+1$, and $k+1$ for Kyber.CPA key generation, encryption, and decryption, respectively. Decapsulation of the CCA-secure KEM includes both Kyber.CPA encryption and Kyber.CPA decryption and thus requires $4k+2$ NTTs.

The Number Theoretic Transform. The number-theoretic transform is a Fourier transform in a finite field, i.e., a multi-point evaluation of a polynomial at powers of a root of unity. In the specific setting of Kyber, the NTT of a polynomial $g = \sum_{i=0}^{n-1} g_i X^i \in R_q$ is defined as

$$\texttt{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$
$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

where $\omega = 3844$ and $\psi = \sqrt{\omega} = 62$. The inverse of this operation is given through

$$\texttt{NTT}^{-1}(\hat{g}) = g = \sum_{i=0}^{n-1} g_i X^i, \text{ with}$$
$$g_i = n^{-1} \psi^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}.$$

With these definitions of $\texttt{NTT}$ and $\texttt{NTT}^{-1}$, the multiplication of two polynomials $f, g \in \mathcal{R}_q$ can be computed as $f \cdot g = \texttt{NTT}^{-1}(\texttt{NTT}(f) \circ \texttt{NTT}(g))$.

The FFT algorithm to compute Fourier transforms with only $\Theta(n \log n)$ operations was introduced by Cooley and Tukey in [13]; only several years later it was pointed out by Goldstine [16] that a similar algorithm had already been described by Gauss in the early 19th century. For a discussion also see [18]. The big picture is that the algorithm iterates through $\log_2 n$ levels, each level performs $n/2$ so-called *butterfly operations*, and each butterfly operation performs a multiplication by a power of ω, one addition, and one subtraction in $\mathbb{Z}_q$. The powers of the root of unity ω are often referred to as the "twiddle factors".

Note that in $\texttt{NTT}$ and $\texttt{NTT}^{-1}$, polynomials are transformed *inplace* and without any additional temporary storage. This comes at a small price: the coefficients of polynomials in NTT domain are in so-called bit-reversed order. This issue can either be addressed by permuting coefficients or by implementing separate algorithms for $\texttt{NTT}$ and $\texttt{NTT}^{-1}$, one that expects input in bitreversed order and

Algorithm 4 CPA KeyGen (v2)

Output: public key $pk = (\rho, \hat{\mathbf{t}})$
Output: secret key $sk = \hat{\mathbf{s}}$
1: $\rho, \sigma \xleftarrow{\$} \{0,1\}^{256} \times \{0,1\}^{256}$
2: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
3: $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\sigma)$
4: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \texttt{NTT}(\mathbf{s}) + \texttt{NTT}(\mathbf{e})$
5: **return** $pk = (\rho, \hat{\mathbf{t}}), sk = \hat{\mathbf{s}}$

Algorithm 5 CPA Encryption (v2)

Input: public key $pk = (\rho, \hat{\mathbf{t}})$
Input: message $m \in \mathcal{R}_q$
Input: randomness $\mu \in \{0,1\}^{256}$
Output: ciphertext $(\mathbf{u}', v')$
1: $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k} \leftarrow \texttt{SampleUniform}(\rho)$
2: $\mathbf{r}, \mathbf{e_1} \in \mathcal{R}_q^k \leftarrow \texttt{SampleCBD}(\mu)$
3: $e_2 \in \mathcal{R}_q \leftarrow \texttt{SampleCBD}(\mu)$
4: $\hat{\mathbf{r}} \leftarrow \texttt{NTT}(\mathbf{r})$
5: $\mathbf{u} \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
6: $v \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + m$
7: **return** $(\texttt{Compress}(\mathbf{u}), \texttt{Compress}(v))$

produces output in normal order and the other one working the other way round. Kyber follows the second approach, i.e., avoids overhead of extra bitreversal operations. For a discussion of the different options, see also [28, Sec. 3.2].

2.2 Kyber v2

In the process of writing this paper, the second round of NIST began and the Kyber team published an updated Kyber specification [7]. We will in the following refer to this updated version as Kyber v2.

The main design decision for round 2 of the NIST competition was to remove the compression of the public key. To compensate for the increased bandwidth requirement, the Kyber team decided to reduce the value of q from 7681 to 3329, a choice that was enabled by the observation from [24] that also this value of q supports very fast NTT-based multiplication of polynomials. Another consequence of the decision to not compress public keys is that public keys can now be transmitted in NTT domain, which saves an NTT operation in encryption (and in the re-encryption during decapsulation of the CCA-secure KEM). Finally, the smaller value of q also requires smaller noise to achieve the same security level. This is why the parameter η of the centered binomial distribution changed to $\eta = 2$ for all security levels; note that this change is hidden by our high-level view of `SampleCBD`. The resulting key-generation and encryption algorithms are given in Algorithm 4 and Algorithm 5; decapsulation is the same as for the round-1 version in this high-level perspective.

From a computational point of view, the most interesting aspect of the changes is the change of the definition of the NTT. In the round-1 version of Kyber, q was chosen such that $\mathbb{Z}_q$ contains 512-th roots of unity. As a consequence, the negacylic NTT of elements of $\mathcal{R}_q$ is a vector of 256 degree-zero polynomials (i.e., scalars). In the round-2 version of Kyber, q is chosen such that $\mathbb{Z}_q$ contains 256-th roots of unity, but not 512-th roots of unity. As a consequence, the NTT of a polynomial $f \in \mathcal{R}_q$ is a vector of 128 polynomials of degree at most 1, i.e., with

2 coefficients each. Specifically, [7, Sec. 1.1] defines the NTT of a polynomial $f \in \mathcal{R}_q$ as

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

where coefficients $\hat{f}_i$ are defined as

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2\mathsf{br}_7(i)+1)j}, \text{ and}$$
$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2\mathsf{br}_7(i)+1)j}.$$

In this definition $\zeta = 17$ is the first primitive 256th root of unity and br_7 reverses the bits in a 7-bit integer.

Note that with this definition of the NTT, the "pointwise" multiplication of two polynomials denoted by $\circ$ now consists of performing 128 multiplications of linear polynomials modulo $X^2 - \zeta^{2\mathsf{br}_7(i)+1}$.

2.3 ARM Cortex-M4

Our target platform is the ARM Cortex-M4, which NIST recommended as the reference platform for evaluation of post-quantum candidates on microcontrollers. It implements the `ARMv7E-M` instruction set which features 16 general purpose 32-bit registers of which 14 are usable by the developer; the other two are used for program counter and stack pointer. Unlike the `ARMv7-M`, the `ARMv7E-M` provides powerful DSP instructions that perform arithmetic operations on two 16-bit halfwords in parallel, which proved to be very beneficial for the other post-quantum KEMs Saber and NTRU-HRSS-KEM [20, 23]. While Kyber does not benefit from the `smlad` instruction, we make use of parallel additions and subtractions using `uadd16`, `usub16`, `sasx`, and `ssax`. Another feature that we extensively use throughout our optimization is the barrel-shifter which allows to shift or rotate one of the arguments in arithmetic instructions without increasing the cycle count.

Specifically, we use the STM32F4DISCOVERY that is also used by a large number of previous optimization papers and the benchmarking and testing framework `pqm4` [21]. It comes with 192 KiB of memory, 1 MiB of flash, and can operate at frequencies of up to 168 MHz. Compared to other ARM platforms like the Cortex-M0, our target platforms can be considered at the higher end of microcontrollers. The RAM and flash are sufficient to implement and evaluate almost all of the second round NIST candidates.

3 Optimizing for speed

In this section we describe the optimizations we apply to speed up the computation of Kyber on the ARM Cortex-M4. Optimizations targeting the reduction of RAM

usage will be presented in 4. The starting point of our optimization efforts is the optimized implementation for the Cortex-M4 by the Kyber authors [5], which is the same as the C reference implementation except for a hand-optimized NTT operation and which is included in the `pqm4` framework [21].

3.1 Link-time optimization

While experimenting with the Kyber implementation from [5], we realized that its performance is heavily penalized in `pqm4` because a number of small functions (in particular modular reductions) are implemented in different files than where they are used. Since `pqm4` compiles all source files separately to object files, the compiler cannot inline those functions, which creates a large overhead from function calls. A simple, but not very elegant solution would be to place all source code in one large file and this indeed results in a speedup of about 5%.

A similar behaviour can be achieved by adding the link-time optimization compiler flag `-flto`, which adds additional information in object files to allow optimization when those are linked together. Since `-flto` consistently improves performance for implementations of Kyber, we use it throughout our experiments.

We contacted the authors of `pqm4` [21] to include `-flto` as a default option. However, their benchmarks show that not all schemes benefit from `-flto`. Some schemes get significantly slower, while others have a up to 60% increase in stack consumption. Therefore, `-flto` was not turned on by default in `pqm4`.

3.2 Speeding up the NTT

In the following we describe our optimization strategy for the NTT, which includes a careful combination of known techniques with new micro-architecture specific improvements.

Representation of polynomials. Polynomials in $\mathcal{R}_q$ have 256 coefficients in $\mathbb{Z}_q$, where q is the 13-bit prime 7681 (or 3329 for Kyber v2). Is is natural to represent polynomials as an array of length 256 of 16-bit integers. Inspired by [30] and unlike the implementation by the Kyber authors or the optimized NewHope implementation described in [4], we use an array of *signed* 16-bit integers to represent elements of $\mathcal{R}_q$. We will later discuss the effect of this choice on modular reductions; one immediate advantage of using signed representation is that during subtractions in $\mathbb{Z}_q$ we do not have to worry about underflows. Compared to using unsigned integers we thus trivially save an addition of a multiple of q before subtractions.

Merging NTT layers. Similar to, e.g., [17] and [4], we merge several layers of the NTT transformation, i.e., we load four coefficients into registers at once, perform four butterfly operations on them, and store them back. This drastically reduces the number of loads and stores. However, it turns out that merging three layers of the NTT as proposed in [4] is not optimal, since there are not enough registers to fit the constants required in the Montgomery and Barrett reductions (see below). In [4] this is solved by reloading the constants for each butterfly, but

Algorithm 6 Original unsigned Montgomery reduction [5]; using Montgomery factor $\beta = 2^{18}$.

Input: a (32 bit)
Output: reduced a (16 bit)

1: `mul` t, a, q^{-1}
2: `and` t, #0x3ffff
3: `mla` a, t, q, a ▷ $a \leftarrow a + t \cdot q$
4: `lsr` a, #18

Algorithm 7 Signed Montgomery reduction (this work, adapted from [30]); using Montgomery factor $\beta = 2^{16}$.

Input: a (32 bit)
Output: reduced a (16 bit)

1: `smulbb` t, a, q^{-1} ▷ $t \leftarrow (a \mod \beta) \cdot q^{-1}$
2: `smulbb` t, t, q ▷ $t \leftarrow (t \mod \beta) \cdot q$
3: `usub16` a, a, t ▷ $a_{top} \leftarrow \left\lfloor \frac{a}{2^{16}} \right\rfloor - \left\lfloor \frac{t}{2^{16}} \right\rfloor$

the cost for these loads is larger than the savings from fewer loads and stores of coefficients. We instead merge only two layers which allows us to still keep all constants in registers and still save 50% of load and store operations.

Precomputation of twiddle factors. Like most speed-optimized NTT implementations before, we precompute all powers of ω and store those in flash. For more efficient modular reduction after multiplication by the twiddle factors, we follow an approach first introduced in [3] and store twiddle factors in Montgomery representation [25]. More specifically, our optimizations are largely inspired by the refined approach described in [30] and we use the same Montgomery factor $\beta = 2^{16}$. We then reorder the twiddle factors in our table such that they can be picked up sequentially in the NTT computation; increasing the pointer to the twiddle factors after each load is free in `ARMv7E-M`. Since we need three twiddle factors per two (merged) layers, we pack two of them into one register, which saves one load operation and one register. The twiddle factors are only used in multiplications with 16-bit coefficients which allows to use `smulbb` and `smulbt` to multiply by the upper or the lower twiddle factor inside that register.

Montgomery reductions. After the multiplication in each butterfly, we need to reduce the 32-bit product to 16-bit. This is done using a signed Montgomery reduction tailored to q. It turns out that the signed Montgomery reduction as proposed in [30] can be implemented in three clock cycles (Algorithm 7) on the ARM Cortex-M4 and as such is one clock cycle faster than the unsigned Montgomery reduction in [5] (Algorithm 6).

Unrolling. As usual we fully unroll the outer loop of the NTT iterating over the NTT levels. Additionally, to save an additional register, we unroll one of the inner loops as well. Depending on the current level, we unroll the loop with the least iterations to minimize the code-size increase. While this is also saving a small number of cycles, the performance gains by having an additional register are much more significant.

Packing. Since q is well below 16-bits, polynomials are usually stored as `int16_t` arrays. Since our target platform is a 32-bit architecture it seems wasteful to only load one 16-bit coefficient into 32-bit registers. Loading and storing two

coefficients at once saves half of the load and store operations. However, the available vector instruction in `ARMv7E-M` are quite limited. For example, there is no dedicated instruction performing two 16-bit multiplications yielding two 32-bit results. Still some operations can be performed in parallel. Therefore, we implement "double" butterflies, i.e., butterflies which operate on packed arguments and return a packed result. By doing this, we can for example perform two additions and subtractions in one clock cycles using `uadd16` and `usub16`. Unfortunately, some operations (e.g., the Barrett reduction) are more than twice as expensive to implement on packed arguments. Nonetheless, we achieve a speed-up in every butterfly by using packing.

Instruction Alignment. Since some instructions available in `ARMv7E-M` are 16-bit Thumb instructions, it is possible that a single Thumb instruction unaligns many following 32-bit ARM instructions which results in a vast performance penalty. Therefore, we make sure our code is as aligned as possible. This can be done by aligning the start of the function using `.align 2` (`.align n` aligns to 2^n bytes) and padding each sole Thumb instruction to 32-bit using the `.w` suffix.

Recent improvements proposed in [24]. Very recent work proposed yet another more efficient NTT in AVX2 [24] which can also be adapted to Kyber. The major speed-up that [24] achieved over [30] in the NTT stems form further optimizing the Montgomery reduction. Lyubashevsky and Seiler save an additional multiplication by avoiding the multiplication by q^{-1} and instead multiplying each of the precomputed twiddle factors by q^{-1}. This is possible since each product of a polynomial coefficient a_i by a twiddle factor is implemented through two separate multiplication instructions, one computing the low half and one computing the high half of the product. Since the low half of the product is multiplied by q^{-1} mod β inside the Montgomery reduction, one can precompute the product of q^{-1} and the corresponding twiddle factor and use this constant for the low product. This saves another multiplication instruction in the Montgomery reduction, but requires to store twice as many precomputed twiddles. Unfortunately, this does not carry over to our Cortex-M4 implementation since the low and high product are not computed separately, but in a single instruction. Doing these multiplications separately with different constants would be possible, but require an additional clock cycle and thus not save anything.

3.3 Optimizing matrix-vector multiplication

Besides the NTT, another fairly expensive operation in Kyber is the matrix-vector multiplication in line 5 of Algorithm 1 and line 5 of Algorithm 2. We also optimize this operation in C. Since this optimization depends on the stack-reduction strategy, we describe it in Section 4.

3.4 Optimized Keccak

As we will see in Subsection 5.3, even before our optimization of the NTT and matrix-vector multiplication, most of the cycles of the Kyber computation are

spent in hashing and pseudorandom-number generation, which both boil down to the Keccak permutation [8]. For all derivatives of Keccak inside Kyber (i.e., SHA3-256, SHA3-512, SHAKE-128, and SHAKE-256) we use the highly optimized code from the eXtended Keccak Code Package [14], which is also included in the `pqm4` framework.

3.5 Kyber v2

Various changes in the updated Kyber specification have an impact on performance, but all the optimizations presented above still apply with minor modifications: The smaller q allows to be more lazy with Barrett reductions in the `NTT` and $\texttt{NTT}^{-1}$ which improves performance. Additionally, both the `NTT` and $\texttt{NTT}^{-1}$ only require 7 instead of 8 layers of butterfly operations which saves roughly 1/8 of the cycles. However, the multiplication of polynomials in the NTT domain is no longer a pointwise multiplication and consequently becomes more expensive. These two changes approximately cancel each other out.

4 Decreasing Stack Usage

In addition to being fast, NTT-based multiplication provides the additional benefit of being entirely in-place; no additional stack space is needed. This presents a major advantage compared to for example $\mathbb{Z}_{2^k}[X]/(f(X))$, where the fastest multiplication methods use a combination of Toom-Cook [12, 32] and Karatsuba's [22] algorithm which comes with a rather large memory footprint. The existing implementation of the NTT in Kyber were already in-place and the changes we applied to them did not change this. Therefore, we also optimized the C-code implementing the remainder of the scheme to use less stack space, making this implementation of Kyber particularly suitable for memory constrained devices. We analyzed which stack space requirements can be eliminated at no or very little computational cost, i.e., without recomputations.

Changes to Kyber.CCAKEM. Kyber uses a FO-transformation to transform a CPA-secure PKE into a CCA-secure KEM. The reference implementation of decapsulation does so by first decrypting the ciphertext and then re-encrypting the obtained plaintext. This produces a ciphertext which is then compared to the original. Only if they are equal, the shared secret key is returned. We eliminate this additional ciphertext on the stack by inlining the comparison into CPA encryption in a constant-time manner. This function is only used for re-encrypting and does not return a ciphertext, but rather a boolean value that indicates the ciphertexts were equal. The actual re-encrypted ciphertext is computed and compared byte per byte. This not only saves a considerable amount of stack space, but also slightly improves the speed.

Changes to Kyber.CPAPKE. The remaining changes were made in the C code of Kyber's CPA key generation (Algorithm 1), encryption (Algorithm 2) and decryption (Algorithm 3), where we reduced the number of polynomials that

are kept in memory at the same time. In the reference implementation of key generation and encryption, firstly, the public matrix $\hat{\mathbf{A}}$ of $k \times k$ polynomials is sampled directly in NTT domain and stored in memory. Then, vectors of noise polynomials are sampled from a centered binomial distribution. Finally, all computations are performed. We optimize this by merging the sampling and the computations, i.e., we sample the required arguments on the fly where possible.

Generating and multiplying $\hat{\mathbf{A}}$. Since a polynomial in Kyber has 256 coefficients each represented by 16 bits, storing one polynomial consumes 512 bytes of memory. Because the size of the matrix $\hat{\mathbf{A}}$ grows quadratically with k, its k^2 polynomials account for the majority of Kyber's stack usage. However, the matrix $\hat{\mathbf{A}}$ is only required once for matrix-vector pointwise multiplication and accumulation (see e.g., line 4 of Algorithm 1). The memory footprint can be reduced using an approach that reduces the storage requirements of $\hat{\mathbf{A}}$ to only the state of the extendable output function for one polynomial of $\hat{\mathbf{A}}$ at a time, allowing to generate a small number of coefficients for multiplication.

In this approach, the polynomials of output vectors $\mathbf{t}$ and $\mathbf{u}$ are serialized one at a time. The vector operands $\hat{\mathbf{s}}$ and $\hat{\mathbf{r}}$ are used k times in the matrix-vector multiplication. Therefore, we decided to keep those in memory throughout the computation. Only maintaining one polynomial of those in memory would require re-sampling and transforming them to NTT domain k times which would introduce a significant performance penalty.

For key generation we require $k + 1$ polynomials, for encryption we require $k + 1$ polynomials, and for decryption we only use 3 polynomials regardless of k, but since decapsulation calls both CPA encryption and decryption, the stack usage is determined by encryption.

Adding noise. The noise polynomials $\mathbf{e}$, $\mathbf{e_1}$, and e_2 are only used once and are sampled from a centered binomial distribution using an extendable output function (XOF). We sample the coefficients of those polynomials on-the-fly without having to store the entire polynomials.

Kyber v2. Our stack optimizations are mostly unaffected by the algorithmic tweaks made by the Kyber team in round-2. However, in key generation (Algorithm 4), the noise vector $\mathbf{e}$ needs to be in NTT domain. Since the NTT transformation requires the entire polynomial $\mathbf{e}$ in memory; the on-the-fly sampling is no longer possible. Therefore, key generation requires an additional polynomial, i.e., $k + 2$ in total.

5 Results

For our experiments we use the STM32F4DISCOVERY together with an extended version of the `pqm4` [21] benchmarking framework. Particularly all cycles counts and stack measurements are those reported by `pqm4`, i.e., running the schemes at a low frequency of 24 MHz to not be impacted by memory wait states due to a slow memory controller. This allows to compare those numbers to boards different from the STM32F4DISCOVERY. We extend `pqm4` to also report cycles

Table 1. Cycle counts for NTT, NTT^{-1}, and the full polynomial multiplication ($\text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$). We outperform the current speed record by more than a factor of two for NTT and NTT^{-1}. The parameter changes in Kyber v2 further speed-up the polynomial multiplication.

	implementation	NTT [cycles]	NTT^{-1} [cycles]	polymul[cycles]
Kyber v1	[5]	21 855	23 622	
	This work	9 452 (−56.8%)	10 373 (−56.1%)	32 576
Kyber v2	This work	7 725	9 347	27 873

spent in hashing. Similar as pqm4 we use arm-none-eabi-gcc at version 8.2.0 [3] and set the optimization option to -O3.

We noticed that pqm4 suffered a serious performance penalty due to how it is using the 128 KiB memory of STM32F4DISCOVERY. pqm4 down-clocks the STM32F4DISCOVERY, such that all accesses to RAM should take the same number of cycles. However, according to [31] the 128 KiB of RAM are divided into SRAM1 which consists of 112 KiB and SRAM2 consisting of 16 KiB. In our experiments we noticed that memory accesses to SRAM2 are slower than to SRAM1, i.e., SRAM2 memory accesses cause wait states even at the low benchmarking frequency. At the time of writing pqm4 places the stack into SRAM2 which eventually grows into SRAM1. As a consequence of this, reducing memory consumption leads to the entire scheme fitting in SRAM2 introducing vast performance penalty. To account for this effect, we consistently place the stack in SRAM1 for all benchmarks. Consequently, the numbers in the following differ from this reported in pqm4. For fair comparison we re-benchmarked all implementations that were integrated pqm4 and indicate which benchmark results from related work were not performed using this way of benchmarking. We reported the problem to the authors of pqm4 and it is going to be resolved in a future version of pqm4.

In this section we present our results for Kyber. We start by benchmarking the NTT and polynomial multiplication in isolation and then report results for key generation, encapsulation, and decapsulation for all parameter sets of Kyber. All numbers reported in this section refer to the CCA-secure Kyber.

5.1 NTT and Polynomial Multiplication

Table 1 presents our new speed records for the computation of the NTT. Our optimized Kyber v1 NTT and NTT^{-1} are more than a factor two faster than the speed records [5]. Combining NTT and NTT^{-1} to perform a full polynomial multiplication in $\mathcal{R}_q$, i.e., computing $\text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$ requires 32 576 clock cycles.

[3] We also benchmarked our code using the February 2019 release of arm-none-eabi-gcc (8.3.0) which produced the same results.

Table 2. Cycle counts for all three security levels of Kyber compared to [5]. Link time optimization does benefit Kyber consistently, but our optimizations go far beyond. Kyber v2 is even faster, mainly due to algorithmic changes.

scheme	impl.	KeyGen cycles	Encaps cycles	Decaps cycles
Kyber-512 (v1)	[5]	666k	904k	934k
	`lto`[a]	637k (−4.3%)	866k (−4.1%)	881k (−5.6%)
	This work	575k (−13.7%)	763k (−15.6%)	730k (−21.8%)
Kyber-512 (v2)	This work	499k	634k	597k
Kyber-768 (v1)	[5]	1 098k	1 384k	1 417k
	`lto`[a]	1 048k (−4.6%)	1 325k (−4.3%)	1 339k (−5.5%)
	This work	946k (−13.9%)	1 167k (−15.7%)	1 117k (−21.1%)
Kyber-768 (v2)	This work	947k	1 113k	1 059k
Kyber-1024 (v1)	[5]	1 730k	2 083k	2 134k
	`lto`[a]	1 630k (−5.8%)	1 970k (−5.4%)	1 994k (−6.6%)
	This work	1 483k (−14.2%)	1 753k (−15.8%)	1 698k (−20.4%)
Kyber-1024 (v2)	This work	1 525k	1 732k	1 653k

[a] Only adding the compiler flag `-flto`.

In Kyber v2 only 7 out of 8 layers of the NTT are computed, which reduces the run-time to roughly 7/8 of the cycles. Computing $\texttt{NTT}^{-1}(\texttt{NTT}(a) \circ \texttt{NTT}(b))$ is considerably (14%) faster even though $\circ$ becomes more expensive.

The fastest multiplication in $\mathbb{Z}_{2^{13}}/(X^{256}+1)$, which has the same dimension as $\mathcal{R}_q$, using Toom–Cook [12,32] and Karatsuba [22] reported by Kannwischer–Rijneveld–Schwabe [20] requires 38 215 clock cycles. We outperform this by 27%. More importantly, Toom–Cook and Karatsuba multiplication require a significant amount of additional memory for intermediate values. For $\mathbb{Z}_{2^{13}}/(X^{256}+1)$, [20] reports 3 800 bytes of intermediate values which excludes the non-reduced result polynomial of 1 022 bytes[4]. Our polynomial multiplication is entirely in place.

In comparison to the performance presented in [4], our NTT is more than a factor of 1.8 faster (when applying the same scaling to accommodate for the different dimension that was also used in [4]). Most of the techniques we present also apply to the NewHope parameters targeted in [4], but some of the speedup we achieve is specific to the smaller value of q (NewHope uses $q = 12289$).

5.2 Kyber.CCA

Table 2 presents the cycle counts for all our implementations in comparison to the existing speed records [5]. By just turning on `-flto`, we achieve speedups of $4-7\%$ mainly caused due to in-lining modular reductions. The speed-ups achieved by applying our speed optimizations are $14-23\%$ and, thus, go far beyond what the compiler achieves. Our implementation of the round one variants of Kyber achieve the lowest cycle counts reported.

[4] $2n-1$ coefficients of 2 bytes each

Table 3. Stack usage for all three security levels of Kyber comparing our optimized implementations to [5]. For our stack-optimized implementation we notice a significant decrease of stack usage across all variants. The stack use of key generation of version 2 is roughly one polynomial (512 bytes) larger than in version 1. This is due to choice of Kyber's authors to represent the public key in the NTT domain.

scheme	impl.	KeyGen	Encaps	Decaps
		bytes	bytes	bytes
Kyber-512 (v1)	[5]	6 448	9 112	9 920
	This work	2 632 (−59%)	2 672 (−71%)	2 736 (−72%)
Kyber-512 (v2)	This work	3 136	2 720	2 744
Kyber-768 (v1)	[5]	10 544	13 720	14 880
	This work	3 072 (−71%)	3 120 (−77%)	3 176 (−79%)
Kyber-768 (v2)	This work	3 648	3 232	3 248
Kyber-1024 (v1)	[5]	15 664	19 352	20 864
	This work	3 520 (−78%)	3 568 (−82%)	3 624 (−83%)
Kyber-1024 (v2)	This work	4 160	3 752	3 776

As a result of the optimizations described in Section 4, we were able to reduce the stack usage of all Kyber variants significantly (see Table 3). Prior to our optimizations $k^2 + 3k$, $k^2 + 4k + 3$, and $2k + 2$ polynomials were used by key generation, encryption, and decryption respectively. Our optimizations were able to reduce this to $k + 1$ for all. Therefore, we notice a more considerable reduction for the higher security levels of Kyber.

Kyber v2. With our optimizations applied to the round two versions of Kyber, the cycle counts are comparable to round 1 if not faster. Similarly, stack size reductions are very comparable with the reductions made in round 1. The exception is the key generation procedure which uses $k + 2$ polynomials instead of $k + 1$ as described in Section 4.

5.3 Profiling

Table 4 contains the profiling information of our implementations for all parameter sets of Kyber v1 and Kyber v2. We observe the following:

Dominance of Hashing. Note that in the original implementation already 54% to 69% of execution time are spent in highly hand-optimized assembly implementation of the Keccak. This limits the speed-ups to be obtained since there is nothing or very little to be gained for this large fraction of the execution time. Our implementations spend the same time in hashing as the previous implementation, but this accounts for 64% to 81% of the total cycle counts. This confirms what previous work concluded [20, 29]: Post-quantum key encapsulation schemes are vastly dominated by hashing and having a hardware-accelerated Keccak permutation would speed-up the majority of schemes significantly. Kyber v2 spends significantly less time in Keccak which is due to the change of the

Table 4. Profiling of Kyber before and after applying all our optimizations. The run-time is vastly dominated by hashing. The cycles spent in NTT reduced notably. Only a small portion of the run-time is still spent in non-optimized code.

	impl.		total	Keccak	NTT	NTT $^{-1}$
Kyber-512 (v1)	[5]	**K:**	666k	453k (68%)	44k (7%)	47k (7%)
		E:	904k	596k (66%)	87k (10%)	71k (8%)
		D:	934k	506k (54%)	131k (14%)	95k (10%)
	This work	**K:**	575k	453k (79%)	19k (3%)	21k (4%)
		E:	763k	596k (78%)	38k (5%)	31k (4%)
		D:	730k	506k (69%)	57k (8%)	42k (6%)
Kyber-512 (v2)	This work	**K:**	499k	354k (71%)	31k (6%)	0 (0%)
		E:	634k	472k (74%)	15k (2%)	28k (4%)
		D:	597k	381k (64%)	31k (5%)	37k (6%)
Kyber-768 (v1)	[5]	**K:**	1 098k	754k (69%)	66k (6%)	71k (6%)
		E:	1 384k	922k (67%)	131k (9%)	95k (7%)
		D:	1 417k	794k (56%)	197k (14%)	118k (8%)
	This work	**K:**	946k	754k (80%)	28k (3%)	31k (3%)
		E:	1 167k	922k (79%)	57k (5%)	42k (4%)
		D:	1 117k	794k (71%)	85k (8%)	52k (5%)
Kyber-768 (v2)	This work	**K:**	947k	680k (72%)	46k (5%)	0 (0%)
		E:	1 113k	836k (75%)	23k (2%)	37k (3%)
		D:	1 059k	708k (67%)	46k (4%)	47k (4%)
Kyber-1024 (v1)	[5]	**K:**	1 730k	1 197k (69%)	87k (5%)	95k (5%)
		E:	2 083k	1 403k (67%)	175k (8%)	118k (6%)
		D:	2 134k	1 249k (59%)	262k (12%)	142k (7%)
	This work	**K:**	1 483k	1 197k (81%)	38k (3%)	42k (3%)
		E:	1 753k	1 403k (80%)	76k (4%)	52k (3%)
		D:	1 698k	1 249k (74%)	113k (7%)	62k (4%)
Kyber-1024 (v2)	This work	**K:**	1 525k	1 112k (73%)	62k (4%)	0 (0%)
		E:	1 732k	1 305k (75%)	31k (2%)	47k (3%)
		D:	1 653k	1 139k (69%)	62k (4%)	56k (3%)

parameters q and η. Both allow for a more efficient sampling routine that uses less SHAKE output and, thus, less Keccak permutations.

NTT. Prior to our optimizations 10% to 24% were spent in the NTT and NTT^{-1}. We speed-up those parts of the code by more than a factor of two and, consequently, they only account for 5% to 14% of the cycles in our optimized implementations.

5.4 Comparison to other PQC schemes on Cortex-M4

Compared to other implementations of NIST PQC KEM candidates on the ARM Cortex-M4 (Table 5), our Kyber implementation has both the smallest memory footprint and lowest cycle count for the sum of key generation, encapsulation and decapsulation. Both our stack-optimized implementations of Kyber-768 outperform all other implementations by large margins in terms of stack usage.

We also note a performance gap between the fastest implementation of Saber, reported in [20], and the stack-optimized implementation [23], whereas our implementations do not suffer any slow-down due to our stack optimizations.

Acknowledgments

The authors would like to thank Pedro Massolino, Joost Rijneveld, and Ko Stoffelen for their help with obtaining reasonable cycle counts on the ARM Cortex-M4.

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status report on the first round of the NIST post-quantum cryptography standardization process. National Institute of Standards and Technology Internal Report 8240 (2019), https://doi.org/10.6028/NIST.IR.8240 1
2. Alkim, E., Avanzi, R., Bos, J., Ducas, L., de la Piedra, A., Pöppelmann, T., Schwabe, P., Stebila, D.: NewHope: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2017), https://cryptojedi.org/papers/#newhopenist 2
3. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange – a new hope. In: Holz, T., Savage, S. (eds.) Proceedings of the 25th USENIX Security Symposium. USENIX Association (2016), https://eprint.iacr.org/2015/1092 2, 9
4. Alkim, E., Jakubeit, P., Schwabe, P.: A new hope on ARM Cortex-M. In: Carlet, C., Hasan, A., Saraswat, V. (eds.) Security, Privacy, and Advanced Cryptography Engineering. LNCS, vol. 10076, pp. 332–349. Springer (2016), http://cryptojedi.org/papers/#newhopearm 2, 3, 8, 14, 20
5. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: ARM Cortex-M4 optimized implementation of Kyber, https://github.com/pq-crystals/kyber/tree/cm4/cm4 (accessed 2019-03-07) 8, 9, 13, 14, 15, 16
6. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS–Kyber: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2017), https://pq-crystals.org/kyber 4
7. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS–Kyber: Algorithm specification and supporting documentation (version 2.0). Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), https://pq-crystals.org/kyber 6, 7
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak reference. Submission to the NIST SHA-3 competition (round 3) (2011), https://keccak.team/files/Keccak-reference-3.0.pdf 11

9. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS – kyber: A cca-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367. IEEE (2018), https://eprint.iacr.org/2017/634 4
10. Bos, J.W., Friedberger, S., Martinoli, M., Oswald, E., Stam, M.: Fly, you fool! Faster Frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Report 2018/1116 (2018), https://eprint.iacr.org/2018/1116 20
11. de Clercq, R., Roy, S.S., Vercauteren, F., Verbauwhede, I.: Efficient software implementation of ring-LWE encryption. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2015. pp. 339–344. EDA Consortium (2015), http://eprint.iacr.org/2014/725 2
12. Cook, S.: On the Minimum Computation Time of Functions. Ph.D. thesis, Harvard University (1966) 2, 11, 14
13. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of computation 19(90), 297–301 (1965), https://www.jstor.org/stable/2003354 5
14. Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: eXtended Keccak Code Package, https://github.com/XKCP/XKCP (accessed 2019-03-07) 11
15. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO '99. LNCS, vol. 1666, pp. 537–554. Springer (1999), http://dx.doi.org/10.1007/3-540-48405-1_34 4
16. Goldstine, H.H.: A History of Numerical Analysis from the 16th through the 19th Century. Springer (1977) 5
17. Güneysu, T., Oder, T., Pöppelmann, T., Schwabe, P.: Software speed records for lattice-based signatures. In: Gaborit, P. (ed.) Post-Quantum Cryptography. Lecture Notes in Computer Science, vol. 7932, pp. 67–82. Springer-Verlag Berlin Heidelberg (2013), document ID: d67aa537a6de60813845a45505c313, http://cryptojedi.org/papers/#lattisigns 8
18. Heideman, M.T., Johnson, D.H., Burrus, C.S.: Gauss and the history of the fast fourier transform. IEEE ASSP Magazine 1(4) (1984), http://www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf 5
19. Hülsing, A., Rijneveld, J., Schanck, J.M., Schwabe, P.: NTRU-KEM-HRSS: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2017), https://ntru-hrss.org 2
20. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates (2018), https://eprint.iacr.org/2018/1018 2, 7, 14, 15, 17, 20
21. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4 (accessed 2019-03-07) 2, 3, 7, 8, 12, 20
22. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Physics Doklady 7, 595–596 (1963), translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. Scanned version on http://cr.yp.to/bib/1963/karatsuba.html 2, 11, 14
23. Karmakar, A., Mera, J.M.B., Roy, S.S., Verbauwhede, I.: Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. IACR Transactions on Cryptographic Hardware and Embedded Systems 2018(3), 243–266 (2018), https://eprint.iacr.org/2018/682 7, 17, 20

24. Lyubashevsky, V., Seiler, G.: NTTRU: Truly fast NTRU using NTT. Cryptology ePrint Archive, Report 2019/040 (2019), https://eprint.iacr.org/2019/040 2, 6, 10
25. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985), http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf 9
26. National Institute for Standards and Technology: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2017), https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf 1
27. Oder, T., Pöppelmann, T., Güneysu, T.: Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6. ACM (2014), https://www.sha.rub.de/media/attachments/files/2014/06/bliss_arm.pdf 2
28. Pöppelmann, T., Oder, T., Güneysu, T.: High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) Progress in Cryptology – LATINCRYPT 2015. LNCS, vol. 9230, pp. 346–365. Springer (2015), extended version at https://eprint.iacr.org/2015/382 6
29. Saarinen, M.J.O., Bhattacharya, S., Garcia-Morchon, O., Rietman, R., Tolhuizen, L., Zhang, Z.: Shorter messages and faster post-quantum encryption with Round5 on Cortex M. Cryptology ePrint Archive, Report 2018/723 (2018), https://eprint.iacr.org/2018/723 15, 20
30. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039 (2018), https://eprint.iacr.org/2018/039 2, 8, 9, 10
31. Reference manual for STM32F405/415, STM32F407/417, STM32F427/437, and STM32F429/439 advanced ARM-based 32-bit MCUs (2019), https://www.st.com/resource/en/reference_manual/dm00031020.pdf 13
32. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers. Soviet Mathematics Doklady 3, 714–716 (1963), www.de.ufpe.br/~toom/my-articles/engmat/MULT-E.PDF 2, 11, 14
33. Zhang, Z., Chen, C., Hoffstein, J., Whyte, W.: NTRUEncrypt: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2017), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions 2

Table 5. Performance results of Kyber-768 in comparison to other round two candidates of NISTPQC optimized for the Cortex-M4. Prior to this work the fasted scheme in terms of encapsulation was NTRU-HRSS-KEM, whereas key generation is (still) fastest for Saber. The best memory footprints were achieved by R5ND_3PKEb and the memory optimized variant of Saber. Note that Saber, R5ND_3PKEb, and NTRU-KEM-743 are claiming NIST security level 3, whereas NTRU-HRSS-KEM claims NIST security level 1.

scheme	impl.	runtime cycles		stack usage bytes	
Kyber-768 (v1)	This work	**K:**	946k	**K:**	3 072
		E:	1 167k	**E:**	3 120
		D:	1 117k	**D:**	3 176
Kyber-768 (v2)	This work	**K:**	947k	**K:**	3 648
		E:	1 113k	**E:**	3 232
		D:	1 059k	**D:**	3 248
Frodo-AES128	[10]	**K:**	41 681k	**K:**	31 116
		E:	45 758k	**E:**	51 444
		D:	46 720k	**D:**	61 820
Frodo-cSHAKE128	[10]	**K:**	81 300k	**K:**	26 272
		E:	86 255k	**E:**	41 472
		D:	87 212k	**D:**	51 848
Saber	[20] [a]	**K:**	902k	**K:**	13 248
		E:	1 173k	**E:**	15 528
		D:	1 217k	**D:**	16 624
	[23] [b]	**K:**	1 165k	**K:**	6 931
		E:	1 530k	**E:**	7 019
		D:	1 635k	**D:**	8 115
R5ND_3PKEb	[29] [c]	**K:**	1 032k	**K:**	6 796
		E:	1 510k	**E:**	8 908
		D:	1 913k	**D:**	4 296
NewHope1024CCA	[4, 21] [a,d]	**K:**	1 221k	**K:**	11 152
		E:	1 902k	**E:**	17 448
		D:	1 926k	**D:**	19 648
NTRU-HRSS-KEM	[20] [a]	**K:**	145 986k	**K:**	23 396
		E:	406k	**E:**	19 492
		D:	827k	**D:**	22 140
NTRU-KEM-743	[20] [a]	**K:**	5 203k	**K:**	25 320
		E:	1 603k	**E:**	23 808
		D:	1 884k	**D:**	28 472

[a] Re-benchmarked in SRAM1 (see beginning of Section 5)
[b] Optimized for stack consumption
[c] Since R5ND_3PKEb does not report any stack usage, we report the numbers from https://github.com/mupq/pqm4/pull/16
[d] NTT assembly implementation from [4] with reference implementation in `pqm4` [21]