IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 3, pp. 336–357. DOI:10.13154/tches.v2020.i3.336-357

Cortex-M4 optimizations for {R,M}LWE schemes

Erdem Alkim[1,2], Yusuf Alper Bilgin[3,4], Murat Cenk[4] and François Gérard[5]

[1] Department of Computer Engineering, Ondokuz Mayıs University, Samsun, Turkey, erdemalkim@gmail.com
[2] Fraunhofer SIT, Darmstadt, Germany
[3] Aselsan Inc., Ankara, Turkey, y.alperbilgin@gmail.com
[4] Institute of Applied Mathematics, Middle East Technical University, Ankara, Turkey, mcenk@metu.edu.tr
[5] Université libre de Bruxelles, Brussels, Belgium, fragerar@ulb.ac.be

Abstract. This paper proposes various optimizations for lattice-based key encapsulation mechanisms (KEM) using the Number Theoretic Transform (NTT) on the popular ARM Cortex-M4 microcontroller. Improvements come in the form of a faster code using more efficient modular reductions, optimized small-degree polynomial multiplications, and more aggressive layer merging in the NTT, but also in the form of reduced stack usage. We test our optimizations in software implementations of Kyber and NewHope, both round 2 candidates in the NIST post-quantum project, and also NewHope-Compact, a recently proposed variant of NewHope with smaller parameters. Our software is the first implementation of NewHope-Compact on the Cortex-M4 and shows speed improvements over previous high-speed implementations of Kyber and NewHope. Moreover, it gives a common framework to compare those schemes with the same level of optimization. Our results show that NewHope-Compact is the fastest scheme, followed by Kyber, and finally NewHope, which seems to suffer from its large modulus and error distribution for small dimensions.

Keywords: ARM Cortex-M4 · Post-quantum key encapsulation · lattice-based cryptography · RLWE · MLWE · NTT · Kyber · NewHope · NewHope-Compact

1 Introduction

The interest in post-quantum cryptography, i.e., cryptography resisting adversaries equipped with both classical and quantum computers, has grown significantly among the research community in the last few years. This growth is partially driven by the NIST post-quantum standardization project aiming to create a formal environment in which concrete instantiations of several post-quantum techniques for signatures and key encapsulation can be analyzed and compared to each other concerning several metrics. The first round of the project took place mainly during 2018 and assessed different possible quantum-safe algorithms. In early 2019, 26 out of the 69 initial algorithms advanced to the second round of the project. In this second round, the practical performance of the candidates will play an important role in the selection for a hypothetical future standardization.

While most of the candidates already have optimized versions of their code targeting large CPUs, which often feature vector extensions such as AVX2, implementations for such architectures usually do not consider the memory usage or code size as a bottleneck. Instead, they usually are solely optimized for the computation time. However, small embedded devices can be greatly impacted by the switch toward a post-quantum paradigm.

Received: 2020-01-15 Accepted: 2019-03-15 Published: 2020-06-19

They often offer less memory, low computing power, and even have the drawback to be more susceptible to side-channel attacks. Hence, in recent papers, researchers focused more and more on small devices, e.g., based on the popular ARM Cortex-M4 processor, to assess the performance on embedded platforms. This microcontroller has the advantage of having large enough memory to support public-key algorithms, while being still reasonably small and cheap in the grand scheme of computing. Its popularity led to the development of pqm4 [KRSS], a library aiming to offer a common framework for benchmarking implementations of post-quantum algorithms on this platform.

Contributions: In this paper, we describe an optimized Cortex-M4 implementation of NewHope, Kyber, and a recently proposed variant of those schemes called NewHope-Compact. We present various optimizations, mainly in terms of speed and stack usage on the ARM microcontroller. Since those schemes share structural similarities, general improvements are applicable to all of them. Our implementation outperforms the current state-of-the-art for Kyber and NewHope while giving a unified framework to compare the three schemes, as they use the same level of optimization, which was not the case in previous works [KRSS]. Our contributions are listed as follows:

- We propose a 2-cycle modular reduction implementation for the Montgomery arithmetic, which translates subtraction to addition to allow the use of special instructions.
- We show that small-degree polynomial multiplications can be implemented efficiently by using lazy-reduction techniques. Hence, we show that early termination of the Number Theoretic Transform (NTT) can be implemented more efficiently when the base multiplication has a degree higher than 2.
- We show that even though the target architecture has only 14 usable registers, 16 coefficients can be used during the butterfly layers. This allowed us to merge up to four layers of the NTT and reduce the number of load and store instructions.
- We give several trade-offs between speed and other metrics. We show that the stack usage of key generation can be reduced by adding the error vector on-the-fly at the cost of an extra NTT computation. We also implement a well known idea to store the secret key as a seed and re-expand it during decapsulation.

Availability of the software: All source code is available at https://github.com/erdemalkim/NewHope-Compact-M4. The source code of Kyber and NewHope have already been pushed to pqm4.

Organization of this paper: Section 2 gives some brief background on the key-encapsulation schemes NewHope, NewHope-Compact, and Kyber. It also describes recent advances on the NTT and the algorithms most commonly used for efficient and constant-time implementations of modular reductions inside the NTT. Section 3 provides the details of our implementation and optimizations to achieve a better performance while using an as small as possible stack space. It also gives a trade-off between secret-key size and speed. Our performance results for NewHope, NewHope-Compact, and Kyber as well as a comparison with previous implementations of those schemes are presented in Section 4. Finally, in Section 5, we conclude the paper.

2 Preliminaries

In this section, we provide the required background that is necessary for understanding the remainder of this paper.

Algorithm 1 NewHope-CPA-PKE key generation

```
Output: public key $pk = (\hat{b}', \rho)$
Output: secret key $sk = \hat{s}$

1: $seed \xleftarrow{\$} \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \mathsf{SHAKE256}(64, seed)$
3: $\hat{a} \leftarrow \mathsf{GenA}(\rho)$
4: $s \leftarrow \mathsf{Sample}(\sigma, 0)$
5: $e \leftarrow \mathsf{Sample}(\sigma, 1)$
6: $\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e)$
7: return $pk = (\hat{b}, \rho), sk = \hat{s}$
```

Algorithm 2 NewHope-CPA-PKE encryption

```
Input: public key $pk = (\hat{b}, \rho)$
Input: message $\mu$ encoded in $\mathcal{R}_q$
Input: seed $coin \in \{0, \cdots, 255\}^{32}$
Output: ciphertext $(\hat{u}', h)$

1: $\hat{a} \leftarrow \mathsf{GenA}(\rho)$
2: $s' \leftarrow \mathsf{Sample}(coin, 0)$
3: $e' \leftarrow \mathsf{Sample}(coin, 1)$
4: $e'' \leftarrow \mathsf{Sample}(coin, 2)$
5: $\hat{t} \leftarrow \mathsf{NTT}(s')$
6: $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \mathsf{NTT}(e')$
7: $v' \leftarrow \mathsf{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + \mu$
8: return $c = (\hat{u}, \mathsf{Compress}(v'))$
```

Algorithm 3 NewHope-CPA-PKE decryption

```
Input: ciphertext $c = (\hat{u}, h)$
Input: secret key $sk = \hat{s}$
Output: message $\mu \in \{0, \cdots, 255\}^{32}$

1: $v' \leftarrow \mathsf{Decompress}(h)$
2: return $\mu = \mathsf{Decode}(v' - \mathsf{NTT}^{-1}(\hat{u} \circ \hat{s}))$
```

2.1 Notation

Let $\mathcal{R} = \mathbb{Z}[X]/(X^n+1)$ be the ring of integer polynomials modulo X^n+1. Then, we define $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$ as a special case of $\mathcal{R}$ such that every coefficient is reduced modulo q. Note that n and q are selected in such a way that performing an efficient NTT of an element in $\mathcal{R}_q$ is possible. For that reason, n is selected as a power of 2 such that X^n+1 is the $2n$-th cyclotomic polynomial, and q is selected as a prime allowing an efficient NTT implementation. We represent an element $a \in \mathcal{R}_q$ as $a = \sum_{i=0}^{n-1} a_i X^i$, where a_i is in $\mathbb{Z}_q$. Moreover, bold lower-case letters such as $\mathbf{v}$ denote a column vector and bold upper-case letters such as $\mathbf{A}$ denote a matrix with entries in $\mathcal{R}_q$. The representations of polynomials, vectors, and matrices in the NTT domain are denoted as $\hat{a}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{A}}$, respectively.

2.2 NewHope

NewHope [AAB+19, ADPS16b] is one of the NIST post-quantum standardization candidates whose security is based on the hardness of solving the Ring Learning With Errors (RLWE) problem [LPR10, LPR13]. This cryptosystem includes both an adaptive chosen-plaintext attack (CPA) secure key-encapsulation mechanism (KEM), referred to as NewHope-CPA-KEM, and an adaptive chosen-ciphertext attack (CCA) secure KEM, referred to as NewHope-CCA-KEM. Both versions are based on the previously proposed NewHope-Simple [ADPS16a] scheme, which was designed as a semantically secure public-key encryption (PKE) scheme and referred to as NewHope-CPA-PKE. Key generation, encryption, and decryption functions of NewHope-CPA-PKE are presented in Algorithm 1, Algorithm 2, and Algorithm 3. The constructions of NewHope-CPA-KEM and NewHope-CCA-KEM using NewHope-CPA-PKE are out of the scope of this work; we refer to [AAB+19, Alg. 16-21] for more information.

Typically, the most time-consuming part of all Learning With Errors (LWE) variant cryptosystems is the hashing used for the randomness generation. This randomness is required inside the GenA and the Sample functions. Moreover, both CPA and CCA secure versions of KEM constructions also require hashing. Note that the encode, decode,

Table 1: Parameters of NewHope512 and NewHope1024 and derived high-level properties [AAB+19].

Parameter Set	NewHope512	NewHope1024
Dimension n	512	1024
Modulus q	12289	12289
Noise Parameter k	8	8
NTT parameter γ	10968	7
Decryption error probability	2^{-213}	2^{-216}
Claimed post-quantum bit security	101	233
NIST Security Strength Category	1	5

compress, and decompress functions perform bit/byte-level manipulation, thus making them relatively inexpensive. Apart from hashing, the main cost of NewHope is multiplication in $\mathcal{R}_q$. NewHope selects its parameters n and q to enable fast polynomial multiplication in $\mathcal{R}_q$ by utilizing the NTT. The parameter sets are provided in Table 1. The modulus q is selected such that $q \equiv 1 \pmod{2n}$, ensuring the existence of the n-th root of unity ω and the $2n$-th root of unity $\gamma = \sqrt{\omega}$, which is a prerequisite for the NTT.

Polynomial multiplication utilizing the NTT: A forward NTT is performed to transform all of the coefficients to the NTT domain, and the inverse of this operation, NTT^{-1}, is performed to carry all coefficients to the normal domain again. The formulae for these two operations are given as follows:

$$\mathsf{NTT}(a) = \hat{a} = \sum_{i=0}^{n-1} \hat{a}_i X^i, \text{ where } \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \mod q,$$
$$\mathsf{NTT}^{-1}(\hat{a}) = a = \sum_{i=0}^{n-1} a_i X^i, \text{ where } a_i = \big(n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij}\big) \mod q.$$

The multiplication of $a, b \in \mathcal{R}_q$ can be computed as $ab = \mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$, where $\circ$ denotes the coefficient-wise multiplication. To perform the NTT and the NTT^{-1} operations faster, the Kyber and NewHope reference implementations used two different approaches. Kyber makes use of the Cooley-Tukey butterfly [CT65] in the NTT and the Gentleman-Sande butterfly [GS66] in the NTT^{-1}, while NewHope utilizes the Gentleman-Sande butterfly in both cases. The main reason that NewHope selects the second one is that it allows more aggressive lazy reductions when unsigned integers are used in the implementation. However, the NewHope reference implementation requires both input and output of the NTT^{-1} to be in the normal order, hence requiring an extra bitreversal.

NewHope-Compact: Alkim, Bilgin, and Cenk proposed a compact and fast instantiation of NewHope called NewHope-Compact [ABC19]. They presented three new parameter sets, which are shown in Table 2. As can be seen from the new parameter sets, they reduced the modulus q from 12289 to 3329 while preserving the same security level by the adjustment of the noise parameter k. Due to the change in q, the $2n$-th root of unity γ does not exist anymore, i.e., $q \not\equiv 1 \pmod{2n}$. However, even in this situation, recent results [ABC19, ABD+19, LS19, ZXZ+18] show that a fast NTT is still possible. The use of the NTT in this situation is achieved in [ABC19] by selecting γ as the 256-th root of unity so that a 7-level NTT is possible. Then, at the end, instead of having $n = 512$ or 1024

Table 2: Parameters of NewHope-Compact512, NewHope-Compact768 and NewHope-Compact1024 and derived high-level properties [ABC19].

Parameter Set	NH-Compact512	NH-Compact768	NH-Compact1024
Dimension n	512	768	1024
Modulus q	3329	3457	3329
Noise Parameter k	2	2	2
NTT parameter γ	17	55	17
Decryption error probability	2^{-256}	2^{-170}	2^{-181}
Claimed post-quantum bit security	100	163	230
NIST Security Strength Category	1	3	5

integer coefficients, i.e., degree zero polynomials, NewHope-Compact has 128 degree three or degree seven polynomials, respectively. Then, coefficient-wise multiplications in the NTT domain are performed on small-degree polynomials modulo $(X^4 - r)$ for $n = 512$ or $(X^8 - r)$ for $n = 1024$, where r is a power of γ, instead of multiplication of integer coefficients. They used a one-iteration Karatsuba method [WP06] for the multiplication of small-degree polynomials. The pseudocode of this step can be found in [ABC19, Alg. 5]. Note that the only parts changed from NewHope to NewHope-Compact are the definition of the NTT and the coefficient-wise multiplication. Therefore, one can still refer to Algorithm 1, Algorithm 2, and Algorithm 3 for the definition of key generation, encryption, and decryption, respectively, since these algorithms are written from a high-level perspective, and the mentioned changes are hidden inside internal functions.

Another contribution of [ABC19], as can be seen in Table 2, is the proposal of a new security level for NewHope, which is referred to as NewHope-Compact768. This new security level is made possible by using a different ring structure $\mathbb{Z}_q[X]/(X^{768} - x^{384} + 1)$, first proposed by [LS19]. NewHope-Compact selects q as 3457, which allows a similar implementation with other parameter sets. They applied a trick at the first level of the NTT to switch the regular ring structure $\mathcal{R}_q$ and a similar one at the last level of the NTT^{-1}. This trick uses the factorization of $(X^2 - X + 1)$ as $(X - \zeta_1)$ and $(X - \zeta_2)$, where ζ_1 and ζ_2 are both sixth roots of unity. We refer to [ABC19, Appendix A] and [LS19, Sec. 4.1] for more details. In the end, the one-iteration Karatsuba method is applied to perform coefficient-wise multiplications for the small-degree polynomials modulo $(X^6 - r)$, where r is a power of γ.

2.3 Kyber

Kyber [ABD$^+$19, BDK$^+$18] is a post-quantum KEM whose security relies on the hardness of the Module Learning With Errors (MLWE) problem [LS15]. Kyber constructs a CCA-secure KEM Kyber.CCAKEM by using a CPA-secure PKE, which is referred to as Kyber.CPAPKE, using a variant of the Fujisaki-Okamoto transform [FO99]. We refer to [ABD$^+$19, Alg. 7-9] for a description of Kyber.CCAKEM. The key generation, encryption, and decryption functions of Kyber.CPAPKE are presented in Algorithm 4, Algorithm 5, and Algorithm 6, respectively.

Similar to other LWE-based systems, such as NewHope, the most time-consuming part of Kyber is the hashing used for randomness generation. Aside from that, the most costly operation is the multiplication in $\mathcal{R}_q$. Kyber also utilizes the NTT to speed up this operation. The modulus q and the dimension n are selected as $q = 3329$ and

Algorithm 4 Kyber.CPAPKE key generation

Output: public key $pk = (\hat{\mathbf{b}}, \rho)$
Output: secret key $sk = \hat{\mathbf{s}}$

```
1: $seed \xleftarrow{\$} \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \mathsf{SHAKE256}(64, seed)$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{GenMatrixA}(\rho)$
4: $\mathbf{s} \leftarrow \mathsf{SampleVec}(\sigma, 0)$
5: $\mathbf{e} \leftarrow \mathsf{SampleVec}(\sigma, 1)$
6: $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{s}) + \mathsf{NTT}(\mathbf{e})$
7: return $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$
```

Algorithm 5 Kyber.CPAPKE encryption

Input: public key $pk = (\hat{\mathbf{b}}, \rho)$
Input: message $\mu \in \mathcal{R}_q$
Input: seed $coin \in \{0, \cdots, 255\}^{32}$
Output: ciphertext $(\hat{\mathbf{u}}', h)$

```
1: $\hat{\mathbf{A}} \leftarrow \mathsf{GenMatrixA}(\rho)$
2: $\mathbf{s}' \leftarrow \mathsf{SampleVec}(coin, 0)$
3: $\mathbf{e}' \leftarrow \mathsf{SampleVec}(coin, 1)$
4: $e'' \leftarrow \mathsf{SampleVec}(coin, 2)$
5: $\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
6: $\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}) + \mathbf{e}'$
7: $v' \leftarrow \mathsf{NTT}^{-1}((\hat{\mathbf{b}}^T \circ \hat{\mathbf{t}}) + e'' + \mu$
8: return $(\mathsf{Compress}(\mathbf{u}), \mathsf{Compress}(v'))$
```

Algorithm 6 Kyber.CPAPKE decryption

Input: ciphertext $c = (\mathbf{u}', h)$
Input: secret key $sk = \hat{\mathbf{s}}$
Output: message $\mu \in \mathcal{R}_q$

```
1: $\mathbf{u} \leftarrow \mathsf{Decompress}(\mathbf{u}')$
2: $v' \leftarrow \mathsf{Decompress}(h)$
3: return $\mu = v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u}))$
```

$n = 256$ for all parameter sets of Kyber. This enables a 7-level NTT with the parameter $\gamma = 17$. After performing the 7-level NTT, there are 128 degree-one polynomials. Therefore, coefficient-wise multiplications are performed on these degree-one polynomials modulo $(X^2 - r)$, where r is a power of γ, by using the schoolbook method.

2.4 FFT Trick

In this work, we used what is known in the recent literature as the FFT trick [Sei18]. The idea is to map the ring

$$\mathbb{Z}_q[X]/\langle X^n - \gamma^n \rangle$$

to

$$\mathbb{Z}_q[X]/\langle X^{n/2} - \gamma^{n/2} \rangle \times \mathbb{Z}_q[X]/\langle X^{n/2} + \gamma^{n/2} \rangle$$

by computing the straightforward CRT map

$$p \mapsto (p \bmod X^{n/2} - \gamma^{n/2}, p \bmod X^{n/2} + \gamma^{n/2}).$$

Since $X^{n/2} + \gamma^{n/2} = X^{n/2} - \gamma^{n+n/2}$, the same map can be computed again on both components until reaching a product of rings in which the multiplication is cheap.

The core operation is to reduce a polynomial p modulo both $X^{n/2} + \zeta$ and $X^{n/2} - \zeta$ (where ζ is an arbitrary power of γ) at the same time which is called a butterfly and is depicted in Figure 1. The computation of the NTT consists of applying $n/2$ butterflies to pairs of coefficients of the whole polynomial iteratively between 1 and $\log_2 n$ times, each iteration being referred to as a layer. Figure 2 depicts a full NTT consisting of 9 layers mapping the ring $\mathbb{Z}_q[X]/\langle X^{512} + 1 \rangle$ to a product of rings of the form $\mathbb{Z}_q[X]/\langle X - \zeta \rangle$. In NewHope-Compact and Kyber, the NTT is stopped earlier because $\mathbb{Z}_q$ does not offer enough high-order roots of unity to compute all the layers. This means that the NTT itself is less expensive, but the base operation in the product of rings is more costly. This base operation is a polynomial multiplication in rings of the form $\mathbb{Z}_q[X]/\langle X^a - r \rangle$ for a in $\{2, 4, 6, 8\}$ depending on the algorithm and parameter set used.

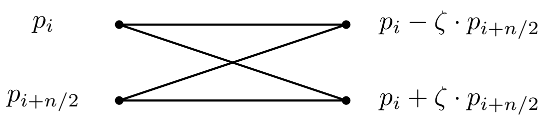

Figure 1: Cooley-Tukey Butterfly

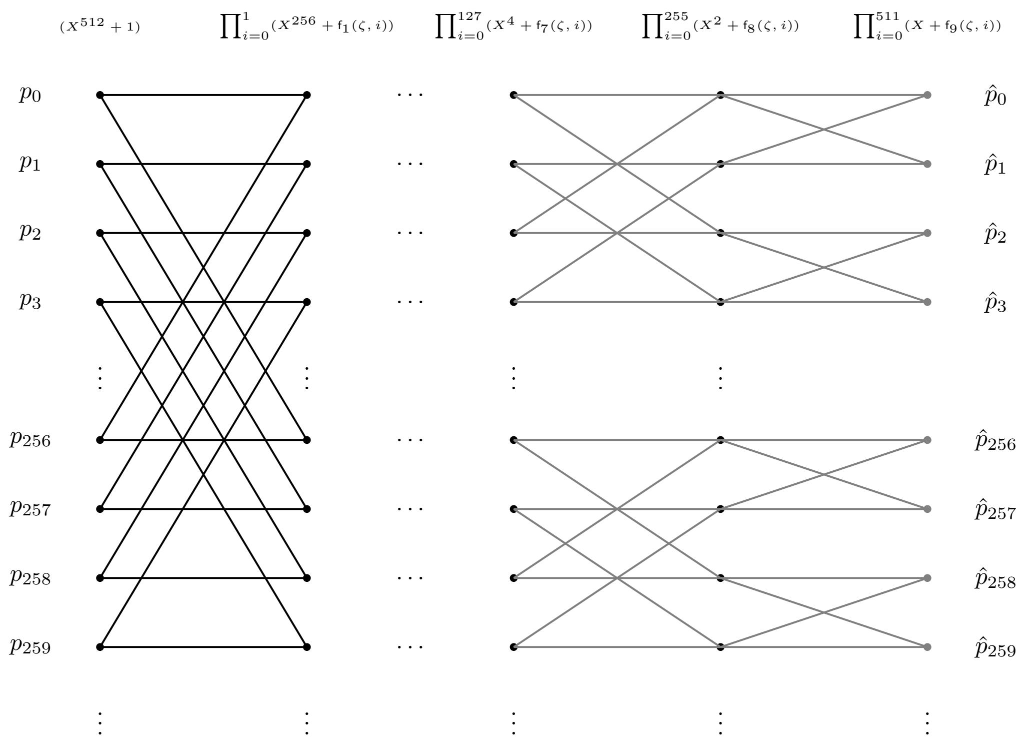
$(X^{512}+1)$
$\prod_{i=0}^{1}(X^{256}+\mathsf{f}_1(\zeta,i))$
$\prod_{i=0}^{127}(X^{4}+\mathsf{f}_7(\zeta,i))$
$\prod_{i=0}^{255}(X^{2}+\mathsf{f}_8(\zeta,i))$
$\prod_{i=0}^{511}(X+\mathsf{f}_9(\zeta,i))$
p_0
p_1
p_2
p_3
p_{256}
p_{257}
p_{258}
p_{259}
$\hat{p}_0$
$\hat{p}_1$
$\hat{p}_2$
$\hat{p}_3$
$\hat{p}_{256}$
$\hat{p}_{257}$
$\hat{p}_{258}$
$\hat{p}_{259}$

Figure 2: Full NTT on a degree 511 polynomial. The function $f_j(\zeta, i) = \zeta^{\mathsf{brv}(2^j-1+i)}$ selects the correct root to compute the isomorphism. All those roots are usually precomputed and correctly ordered in a table. Techniques to reduce q skip some levels: for example, using $q = 3329$ as in NewHope-Compact requires to skip the last two layers (grey).

2.5 Montgomery and Barrett Reductions

Montgomery [Mon85] and Barrett [Bar86] reductions are beneficial for efficient and constant-time implementations of reductions in $\mathcal{R}_q$. Efficient versions for signed integers of these reduction algorithms were presented in [Sei18, Alg. 3, 5], and they are recalled in Algorithm 7 and Algorithm 8. Moreover, assembly implementations of these reduction algorithms on the Cortex-M4 are provided in Algorithm 9, [BKS19, Alg. 7], and Algorithm 10. There are two important things to consider when using these efficient reduction algorithms: Firstly, while the Montgomery reduction gives outputs between $-q$ and q, the Barrett reduction gives outputs between 0 and q. Secondly, the Montgomery reduction cannot handle all signed words. It accepts inputs in the range of $-\frac{\beta}{2}q$ to $\frac{\beta}{2}q$, where β is selected as 2^{16} for efficient implementations.

Algorithm 7 Signed Montgomery reduction [Sei18] using Montgomery factor $\beta = 2^{16}$.

Input: odd q where $0 < q < \frac{\beta}{2}$, and a where $-\frac{\beta}{2}q \leq a = a_1\beta + a_0 < \frac{\beta}{2}q$ and $0 < a_0 < \beta$
Output: r' where $r' = \beta^{-1}a \pmod q$, and $-q < r' < q$

```
1: $m \leftarrow a_0 q^{-1} \ (\mathrm{mod}^{\pm}\ \beta)$        ▷ signed low product, $q^{-1}$ precomputed
2: $t_1 \leftarrow \left\lfloor \frac{mq}{\beta} \right\rfloor$        ▷ signed high product
3: $r' \leftarrow a_1 - t_1$
```

Algorithm 8 Signed Barrett reduction [Sei18] using $\beta = 2^{16}$.

Input: odd q where $0 < q < \frac{\beta}{2}$, and a where $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$
Output: r where $r = a \pmod q$, and $0 \leq r \leq q$

```
1: $v \leftarrow \left\lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \right\rceil$        ▷ precomputed
2: $t \leftarrow \left\lfloor \frac{av}{2^{\log(q)-1} \cdot \beta} \right\rfloor$        ▷ signed high product and arithmetic right shift
3: $t \leftarrow tq \mod \beta$        ▷ signed low product
4: $r \leftarrow a - t$
```

Algorithm 9 Signed Montgomery reduction [BKS19] using Montgomery factor $\beta = 2^{16}$.

Input: a where $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$
Output: reduced $a \to r'$ where $r' = \beta^{-1}a \pmod q$, and $-q < r' < q$

```
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 $a, a, t$        ▷ $a_{top} \leftarrow \left\lfloor \frac{a}{2^{16}} \right\rfloor - \left\lfloor \frac{t}{2^{16}} \right\rfloor$
```

Algorithm 10 Barrett reduction on packed argument using $\beta = 2^{16}$.

Input: a (32 bit signed integer where a_{top} and a_{bottom} contains two different coefficients)
Output: $r = r_{top} \mid r_{bottom}$ where $r_{top} \equiv a_{top} \pmod q$, $r_{bottom} \equiv a_{bottom} \pmod q$, $0 \leq r_{top}, r_{bottom} \leq q$ and $0 \leq q < 2^{15}$

```
1: movw v, const        ▷ v ← const where const=$\left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \cdot 2^{\beta}}{q} \right\rceil$ and precomputed
2: smulbb $t_1, a, v$        ▷ $t_1 \leftarrow a_{bottom} \cdot v$
3: smultb $t_2, a, v$        ▷ $t_2 \leftarrow a_{top} \cdot v$
4: asr $t_1, t_1$, #$(\log(\beta) + \lfloor \log(q) \rfloor - 1)$        ▷ $t_1 \leftarrow t_1 >> (\log(\beta) + \lfloor \log(q) \rfloor - 1)$
5: asr $t_2, t_2$, #$(\log(\beta) + \lfloor \log(q) \rfloor - 1)$        ▷ $t_2 \leftarrow t_2 >> (\log(\beta) + \lfloor \log(q) \rfloor - 1)$
6: smulbb $t_1, t_1, q$        ▷ $t_1 \leftarrow t_1 \cdot q$
7: smulbb $t_2, t_2, q$        ▷ $t_2 \leftarrow t_2 \cdot q$
8: pkhbt $t, t_1, t_2$, lsl#16        ▷ $t \leftarrow (t_1 \& 0xFFFFu) | (t_2 << 16)$
9: usub16 $r, a, t$        ▷ $r_{top} \leftarrow a_{top} - t_{top}$ and $r_{bottom} \leftarrow a_{bottom} - t_{bottom}$
```

2.6 ARM Cortex-M4

Our target platform is the STM32F4DISCOVERY [STM] development board featuring a 32-bit ARM Cortex-M4 [ARM], which is the selected platform by NIST to evaluate post-quantum candidates on microcontrollers. The Cortex-M4 implements the ARMv7E-M instruction set and provides special Digital Signal Processing (DSP) instructions. These DSP extensions offer Single Instruction Multiple Data (SIMD) instructions that can perform arithmetic operations on two halfwords or four bytes in parallel. These instructions have been shown to be very beneficial to speed up post-quantum algorithms [AJS16, BFM$^+$18, BKS19, KRS19, KRSS, KMRV18, SBG$^+$18]. This architecture comes with the restriction of a limited number of registers, which is 16 general-purpose 32-bit registers, out of which only 14 are available for the developer. The Cortex-M4 is also used by the benchmarking and testing framework `pqm4` [KRSS].

3 Implementation Details

This section first describes our optimizations to speed-up the computation of the polynomial multiplication in $\mathcal{R}_q$. This includes both the computation of the NTT and the NTT^{-1} as well as coefficient-wise multiplication of polynomials modulo $(X^d - r)$ where $d = 1$ in NewHope, $d = 2$ in Kyber, and $d = 4$, 6, and 8 in NewHope-Compact for $n = 512$, 768, and 1024, respectively. Then, we present our implementation techniques that decrease the stack usage of NewHope and NewHope-Compact. We use on-the-fly generation of $\hat{a}$ during arithmetic operations in the NTT domain proposed by [BKS19] for Kyber. We also introduce a new method for key generation, which adds the error polynomial in the normal domain instead of in the NTT domain. Finally, we provide a trade-off between the size of the secret key and the performance.

3.1 Optimization of Polynomial Multiplication for Speed

Polynomial multiplication in $\mathcal{R}_q$ is one of the most time-consuming parts of KEMs whose security relies on RLWE/Ring Learning With Rounding (RLWR) or MLWE/Module Learning With Rounding (MLWR) problems. Some of them, specifically NewHope and Kyber, utilize the NTT for ring arithmetic in order to have a fast and efficient implementation. We present an optimized assembly implementation of polynomial multiplication on the Cortex-M4, which can be used by all RLWE/RLWR-MLWE/MLWR schemes utilizing the NTT for the ring arithmetic and have a modulus smaller than 2^{15}. Although some of the techniques described in this section are specific to the rings used by NewHope, NewHope-Compact, or Kyber, adapting an implementation of one to another with some minor changes is possible. We indicate such points when appropriate.

Representation of polynomials and packing: Similar to [BKS19], we represent polynomials in $\mathcal{R}_q$ as an array composed of signed 16-bit integers. This representation was first introduced in an AVX2 implementation of Kyber by [Sei18]. In their reference implementations, Kyber and NewHope-Compact are already implemented by using signed halfword integers. However, [AJS16] preferred using unsigned values for NewHope on the Cortex-M4. As discussed previously in Section 2.2, NewHope utilizes the Gentleman-Sande butterfly in both the NTT and the NTT^{-1}, while the other two schemes use the Cooley-Tukey butterfly in the NTT and the Gentleman-Sande butterfly in the NTT^{-1}. We decided to follow the approach used by Kyber and NewHope-Compact, since it has been shown in [Sei18] that using this approach is more efficient when the coefficients of the polynomials are represented as signed halfwords. Moreover, the Cortex-M4 is a 32-bit architecture, while the polynomial coefficients are below 16 bits.

Therefore, in order to fully utilize the properties of the Cortex-M4 platform, we packed two coefficients into one register. Thereby, we can utilize SIMD instructions and perform addition/subtraction on two halfwords in parallel by using `uadd16` or `usub16`. Moreover, similar to [BKS19], we implement a double butterfly, which takes a packed register as input and returns a packed butterfly result.

Montgomery, Barrett, and lazy reductions: The Montgomery reduction, which is given in Algorithm 9, is implemented by [BKS19] in three clock cycles. In this work, we optimize the implementation of the Montgomery reduction such that it can be performed in only two clock cycles. We achieve this by storing $-q^{-1}$ instead of q^{-1} and using the `smlabb` instruction which multiplies two halfwords and adds the 32-bit result to another 32-bit value in one clock cycle. This implementation is given in Algorithm 11. The Kyber implementation of [BKS19] performs 3200 Montgomery reductions (1792 in the two NTT, 896 in NTT^{-1}, 512 in base multiplication) in a full polynomial multiplication $(\mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b)))$. Therefore, this change saves 3200 clock cycles for a full polynomial multiplication in $\mathcal{R}_q$.

Algorithm 11 Signed Montgomery reduction using Montgomery factor $\beta = 2^{16}$.

Input: a where $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$
Output: reduced $a \to r'$ where $r' = \beta^{-1}a \pmod q$, and $-q < r' < q$

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

Similarly to [BKS19], we used the Barrett reduction given by [Sei18, Alg. 5]. The assembly implementation of this reduction on a packed argument is given in Algorithm 10. It requires nine clock cycles on our Cortex-M4 microcontroller. We also implemented a Montgomery reduction on a packed argument (Algorithm 12). It needs eight clock cycles, which is slightly faster than the Barrett reduction. Each halfword of the input of Algorithm 12 is multiplied by β to be able to get the same result as the Barrett reduction. Note that their outputs are not in the same range. Hence, after the last layer of the NTT and the NTT^{-1}, where we need our output to be between 0 and q, we use the Barrett reduction. In other cases, we use the faster Montgomery reduction.

Algorithm 12 Signed Montgomery reduction on packed argument using Montgomery factor $\beta = 2^{16}$.

Input: a (32 bit signed integer where a_{top} and a_{bottom} contains two different coefficients)
Output: $r = r_{top} \mid r_{bottom}$ where $r_{top} \equiv a_{top} \pmod q$, $r_{bottom} \equiv a_{bottom} \pmod q$

```
1: movw $v$, const               ▷ $v \leftarrow$ const where const=$\beta \pmod q$ and precomputed
2: smulbb $t_1, a, v$
3: smulbb $r_1, t_1, -q^{-1}$    ▷ $r_1 \leftarrow (t_1 \mod \beta) \cdot (-q^{-1})$
4: smlabb $r_1, r_1, q, t_1$     ▷ $r_{1_{top}} \leftarrow \left\lfloor \frac{(r_1 \mod \beta) \cdot q + t_1}{2^{16}} \right\rfloor$
5: smultb $t_2, a, v$
6: smulbb $r_2, t_2, -q^{-1}$    ▷ $r_2 \leftarrow (t_2 \mod \beta) \cdot (-q^{-1})$
7: smlabb $r_2, r_2, q, t_2$     ▷ $r_{2_{top}} \leftarrow \left\lfloor \frac{(r_2 \mod \beta) \cdot q + t_2}{2^{16}} \right\rfloor$
8: pkhtb $r, r_2, r_1$, asr #16  ▷ $r \leftarrow (r_{2_{top}} | (r_{1_{top}} >> 16))$
```

Depending on the size of the modulus and the register size of the underlying architecture, it is not always necessary to reduce the results after an addition or subtraction. Skipping unnecessary reductions is called lazy reduction. It is common that optimized NTT implementations heavily use this technique to speed-up the code. However, those lazy reductions are mostly performed after an addition or subtraction, as stated before. In this work, we also perform lazy reductions during component-wise multiplications, also referred to as base multiplications, for moduli 3329 and 3457 used in Kyber and NewHope-Compact.

The base multiplication for Kyber is given in Algorithm 13. Each (mod q) in Algorithm 13 corresponds to a Montgomery reduction. As we can see, five Montgomery reductions are needed in each base multiplication. We noticed that if both of the coefficients that are multiplied are already reduced modulo q, then the result is much lower than the value that the Montgomery reduction can handle, i.e., $2^{15} \cdot q$ (see Proposition 1). Thus, we can sum up the results of several multiplications before performing a Montgomery reduction, e.g., we can compute $(c[0] \leftarrow ((a[0] \cdot b[0]) + ((a[1] \cdot b[1]) \bmod q) \cdot r) \bmod q)$ instead of applying line 2 of Algorithm 13. Therefore, we save one Montgomery reduction per coefficient, i.e., two per base multiplication. In total, there are 128 base multiplications in a polynomial multiplication in $\mathcal{R}_q$. Thus, we save 256 Montgomery reductions, i.e., 768 clock cycles for the implementation of [BKS19] and 512 clock cycles for our implementation for a polynomial multiplication of Kyber.

Moreover, we can use this lazy-reduction technique for NewHope-Compact. The base multiplications for NewHope-Compact512, NewHope-Compact768, and NewHope-Compact1024 require 4, 6, and 8 sequential additions of the multiplication results, which can be handled as shown in Proposition 1. Therefore, we can skip 3, 5, or 7 Montgomery reductions per coefficient, which sums up to 1536, 3840, or 7168 Montgomery reductions in total for NewHope-Compact512, NewHope-Compact768, or NewHope-Compact1024, respectively. Note that we can also omit the Montgomery reductions after the multiplications in the first layer of the Cooley-Tukey butterflies, where the inputs are polynomials with small coefficients that are sampled from the centered binomial distribution. The results of the first multiplications can only grow up to $\pm 2q$ for Kyber and NewHope-Compact and up to $\pm q$ for NewHope. However, this technique cannot be used if the input polynomial is not a polynomial having small coefficients, i.e., not sampled from the centered binomial distribution. We have such cases in Kyber by design because of the ciphertext compression. We also cause this case in NewHope and NewHope-Compact with the stack-usage optimization explained in Section 3.2.

Algorithm 13 Multiplication of polynomials in $\mathbb{Z}_q[X]/(X^2 - r)$ for Kyber.

Input: a and $b \in \mathbb{Z}_q[X]/(X^2 - r)$ where r is a power of γ.
Output: $c \in \mathbb{Z}_q[X]/(X^2 - r)$.

```
1: function basemul($a$, $b$)
2:     $c[0] \leftarrow (a[0] \cdot b[0]) \bmod q + ((a[1] \cdot b[1]) \bmod q) \cdot r) \bmod q$
3:     $c[1] \leftarrow (a[0] \cdot b[1]) \bmod q + (a[1] \cdot b[0]) \bmod q$
4:     return $c$
5: end function
```

Proposition 1. *Let $-3329 < a_i, b_i < 3329$ where $0 \leq i \leq 8$, and $c = \sum_{i=0}^{8} a_i \cdot b_i$. c is in the range of $(-2^{15} \cdot 3329)$ to $(2^{15} \cdot 3329)$*

Proof. Let $a_i = 3328$ and $b_i = 3328$ for $0 \leq i \leq 8$ be the maximum allowed values. Then, $\sum_{i=0}^{8} a_i \cdot b_i = 99680256$. This is very close to $2^{15} \cdot 3329$ which is the maximum value for the input of the Montgomery reduction. In fact, adding another multiplication of coefficients to the sum, i.e., $3328^2 + 99680256 = 110755840$, will exceed $2^{15} \cdot 3329 = 109084672$. □

Merging NTT layers: Merging multiple NTT layers reduces the number of load and store instructions and gives a noticeable performance improvement on the Cortex-M4

[AJS16, BKS19]. While [AJS16] uses eight registers for eight coefficients and performs three layers of the NTT, [BKS19] uses only four registers for eight coefficients and performs two layers of the NTT without storing and reloading. Both use the remaining registers to store the constants required in the Montgomery and Barrett reductions as well as the loop counter. In this work, we use eight registers to keep 16 coefficients and perform three or four layers of the NTT, depending on the distance between the coefficients being used in the same butterfly on the next layer. In other words, we load coefficients into these eight registers in such a way that a maximum of NTT layers can be performed before storing the results. Thanks to the structure of the NTT used in NewHope and NewHope-Compact, we can merge four layers, since at some point, we need coefficients with distance one, and loading consecutive coefficients from the memory is free with the `ldr` instruction.

Moreover, Kyber uses seven layers of a 256-point NTT, which is different from NewHope-Compact, having seven layers of a 128-point NTT. Consequently, while NewHope-Compact has distance one coefficients on the last layer, Kyber requires distance two coefficients. Thus, the last four layers cannot be merged. Therefore, we merged seven layers as 3+3+1 for Kyber.

Although eight registers are used to store coefficients, we can still spare some registers to store the constants required for the Montgomery reductions, specifically q and $-q^{-1}$. However, we have to reload the Barrett constant (line 1 of Algorithm 10) or the Montgomery constant (line 1 of Algorithm 12) at every use, but note that we do not need them heavily thanks to lazy reductions. We follow a different approach for the loop counter, which will be described in the loop unrolling paragraph. Hence, we save more than we lose, i.e., loading more coefficients and performing more layers in every loop is better than loading the Barrett constant only once and keeping it in the register to be used for all Barrett reductions.

Precomputation of twiddle factors: The powers of the NTT constant γ are often referred to as twiddle factors. It is a common approach to precompute all of these twiddle factors in the Montgomery domain and store them in flash memory. In this work, we use the Montgomery factor $\beta = 2^{16}$, similar to [BKS19, Sei18]. We reorder these constants before storing them in the flash memory to have them appearing in memory in the same order as they are used during the computation. Hence, we can easily load the next one without computing its address. The load instruction on the Cortex-M4 has the ability to move the pointer to the next twiddle factor while fetching the current value from memory. Thus, moving to the next factor has no extra cost. Since the first Montgomery reduction for NewHope and NewHope-Compact can be skipped, as mentioned before, the first twiddle factor for these two schemes should not be stored in the Montgomery domain when this optimization is used. Moreover, since we use the Gentleman-Sande butterfly for the NTT^{-1}, we perform the division with n on half of the coefficients during the last butterfly by just multiplying the twiddle factor(s) for the last layer with n^{-1}.

Unrolling: We unroll the outer loop of the NTT and iterate over the layers as usual. [AJS16, BKS19] spare one register for the loop counter. While [AJS16] uses this loop counter both to check the number of iterations remaining in the loop and to decide which precomputed twiddle factor to use, [BKS19] uses it only to detect when to end the loop. We decided not to spare a register for this loop counter and instead use this freed register to load more coefficients in every loop and merge more NTT layers. Then, we can naively use the `.rept` precompiler directive instead of this loop counter. However, since the `.rept` only repeats the same code, it increases the code size dramatically. Hence, we went back to the loop counter idea again. However, instead of keeping it in a register, we spill it to the stack. Consequently, the code size stays reasonable while we can still load more coefficients in every loop. As an obvious observation, using the `.rept` directive is slightly faster. Hence, it might be useful for some applications where there is plenty of empty memory for storing the code.

Link-time optimization: Link-time optimization is an important option to control optimization, although it may increase the code size. The usual approach without link-time optimization is that each source file is compiled with some optimization level to generate different object files. Then, these optimized object files are linked together to compose an executable file. Although this approach does a good job optimizing source code, it turns out that the linker can perform even better optimization when link-time optimization (`-flto`) is enabled. This option can give a performance boost of around 10%. The critical performance gain is achieved from cross-module function inlining, which is not directly possible without `-flto`. This will tend to increase the code size since inlining functions across source files introduces code duplication. However, it should also be noted that link-time optimization is more effective at identifying unused code or code that has no impact on the output.

The `pqm4` platform does not use `-flto` as default option since it increases stack consumption or results in a slower computation of some schemes while improving the performance of Kyber [BKS19]. Therefore, we have also tested the effect of `-flto` on performance for our implementations of Kyber, NewHope, and NewHope-Compact and realized that they all benefit from it and have a performance boost. However, since assembly-optimized polynomial multiplication was already implemented carefully by inlining all necessary functions such as modular reductions, adding `-flto` has no effect on its performance.

3.2 Optimization of NewHope and NewHope-Compact for Stack Usage

On embedded devices, RAM usage is often a significant bottleneck. Outside of real-time systems, one can always wait for a slow algorithm, but if the algorithm needs more memory than available on the device, it cannot be used. While the Cortex-M4 on our board offers quite a large amount of memory, we decided to optimize for stack usage as well.

Our goal was to reduce the minimum amount of stack space required to compute the cipher while keeping performance mostly unaffected. While it is possible to follow a more aggressive approach to reduce stack usage, such implementations would be considerably slower than ours. The three main metrics regarding implementation are speed, stack usage, and code size. The one to optimize depends on the context in which the cipher will be used. In our work, we tried to optimize the first two while keeping the last one reasonable.

Key generation: The core of the key generation (Algorithm 1) is the computation of $\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e)$. Since each coefficient of the output of the NTT depends on *all* coefficients of the input, all coefficients of s and e must have been generated before proceeding to the addition. Hence, at least two full polynomials should be stored in memory. To reduce the memory usage, we used the observation that polynomial multiplication can be performed on-the-fly in the NTT domain and, likewise, adding an error to a polynomial can be performed on-the-fly in the *normal* domain. Indeed, the operation $\circ$ works sequentially on parts of its inputs (one coefficient at the time for NewHope and four, six, or eight for NewHope-Compact depending on the parameter set used) and does not need all of them in memory at the same time. Similarly, each coefficient of the error polynomial can be computed and added separately, but only if the addition considered is in the normal domain. This is why instead of computing

$$\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e),$$

we compute

$$\hat{b} \leftarrow \mathsf{NTT}(\mathsf{NTT}^{-1}(\hat{a} \circ \mathsf{NTT}(s)) + e)$$

and perform the multiplication and the addition on-the-fly. This requires one more NTT^{-1}, but allows to store only *one* polynomial in memory, containing s and $\hat{b}$ subsequently. This approach reduces stack usage significantly and since our benchmarks show that computing one extra optimized NTT^{-1} only increases the key generation time by around 5%, we believe that this is a good trade-off. This trick can be similarly applied to KYBER. Note that the small *relative* cost of this technique is specific to our context and is mainly due to the fact that hashing is the main performance bottleneck. This issue will be discussed in more detail in Section 4. If a faster hash function is used, the decrease in performance will be higher than 5%. That being said, the *absolute* cost of the trick is always the same with one NTT^{-1}.

Encryption: The encryption procedure (Algorithm 2) is mainly driven by the following computations:

1. $\hat{t} \leftarrow \mathsf{NTT}(s')$
2. $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \mathsf{NTT}(e')$
3. $v' \leftarrow \mathsf{NTT}^{-1}(\mathsf{DecodePoly}(\hat{b}') \circ \hat{t}) + e'' + \mathsf{Encode}(\mu)$

The first two yield a situation similar to the key generation but unfortunately require two polynomials on the stack frame. Indeed, since $\hat{t}$ appears in the second and the last computation, the result of $\hat{a} \circ \hat{t}$ *cannot* be stored in the same memory space as $\hat{t}$ (and since it would need to go through a NTT^{-1}, it does need to be fully stored). Once $\hat{u}$ is computed, it can be packed in the ciphertext and free one of the two polynomials. The last computation is quite friendly for stack usage. Since both the base multiplication and the addition operate on small portions of the polynomial, $e'' + \mathsf{Encode}(\mu)$ can be computed coefficient-wise and $\hat{b}$ can be partially unpacked from the inputs. Thus, Line 3 could technically be computed with one polynomial plus a small overhead in the stack frame. Since two polynomials were already allocated previously and only maximal stack usage is relevant, we actually fully unpack $\hat{b}$. Finally, the stack usage is bigger than the one of the key generation because of the extra polynomial stored.

Decryption: The decryption of NEWHOPE (Algorithm 3) is quite lightweight in terms of stack usage. Unfortunately, the algorithms introduced in the preliminaries are the CPA version of the cipher. Since the CCA transform runs the encryption procedure during decryption, the stack usage is essentially the same as for encryption.

3.3 Trade-offs Between Secret Key Size and Speed

There are different trade-offs between the secret key size and the performance of the scheme. [ABD+19, BDK+18] proposed to store the seed used for all randomness in the key generation if the size of the secret key is critical. However, this requires to perform key generation again during decapsulation and gives a significant performance penalty. As also stated by [ABD+19, BDK+18], another optimization could be storing the secret key in the normal domain instead of the NTT domain. Hence, each coefficient can be compressed to 3 bits, since their possible values are in between -2 and 2. Note that our NTT implementation is fast on the Cortex-M4, so we decided that such optimizations are good trade-offs for this platform. We also observed that sampling the secret polynomial s is fast enough, although it is one of the most time-consuming part of such algorithms is the hashing used for the randomness generation. Note that sampling the secret key from the centered binomial distribution is lightweight in comparison to the generation of the public parameter a since we can extract two coefficients from only one byte by using the centered binomial distribution while uniform sampling needs two bytes to extract only one coefficient. Hence, we decided to store only the 32-byte secret-key seed. Then, the secret key is sampled and transformed to the `NTT` domain again during decapsulation. These operations reduce the secret key size by 736 bytes for Kyber512 and NewHope-Compact512, 864 bytes for NewHope512, 1120 bytes for Kyber768 and NewHope-Compact768, 1504 bytes for Kyber1024 and NewHope-Compact1024, and 1760 bytes for NewHope1024. However, they increase the decapsulation time by around 7% for Kyber, 9% for NewHope-Compact, and 18% for NewHope, while decreasing the key generation time slightly.

4 Results and Comparison

Our optimizations were implemented in the three sibling schemes NewHope, NewHope-Compact, and Kyber. Comparing different schemes across parameter sets is often complicated because performance is always strongly correlated with the targeted security level. Most of the implemented schemes propose parameter sets for NIST security levels 1, 3, and 5, which correspond to 128, 192, and 256 bits of security. Fortunately, since all the schemes involved in our tests are similar and based on {R,M}LWE, the dimension of the underlying lattice problem can be roughly translated into NIST security levels. Hence, we compare them for dimensions 512, 768 (if available), and 1024, which correspond to the three aforementioned security levels.

4.1 Speed Comparison

The results of our benchmarks in terms of speed can be found in Table 3. The code was compiled and run in the same conditions as the schemes benchmarked in `pqm4` [KRSS]. We use the latest `arm-none-eabi-gcc` release (version 9.2.1) that is currently available. We compare the two candidates NewHope and Kyber against their previous Cortex-M4 optimized implementations available in `pqm4` and also add the newcomer NewHope-Compact. One can see that NewHope and Kyber perform around 10% better with our optimizations, while using less stack space (see Table 4). Furthermore, NewHope-Compact is more than 40% faster compared to NewHope and more than 25% faster compared to Kyber for all security levels. This is explained by the two following observations:

- NewHope-Compact is a variant of NewHope using a smaller modulus and distribution, which means an increased performance during polynomial multiplication because of lazy reductions and less hashing needed to sample from the error distribution.
- NewHope-Compact is based on RLWE, while Kyber is based on MLWE. Hence, even though they share similar parameter sets, the inherent performance penalty of using the less structured version of LWE hurts Kyber.

Moreover, some design decisions affect the performance or the stack usage of the scheme. These design decisions are unrolling `NTT` loops by using the `.rept` directive instead of the loop counter (Section 3.1), optimizing the stack usage (Section 3.2), and the trade-off between the size of the secret key and performance (Section 3.3). These three decisions can be easily enabled or disabled. The effects of the last two choices on the performance and the stack usage are given in Table 3. Using the `.rept` directive instead of the loop counter decreases the cycle count by n per `NTT` call, where n is the degree of the polynomial

Table 3: Cycle count comparison for the {R,M}LWE schemes improved by our work. **G**: key generation, **E**: encapsulation, **D**: decapsulation.

Scheme		**Previous work**	**This work Speed**	**This work Stack Opt.**	**This work Short sk**
NewHope	512	**G**: 588 683 [a] **E**: 918 558 [a] **D**: 904 800 [a]	**G**: 561 161 **E**: 865 243 **D**: 820 130	**G**: 578 850 **E**: 865 856 **D**: 820 742	**G**: 555 625 **E**: 865 018 **D**: 968 961
	1024	**G**: 1 161 112 [a] **E**: 1 777 918 [a] **D**: 1 760 470 [a]	**G**: 1 117 398 **E**: 1 687 272 **D**: 1 612 960	**G**: 1 157 331 **E**: 1 689 375 **D**: 1 614 804	**G**: 1 106 350 **E**: 1 686 964 **D**: 1 918 747
NH-Cmpct	512	-	**G**: 335 991 **E**: 531 453 **D**: 484 416	**G**: 349 692 **E**: 532 423 **D**: 484 945	**G**: 330 826 **E**: 531 282 **D**: 526 805
	768	-	**G**: 501 885 **E**: 782 315 **D**: 717 250	**G**: 524 181 **E**: 784 117 **D**: 718 950	**G**: 494 364 **E**: 782 471 **D**: 786 664
	1024	-	**G**: 658 581 **E**: 1 022 903 **D**: 940 023	**G**: 686 225 **E**: 1 025 503 **D**: 941 076	**G**: 648 206 **E**: 1 022 773 **D**: 1 030 809
Kyber	512	**G**: 514 291 [b] **E**: 652 769 [b] **D**: 621 245 [b]	**G**: 452 919 **E**: 586 380 **D**: 542 576	**G**: 461 693 **E**: 586 754 **D**: 543 332	**G**: 446 876 **E**: 586 403 **D**: 579 594
	768	**G**: 976 757 [b] **E**: 1 146 556 [b] **D**: 1 094 849 [b]	**G**: 860 227 **E**: 1 031 603 **D**: 967 124	**G**: 872 140 **E**: 1 030 764 **D**: 966 848	**G**: 850 228 **E**: 1 030 679 **D**: 1 021 439
	1024	**G**: 1 575 052 [b] **E**: 1 779 848 [b] **D**: 1 709 348 [b]	**G**: 1 394 148 **E**: 1 603 776 **D**: 1 522 900	**G**: 1 410 591 **E**: 1 603 988 **D**: 1 523 175	**G**: 1 381 514 **E**: 1 603 772 **D**: 1 595 530

[a] [KRSS], `https://github.com/mupq/pqm4/`, commit `be0c421aaecaad4443071bfcf62ad397d4f40832`.
[b] [BKS19]

for the selected parameters. However, the code size increases by a factor of 10 to 50. Optimizing the stack usage decreases the memory used by key generation while increasing the cycle count of the same function. Finally, applying the method described in Section 3.3 to reduce the size of the secret key increases the cycle count of the decapsulation while decreasing it for the key generation.

4.2 Dominance of Hashing

The speed difference shown in Table 3 might look slim at first sight. This is due to the fact that, as pointed out by previous works, those schemes have been optimized so much that the bottleneck is now the generation of random numbers through hashing instead of the polynomial multiplication procedure. Table 5 shows the time spent hashing for all algorithms and parameter sets. As we can see, with a minimum of 66% for the decapsulation of NewHope-Compact, all the algorithms are severely dominated by hashing. Even if polynomial multiplications were somehow instantaneous, the results of Table 3 would be somewhat similar.

Table 4: Stack usage comparison for the {R,M}LWE schemes improved by our work. **G**: key generation, **E**: encapsulation, **D**: decapsulation.

Scheme	NewHope **(This work)**	NewHope [**KRSS**][a]	NH-Cmpct **(This work)**	Kyber **(This work)**	Kyber [**BKS19**]
512	**G**: 2 056 **E**: 2 864 **D**: 2 880	**G**: 5 960 **E**: 9 168 **D**: 10 296	**G**: 2 160 **E**: 2 984 **D**: 2 984	**G**: 2 392 **E**: 2 344 **D**: 2 360	**G**: 2 952 **E**: 2 552 **D**: 2 560
768	-	-	**G**: 2 600 **E**: 3 936 **D**: 3 936	**G**: 3 240 **E**: 2 856 **D**: 2 864	**G**: 3 848 **E**: 3 128 **D**: 3 072
1024	**G**: 3 072 **E**: 4 904 **D**: 4 920	**G**: 11 080 **E**: 17 360 **D**: 19 576	**G**: 3 176 **E**: 5 024 **D**: 5 024	**G**: 3 776 **E**: 3 744 **D**: 3 760	**G**: 4 360 **E**: 3 584 **D**: 3 592

[a] `https://github.com/mupq/pqm4/`, commit `be0c421aaecaad4443071bfcf62ad397d4f40832`.

Table 5: Time spent hashing. **G**: key generation, **E**: encapsulation, **D**: decapsulation.

Scheme	Dimension 512	Dimension 768	Dimension 1024
NewHope	**G**: 75% **E**: 80% **D**: 72%	-	**G**: 73% **E**: 78% **D**: 71%
NewHope-Compact	**G**: 75% **E**: 78% **D**: 67%	**G**: 72% **E**: 77% **D**: 66%	**G**: 73% **E**: 77% **D**: 66%
Kyber	**G**: 76% **E**: 80% **D**: 69%	**G**: 77% **E**: 80% **D**: 72%	**G**: 78% **E**: 80% **D**: 73%

4.3 Comparing Polynomial Multiplications

The reader might wonder why to bother optimizing polynomial multiplications further if it is not the bottleneck anymore. The reason is twofold: First, `Keccak` (`SHA-3`) is used to expand the seed in every scheme implemented. However, the choice of the seed expansion algorithm is somewhat orthogonal to the scheme and does not affect post-quantum assumptions. Hence, using a faster hash function would reduce the impact of hashing on the performance. Furthermore, it might be unnecessary to use a cryptographic hash function to generate the public parameters. For instance, [BFM$^+$18] uses a faster, non-cryptographic RNG to speed-up a scheme based on LWE. Second, even if `Keccak` is used, since its usage will likely grow in all future cryptographic applications, we might eventually see hardware acceleration for it on a lot of architectures. This would naturally drastically decrease the time spent hashing in our schemes and make the polynomial multiplication the most important optimization target again. Recall that, as stated in Section 3.2, this would increase the relative cost of the reduced stack usage trick used in the key generation. Nevertheless, we think that outside of unrealistically fast polynomial generation, the trade-off can still be useful.

Since our work is the first Cortex-M4 implementation of NewHope-Compact, we do not have any point of comparison for our technique for this scheme. Table 6 shows the speed-up for the dimension of the NTT used in all parameter sets of NewHope and Kyber and the cycle count of all subroutines of the polynomial multiplication for each algorithm and dimension. The total cost of multiplication operations for each scheme is presented in Table 7. This table was obtained by summing all the time spent in the three multiplication subroutines: NTT, NTT^{-1}, and $\circ$. Note that the stack optimized version of our implementation is used in this table to show its actual impact on the performance. It can be seen that Kyber and NewHope-Compact have similar performance, while NewHope is slower. This is mainly due to the extra layers of the NTT and the increased number of reductions caused by the larger modulus. Note that our NTT^{-1} cycles do not include any bitreversal operation, because we need NTT^{-1} to output a bitreversed order for the stack optimization (Section 3.2). To be able to verify test vectors with the reference implementation of NewHope, we have implemented a separate bitreversal operation that takes roughly $4n$-cycle for the selected parameter set, which is not included in the NTT^{-1}. It can be seen that our implementation of the NewHope NTT is slightly slower compared to the implementation from [KRSS], while we have noticeably better performance for the NTT^{-1}. Table 7 shows that even though we have a slower NTT, the total number of cycles spent in polynomial operations is reduced compared to [KRSS].

Table 6: Comparison of the polynomial multiplication functions of all the schemes. Kyber actually uses the exact same NTT code for all dimensions.

Scheme	Dimension	NTT	NTT^{-1}	$\circ$
NewHope	512	28662	22836	4736
	512 ([KRSS][a])	29767	35813	5459
	1024	63387	49880	9396
	1024 ([KRSS][a])	59752	71942	10836
	1024 ([AJS16])	86769	97340	14977
NewHope-Compact	512	12799	13052	7052
	768	19647	21226	12749
	1024	25536	26039	18510
Kyber	256	6847	6975	2317
	256 ([BKS19])	7754	9377	3076

[a] https://github.com/mupq/pqm4/, commit be0c421aaecaad4443071bfcf62ad397d4f40832.

Table 7: Total time spent in polynomial multiplication subroutines (NTT, NTT^{-1}, and $\circ$).

Scheme	Dimension	KeyGen	Encaps	Decaps
NewHope	512	84896	89632	117204
	512 ([KRSS][a])	64993	106265	147537
	1024	186050	195446	254722
	1024 ([KRSS][a])	130340	213118	295896
NewHope-Compact	512	45702	52754	72858
	768	73269	86018	119993
	1024	95621	114131	158680
Kyber	512	50606	48521	73824
	512 ([BKS19])	43320	62095	93132
	768	82860	76245	110712
	768 ([BKS19])	74208	97682	139549
	1024	119748	108603	152234
	1024 ([BKS19])	111248	139421	192118

[a] https://github.com/mupq/pqm4/, commit be0c421aaecaad4443071bfcf62ad397d4f40832.

5 Conclusion

In this work, we proposed several optimizations for {R,M}LWE schemes on the Cortex-M4. Among them, some are direct improvements over the current literature, while others are trade-offs that are up to the user of the scheme to deem needed or not. The core speed optimizations are due to a more aggressive layer-merging strategy and a minimization of the number of reductions in the base multiplication. Our implementation has already been integrated into the `pqm4` library. We also provide a comparison of the proposed trade-offs. The code is written in a modular fashion that allows the user to easily switch between versions.

Our results show that all optimization techniques have advantages and disadvantages and might be useful for different applications. Note that our default option includes only the stack usage optimization since our goal is having a fast implementation while using a stack space as small as possible. One interesting point is that our NTT implementation for NewHope has slightly slower performance than the one reported in [KRSS], suggesting that there is still room for improvement.

The time spent during polynomial operations has two main bottlenecks, namely modular reduction and memory access operations among layers of the NTT. This paper solves these bottlenecks by proposing a 2-cycle implementation of the Montgomery modular reduction and by increasing the number of merged layers. While this paper focuses on this strategy, keeping coefficients in 16-bit signed integers requires performing modular reduction regularly. It would be interesting to see if using 32-bit signed integers with proper modular reduction can improve the performance. This approach would require to store one coefficient per register, which means that fewer layers can be merged, but the number of modular reductions might be reduced since lazy reduction can be applied more aggressively. Such an implementation was recently proposed for the RISC-V architecture [AEL+20]. The authors propose an implementation which uses the Barrett reduction after both addition/subtraction and multiplication. Their reduction implementation can reduce 32-bit numbers, so it allows more aggressive use of lazy reduction at the cost of using more registers. The Cortex-M4 has a 32-bit multiplier. Hence, it might be interesting to evaluate the performance of that implementation on the Cortex-M4.

Acknowledgments

The authors would like to thank anonymous TCHES reviewers for their valuable comments on drafts of this paper. We are grateful to Peter Pessl and Ruben Niederhagen for helping us to improve our paper.

The work of EA was partially supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, and was partially carried out during his tenure of the ERCIM ‘Alain Bensoussan’ Fellowship Programme.

References

[AAB+19] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope - Algorithm Specifications And Supporting Documentation (version 1.03). Technical report, NIST Post-Quantum Cryptography Standardization Project, 2019. https://newhopecrypto.org/.

[ABC19] Erdem Alkim, Yusuf Alper Bilgin, and Murat Cenk. Compact and simple RLWE based key encapsulation mechanism. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*, volume 11774 of *Lecture Notes in Computer Science*, pages 237–256, Santiago de Chile, Chile, October 2–4 2019. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-030-30530-7_12.

[ABD$^+$19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER - Algorithm Specifications And Supporting Documentation (version 2.0). Technical report, NIST Post-Quantum Cryptography Standardization Project, 2019. https://pq-crystals.org/kyber/.

[ADPS16a] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. http://eprint.iacr.org/2016/1157.

[ADPS16b] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343, Austin, TX, USA, August 10–12 2016. USENIX Association. https://eprint.iacr.org/2015/1092.

[AEL$^+$20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic - accelerating Kyber and NewHope on RISC-V. Cryptology ePrint Archive, Report 2020/049, 2020. https://eprint.iacr.org/2020/049.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. NewHope on ARM Cortex-M. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 332–349. Springer, 2016. https://eprint.iacr.org/2016/758.

[ARM] ARM. ARM Cortex-M4. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986. https://doi.org/10.1007/3-540-47721-7_24.

[BDK$^+$18] Joppe W. 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 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018. https://eprint.iacr.org/2017/634.

[BFM$^+$18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! faster frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Report 2018/1116, 2018. https://eprint.iacr.org/2018/1116.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228, Rabat, Morocco, July 9–11 2019. Springer, Heidelberg, Germany. https://eprint.iacr.org/2019/489.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. http://www.jstor.org/stable/2003354.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_34.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. ACM. http://doi.acm.org/10.1145/1464291.1464352.

[KMRV18] Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. 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.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on cortex-M4 to speed up NIST PQC candidates. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301, Bogota, Colombia, June 5–7 2019. Springer, Heidelberg, Germany. https://eprint.iacr.org/2018/1018.

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

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010. https://doi.org/10.1007/978-3-642-13190-5_1.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013. https://doi.org/10.1145/2535925.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015. https://eprint.iacr.org/2012/090.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast ntru using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019. https://eprint.iacr.org/2019/040.

[Mon85] Peter L. Montgomery. 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.

[SBG+18] Markku-Juhani O. Saarinen, Sauvik Bhattacharya, Óscar García-Morchón, Ronald Rietman, Ludo Tolhuizen, and Zhenfei Zhang. Shorter messages and faster post-quantum encryption with Round5 on cortex M. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers.*, volume 11389 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2018. https://eprint.iacr.org/2018/723.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[STM] STMicroelectronics. STM32F4DISCOVERY kit. https://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[WP06] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006. http://eprint.iacr.org/2006/224.

[ZXZ+18] Shuai Zhou, Haiyang Xue, Daode Zhang, Kunpeng Wang, Xianhui Lu, Bao Li, and Jingnan He. Preprocess-then-NTT Technique and Its Applications to Kyber and NewHope. In Fuchun Guo, Xinyi Huang, and Moti Yung, editors, *Information Security and Cryptology - 14th International Conference, Inscrypt 2018, Fuzhou, China, December 14-17, 2018, Revised Selected Papers*, volume 11449 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 2018. https://eprint.iacr.org/2018/995.