Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates

Matthias J. Kannwischer, Joost Rijneveld and Peter Schwabe*

Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu, joost@joostrijneveld.nl, peter@cryptojedi.org

Abstract. In this paper we optimize multiplication of polynomials in $\mathbb{Z}_{2^m}[x]$ on the ARM Cortex-M4 microprocessor. We use these optimized multiplication routines to speed up the NIST post-quantum candidates RLizard, NTRU-HRSS, NTRUEncrypt, Saber, and Kindi. For most of those schemes the only previous implementation that executes on the Cortex-M4 is the reference implementation submitted to NIST; for some of those schemes our optimized software is more than factor of 20 faster. One of the schemes, namely Saber, has been optimized on the Cortex-M4 in a CHES 2018 paper; the multiplication routine for Saber we present here outperforms the multiplication from that paper by 37%, yielding speedups of 17% for key generation, 15% for encapsulation and 18% for decapsulation. Out of the five schemes optimized in this paper, the best performance for encapsulation and decapsulation is achieved by NTRU-HRSS. Specifically, encapsulation takes just over 430 000 cycles, which is more than twice as fast as for any other NIST candidate that has previously been optimized on the ARM Cortex-M4.

Keywords: ARM Cortex-M4, Karatsuba, Toom, lattice-based KEMs, NTRU

1 Introduction

In November 2017 the NIST post-quantum project [NIS16b] received 69 "complete and proper" proposals for future standardization of a suite of post-quantum cryptosystems. By September 2018, five of those 69 have been withdrawn. Out of the remaining 64 proposals, 22 are lattice-based public-key encryption schemes or key-encapsulation mechanisms (KEMs)[1]. Most of those lattice-based schemes use structured lattices and, as a consequence, require fast arithmetic in a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/f$ for some n-coefficient polynomial $f \in \mathbb{Z}_q[x]$. Typically the largest performance bottleneck of these schemes is multiplication in $\mathcal{R}_q$.

Many proposals, for example NewHope [ADPS16, AAB+17], Kyber [ABD+17], and LIMA [SAL+17], choose q, n, and f such that multiplication in $\mathcal{R}_q$ can be done via very fast number-theoretic transforms. However, six schemes choose $q = 2^k$ which requires using a different algorithm for multiplication in $\mathcal{R}_q$. Specifically those six schemes are Round2 [GMZB+17], Saber [DKRV17], NTRU-HRSS [HRSS17b], NTRUEncrypt [ZCHW17], Kindi [Ban17], and RLizard [CPL+17]. Round2 recently merged with Hila5 [Saa17] into Round5 [BGML+18] and the Round5 team presented optimized software for the ARM Cortex-M4 processor in [SBGM+18]; the multiplication in Round5 has more structure, allowing for a specialized high-speed routine. In this paper we optimize the other five

*This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE) and by COST (European Cooperation in Science and Technology) through COST Action IC1403 (CRYPTACUS). Date: October 19, 2018

[1]see https://www.safecrypto.eu/pqclounge/

schemes relying on arithmetic in $\mathcal{R}_q$ with a power-of-two q on the same platform. Note that Saber has previously been optimized on the ARM Cortex-M4 [KMRV18] as well; our polynomial multiplication implementation outperforms the results by 37% which improves the overall performance of key generation by 17%, encapsulation by 15%, and decapsulation by 18%. For the other four schemes the only software that was readily available for the Cortex-M4 was the reference implementation and, unsurprisingly, our carefully optimized code significantly outperforms these implementations. For example, our optimized implementations of RLizard-1024 and Kindi-256-3-4-2 encapsulation and decapsulation are more than a factor of 20 faster than the reference implementation. Our implementation of NTRU-HRSS encapsulation and decapsulation solidly outperform the optimized Round5 software presented in [SBGM+18], which calls into question the statement from [SBGM+18] that *"Round5 offers not only the shortest key and ciphertext sizes among lattice-based candidates, but also has leading performance and implementation size characteristics"*.

We achieve our results by systematically exploring different combinations of Toom-3, Toom-4, and Karatsuba decomposition of multiplication in $\mathcal{R}_q$, and by carefully hand-optimizing multiplication of low-degree polynomial multiplication at the bottom of the Toom/Karatsuba decomposition. The exploration of the different approaches is automated through a set of Python scripts that generate optimized assembly given the parameters $q = 2^k$ for $k \leq 16$ and $n \leq 1024$. These Python scripts may be of independent interest for a similar design-space exploration on different architectures.

Organization of this paper. In Section 2 we briefly recall the five NIST candidates that we optimize in this paper and give the necessary background on the target microarchitecture, i.e., the ARM Cortex-M4. In Section 3 we first detail our approach to explore different Toom and Karatsuba decomposition strategies for multiplication in $\mathcal{R}_q$ and then explain how we hand-optimized schoolbook multiplications of low-degree polynomials. Finally, Section 4 presents performance results for stand-alone multiplication in $\mathcal{R}_q$ for the different parameter sets, and for the five NIST candidates.

Availability of the software. We place all software presented in this paper, including the Python scripts used for design-space exploration, into the public domain. The software is available at `https://github.com/mupq/polymul-z2mx-m4` and the optimized implementations have been integrated into the `pqm4` framework [KRSS].

2 Preliminaries

In this section, we briefly review the five NIST candidates that we optimize in this paper. Readers interested in the multiplication routine outside the context of NIST submissions are encouraged to skip ahead to Subsection 2.2, where we introduce the targeted Cortex-M4 platform and give context that is relevant to interpret the benchmark results.

2.1 Cryptosystems targeted in this paper

Notation. The full specification of each of the five CCA-secure KEMs would take several pages, so for the sake of brevity we leave out various details. In particular, all five schemes build a CCA-secure KEM from an encryption scheme; for all but NTRUEncrypt, this encryption scheme is only passively secure. In our description we focus only on the encryption schemes underlying the KEM and highlight the multiplications in $\mathcal{R}_q$—the main target of our optimization effort—by denoting those multiplications with $\circledast$. In general, we denote scalar multiplications with $\cdot$ and polynomial multiplications with $*$.

Similarly, we do not go into any detail with respect to the sampling of random bit strings, polynomials, or matrices, and simply denote all of these functions as $\mathsf{Sample}_{\mathcal{R}}$,

Algorithm 1 RLizard.KeyGen ()

```
1: $a, s, e \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $b \leftarrow -a \circledast s + e \in \mathcal{R}_q$
3: return $(\mathsf{pk} = (a, b), \mathsf{sk} = s)$
```

Algorithm 2 RLizard.Enc $(m, (a, b))$

```
1: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $c'_1 \leftarrow a \circledast r \in \mathcal{R}_q$
3: $c'_2 \leftarrow b \circledast r \in \mathcal{R}_q$
4: $c_1 \leftarrow \lfloor (p/q) \cdot c'_1 \rceil \in \mathcal{R}_p$
5: $c_2 \leftarrow \lfloor (p/q) \cdot ((q/2) \cdot m + c'_2) \rceil \in \mathcal{R}_p$
6: return $(c_1, c_2)$
```

Algorithm 3 RLizard.Dec $((c_1, c_2), s)$

```
1: $m' \leftarrow \lfloor (2/p) \cdot (c_2 + c_1 \circledast s) \rceil \in \mathcal{R}_2$
2: return $m'$
```

where $\mathcal{R}$ is the set from which the elements are drawn. While we specify a set to which the sampled elements belong, we leave the distribution according to which they are sampled unspecified. Where deterministic sampling from a specific seed is relevant, $\mathsf{Sample}_{\mathcal{R}}$ is parameterized with this seed.

Finally, many schemes make use of rounding coefficients of polynomials. We denote any such rounding operation by $\lfloor \dots \rceil$, specify the domain in which the result lives, but again omit the details of how the rounding operation is defined.

2.1.1 RLizard

RLizard is part of the Lizard submission to NIST [CPL+17]. It is a cryptosystem based on the Ring-Learning-with-Errors (Ring-LWE) and Ring-Learning-with-Rounding (Ring-LWR) problems. As the names suggest, these problems are closely related, and efficient reductions exist [BPR12, BGM+16] that connect the two. The submission motivates the choice for the Learning-with-Rounding problem by stressing its deterministic encryption routine and reduced ciphertext size compared to Learning-with-Errors. RLizard.KEM is a CCA-secure KEM that is constructed by applying Dent's variant of the FO transform [FO99, Den03] to the RLizard CPA-secure public-key encryption scheme, which is summarized in Algorithms 1, 2, and 3.

The main structure underlying RLizard is the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, but coefficients of the ciphertext are ultimately reduced to $\mathcal{R}_p$, where $p < q$. We consider the parameter set where $n = 1024$, $q = 2048$ and $p = 512$. In the submission the derived KEM is referred to as `RING_CATEGORY3_N1024` – for clarity, we denote it as RLizard-1024 from this point onwards.

All multiplications that appear in Algorithms 1, 2, and 3 fit the structure that we target in this work. From a performance perspective, the remaining mathematical operations are practically negligible.

2.1.2 NTRU-HRSS-KEM

The NTRU-HRSS scheme [HRSS17a] is based on the 'classic' NTRU cryptosystem [HPS98]. It starts from the CPA-secure NTRU encryption scheme, and, like RLizard, applies Dent's variant of the FO transform [FO99, Den03] to construct a CCA-secure KEM. By restricting the parameter space compared to traditional NTRU, the scheme is simplified and avoids implementation pitfalls such as decryption failures and fixed-weight sampling.

We look at the concrete instance as submitted to NIST [HRSS17b], i.e., fix the parameters to $p = 3$, $q = 8192$ and $n = 701$. NTRU-HRSS relies on arithmetic in a number of different rings. Glossing over the technicalities (see Sections 2 and 3 of [HRSS17a]), we reuse the notation to define $\Phi_d = 1 + x^1 + x^2 + \dots + x^{d-1}$, and then define $\mathcal{R}_p = \mathbb{Z}[x]_p/\Phi_n$, $\mathcal{R}'_q = \mathbb{Z}[x]_q/\Phi_n$ and $\mathcal{R}_q = \mathbb{Z}[x]_q/(x^n - 1)$, but abstract away the transitions between rings.

Algorithm 4 NTRU-HRSS.KeyGen ()
```
1: $f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_p}$
2: $f_p^{-1} \leftarrow f^{-1} \in \mathcal{R}_p$
3: $f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}'_q$        $\triangleright$ Uses mult. in $\mathcal{R}_q$
4: $h \leftarrow \Phi_1 * g \circledast f_q^{-1} \in \mathcal{R}_q$
5: **return** $(\mathsf{pk} = p \cdot h, \mathsf{sk} = (f, f_p^{-1})$
```

Algorithm 5 NTRU-HRSS.Enc $(m, (p \cdot h))$
```
1: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $c \leftarrow h' \circledast r + m \in \mathcal{R}_q$
3: **return** $c$
```

Algorithm 6 NTRU-HRSS.Dec $\left(c, (f, f_p^{-1})\right)$
```
1: $v \leftarrow c \circledast f \in \mathcal{R}_q$
2: $m' \leftarrow v \circledast f_p^{-1} \in \mathcal{R}_p$
3: **return** $m$
```

Algorithms 4, 5, and 6 show that the scheme requires several multiplications and inversions. For this paper, the relevant operations are the multiplications in $\mathcal{R}'_q$ and $\mathcal{R}_q$. However, we can use the same routine to perform the multiplication in $\mathcal{R}_p$: as observed in [HRSS17a], the large difference between p and q ensures that this is possible without introducing errors (as the sum of coefficients modulo p never exceeds q). Furthermore, as the inversion in $\mathcal{R}'_q$ can be performed using multiplications [HRSS17a], this benefits from the same optimization.

2.1.3 NTRUEncrypt

The NTRUEncrypt scheme [ZCHW17] is also based on the standard NTRU construction [HPS98], but chooses parameters based on a recent revisiting [HPS⁺17]. NTRUEncrypt builds a CCA-secure KEM from a CCA-secure PKE; this public-key encryption scheme uses the NAEP transform [HGSSW03]. Note that the PKE description already includes re-encryption, which is typically introduced to decapsulation as part of the CCA transform and is thus not included for the other schemes.

The NIST submission of NTRUEncrypt [ZCHW17] presents several instantiations, but we limit ourselves to the instances where $q = 2^k$. We look at the parameter set NTRU-KEM-743, where $p = 3$, $q = 2048$, and $n = 743$; the arithmetic takes place in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1)$, but coefficients are also reduced modulo p when moving to $\mathcal{R}_p$. The optimizations in this work also carry over to the smaller NTRU-KEM-443 parameter set, but not to NTRU-KEM-1024 (which uses a prime q).

The full specification includes a padding mechanism that we omit for brevity. As before, the relevant multiplication occurs when the noise polynomial r is multiplied with the public key h, but we also utilize our multiplication routine for the other multiplication in Dec. See the algorithmic descriptions in Algorithm 7, 8, and 9, below.

2.1.4 Saber

Like Lizard and RLizard, Saber [DKRV17] also relies on the Learning-with-Rounding problem. Rather than directly targeting LWR or the ring variant, it positions itself in the middle-ground formed by the Module-LWR problem. The submission conforms to the common pattern of proposing a PKE scheme, and then applying an FO variant [HHK17] to obtain a CCA-secure KEM. We give descriptions of the PKE scheme in Algorithm 10, 11 and 12, below. for the context of this work only the types, and not the precise valuations, are relevant.

Algorithm 7 NTRUEncrypt.KeyGen ()

```
1: $f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
2: $h \leftarrow (p \cdot g)/(p \cdot f + 1) \mod q$
3: return $(\mathsf{pk} = h, \mathsf{sk} = (f, h))$
```

Algorithm 8 NTRUEncrypt.Enc (m, h)

```
1: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
2: $t \leftarrow r \circledast h$
3: $m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
4: $m' \leftarrow m - m_{mask} \mod p$
5: $c \leftarrow t + m'$
6: return $c$
```

Algorithm 9 NTRUEncrypt.Dec $(c, (f, h))$

```
1: $m' \leftarrow f \circledast c \mod p$
2: $t \leftarrow c - m$
3: $m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
4: $m \leftarrow m' + m_{mask} \mod p$
5: $r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
6: if $p \cdot r \circledast h = t$ then
7:     return $m$
8: else
9:     return $\perp$
10: end if
```

Algorithm 10 Saber.KeyGen ()

```
1: $\rho \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
2: $A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho)$
3: $s \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
4: $b \leftarrow \lfloor A \circledast s + h \rceil \in \mathcal{R}_p^{\ell}$
5: return $(\mathsf{pk} = (\rho, b), \mathsf{sk} = s)$
```

Algorithm 11 Saber.Enc $(m, (\rho, b))$

```
1: $A \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{l \times l}}(\rho)$
2: $s' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
3: $b' \leftarrow \lfloor A \circledast s' + h \rceil \in \mathcal{R}_p^{\ell}$
4: $v' \leftarrow b \circledast \lfloor s' \rceil \in \mathcal{R}_p$
5: $c_m \leftarrow \lfloor v' + (p/2) \cdot m \rceil \in \mathcal{R}_{2t}$
6: return $(c_m, b')$
```

Algorithm 12 Saber.Dec $((c_m, b'), s)$

```
1: $v \leftarrow b' \circledast \lfloor s \rceil \in \mathcal{R}_p$
2: $m' \leftarrow \lfloor v - (p/(2t)) \cdot c_m + h \rceil \in \mathcal{R}_2$
3: return $m'$
```

Like RLizard, Saber operates in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, and in the smaller $\mathcal{R}_p$. Because of the Module-LWR structure, however, n is fixed to 256 for all parameter sets. Instead of varying the dimension of the polynomial, Saber variants use matrices of varying sizes with entries in the polynomial ring (denoted $\mathcal{R}^{\ell \times k}$). With the fixed $q = 8192$, this ensures that an optimized routine for multiplication in $\mathcal{R}_q$ directly applies to the smaller LightSaber and the larger FireSaber instances as well. Other parameters p and t are powers of 2 smaller than q; for the Saber instance[2], $p = 1024$ and $t = 8$. The vector h is a fixed constant in $\mathcal{R}_q^{\ell}$.

Note that some of the multiplications in Saber are in $\mathcal{R}_q$ and some are in $\mathcal{R}_p$; in our software both use the same routine. As we will explain in Section 3, the smaller value of p would in principle allow us to explore a larger design space for multiplications in $\mathcal{R}_p$; however, for the small value of $n = 256$ there is nothing to be gained in the additional multiplication approaches.

[2]Note that both the scheme and the category 3 parameter set are called Saber.

2.1.5 KINDI

In the same vein as Saber, Kindi [Ban17] is based on a matrix of polynomials, relating it to the Module-LWE problem. Somewhat more intricate than the standard approach, however, it relies on a trapdoor construction, and constructs a CPA-secure PKE that is already close to a key-encapsulation mechanism.

Kindi operates in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ with $q = 2^k$, the more general $\mathcal{R}_b = \mathbb{Z}_b[x]/(x^n+1)$ for some integer b, and in the polynomial ring with integer coefficients $\mathcal{R} = \mathbb{Z}[x]/(x^n+1)$. The relevant arithmetic primarily happens in the ring $\mathcal{R}_q$, though, meaning that the performance of Kindi still considerably improves as a consequence of this work. We consider the parameter set Kindi-256-3-4-2, where $n = 256$ and $q = 2^{14}$.

In Algorithms 13, 14 and 15, we list the PKE. Here, $\mathsf{g} \in \mathcal{R}_q$ is a constant, $\ell = 3$, $p = 4$, and $[p] \in \mathcal{R}_q$ is constant with all coefficients equal to p. We omit public key compression and message encoding for ease of exposition. To obtain a CCA-secure KEM, a slightly simplified version of the modular FO variant [HHK17] is used: as Kindi exhibits a KEM-like structure and already includes re-encryption in Dec, this results in merely having to add hash-function calls.

Algorithm 13 Kindi.KeyGen()

```
1: $\mu \leftarrow \mathsf{Sample}_{\{0,1\}^{256}}$
2: $\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell\times\ell}}(\mu)$
3: $\mathsf{r}, \mathsf{r}' \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell}}$
4: $\mathsf{b} \leftarrow \mathsf{A} \circledast \mathsf{r} + \mathsf{r}'$
5: return $(\mathsf{pk} = (\mathsf{b}, \mu), \mathsf{sk} = (\mathsf{r}, \mathsf{b}, \mu))$
```

Algorithm 14 Kindi.Enc$(m, (\mathsf{b}, \mu))$

```
1: $s_1 \leftarrow \mathsf{Sample}_{\mathcal{R}_2}$
2: $\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell\times\ell}}(\mu)$
3: $\mathsf{p} \leftarrow b + \mathsf{g}$
4: $\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
5: $(s_2, \ldots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
6: $\mathsf{s} \leftarrow (s_1 + 2\cdot\bar{s_1} - [p], s_2 - [p], \ldots, s_\ell - [p]) \in \mathcal{R}_q^{\ell}$
7: $\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
8: $\mathsf{u} \leftarrow \bar{u} \oplus m$
9: $\mathsf{e} \leftarrow (\mathsf{u}_1 - [p], \ldots, \mathsf{u}_\ell - [p]) \in \mathcal{R}_q^{\ell}$
10: $e_{\ell+1} \leftarrow u_{\ell+1} - [p]$
11: $(\mathsf{c}, c_{\ell+1}) \leftarrow (\mathsf{A} \circledast \mathsf{s} + \mathsf{e}, \mathsf{p} \circledast \mathsf{s} + \mathsf{g}\cdot[p] + \mathsf{e}) \in \mathcal{R}_q^{\ell+1}$
12: return $(\mathsf{c}, c_{\ell+1})$
```

2.2 ARM Cortex-M4

Our target platform is the ARM Cortex-M4 which implements the 32-bit ARMv7E-M architecture. It has 16 general purpose registers of which 14 are freely usable by the developer. In contrast to smaller architectures like the Cortex-M3, the Cortex-M4 supports the DSP instructions `smuad`, `smuadx`, `smlad`, and `smladx`, which we use to significantly speed up low-degree polynomial multiplication using the schoolbook method. Those low-degree multiplication routines are used as a core building block for higher-degree

Algorithm 15 $\mathsf{Kindi.Dec}(\mathsf{r}, \mathsf{b}, \mu, (\mathsf{c}, c_{\ell+1}))$

```
1: $\mathsf{A} \leftarrow \mathsf{Sample}_{\mathcal{R}_q^{\ell\times\ell}}(\mu)$
2: $\mathsf{p} \leftarrow \mathsf{b} + \mathsf{g}$
3: $\mathsf{v} \leftarrow c_{\ell+1} - \mathsf{c} \circledast \mathsf{r}$
4: $s_1 \leftarrow (\lfloor \mathsf{v}_1/2^{\log q-1}\rceil, \ldots, \lfloor \mathsf{v}_n/2^{\log q-1}\rceil) \in \mathcal{R}_2$
5: $\bar{s_1} \leftarrow \mathsf{Sample}_{\mathcal{R}_p}(s_1)$
6: $(s_2, \ldots, s_\ell) \leftarrow \mathsf{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)$
7: $\mathsf{s} \leftarrow (s_1 + 2\cdot\bar{s_1} - [p], s_2 - [p], \ldots, s_\ell - [p])$
8: $\bar{u} \leftarrow \mathsf{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)$
9: $(\mathsf{e}, e_{\ell+1}) \leftarrow (\mathsf{c} - \mathsf{A} \circledast \mathsf{s}, c_{\ell+1} - \mathsf{p} \circledast \mathsf{s}) \in \mathcal{R}_q^{\ell+1}$
10: $\mathsf{u} \leftarrow (\mathsf{e}_1 + [p], \ldots \mathsf{e}_\ell + [p])$
11: $u_{\ell+1} \leftarrow e_{\ell+1} + [p]$
12: $m \leftarrow \mathsf{u} \oplus \bar{u}$
13: **return** $m$
```

polynomial multiplication. The DSP instructions perform two half-word multiplications, accumulate the two products and optionally accumulate another 32-bit word in one clock cycle (as illustrated in Table 1).

There is strong synergy between these DSP instructions and the fact that loading a 32-bit word using `ldr` is as expensive as loading a halfword using `ldrh`. Related to this, it is important to perform load operations sequentially (i.e., uninterrupted by other instructions) when possible, as this has a pipelining benefit. This shows in the `ldm` instruction, but also when simply adjoining multiple `ldr` instructions. While the same behavior occurs for store instructions, combining loads and stores only incurs pipelining benefits when stores follow loads, but not when loads follow stores.

The ARMv7E-M instruction set contains support for the so-called Thumb instructions subset, describing operations that can be encoded in 16 bits, such as simple arithmetic and memory operations with register parameters. Using these instructions has an obvious benefit for code size, but comes at the cost of introducing misalignment: instruction fetching is significantly more expensive when instruction offsets are not aligned to multiples of four bytes. To combat this, Thumb instructions can be optionally expanded to full-word width using the `.w` suffix.

Benchmarking platform. For our experiments we use the STM32F4DISCOVERY which features 1 MiB of Flash ROM, 192 KiB of RAM (128 KiB of which are contiguous) running at a maximum frequency of 168 MHz. For benchmarking we use the reduced clock frequency of 24 MHz to not be impacted by wait states caused by slow memory [SS17]. We use the GNU ARM Embedded Toolchain[3] (`arm-none-eabi`) with `arm-none-eabi-gcc-8.2.0`. All source files are compiled with the optimization flag `-O3`.

Table 1: Relevant dual 16-bit multiplication instructions supported by the ARM Cortex-M4

instruction	semantics
`smuad Ra, Rb, Rc`	$\mathtt{Ra} \leftarrow \mathtt{Rb_L} \cdot \mathtt{Rc_L} + \mathtt{Rb_H} \cdot \mathtt{Rc_H}$
`smuadx Ra, Rb, Rc`	$\mathtt{Ra} \leftarrow \mathtt{Rb_L} \cdot \mathtt{Rc_H} + \mathtt{Rb_H} \cdot \mathtt{Rc_L}$
`smlad Ra, Rb, Rc, Rd`	$\mathtt{Ra} \leftarrow \mathtt{Rb_L} \cdot \mathtt{Rc_L} + \mathtt{Rb_H} \cdot \mathtt{Rc_H} + \mathtt{Rd}$
`smladx Ra, Rb, Rc, Rd`	$\mathtt{Ra} \leftarrow \mathtt{Rb_L} \cdot \mathtt{Rc_H} + \mathtt{Rb_H} \cdot \mathtt{Rc_L} + \mathtt{Rd}$

[3]https://developer.arm.com/open-source/gnu-toolchain/gnu-rm

3 Multiplication in $\mathbb{Z}_{2^m}[x]$

As discussed in the previous sections, we focus on multiplication in $\mathcal{R}_q$, where $q = 2^m$. In particular, we approach this by looking at the non-reduced multiplication in $\mathbb{Z}_{2^m}[x]$, as this is identical across all schemes we investigate. The reduction is then done outside of our optimized polynomial multiplication in plain C.

In this section, we describe the way we break down such a multiplication for a specific number of coefficients n, modulo a specific q. This is done using combinations of Toom-Cook's and Karatsuba's multiplication algorithms, which we briefly discuss below. For a given n and q, there are multiple possible approaches; we explore the entire space and select the optimum for each parameter set. To do this, we use Python scripts that generate carefully optimized dedicated assembly functions for all combinations, generalizing this to arbitrary-degree polynomials (with degree below 1024). The main generation script is parameterized by the degree, the Toom method (see the next subsection; Toom-3, Toom-4, both Toom-4 and Toom-3 or no Toom layer at all), and the threshold at which to switch from Karatsuba to schoolbook multiplication. See Section 4.1 for a detailed analysis of these results, and the `README.md` file included as part of the supplied code package for more details on the usage of the code generation scripts.

3.1 Toom/Karatsuba strategies

The naive schoolbook approach to multiply of two polynomials with n coefficients results in n^2 multiplications in $\mathbb{Z}_q$. For large n, this is quite a very costly method.

Instead, using well-known algorithms by Karatsuba [KO63] and, more generally, Toom-Cook [Too63, Coo66], it is possible to trade some of these multiplications for additions and subtractions. Both algorithms have originally been introduced for the multiplication of large integers, but straight-forwardly translate to polynomial multiplication. Karatsuba's method breaks a multiplication of n-coefficient polynomials into three (instead of four) multiplications of polynomials with $\frac{n}{2}$ coefficients. Toom-Cook is a generalization of this approach. For this work we concern ourselves with Toom-3, which breaks down a multiplication of n-coefficient polynomials into five (rather than nine) multiplications of polynomials with $\frac{n}{3}$ coefficients, and Toom-4, which breaks down a multiplication of n-coefficient polynomials into seven multiplications of $\frac{n}{4}$-coefficient multiplications.

While asymptotically Toom-4 is more efficient than Toom-3 (which, in turn, is more efficient than Karatsuba), in practice the additions and subtractions also impact the runtime. Furthermore, the increased and more complex memory-access patterns significantly influence performance. Thus, for a given n it is not immediately obvious in general which approach is the fastest. We first evaluate whether to decompose using a layer of Toom-4, Toom-3, both Toom-4 and Toom-3, or no Toom at all. We then repeatedly apply Karatsuba's method to break down the multiplications, up to the threshold at which it becomes inefficient and the "naive" schoolbook method becomes the fastest approach.

Toom-Cook. In order to evaluate the optimal Toom strategy for a given n, we implement Toom-4 and Toom-3. Both Toom-4 and Toom-3 work by splitting the operands into polynomials of four or three coefficients (which, in our setting, are themselves polynomials), evaluating these polynomials at a certain points, performing point-wise multiplication, and interpolating the resulting polynomial. As an example, we consider Toom-3. We first split the input polynomial $A(X)$ into $A(Y) = A_2Y^2 + A_1Y + A_0$. Each of these limbs A_i then has $\frac{n}{3}$ coefficients. We refer to the the product of $A(Y)$ and $B(Y)$ as $C(Y)$, which consists of five limbs ($C(Y) = C_4Y^4 + C_3Y^3 + C_2Y^2 + C_1Y + C_0$). This resulting polynomial C is found by evaluating $A(Y)$ and $B(Y)$ at five points, multiplying the results, and then interpolating $C(Y)$ from these products. The interpolation step consists of solving a system of linear equations for the five unknown limbs C_i.

Algorithm 16 Toom-3 Interpolation

```
1: $C_0 = C(0)$
2: $C_4 = C(\infty)$
3: $V_3 = (C(-2) - C(1))/3$
4: $V_1 = (C(1) - C(-1)/2$
5: $V_2 = (C(-1) - C(0)$
6: $C_3 = (V_2 - V_3)/2 + 2 \cdot C_4$
7: $C_2 = V_2 + V_1 - C_4$
8: $C_1 = V1 - C_3$
```

Algorithm 17 Toom-4 Interpolation

```
1: $C_0 = C(0)$
2: $C_6 = C(\infty)$
3: $V_0 = (C(1) + C(-1))/2 - C_0 - C_6$
4: $V_1 = (C(2) + C(-2) - 2 \cdot C_0 - 128 \cdot C_6)/8$
5: $C_4 = (V_1 - V_0)/3$
6: $C_2 = (V_0 - C_4)$
7: $V_0 = (C(1) - C(-1))/2$
8: $V_1 = ((C(2) - C(-2))/4 - V_0)/3$
9: $V_2 = C(3) - C_0 - 9 \cdot C_2 - 81 \cdot C_4 - 729 \cdot C_6$
10: $V_2 = (V_2 - V_0)/8 - V_1$
11: $C_5 = V_2/5$
12: $C_3 = V_1 - V_2$
13: $C_1 = V_0 - C_3 - C_5$
```

For Toom-3, we use the evaluation points $\{0, 1, -1, -2, \infty\}$. Note that each C_i has $2 \cdot \frac{n}{3} - 1$ coefficients and, thus, the C_i's are convoluted to obtain $C(X)$. Toom-4 works in a similar fashion, but with four limbs and, thus, seven resultant limbs. We use the evaluation points $\{0, 1, -1, 2, -2, 3, \infty\}$.

The interpolation sequence used for Toom-3 and Toom-4 is illustrated in Algorithm 16 and Algorithm 17: it defines how to obtain the limbs C_i from the results of the smaller point-wise multiplications. We implement both evaluation and interpolation completely unrolled and make extensive use of parallel 16-bit instructions. For the sake of simplicity, we restrict our implementation to even limb sizes; this is the case for any n that is relevant in this work. In case n is not divisible by three (or, respectively, four) the most significant limb is padded with zeros.

It is important to note that there is a loss in precision when using Toom's method, since the interpolation involves divisions. While divisions by three and five can be replaced by multiplications by their inverses modulo 2^{16}, i.e., 43691 and 52429, this is not possible for divisions by powers of two. Due to the division by two in line 6 of Algorithm 16, Toom-3 loses one bit of precision. The division by eight in line 10 of Algorithm 17 leads to a loss of three bits of precision in Toom-4. Since our Karatsuba and schoolbook implementations operate in $\mathbb{Z}_{2^{16}}[x]$, this imposes constraints on the values of q for which our implementations can be used. Toom-3 can be used for $q \le 2^{15}$, Toom-4 can be used for $q \le 2^{13}$ and a combination of both is only possible if $q \le 2^{12}$. This also prevents us from using higher-order Toom methods for any cryptographically relevant q, since they result in even more loss of precision. While switching to 32-bit arithmetic would allow using higher order Toom, this would slow down Karatsuba and schoolbook multiplication significantly, since they would increase load-store overhead and would no longer be able to make use of DSP instructions. Therefore, we restrict our analysis to Toom-4 and Toom-3.

Karatsuba. Karatsuba's algorithm can be seen as a specific instance of the Toom-Cook algorithm described above, but is often considered separately for ease of exposition.

The operands A and B are split into two parts: $A = A_1 x^{n/2} + A_0$ and $B = B_1 x^{n/2} + B_0$. The product C can be split similarly, as $C = C_2 x^n + C_1 x^{n/2} + C_0$. The subproducts $C_0 = A_0 * B_0$ and $C_2 = A_1 * B_1$ follow immediately, but rather than computing $C_1 = (A_0 * B_1) + (A_1 * B_0)$, one computes the equivalent $C_1 = (A_0 + B_0) * (B_0 + B_1) - C_0 - C_2$. This saves a multiplication, at the cost of several extra additions. Note that careful consideration of these polynomial additions can save several individual coefficient additions, as some resulting terms are guaranteed to be zero.

The call to the topmost Karatsuba layer is implemented as a function call, but from that point on, we recursively inline the separate layers. Upon reaching the threshold at which the schoolbook approach takes precedence, we adhere to a self-imposed ABI of passing source and destination pointers, and call one of the schoolbook approaches discussed below as a function. This provides a trade-off that keeps code size reasonable and is flexible to implement and experiment with, but does imply that the register allocation between the final Karatsuba layer and the underlying schoolbook is disjoint; it may prove worthwhile to look into this for specific n rather than in a general approach.

In particular because we perform several nested layers of Karatsuba multiplication, it is important to carefully manage memory usage. We do not go for a completely in-place approach (as is done in [KMRV18]), but instead allocate stack space for the sums of the respective high and low limbs, as well as C_1, relying on the input and output buffers for all other terms. This leads to effective memory usage without reducing performance.

Assembly-level optimizations. For both Toom and Karatsuba, the typical operations require adding and subtracting polynomials of moderate size from a given address. We stress the importance of careful pipelining, loading and storing 16-bit coefficients pairwise into full-word registers, and using `uadd16` and `usub16` arithmetic operations. For memory operations, we rely on offset-based instructions, in particular for the more intricate memory access patterns in the interpolation step. This leads to a slight increase in code size compared to using `ldm` and `stm`, (and some bookkeeping for polynomials exceeding the maximal offset of 4095 bytes), but ensures that all address computations are done during code generation.

Like in Toom, several steps in Karatsuba's algorithm require adding or subtracting polynomials. Unlike Toom, we do not restrict the dimensions at all: the implementation can work on unbalanced splits, and thus polynomials of unequal length. To not waste any memory or cycles here (e.g., by applying common refinement approaches), the Python script becomes a rather complex composition of conditionals; rather than trying to combine pairs of 16-bit additions into `uadd16` operations on the fly, we run a post-processing step over the scheduled instructions to do so.

As mentioned in Section 2.2, instruction alignment is important when using 16-bit instructions. Rather than taking this into consideration during code generation, manually ensuring that 32-bit instructions only start at a word bound, we also use a post-processing step for this. After compilation, we disassemble the resulting binary and expand Thumb instructions in the cases where they cause misalignment. This allows us to still use the smaller Thumb instructions where possible, but avoids paying the overhead of misalignment. In particular, this is important when an odd number of Thumb instructions is followed by a large block of 32-bit instructions. This applies to the Toom and Karatsuba routines described above, but is equally important for the schoolbook multiplications described below. The alignment post-processing is done using a Python script that is included in our software package, and may be of independent interest.

3.2 Small schoolbook multiplications

Underlying the breakdowns achieved by Karatsuba and Toom-Cook layers, the actual multiplications are performed using schoolbook multiplications of small degree. We carefully investigate several approaches to do this, varying the approaches and implementing distinct generation routines for different n.

As discussed in Section 2.2, we effectively have 14 32-bit registers to work with. For each approach, we keep the polynomial in packed representation, loading all coefficients into the 32-bit registers in pairs. The ARMv7E-M instruction set provides a number of multiplication instructions that efficiently operate on data in this format: both parallel multiplications as well as instructions that operate on a specific combination of high and low halfwords. For $n \leq 10$, all input coefficients can be kept in registers simultaneously, even with registers

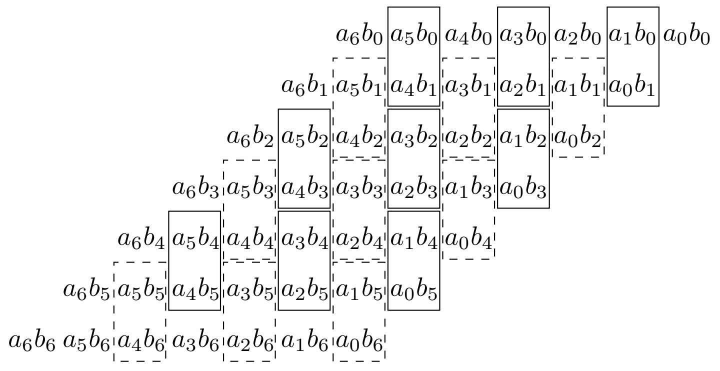
a_6b_0 a_5b_0 a_4b_0 a_3b_0 a_2b_0 a_1b_0 a_0b_0
a_6b_1 a_5b_1 a_4b_1 a_3b_1 a_2b_1 a_1b_1 a_0b_1
a_6b_2 a_5b_2 a_4b_2 a_3b_2 a_2b_2 a_1b_2 a_0b_2
a_6b_3 a_5b_3 a_4b_3 a_3b_3 a_2b_3 a_1b_3 a_0b_3
a_6b_4 a_5b_4 a_4b_4 a_3b_4 a_2b_4 a_1b_4 a_0b_4
a_6b_5 a_5b_5 a_4b_5 a_3b_5 a_2b_5 a_1b_5 a_0b_5
a_6b_6 a_5b_6 a_4b_6 a_3b_6 a_2b_6 a_1b_6 a_0b_6

Figure 1: Pairing coefficients for `smladx` / `smlad` instructions

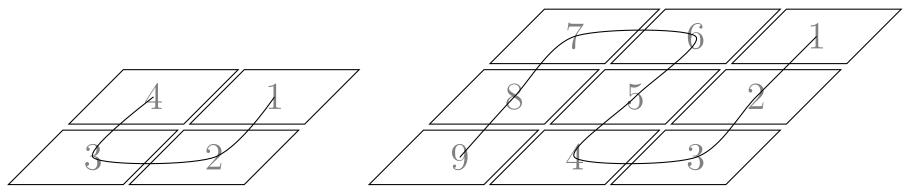

Figure 2: Decomposing larger schoolbook multiplications

remaining to keep the pointers to the source and destination polynomials around. We first compute all coefficients of terms with odd exponents, before using `pkh` instructions to repack one of the input polynomials and computing the remaining coefficients. This ensures that the vast majority of the multiplications can be computed using the two-way parallel multiply-accumulate dual instructions. See Figure 1 for an illustration of this; here, b is repacked to create the dashed pairs. This is somewhat similar to the approach used in [KMRV18], but ends up needing less repacking and memory interaction.

For $n \in \{11, 12\}$, we spill the source pointers to the stack after loading the complete polynomials. For these dimensions the registers are used to their full potential, and by using the DSP instructions we end up needing only 78 multiplications; 66 combined multiplications, 12 single multiplications, and not one dedicated addition instruction. This offsets the extra cost of the 6 packing instructions by a considerable margin. For $n \in \{13, 14\}$, not all coefficients fit in registers at the same time, leading to carefully optimized spill code for the middle columns (i.e., the computation of coefficients around x^n, which are affected by all input coefficients). Manual register allocation becomes somewhat tedious in the cases that involve many spills to the stack (even though writing Python generation scripts allows for convenient renaming). To remedy this, we use bare-bones register allocation functions akin to the scripts in [HRSS17a].

For larger n, the above strategy leads to an excessive amount of register spills. Instead, we compose the multiplication out of a grid of smaller schoolbook multiplications. For $15 \leq n \leq 24$, we compose the multiplication out of four schoolbook multiplications, for $25 \leq n \leq 36$, we use a grid of nine multiplications, etc. Note that we use at most $n = 12$ for the building blocks, given the extra complexity and overhead of the register spills for $n \in \{13, 14\}$. We further remark that it is important to carefully schedule the (re)loading and repacking of input polynomials. We illustrate this in Figure 2.

The approach described above works trivially when n is divisible by $\lceil \frac{n}{12} \rceil$, but leads to a less symmetric pattern for other dimensions. We plug these holes by starting from an n that divides even, and either adding a layer 'around' the parallelogram or nullifying the superfluous operations in a post-processing step.

Table 2: Benchmarks for small schoolbook multiplication routines. The cycle counts include an overhead of approximately 50 cycles for benchmarking.

n	cycles	n	cycles	n	cycles	n	cycles
1	60	13	247	25	996	37	2 112
2	66	14	268	26	1 164	38	2 113
3	72	15	359	27	1 160	39	2 106
4	79	16	360	28	1 287	40	2 107
5	89	17	506	29	1 283	41	2 478
6	97	18	503	30	1 285	42	2 863
7	110	19	549	31	1 350	43	2 868
8	117	20	551	32	1 351	44	2 864
9	135	21	679	33	1 569	45	3 038
10	144	22	677	34	1 701	46	3 039
11	175	23	725	35	1 697	47	3 032
12	184	24	727	36	1 699	48	3 033

See Table 2 and Figure 3 for an overview of the performance of these routines.

4 Results and discussion

In this section we present benchmark results for polynomial multiplication, and for key generation, encapsulation, and decapsulation of the five NIST post-quantum candidates Kindi, NTRUEncrypt, NTRU-HRSS, RLizard, and Saber. For each of the schemes we have tried to select the parameter set which targets NIST security category 3. However, NTRU-HRSS only provides a category 1 parameter set, hence we use this. Furthermore, the reference implementations for the category 3 parameter sets of Kindi require more than 128 KiB of RAM and consequently do not trivially fit our platform (STM32F4DISCOVERY). We use Kindi-256-3-4-2 instead, which targets security category 1. For the definition of NIST security categories see [NIS16a, Sec. 4.A.5].

All cycle counts presented in this section were obtained by using an adapted version of the `pqm4` benchmarking framework [KRSS][4], which uses the built-in 24-bit hardware timer. Stack measurements were also also obtained using the method implemented in `pqm4`, i.e., by writing a canary to the entire memory available for the stack, running the scheme under test and subsequently checking how much of the canary was overwritten.

All benchmarks of polynomial multiplication are multiplications in $\mathbb{Z}_q[x]$, i.e., exclude the reduction which is done outside of our optimized code in C. Moreover, the stack measurements exclude stack space required to store the operands and the result.

4.1 Multiplication results

We first present results for polynomial multiplication as a building block. We report benchmarks for the multiplication for all possible $n < 1024$, using different approaches to evaluate which strategy is optimal.

Figure 3 shows the run-time of our hand-optimized schoolbook implementations and the generated optimized Karatsuba code for small n. For the Karatsuba benchmarks, we have selected the optimal schoolbook threshold, e.g., for $n = 32$ one could either apply one layer of Karatsuba and then use the schoolbook method for $n = 16$ or, alternatively, use two layers of Karatsuba and use schoolbook multiplications for $n = 8$. The former

[4]A copy of the relevant parts of `pqm4` is included in the software package accompanying this paper.

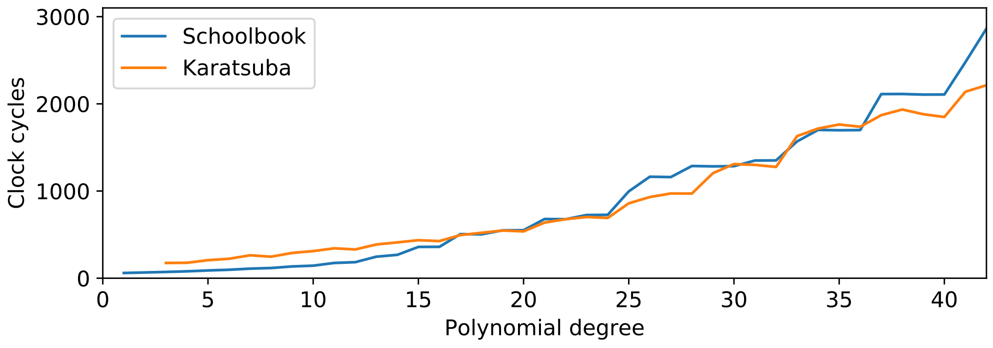

Figure 3: Runtime of generated optimized polynomial multiplication for small n. For $n < 20$ our hand-optimized schoolbook multiplications are clearly superior, for $n > 36$ first applying at least one layer of Karatsuba is faster.

variant is faster in this scenario, which leads to a schoolbook threshold of 16. For each n, we simply iterated over all schoolbook thresholds and selected the fastest variant. The graph shows that directly applying the schoolbook method is superior for $n < 20$, and for $n > 36$ Karatsuba outperforms schoolbook. However, for values in between, the plot is inconclusive. A large cause of this is the amount of hand-optimization that went into some of our schoolbook implementations, but it is also strongly determined by register pressure: there is a large performance hit in the step from $n = 14$ to $n = 15$, which then propagates to dimensions that break down to these schoolbook multiplications using Karatsuba. For cryptographically relevant values we found that the cross-over point is at $n = 22$, i.e., for values $n > 22$ one should use an additional layer of Karatsuba.

Figure 4 shows the performance of the different multiplication approaches for larger n. From a theoretical perspective, one might expect a cleaner line with clearer intersection points: initially the overhead of larger Toom variants is prohibitive, but as n increases it should overtake the lower-degree options, and prove optimal in the limit. While that general trend is visible, one still observes a jagged line. We speculate that the main cause for this is similar to the irregularities in Figure 3: the variance in the increasing cost of the schoolbooks is magnified as n grows larger and specific schoolbook sizes are repeated in the decomposition of large multiplications. Because of the difference in decomposition between Toom-3 and Toom-4, this favors each method for different ranges for n, resulting in alternating optimality. Another factor that is impacted by specific decomposition is the resulting memory access pattern, and, by extension, data alignment, resulting in a large performance penalty. In practice, comparing benchmarks for specific n seems to be the only way to come to conclusive results. In particular, we observe that the lines are not even monotonically increasing; we did not consolidate the results, but note that it is trivially possible to pad a smaller-degree polynomial and use a larger multiplication routine to benefit of a more efficient decomposition.

As Figure 4 does not allow us to identity which method performs best for clear bounds on n, we instead focus on individual n as relevant for the five cryptographic schemes we intend to cover. This restricts n to $\{256, 701, 743, 1024\}$. In Table 3, we report the cycle counts alongside the required additional stack space for each of the multiplication methods. All cycle counts are for polynomial multiplication *excluding* subsequent reduction required to obtain an n-coefficient polynomial; additional cost for reduction differs depending on the specific choice of ring[5]. For the rather small $n = 256$ (Saber, Kindi), we already

[5]A slightly more tailored multiplication routine can potentially reduce memory usage by taking the

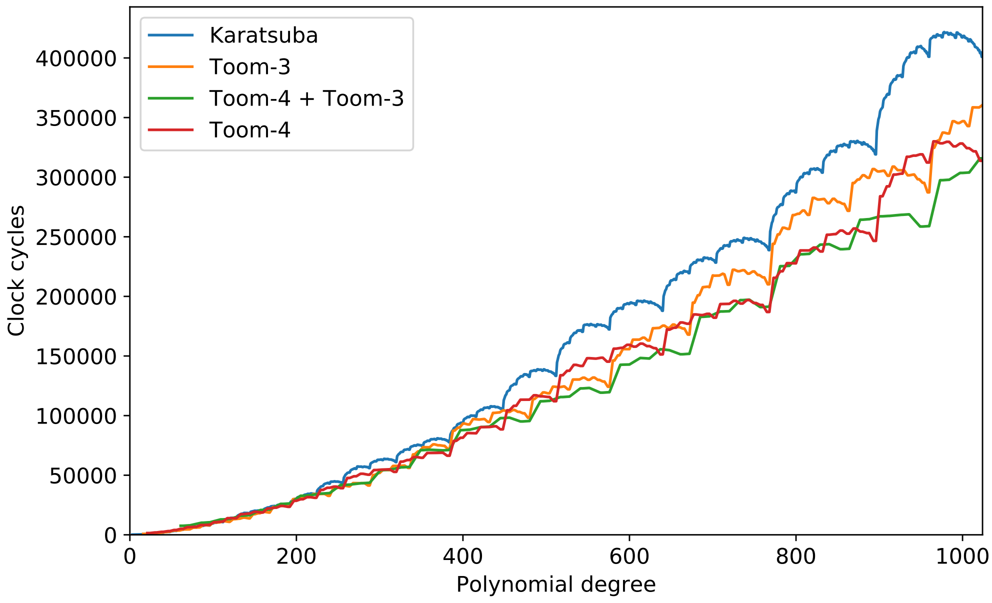

Figure 4: Runtime of different decomposition variants for large-degree multiplications.

see that Toom-4 (followed by two layers of Karatsuba) is slightly faster than directly applying Karatsuba. As the difference is small, however, one might decide to not use a Toom layer at all, at the benefit of a much simpler implementation and considerably reduced stack usage. Toom-4 is not suitable for Kindi ($n = 256, q = 2^{14}$), as q is too large. Again the impact is marginal, though, as Karatsuba is only a few percent slower at this dimension, also performing just above Toom-3. For larger $n \in \{701, 743, 1024\}$ (NTRU-HRSS, NTRUEncrypt,RLizard) applying Toom-4 is most efficient. The second layer ends up in the same range of small n, where it is a close competition between applying Toom-3 or directly switching to recursive Karatsuba.

4.2 Encapsulation and decapsulation results

In this section we present our performance results for RLizard, Saber, Kindi, NTRUEncrypt, and NTRU-HRSS. All the software presented in this section started from the reference implementations submitted to NIST but went considerably further than just replacing the multiplication routines with the optimized routines described in Section 3. For Saber, we considered starting from the already optimized implementation by Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede [KMRV18], but achieved marginally better performance starting from the reference code. We start by describing the changes that apply to the reference implementations; some of these changes might be more generally advisable as updates to reference software.

Memory allocations. The reference implementations of Kindi, RLizard, and NTRUEncrypt make use of dynamic memory allocation on the heap. The RLizard implementation does not free all the allocated memory, which results in memory leaks; also it misinterprets the NIST API and assumes that the public key is always stored right behind the secret key. This may result in reads from uninitialized (or even unallocated) memory. Luckily none

ring structure into account when performing the interpolation step in Toom-Cook.

Table 3: Benchmarks for polynomial multiplication excluding reduction. Fastest approach is highlighted in **bold**. The 'Toom-4 + Toom-3' and 'Toom-4' approaches are not applicable to all parameter sets, as q may be too large.

	approach	schoolbook	clock cycles	stack
Saber ($n = 256,\ q = 2^{13}$)	Karatsuba only	16	41 121	2 020
	Toom-3	11	41 225	3 480
	Toom-4	**16**	**39 124**	**3 800**
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256,\ q = 2^{14}$)	**Karatsuba only**	**16**	**41 121**	**2 020**
	Toom-3	11	41 225	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701,\ q = 2^{13}$)	Karatsuba only	11	230 132	5 676
	Toom-3	15	217 436	9 384
	Toom-4	**11**	**182 129**	**10 596**
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743,\ q = 2^{11}$)	Karatsuba only	12	247 489	6 012
	Toom-3	16	219 061	9 920
	Toom-4	**12**	**196 940**	**11 208**
	Toom-4 + Toom-3	16	197 227	12 152
RLizard-1024 ($n = 1024,\ q = 2^{11}$)	Karatsuba only	16	400 810	8 188
	Toom-3	11	360 589	13 756
	Toom-4	**16**	**313 744**	**15 344**
	Toom-4 + Toom-3	11	315 788	16 816

of the implementations *require* dynamically allocated memory; the size of all allocated memory is reasonably small and known at compile time. We eliminated all dynamic memory allocations and our software thus only relies on the stack to store temporary data. Our benchmarks show that this significantly improves performance, and in more unpredictable environments than bare-metal microcontrollers it makes error handling considerably easier. All resulting C implementations —compiled for an AMD64 desktop computer with `gcc-6.3` and optimization flag `-O3`—pass the `valgrind` memory checker without errors.

Hashing. Aside from arithmetic in $\mathcal{R}_q$, performance of most lattice-based KEMs is largely influenced by performance of hashing and randomness generation. The five NIST candidates we optimize in this paper make use of variants of SHA-3 and SHAKE [NIS15b] and of SHA-512 [NIS15a]. For SHA-3 and SHAKE we use the optimized assembly implementation from `pqm4` [KRSS], which makes use of the optimized Keccak-permutation from the Keccak Code Package [DHP+]. For SHA-512, we include our own optimized implementation, which may be of independent interest, but end up using a C implementation from SUPERCOP [BL]: while our implementation outperforms the compiled C code by a factor two when compiling with `arm-none-eabi-gcc 5.4.1` (as included in Debian 9), the more recent `arm-none-eabi-gcc 8.2.0` achieves identical performance for the C implementation. This is a testament to remarkable progress in compiler optimization and the importance of using recent compiler versions when replicating and comparing results.

Randomness generation. Submissions to NIST are allowed (and requested) to obtain randomness through a call to the externally supplied `randombytes` function. We adopt the approach of `pqm4` [KRSS] and implement this call to the hardware RNG on the STM32F4Discovery.

Table 4: Benchmarks for reference implementations and optimized implementations using fastest multiplication approach. Reporting run time (cycle count) and stack usage (bytes) for key generation (K), encapsulation (E), and decapsulation (D).

KEMs optimized in this paper					
	implementation	**clock cycles**		**stack usage**	
Saber	Reference	**K:**	6 530k	**K:**	12 616
		E:	8 684k	**E:**	14 896
		D:	10 581k	**D:**	15 992
	[KMRV18]	**K:**	1 147k	**K:**	13 883
		E:	1 444k	**E:**	16 667
		D:	1 543k	**D:**	17 763
	This work	**K:**	949k	**K:**	13 248
		E:	1 232k	**E:**	15 528
		D:	1 260k	**D:**	16 624
Kindi-256-3-4-2	Reference	**K:**	21 794k	**K:**	59 864
		E:	28 176k	**E:**	71 000
		D:	37 129k	**D:**	84 096
	This work	**K:**	1 010k	**K:**	44 264
		E:	1 365k	**E:**	55 392
		D:	1 563k	**D:**	64 376
NTRU-HRSS	Reference	**K:**	205 156k	**K:**	10 020
		E:	5 166k	**E:**	8 956
		D:	15 067k	**D:**	10 204
	This work	**K:**	161 790k	**K:**	23 396
		E:	432k	**E:**	19 492
		D:	863k	**D:**	22 140
NTRU-KEM-743	Reference	**K:**	59 815k	**K:**	14 148
		E:	7 540k	**E:**	13 372
		D:	14 229k	**D:**	18 036
	This work	**K:**	5 663k	**K:**	25 320
		E:	1 655k	**E:**	23 808
		D:	1 904k	**D:**	28 472
RLizard-1024	Reference	**K:**	26 423k	**K:**	4 272
		E:	32 156k	**E:**	10 532
		D:	53 181k	**D:**	12 636
	This work	**K:**	537k	**K:**	27 720
		E:	1 358k	**E:**	33 328
		D:	1 740k	**D:**	35 448
Other KEMs submitted to the NIST PQC project					
	implementation	**clock cycles**		**stack usage**	
R5ND_1PKEb	[SBGM+18]	**K:**	658k	**K:**	?
		E:	984k	**E:**	?
		D:	1 265k	**D:**	?
R5ND_3PKEb	[SBGM+18]	**K:**	1 032k	**K:**	?
		E:	1 510k	**E:**	?
		D:	1 913k	**D:**	?
NewHopeCCA1024	[KRSS, AJS16]	**K:**	1 244k	**K:**	11 152
		E:	1 963k	**E:**	17 448
		D:	1 979k	**D:**	19 648
Kyber768	[KRSS]	**K:**	1 200k	**K:**	10 544
		E:	1 446k	**E:**	13 720
		D:	1 477k	**D:**	14 880

Comparison to reference code. Table 4 contains the performance benchmarks for the optimized implementations as well as the reference implementations with the modifications described above. For comparison we also include four KEMs submitted to the NIST post-quantum project that have previously been optimized on the ARM Cortex-M4. For all schemes targeted in this paper we dramatically increase the performance; the improvements go up to a factor of 47 for the key generation of RLizard-1024. Since both Karatsuba and Toom-Cook require storing additional intermediate polynomials on the stack, we increase stack usage for NTRU-HRSS and RLizard. Note that for Kindi-256-3-4-2, NTRU-KEM-743, and Saber, stack usage decreased. The reference implementations of those schemes already contained optimized polynomial multiplication methods, which were implemented in a stack-inefficient manner.

Side-channel resistance. While side-channel resistance was not a focus of this work, we ensured that our polynomial multiplication is protected against timing attacks. More specifically, in the multiplication routines we avoid all data flow from secrets into branch conditions and into memory addresses. The special multiplication routine in [SBGM+18] is less conservative and *does* use secret-dependent lookup indices with a reference to [ARM12] saying that the Cortex-M4 does not have internal data caches. However, it is not clear to us that really all Cortex-M4 cores do not have any data cache; [ARM12] states that the *"Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory. However, it is possible for a SoC design to integrate a system level cache."* Also, it *is* clear that some ARMv7E-M processors (for example, the ARM Cortex-M7) have data caches and our multiplication code is timing-attack protected also on those devices.

To investigate if the full KEM implementations are protected against timing attacks, we compiled the pure C implementations for an AMD64 desktop computer and ran them in `valgrind` with uninitialized secret data. This dynamic analysis reveals if the running software accesses memory at any secret-dependent location or if it branches on any secret-dependent data. Note that passing this analysis without errors is a strong indication, but no guarantee for the absence of timing leakage: the dynamic analysis does not guarantee branch coverage and the analysis is run on a binary compiled for a different architecture (because `valgrind` does not run on the bare-metal STM32F4Discovery). The results of the analysis are the following:

- The RLizard implementation is full of secret-dependent branches already in key generation. The output of `valgrind` states *"More than 10000000 total errors detected. I'm not reporting any more. Final error counts will be inaccurate. Go fix your program!"*
- The NTRU-HRSS implementation contained one small source of timing leakage, which we eliminated. After this modification the software passes without any secret-dependent branches or memory access.
- The NTRUEncrypt implementation makes heavy use of secret-dependent branches in the constant-weight sampling. Eliminating those efficiently (for example by using a sorting network) will result in different test vectors.
- The Saber implementation is free of any secret-dependent branches and access to secret-dependent memory locations. This is consistent with the claims made in [DKRV18].
- The Kindi implementation makes use of secret-dependent branches inside the `round` and `lround` functions of the standard C math library and in a conditional subtraction of q when reducing coefficients. These sources of timing leakage should be reasonably easy to eliminate.

Key-generation performance. The focus of this paper is to improve performance of encapsulation and decapsulation. All KEMs considered in this paper are CCA-secure, so the impact of a poor key-generation performance can in principle be minimized by caching also ephemeral keys for some time. Such caching of ephemeral keys makes software more complex and in some cases also requires changes to higher level protocols; we therefore believe that key-generation performance, also for CCA-secure KEMs, remains an important target of optimization. The key generation of RLizard, Saber, and Kindi is rather straight-forwardly optimized by integrating our fast multiplication. The key generation of NTRUEncrypt and NTRU-HRSS also requires inversions, which we did not optimize in this paper; we believe that further research into efficient inversions for those two schemes will significantly improve their key-generation performance.

Comparison to previous results. To the best of our knowledge, Saber is the only scheme of those considered in this paper that has been optimized for the ARM Cortex-M family in previous work [KMRV18]. Table 4 contains the performance result on the same platform as ours. Our optimized implementation outperforms the CHES 2018 implementation by 17% for key generation, 15% for encapsulation, and 18% for decapsulation. Karmakar, Bermudo Mera, Sinha Roy, and Verbauwhede report 65 459 clock cycles for their optimized 256-coefficient polynomial multiplication, but we note that their polynomial multiplication includes the reduction. Including the reduction, our multiplication requires 40 978 clock cycles, which is 37% faster. On a more granular level, they claim 587 cycles for 16-coefficient schoolbook multiplication, while we require only 360 cycles (see Table 2; this includes approximately 50 cycles of benchmarking overhead).

Several other NIST candidates have been evaluated on the Cortex-M4 family. We also list the performance results in Table 4 for comparison. Most recently Saarinen, Bhattacharya, Garcia, Morchon, Rietman, Tolhuizen, and Zhang published results for Round5[6] on Cortex-M4. The fastest scheme described in this work, targeting NIST security category 1, NTRU-HRSS, is 56% faster for encapsulation and 32% faster for decapsulation compared to the corresponding CCA variant of Round5 at the same security level. The key generation of NTRU-HRSS is considerably slower, but the inversion is also not optimized yet. The fastest scheme implementation described here that targets NIST security category 3, Saber, is 8% faster for key generation, 18% faster for encapsulation, and 34% faster for decapsulation There are also optimized implementations for NewHopeCCA1024 [KRSS, AJS16] and Kyber768 [KRSS]. Both implementations are outperformed by the described implementations of NTRU-HRSS and Saber.

4.3 Profiling of optimized implementations

The speed up achieved by optimizing polynomial multiplication clearly shows that it vastly dominates the runtime of reference implementations. Having replaced this core arithmetic operation with highly optimized assembly, we analyze how much time the optimized implementations still spend in non-optimized code to capture how much performance could still be gained by hand-optimizing scheme-specific procedures. We achieve this by measuring the clock cycles spent in polynomial multiplication, hashing, and random number generation. Table 4.3 contains the results. Still a considerable proportion of encapsulation and decapsulation is spent in polynomial multiplication. However, cycles consumed by hashing and randomness generation become more prominent. In the following we briefly discuss these results and emphasize how one could further speed-up those schemes.

Hashing. For encapsulation, hashing (SHA-3 and SHA-2) dominates the run-time of Kindi-256-3-4-2, NTRU-KEM-743, and Saber. We have already replaced these primitives with the fastest implementations available. Still, all schemes spend a substantial number of

[6]R5ND_{1,3,5}PKEb are the CCA-variants of Round5, whereas R5ND_{1,3,5}KEMb are CPA-secure.

Table 5: Time spent in polynomial multiplication, hashing, and sampling randomness for optimized implementations. Still considerable time is spent in polynomial multiplication, but hashing is more apparent.

scheme		total	polymul		hashing		randombytes	
Saber	**K:**	949k	352k	(37%)	514k	(54%)	2.0k	(<1%)
	E:	1 232k	470k	(38%)	670k	(54%)	0.7k	(<1%)
	D:	1 260k	587k	(46%)	542k	(43%)	0	
Kindi-256-3-4-2	**K:**	1 010k	370k	(37%)	409k	(41%)	1.2k	(<1%)
	E:	1 365k	494k	(36%)	604k	(44%)	0.7k	(<1%)
	D:	1 563k	617k	(40%)	603k	(39%)	0	
NTRU-HRSS	**K:**	161 790k	1 639k	(1%)	88k	(<1%)	0.7k	(<1%)
	E:	432k	182k	(42%)	118k	(27%)	0.7k	(<1%)
	D:	863k	546k	(63%)	74k	(9%)	0	
NTRU-KEM-743	**K:**	5 663k	1 773k	(31%)	0		85k	(2%)
	E:	1 655k	197k	(12%)	1 186k	(73%)	48k	(3%)
	D:	1 904k	394k	(21%)	1 180k	(62%)	0	
RLizard-1024	**K:**	537k	314k	(59%)	0		123k	(22%)
	E:	1 358k	628k	(46%)	630k	(46%)	2.2k	(<1%)
	D:	1 740k	941k	(54%)	630k	(36%)	0	

clock cycles computing hashes. This is partly due to the Fujisaki-Okamoto transformation required to achieve CCA security. An additional amount of hash function calls is required to sample pseudo-random numbers from a seed, which most schemes implement using the SHAKE XOF. Having a hardware accelerator for these hash function would highly benefit all of the examined schemes. While ARM Cortex-M4 platforms with SHA-2 hardware support exist, there are (at the time of writing) none available which have SHA-3 hardware support. Since SHA-3 is extensively used by the majority of post-quantum schemes in general, we consider hardware accelerators for this primitive in small devices essential for the performance of quantum-resistant cryptography on embedded systems.

Randomness generation. Kindi-256-3-4-2, NTRU-HRSS, and Saber do not make use of `randombytes` extensively, but sample a small seed and then expand this seed using SHAKE. Contrary to that, RLizard-1024 and NTRU-KEM-743 sample their entire randomness in KeyGen using `randombytes`. This raises the question which approach is more efficient for our platform. To address this we measure the time required to sample random bytes using `randombytes` and the time required to expand a short seed (32 bytes) to a number of pseudo-random bytes. The former method requires roughly 17 cycles/byte and is linear in the number of bytes sampled. The run-time of the latter depends upon the number of Keccak permutations ('squeezes') required to obtain the output. Consequently, the performance is influenced by how well the desired output length divides by the corresponding sponge rate. In the best case (i.e., the number of required bytes is divisible by the SHAKE rate) this requires 85 cycles/byte for SHAKE-128 and 102 cycles/byte for SHAKE-256 excluding the cycles required to sample the seed. In the worst case (i.e., sampling only one byte) it requires over 15,000 cycles/byte for both SHAKE-128 and SHAKE-256 (exactly one Keccak permutation).

There are, however, important caveats to consider when relying on the hardware number generator. For one, it is unclear what the cryptographic properties of different implementations of the RNG are, and how that affects the security of the various schemes, in particular since most reveal randomness as part of the CCA transform. Secondly, for our discovery board, the RNG guarantees a new random 32-bit word every 40 cycles of the RNG clock [STM18], which is decoupled from the core clock. This is in no way constant across platforms, may lead to wildly varying performance results, and needs to be carefully taken into account when sampling more than four bytes at a time.

5 Conclusion

Profiling of the software after our optimizations shows that for all five primitives still more than 69% of the cycles of encapsulation and decapsulation are are consumed by multiplication, hashing, and randomness generation. The remaining $< 31\%$ are spent in C code that, aside from eliminating dynamic memory allocations and various small fixes, is largely unchanged from the reference implementations. While we believe that there is something to be gained by speeding up these remaining routines in assembly, our results show that for the five lattice-based KEMS targeted in this paper a generally promising optimization strategy, also on other architectures, is to

- speed up multiplications in $\mathcal{R}_q$;
- integrate existing fast implementations of the respective hash algorithms; and
- integrate a fast implementation of `randombytes`.

When targeting a new microarchitecture, most effort is clearly the implementation of fast multiplication. The Python scripts shipping with this paper can be adapted to other architectures and should significantly reduce this effort.

References

[AAB+17] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. Newhope: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://cryptojedi.org/papers/#newhopenist. 1

[ABD+17] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://pq-crystals.org/kyber. 1

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In Thorsten Holz and Stefan Savage, editors, *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. https://eprint.iacr.org/2015/1092. 1

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. A new hope on ARM Cortex-M. In Claude Carlet, Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Advanced Cryptography Engineering*, volume 10076 of *LNCS*, pages 332–349. Springer, 2016. https://eprint.iacr.org/2016/758. 16, 18

[ARM12] Arm cortex-m programming guide to memory barrier instructions, 2012. https://static.docs.arm.com/dai0321/a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf. 17

[Ban17] Rachid El Bansarkhani. KINDI: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. http://kindi-kem.de. 1, 6

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, volume 9562 of *LNCS*, pages 209–224. Springer, 2016. https://eprint.iacr.org/2015/769. 3

[BGML+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. Cryptology ePrint Archive, Report 2018/725, 2018. https://eprint.iacr.org/2018/725. 1

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to (accessed 2018-10-14). 15

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, 2012. https://eprint.iacr.org/2011/401. 3

[Coo66] Stephen Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966. 8

[CPL+17] Jung Hee Cheon, Sangjoon Park, Joohee Lee, Duhyeong Kim, Yongsoo Song, Seungwan Hong, Dongwoo Kim, Jinsu Kim, Seong-Min Hong, Aaram Yun, Jeongsu Kim, Haeryong Park, Eunyoung Choi, Kimoon kim, Jun-Sub Kim, and Jieun Lee. Lizard: 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. 1, 3

[Den03] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *Cryptography and Coding*, volume 2898 of *LNCS*, pages 133–151. Springer, 2003. http://www.cogentcryptography.com/papers/designer.pdf. 3

[DHP+] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP (accessed 2018-10-14). 15

[DKRV17] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: 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. 1, 4

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *Progress in Cryptology – AFRICACRYPT 2018*, volume 10831 of *LNCS*, pages 282–305. Springer, 2018. https://eprint.iacr.org/2018/230. 17

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999. http://dx.doi.org/10.1007/3-540-48405-1_34. 3

[GMZB+17] Oscar Garcia-Morchon, Zhenfei Zhang, Sauvik Bhattacharya, Ronald Rietman, Ludo Tolhuizen, and Jose-Luis Torre-Arce. Round2: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://www.onboardsecurity.com/nist-post-quantum-crypto-submission. 1

[HGSSW03] Nick Howgrave-Graham, Joseph H. Silverman, Ari Singer, and William Whyte. NAEP: Provable security in the presence of decryption failures. Cryptology ePrint Archive, Report 2003/172, 2003. https://eprint.iacr.org/2003/172. 4

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, volume 10677 of *LNCS*, pages 341–371. Springer, 2017. https://eprint.iacr.org/2017/604. 4, 6

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998. http://dx.doi.org/10.1007/BFb0054868. 3, 4

[HPS+17] Jeff Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing parameters for NTRUEncrypt. In Helena Handschuh, editor, *Topics in Cryptology – CT-RSA 2017*, volume 10159 of *LNCS*, pages 3–18. Springer, 2017. https://eprint.iacr.org/2015/708. 4

[HRSS17a] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *LNCS*, pages 232–252. Springer, 2017. https://eprint.iacr.org/2017/667. 3, 4, 11

[HRSS17b] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. Ntru-kem-hrss: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://ntru-hrss.org. 1, 3

[KMRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, 2018. https://eprint.iacr.org/2018/682. 2, 10, 11, 14, 16, 18

[KO63] Anatolii Karatsuba and Yuri Ofman. 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. 8

[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 (accessed 2018-10-14). 2, 12, 15, 16, 18

[NIS15a] FIPS PUB 180-4: Secure hash standard, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. 15

[NIS15b] FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 15

[NIS16a] Submission requirements and evaluation criteria for the post - quantum cryptography standardization process, 2016. https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf. 12

[NIS16b] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 1

[Saa17] Markku-Juhani O. Saarinen. Hila5: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://mjos.fi/hila5. 1

[SAL+17] Nigel P. Smart, Martin R. Albrecht, Yehuda Lindell, Emmanuela Orsini, Valery Osheter, Kenny Paterson, and Guy Peer. Lima: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://lima-pq.github.io. 1

[SBGM+18] Markku-Juhani O. Saarinen, Sauvik Bhattacharya, Oscar Garcia-Morchon, Ronald Rietman, Ludo Tolhuizen, and Zhenfei Zhang. 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, Version: 13-Oct-2018 08:50:18 UTC. 1, 2, 16, 17

[SS17] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptology – SAC 2016*, volume 10532 of *LNCS*, pages 180–194. Springer, 2017. https://eprint.iacr.org/2016/714. 7

[STM18] STMicroelectronics. Reference manual – STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced arm-based 32-bit MCUs. Technical Report RM0090 Rev 17, 2018. https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf. 19

[Too63] Andrei L. Toom. 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. 8

[ZCHW17] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. 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. 1, 4