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 42%, yielding speedups of 22% for key generation, 20% for encapsulation and 22% 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 400 000 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 By December 2018, five of those 69 have been withdrawn.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). 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

* 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: April 9, 2019

$q = 2^k$ 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 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 42% which improves the overall performance of key generation by 22%, encapsulation by 20%, and decapsulation by 22%. 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. Our implementation of NTRU-HRSS encapsulation and decapsulation solidly outperform the optimized Round5 software presented in [SBGM+18].

We achieve our results by systematically exploring different combinations of Toom-3, Toom-4, and Karatsuba decomposition [Too63,Coo66,KO63] 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 implementations have been integrated into the `pqm4` framework [KRSS].

Second round of NISTPQC. Since this paper first appeared online NIST announced the second round candidates of the post-quantum competition. While Kindi and RLizard are no longer under consideration by NIST, Saber, NTRU-HRSS, and NTRUEncrypt made it to the second round. NTRU-HRSS and NTRUEncrypt were merged into the new scheme NTRU. The optimizations presented in this paper carry over directly to the second round schemes.

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 this section, we highlight the relevant aspects; see Appendix A for algorithmic descriptions.

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 descriptions, 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 $\mathsf{Sample}_{\mathcal{R}}$, 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.

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]. 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 PKE scheme.

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 in RLizard fit the structure that we target in this work.

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 $\Phi_d = 1 + x^1 + x^2 + \cdots + x^{d-1}$, 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.

The scheme requires several multiplications and inversions. For this paper, we focus on multiplications in $\mathcal{R}'_q$ and $\mathcal{R}_q$. However, the same routine can be used to perform the multiplication in $\mathcal{R}_p$. Furthermore, as the inversion in $\mathcal{R}'_q$ can be performed using multiplications [HRSS17a], this benefits from the same optimization.

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].

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). 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.

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.

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

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`	$\mathtt{Ra} \leftarrow \mathtt{Rb_L} \cdot \mathtt{Rc_H} + \mathtt{Rb_H} \cdot \mathtt{Rc_L} + \mathtt{Rd}$

as well. Other parameters p and t are powers of 2 smaller than q; for the Saber instance[1], $p = 1024$ 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.

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 $q = 2^{14}$.

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 adding hash-function calls.

2.2 ARM Cortex-M4

Our target platform is the ARM Cortex-M4 which implements the 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 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

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

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 16-bit Thumb instructions, 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 expanded to full-word width using the `.w` suffix.

Benchmarking platform. In 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[2] (`arm-none-eabi`) with `arm-none-eabi-gcc-8.3.0`. All source files are compiled with the optimization flag `-O3`.

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 done outside of our optimized polynomial multiplication.

Here, 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. For a given n and q, there are multiple possible approaches; we explore the entire space and select the optimum for each parameter set. We use Python scripts that generate optimized assembly functions for all combinations, for arbitrary-degree polynomials (with degree below 1024). These scripts are 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.

3.1 Toom/Karatsuba strategies

The naive schoolbook approach to multiply two polynomials with n coefficients results in n^2 multiplications in $\mathbb{Z}_q$. Using well-known algorithms by Karatsuba [KO63] and Toom-Cook [Too63,Coo66], it is possible to trade some of these

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

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, breaking down a multiplication of n-coefficient polynomials into seven multiplications of $\frac{n}{4}$ coefficients.

Toom-Cook. It is important to note that there is a loss in precision when using Toom's method, as it involves division over the integers. 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. Consequently, Toom-3 loses one bit of precision, and Toom-4 loses three bits. 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 $q \leq 2^{15}$, Toom-4 can be used for $q \leq 2^{13}$. These losses accumulate, and a combination of both is only possible if $q \leq 2^{12}$. This also rules out higher-order Toom methods. While switching to 32-bit arithmetic would allow using higher order Toom, this slows down Karatsuba and the schoolbooks significantly by increasing load-store overhead and ruling out DSP instructions.

While asymptotically Toom-4 is more efficient than Toom-3 and Karatsuba, in practice the additions and subtractions also impact the run-time. The increased and more complex memory-access patterns also 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.

Karatsuba. The call to the topmost Karatsuba layer is 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 jump to the schoolbook multiplication as an explicit subroutine. 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.

Note that we only applied Karatsuba's method to split polynomials in two parts (i.e., not more), and did not combine operations across recursive calls. See [WP06] for details on a more general approach.[3]

[3] The approach by Weimerskirch and Paar provides a middle ground between Karatsuba and Toom-Cook. While allowing for a wider range of splits than traditional Karatsuba and a more efficient way of dealing with the newly introduced additions, it does

As 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 high and low limbs, 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. We rely on offset-based instructions for memory operations, in particular for the more intricate memory access patterns in Toom and Karatsuba. 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 addresses are computed during code generation.

For ease of implementation, our code generator for Toom is restricted to dimensions that divide without remainder. For Karatsuba, we do not restrict the dimensions at all: the implementation can work on unbalanced splits, and thus polynomials of unequal length. In order not to 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.

Rather than considering alignment to 32-bit word boundaries during code generation, we use a post-processing step. After compilation, we disassemble the resulting binary and expand Thumb instructions in the cases where they cause misalignment. This allows using 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. 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

We carefully investigate several approaches to perform the small-degree schoolbook multiplications that underlie Karatsuba and Toom-Cook, varying the approaches and implementing distinct generation routines for different n.

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 multiplication instructions that efficiently operate on data in this format: parallel multiplications, but also instructions that operate on a specific halfwords. For $n \leq 10$, all input coefficients can be kept in registers simultaneously, with

come at the cost of more small-sized multiplications than similarly-sized Toom-Cook instances. A key advantage, though, is the fact that this approach does not introduce divisions that lead to a loss of precision. This could be relevant in particular for multiplications where both n and q are large.

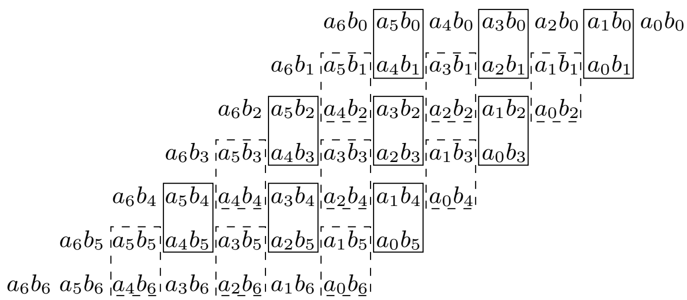

Fig. 1. Pairing coefficients to reduce the number of multiplications, using `smladx` / `smlad` instructions. Dashed boxes represent multiplications involving repacked b.

registers 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. At 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 a single dedicated addition instruction. This offsets the extra cost of the 6 packing instructions considerably. For $n \in \{13, 14\}$, not all coefficients fit in registers at the same time, leading to spills for the middle columns (i.e., the computation of coefficients around x^n, which are affected by all input coefficients). Even when using the Python abstraction layer, manual register allocation becomes somewhat tedious in the cases that involve many spills to the stack. 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 of a grid of smaller instances. For $15 \leq n \leq 24$, we compose the multiplication out of four smaller 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 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.

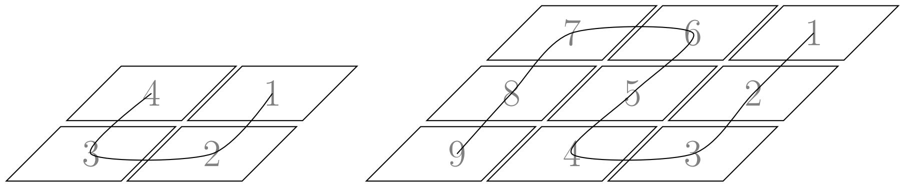

Fig. 2. Decomposing larger schoolbook multiplications

Figure 3 shows the performance of these routines; see Table 5 for more details.

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], 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.

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 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 $n > 36$

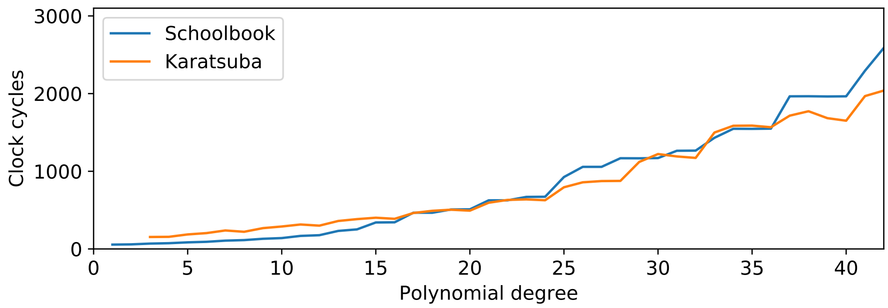

Fig. 3. Runtime of generated optimized polynomial multiplication for small n. For $n < 20$ our hand-optimized schoolbook multiplications are clearly superior, for $n > 36$ first applying at least one layer of Karatsuba is faster.

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. 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; 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 2, 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.

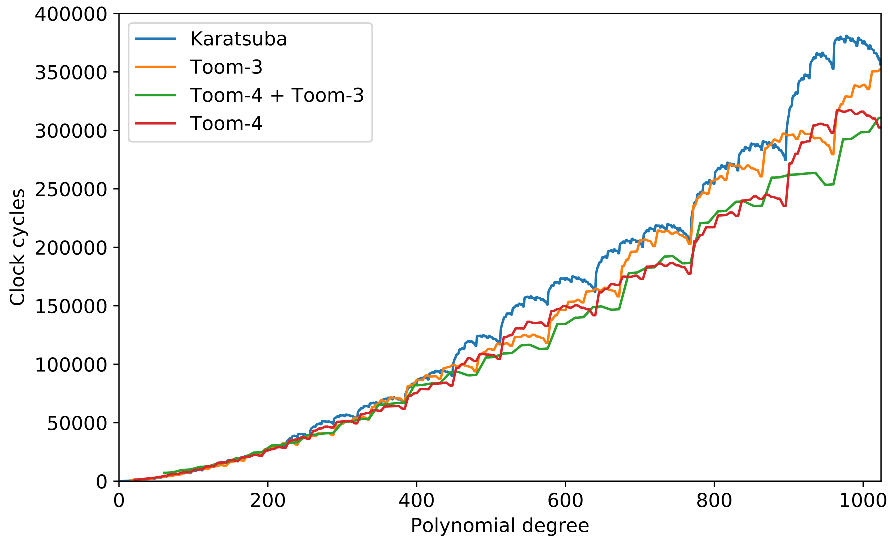

Fig. 4. Runtime of different decomposition variants for large-degree multiplications.

While there is some performance benefit to performing the reduction inline, the main gain is in stack usage. For the Toom variants, this allows for in-place recomposition, reducing stack usage by roughly $2n$ coefficients. This is not trivial for Karatsuba, though, introducing some additional complexity. We leave this for future work.

For the rather small $n = 256$ (Saber, Kindi), we already 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

Table 2. 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 usage** [bytes]
Saber ($n = 256,\ q = 2^{13}$)	Karatsuba only	16	38 000	2 020
	Toom-3	11	39 043	3 480
	Toom-4	**16**	**36 274**	**3 800**
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256,\ q = 2^{14}$)	**Karatsuba only**	**16**	**38 000**	**2 020**
	Toom-3	11	39 043	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701,\ q = 2^{13}$)	Karatsuba only	11	202 889	5 676
	Toom-3	15	205 947	9 384
	Toom-4	**11**	**172 882**	**10 596**
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743,\ q = 2^{11}$)	Karatsuba only	12	217 130	6 012
	Toom-3	16	211 588	9 920
	Toom-4	**12**	**186 639**	**11 208**
	Toom-4 + Toom-3	16	192 503	12 152
RLizard-1024 ($n = 1024,\ q = 2^{11}$)	Karatsuba only	16	356 046	8 188
	Toom-3	11	352 770	13 756
	Toom-4	**16**	**302 504**	**15 344**
	Toom-4 + Toom-3	11	310 712	16 816

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 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.

Hashing. 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

Table 3. 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** [bytes]	
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:**	$895k$	**K:**	13 248
		E:	$1\,161k$	**E:**	15 528
		D:	$1\,204k$	**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:**	$969k$	**K:**	44 264
		E:	$1\,320k$	**E:**	55 392
		D:	$1\,517k$	**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:**	$145\,963k$	**K:**	23 396
		E:	$404k$	**E:**	19 492
		D:	$819k$	**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:**	$5\,198k$	**K:**	25 320
		E:	$1\,601k$	**E:**	23 808
		D:	$1\,881k$	**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:**	$525k$	**K:**	27 720
		E:	$1\,345k$	**E:**	33 328
		D:	$1\,716k$	**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

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 use a C implementation from SUPERCOP [BL].

Comparison to reference code. Table 3 contains the performance benchmarks for the optimized implementations as well as the reference implementations with the modifications described above. For all schemes targeted in this paper we dramatically increase the performance; the improvements go up to a factor of 49 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 all schemes except Kindi-256-3-4-2. The reference implementations of Kindi-256-3-4-2 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.

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 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 3 contains the performance result on the same platform as ours. Our optimized implementation outperforms the CHES 2018 implementation by 22% for key generation, 20% for encapsulation, and 22% 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 38 215 clock cycles, which is 42% faster. On a more granular level, they claim 587 cycles for 16-coefficient schoolbook multiplication, while we require only 343 cycles (see Table 5; 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 3 for comparison. Most recently, record-setting results were published for Round5[4] on Cortex-M4 [SBGM+18]. The fastest scheme described in our work, targeting NIST security category 1, NTRU-HRSS, is 59% faster for encapsulation and 35% 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 its inversion is not optimized yet. The fastest scheme implementation described here that targets NIST security category 3, Saber, is 13% faster for key generation, 23% faster for encapsulation, and 37% faster for decapsulation There are also optimized implementations for NewHopeCCA1024 [KRSS,AJS16] and Kyber768 [KRSS]. Both implementations are outperformed by NTRU-HRSS and Saber.

4.3 Profiling of optimized implementations

Table 4. 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 [cycles]	polymul [cycles]	hashing [cycles]	randombytes [cycles]
Saber	**K:**	895k	327k (37 %)	475k (53 %)	2.0k (<1 %)
	E:	1 161k	435k (38 %)	615k (53 %)	0.6k (<1 %)
	D:	1 204k	544k (45 %)	500k (42 %)	0
Kindi-256-3-4-2	**K:**	969k	342k (35 %)	409k (42 %)	1.2k (<1 %)
	E:	1 320k	456k (35 %)	604k (46 %)	0.6k (<1 %)
	D:	1 517k	570k (38 %)	603k (40 %)	0
NTRU-HRSS	**K:**	145 963k	1 556k (1 %)	80k (<1 %)	0.6k (<1 %)
	E:	404k	173k (43 %)	107k (26 %)	0.6k (<1 %)
	D:	819k	519k (63 %)	67k (8 %)	0
NTRU-KEM-743	**K:**	5 198k	1 680k (32 %)	0	85k (2 %)
	E:	1 601k	187k (12 %)	1 171k (73 %)	46k (3 %)
	D:	1 881k	373k (20 %)	1 172k (63 %)	0
RLizard-1024	**K:**	525k	303k (58 %)	0	123k (23 %)
	E:	1 345k	605k (45 %)	628k (47 %)	2.2k (<1 %)
	D:	1 716k	908k (53 %)	628k (36 %)	0

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

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 shows that 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 runtime of Kindi-256-3-4-2, NTRU-KEM-743, and Saber. We have replaced these primitives with the fastest implementations available. Still, all schemes spend a substantial number of clock cycles computing hashes. This is partly due to the Fujisaki-Okamoto transformation required to achieve CCA security. Further hash function calls are 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.

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 using SHAKE. RLizard-1024 and NTRU-KEM-743 directly sample their randomness `randombytes`. As we implement `randombytes` using the hardware RNG on the STM32F4Discovery, it is more efficient than using SHAKE to expand a seed. There are, however, important caveats to consider when only using the hardware number generator. It is unclear what the cryptographic properties of such an RNG are, and how this affects the security of the various schemes, in particular since most reveal randomness as part of the CCA transform.

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. 14, 16

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. 15

Ban17. Rachid El Bansarkhani. KINDI: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. http://kindi-kem.de. 2, 5

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. 2

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. 2, 6

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. 2, 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, 4

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. 2, 4

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, 4

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. 2

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, 5

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. 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. 4, 9

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. 2, 4

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, 8, 9, 13, 14, 15

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. 2, 6

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, 10, 14, 15, 16

NIS15a. FIPS PUB 180-4: Secure hash standard, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. 13

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. 13

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. 10

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. 2

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. 2, 14, 15, 16

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. 6

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. 2, 6

WP06. André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. 2006. https://eprint.iacr.org/2003/172. 7

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. 2, 4

A Algorithmic descriptions

A.1 RLizard

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 $\mathsf{RLizard.Enc}\,(m,(a,b))$

$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
$c_1' \leftarrow a \circledast r \in \mathcal{R}_q$
$c_2' \leftarrow b \circledast r \in \mathcal{R}_q$
$c_1 \leftarrow \lfloor (p/q) \cdot c_1' \rceil \in \mathcal{R}_p$
$c_2 \leftarrow \lfloor (p/q) \cdot ((q/2) \cdot m + c_2') \rceil \in \mathcal{R}_p$
return (c_1, c_2)

Algorithm 3 $\mathsf{RLizard.Dec}\,((c_1,c_2),s)$

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

A.2 NTRU-HRSS-KEM

Algorithm 4 $\mathsf{NTRU\text{-}HRSS.KeyGen}\,()$

$f, g \leftarrow \mathsf{Sample}_{\mathcal{R}_p}$
$f_p^{-1} \leftarrow f^{-1} \in \mathcal{R}_p$
$f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}_q'$ ▷ Uses mult. in $\mathcal{R}_q$
$h \leftarrow \Phi_1 * g \circledast f_q^{-1} \in \mathcal{R}_q$
return $(\mathsf{pk} = p \cdot h, \mathsf{sk} = (f, f_p^{-1})$

Algorithm 5 $\mathsf{NTRU\text{-}HRSS.Enc}\,(m,(p \cdot h))$

$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}$
$c \leftarrow h' \circledast r + m \in \mathcal{R}_q$
return c

Algorithm 6 $\mathsf{NTRU\text{-}HRSS.Dec}\left(c,(f,f_p^{-1})\right)$

$v \leftarrow c \circledast f \in \mathcal{R}_q$
$m' \leftarrow v \circledast f_p^{-1} \in \mathcal{R}_p$
return m

A.3 NTRUEncrypt

Algorithm 7 $\mathsf{NTRUEncrypt.KeyGen}\,()$

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

Algorithm 8 NTRUEncrypt.Enc (m, h)

$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
$t \leftarrow r \circledast h$
$m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
$m' \leftarrow m - m_{mask} \mod p$
$c \leftarrow t + m'$
return c

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

$m' \leftarrow f \circledast c \mod p$
$t \leftarrow c - m$
$m_{mask} \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(t)$
$m \leftarrow m' + m_{mask} \mod p$
$r \leftarrow \mathsf{Sample}_{\mathcal{R}_q}(m, h)$
if $p \cdot r \circledast h = t$ **then**
    return m
else
    return $\perp$
end if

A.4 Saber

Algorithm 10 Saber.KeyGen $()$

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

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

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

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

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

A.5 KINDI

Algorithm 13 $\mathsf{Kindi.KeyGen}()$

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

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

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

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

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

B Schoolbook multiplication benchmarks

Table 5. 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	56	13	232	25	926	37	1 965
2	59	14	252	26	1 057	38	1 966
3	69	15	341	27	1 057	39	1 963
4	74	16	343	28	1 168	40	1 965
5	85	17	467	29	1 167	41	2 294
6	92	18	466	30	1 170	42	2 588
7	107	19	508	31	1 264	43	2 595
8	114	20	510	32	1 266	44	2 594
9	131	21	626	33	1 431	45	2 824
10	140	22	626	34	1 547	46	2 825
11	168	23	670	35	1 546	47	2 822
12	177	24	672	36	1 549	48	2 824