Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography

Patrick Longa and Michael Naehrig

Microsoft Research, USA
{plonga,mnaehrig}@microsoft.com

Abstract. The Number Theoretic Transform (NTT) provides efficient algorithms for cyclic and nega-cyclic convolutions, which have many applications in computer arithmetic, e.g., for multiplying large integers and large degree polynomials. It is commonly used in cryptographic schemes that are based on the hardness of the Ring Learning With Errors (R-LWE) problem to efficiently implement modular polynomial multiplication.
We present a new modular reduction technique that is tailored for the special moduli required by the NTT. Based on this reduction, we speed up the NTT and propose faster, multi-purpose algorithms. We present two implementations of these algorithms: a portable C implementation and a high-speed implementation using assembly with AVX2 instructions. In comparison to state-of-the-art implementations of the NTT using the same parameters, our C and assembly implementations achieve factor-1.90 and factor-1.25 speedups, respectively. To demonstrate the improved efficiency in an application example, we benchmarked the algorithms in the context of the R-LWE key exchange protocol that has recently been proposed by Alkim, Ducas, Pöppelmann and Schwabe. In this case, our C and assembly implementations compute the full key exchange 1.40 and 1.34 times faster, respectively. These results are achieved with full protection against timing attacks.

Keywords: Post-quantum cryptography, number theoretic transform (NTT), ring learning with errors (R-LWE), fast modular reduction, efficient implementation.

1 Introduction

Fast Fourier Transform (FFT) algorithms to compute the Discrete Fourier Transform (DFT) have countless applications ranging from digital signal processing to the fast multiplication of large integers. The cyclic convolution of two integer sequences of length n can be computed by applying an FFT algorithm to both, then multiplying the resulting DFT sequences of length n coefficient-wise and transforming the result back via an inverse FFT. This operation corresponds to the product of the corresponding polynomials modulo $X^n - 1$, and for large n, a computation via FFTs as above was suggested to be used in the ring-based encryption scheme NTRUEncrypt in [14].

When the sequence (or polynomial) coefficients are specialized to come from a finite field, the DFT is called the Number Theoretic Transform (NTT) [8] and can be computed with FFT algorithms that work over this specific finite field. Polynomial multiplication over a finite field is one of the fundamental operations required in cryptographic schemes based on the Ring Learning With Errors (R-LWE) problem, and the NTT has shown to be a powerful tool that enables this operation to be computed in quasi-polynomial complexity.

R-LWE-based cryptography. Since its introduction by Regev [27], the Learning With Errors (LWE) problem has been used as the foundation for many new lattice-based constructions with a variety of cryptographic functionalities. It is currently believed to be sufficiently hard, even for attackers running a large scale quantum computer. Hence cryptographic schemes with security based on the hardness of the LWE problem are promising candidates for post-quantum (or quantum-safe) cryptography.

The Ring LWE (R-LWE) problem, introduced by Lyubashevsky, Peikert and Regev [20], is a special instance of the LWE problem that is essentially obtained by adding a ring structure to the underlying lattice. R-LWE-based schemes have been proposed for public-key encryption [30, 20, 24], digital signatures [18, 9], and key exchange [24, 31, 5, 2]. Furthermore, the most efficient proposals for (fully) homomorphic encryption are also based on R-LWE, e.g., [6].

The advantage of R-LWE over LWE is a significant increase in efficiency. When working with vectors of dimension n, it allows a factor n space reduction and the possibility of using FFT algorithms to compute polynomial products instead of matrix-vector or matrix-matrix operations; this leads to an improvement from roughly n^2 base ring multiplications to roughly $n \log n$ such multiplications.

One particularly efficient parameter instantiation in the context of R-LWE is such that the dimension n is a power of 2 and polynomial products are taken modulo the $2n$-th cyclotomic polynomial $X^n + 1$ with coefficients modulo a prime q. Here, the polynomial product corresponds to a nega-cyclic convolution of the coefficient sequences. In this setting, the NTT is usually computed with a special type of FFT algorithm that can be used efficiently when q is a prime that satisfies the congruence condition $q \equiv 1 \bmod 2n$ (cf. [19, §2.1]), which in turn means that the underlying finite field contains primitive $2n$-th roots of unity. Many state-of-the-art instantiations of R-LWE-based cryptography choose n and q as above in order to harness the efficiency of the NTT; for example, the BLISS signature implementations (I-IV) set $n = 512$ and $q = 12289$ [9] and the fastest R-LWE-based key exchange implementation to date sets $n = 1024$ and $q = 12289$ [2].

Our contributions. We present a new modular reduction algorithm for the special moduli that are required to invoke the NTT. While this new routine can be used to replace existing modular reduction algorithms and give standalone performance improvements, we further show that calling it inside a modified NTT algorithm can give rise to additional speedups. We illustrate these improvements by providing and benchmarking both our portable C and AVX2 assembly implementations. When setting $n = 1024$ and $q = 12289$, our C and AVX2 implementations of the NTT are up to 1.86 and 1.21 times faster, respectively, than the previously fastest, constant-time NTT implementations by Alkim et al. [2] that use the same parameters. Similarly, our inverse NTT implementations are 1.90 and 1.25 times faster (respect.) than similar implementations presented in [2]. Our software is publicly available [17].

Given the ubiquity of the NTT in (both the existing and foreseeable) high-speed instantiations of R-LWE-based primitives, we emphasize that an improved NTT simultaneously improves a large portion of all lattice-based cryptographic proposals. While our algorithm will give a solid speedup to signature schemes like Lyubashevsky's [18] and BLISS [9], it will give a more drastic overall improvement in common encryption and key exchange schemes. In these scenarios, there are different ways of removing the need for obtaining high-precision samples from a Gaussian distribution [21], for example, the number of R-LWE samples per secret can be bounded, or one can use the Kullback-Leibler or Renyi divergences [3]. Subsequently, the cost of sampling the error distribution decreases dramatically, and the NTT becomes the bottleneck of the overall computation.

To highlight the practical benefits of the new approach in an example of a cryptographic protocol, we implemented the recent instantiation of Peikert's key exchange [24] due to Alkim, Ducas, Pöppelmann and Schwabe [2], and show that the overall key exchange is approximately 1.40 times faster (portable C implementation) and 1.34 times faster (AVX2 assembly implementation) using our improved NTT. Beyond the faster modular reduction itself, the specific improvements over the approach in [2] that have led to this speedup are as follows:

- The new modular reduction algorithm allows coefficients to grow up to 32 bits in size, which eliminates the need for modular reductions after any addition during the NTT. As a consequence, reductions are only carried out after multiplications.
- The new modular reduction is very flexible and enables efficient implementations using either integer arithmetic or floating point arithmetic. Since it minimizes the use of multiplications, using the higher throughput of floating point instructions on the latest Intel processors does not have as big an impact as for more multiplication-heavy methods like Montgomery reduction. Hence, the method is especially attractive for implementations with a focus on simplicity, particularly in plain C.
- Related to the previous point, our implementation uses signed integer arithmetic in the NTT. This allows for signed integers to represent error polynomials and secret keys, which saves conversions from negative to positive integers (e.g., this reduces the number of additions during error sampling and before modular reductions in the NTT).
- We show how to merge the scaling by n^{-1} with our conversion from redundant to standard integer representation at the end of the inverse NTT. In addition, by pulling this conversion into the last stage of the inverse NTT, we eliminate $n/2$ multiplications and reductions, all at the cost of precomputing only two integers.

Organization. Section 2 gives the background on R-LWE and the NTT. Section 3 contains our two main contributions: the improved modular reduction and NTT algorithms. Section 4 revises the details in the R-LWE key exchange scheme from [2], which is used as a case study to give a practical instance where our improved NTT gives rise to faster cryptography. Finally, Section 5 provides a performance analysis and benchmarks.

2 Preliminaries

This section provides details about the ring structure in the R-LWE setting, the NTT, and the FFT algorithm to compute the NTT and its inverse. The original proposal of R-LWE [20] restricts to cyclotomic rings, i.e. rings generated over the integers by primitive roots of unity. We immediately focus on 2-power cyclotomic rings as this is the most commonly used case and seems to provide the most efficient arithmetic. It is also the most natural application for the NTT within the class of cyclotomic rings and we describe the NTT algorithm for this specific choice. As is often done in the literature, in this paper we use the term NTT simultaneously for naming the number theoretic transform as well as an FFT algorithm to compute it.

2.1 The Ring Learning With Errors (R-LWE) setting

Let $N = 2^d$, $d > 1$ be a power of two and let $n = \varphi(N) = 2^{d-1} = N/2$. Then the N-th cyclotomic polynomial is given by $\Phi_N(X) = X^n + 1$. Let R be the ring of cyclotomic integers, i.e. $R = \mathbb{Z}[X]/(\Phi_N(X)) = \mathbb{Z}[X]/(X^n + 1)$. We identify the elements of R with their representatives of minimal degree. Thus, any element $a \in R$ is a polynomial of degree at most $n-1$ with integer coefficients, written as $a = \sum_{i=0}^{n-1} a_i X^i$, $a_i \in \mathbb{Z}$. Furthermore, let $q \in \mathbb{Z}$ be a positive integer modulus such that $q \equiv 1 \pmod{N}$. The quotient ring $R/(q)$, obtained by taking the elements of R modulo q, is isomorphic to the ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$. With the above identification, the ring R_q consists of all polynomials of degree at most $n-1$ with coefficients from the ring $\mathbb{Z}_q$, i.e., for any $a \in R_q$, we write $a = \sum_{i=0}^{n-1} a_i X^i$, $a_i \in \mathbb{Z}_q$. We use the same symbol a to denote both the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ and the sequence $a = (a[0], a[1], \ldots, a[n-1]) \in \mathbb{Z}_q^n$.

2.2 The Number Theoretic Transform (NTT)

The NTT is a specialized version of the discrete Fourier transform, in which the coefficient ring is taken to be a finite field (or ring) containing the right roots of unity. It can be viewed as an exact version of the complex DFT, avoiding round-off errors for exact convolutions of integer sequences. While Gauss apparently used similar techniques already in [11], laying the ground work for modern FFT algorithms to compute the DFT and therefore the NTT is usually attributed to Cooley and Tukey's seminal paper [8].

Notation and background. With parameters as above, i.e. n being a power of 2 and q a prime with $q \equiv 1 \pmod{2n}$, let $a = (a[0], ..., a[n-1]) \in \mathbb{Z}_q^n$, and let ω be a primitive n-th root of unity in $\mathbb{Z}_q$, which means that $\omega^n \equiv 1 \pmod{q}$. The forward transformation $\tilde{a} = \mathrm{NTT}(a)$ is defined as $\tilde{a}[i] = \sum_{j=0}^{n-1} a[j]\omega^{ij} \bmod q$ for $i = 0, 1, ..., n-1$. The inverse transformation is given by $b = \mathrm{INTT}(\tilde{a})$, where $b[i] = n^{-1} \sum_{j=0}^{n-1} \tilde{a}[j]\omega^{-ij} \bmod q$ for $i = 0, 1, ..., n-1$, and we have $\mathrm{INTT}(\mathrm{NTT}(a)) = a$.

As mentioned above, the NTT can be used directly to perform the main operation in R-LWE-based cryptography, that is, polynomial multiplication in $R_q = \mathbb{Z}_q[X]/(X^n+1)$. However, since applying the NTT transform as described above provides a cyclic convolution, computing $c = a \cdot b \bmod (X^n+1)$ with two polynomials a and b would require applying the NTT of length $2n$ and thus n zeros to be appended to each input; this effectively doubles the length of the inputs and also requires the computation of an explicit reduction modulo X^n+1. To avoid these issues, one can exploit the *negative wrapped convolution*: let ψ be a primitive $2n$-th root of unity in $\mathbb{Z}_q$ such that $\psi^2 = \omega$, and let $a = (a[0], ..., a[n-1])$, $b = (b[0], ..., b[n-1]) \in \mathbb{Z}_q^n$ be two vectors. Also, define $\hat{a} = (a[0], \psi a[1] ..., \psi^{n-1} a[n-1])$ and $\hat{b} = (b[0], \psi b[1] ..., \psi^{n-1} b[n-1])$. The negative wrapped convolution of a and b is defined as $c = (1, \psi^{-1}, \psi^{-2}, ..., \psi^{-(n-1)}) \circ \mathrm{INTT}(\mathrm{NTT}(\hat{a}) \circ \mathrm{NTT}(\hat{b}))$, where $\circ$ denotes component-wise multiplication. This operation satisfies $c = a \cdot b$ in R_q, and thus it allows us to compute a full polynomial multiplication that implicitly includes the reduction modulo X^n+1, without increasing the length of the inputs.

Previous optimizations. Some additional optimizations are available to the NTT-based polynomial multiplication. Previous works explain how to merge multiplications by the powers of ω with the powers of ψ and ψ^{-1} inside the NTT. Consequently, important savings can be achieved by precomputing and storing in memory the values related to these parameters. In particular, Roy et al. [28] showed how to merge the powers of ψ with the powers of ω in the forward transformation. This merging did not pose any difficulty in the case of the well-known *decimation-in-time* NTT, which is based on the Cooley-Tukey butterfly [8] that was used in the first implementations of R-LWE-based schemes. Similarly, Pöppelmann et al. [26] showed how to merge the powers of ψ^{-1} with the powers of ω in the inverse transformation. In this case, however, it was necessary to switch from a decimation-in-time NTT to a *decimation-in-frequency* NTT [12], which is based on the Gentleman-Sade (GS) butterfly. In this work we exploit the combination of both transformations for optimal performance.

Other optimizations focus on the NTT's butterfly computation. Relevant examples are the use of precomputed quotients, as exploited in Shoup's butterfly algorithm [29], and the use of redundant representations that enable the elimination of several conditional modular corrections, as shown by Harvey [13]. In particular, Harvey showed how to apply the latter technique on Shoup's butterfly and on a butterfly variant based on Montgomery arithmetic. In Section 5, we compare our improved NTT algorithms with the approaches by Melchor et al. [1] and Alkim et al. [2], both of which adopted and specialized Harvey's butterfly algorithms.

Several works in the literature (e.g., [25, 28, 16, 2]) have applied a relatively expensive re-ordering or bit-reversal step before or after the NTT computation. This is due to the restrictive nature of certain forward and inverse algorithms that only accept inputs in standard ordering and produce results in bit-reversed ordering. However, Chu and George [7] showed how to also derive forward and inverse FFT algorithms working for the reversed case, i.e., accepting inputs in bit-reversed ordering and producing outputs in standard ordering. Accordingly, [26] adapted and suitably combined the algorithms in the context of NTTs in order to eliminate the need of the bit-reversal step.

From hereon, we denote by $\texttt{NTT} := \mathrm{NTT}_{\mathrm{CT},\Psi_{\mathrm{rev}}}$ an algorithm that computes the forward transformation based on the Cooley-Tukey butterfly that absorbs the powers of ψ in bit-reversed ordering. This function receives the inputs in standard ordering and produces a result in bit-reversed ordering. Similarly, we denote by $\texttt{INTT} := \mathrm{INTT}_{\mathrm{GS},\Psi^{-1}_{\mathrm{rev}}}$ an algorithm computing the inverse transformation based on the Gentleman-Sade butterfly that absorbs the powers of ψ^{-1} in the bit-reversed ordering. This function receives the inputs in bit-reversed ordering and produces an output in standard ordering. Following Pöppelmann et al. [26], the combination of these two functions eliminates any need for a bit-reversal step. Optimized algorithms for the forward and inverse NTT are presented in Algorithms 1 and 2, respectively. These algorithms are based on the ones detailed in [26, App. A.1]. Note that we have applied a few modifications and corrected some typos.

Pöppelmann et al. [26] avoid the final scaling by n^{-1} during the inverse NTT by shifting the computation to a polynomial transformation that is (in their target application of BLISS signatures) assumedly performed offline. In general, however, that assumption does not necessarily hold; for example, in [2], all of the polynomials to be multiplied are generated *fresh* per key exchange connection. Accordingly, Algorithm 2 includes scaling by n^{-1}.

3 Modular Reduction and Speeding up the NTT

Most FFT algorithms to compute the NTT over a finite field or ring need certain roots of unity. In the specific setting discussed in the previous section, one needs primitive $2n$-th roots of unity to exist[1] modulo q, which imposes a congruence condition on q, namely $q \equiv 1 \pmod{2n}$. The parameters for R-LWE-based cryptosystems tend to have relatively large dimension n

[1] For an algorithm that does not require such roots, but has the disadvantage of needing to pad the inputs to double length to compute nega-cyclic convolutions, see Nussbaumers algorithm ([23] and [15, Exercise 4.6.4.59]).

Algorithm 1 Function `NTT` based on the Cooley-Tukey (CT) butterfly.

Input: A vector $a = (a[0], a[1], ..., a[n-1]) \in \mathbb{Z}_q^n$ in standard ordering, where q is a prime such that $q \equiv 1 \bmod 2n$ and n is a power of two, and a precomputed table $\Psi_{rev} \in \mathbb{Z}_q^n$ storing powers of ψ in bit-reversed order.
Output: $a \leftarrow \texttt{NTT}(a)$ in bit-reversed ordering.

```
 1: $k = n$
 2: for ($m = 1;\ m < n;\ m = 2m$) do
 3:     $k = k/2$
 4:     for ($i = 0;\ i < m;\ i{+}{+}$) do
 5:         $j_1 = 2 \cdot i \cdot k$
 6:         $j_2 = j_1 + k - 1$
 7:         $S = \Psi_{rev}[m + i]$
 8:         for ($j = j_1;\ j \leq j_2;\ j{+}{+}$) do
 9:             $U = a[j]$
10:             $V = a[j + k] \cdot S$
11:             $a[j] = U + V \bmod q$
12:             $a[j + k] = U - V \bmod q$
13:         end for
14:     end for
15: end for
16: return $a$
```

Algorithm 2 Function `INTT` based on the Gentleman-Sade (GS) butterfly.

Input: A vector $a = (a[0], a[1], ..., a[n-1]) \in \mathbb{Z}_q^n$ in bit-reversed ordering, where q is a prime such that $q \equiv 1 \bmod 2n$ and n is a power of two, and a precomputed table $\Psi_{rev}^{-1} \in \mathbb{Z}_q^n$ storing powers of ψ^{-1} in bit-reversed order.
Output: $a \leftarrow \texttt{INTT}(a)$ in standard ordering.

```
 1: $k = 1$
 2: for ($m = n;\ m > 1;\ m = m/2$) do
 3:     $j_1 = 0$
 4:     $h = m/2$
 5:     for ($i = 0;\ i < h;\ i{+}{+}$) do
 6:         $j_2 = j_1 + k - 1$
 7:         $S = \Psi_{rev}^{-1}[h + i]$
 8:         for ($j = j_1;\ j \leq j_2;\ j{+}{+}$) do
 9:             $U = a[j]$
10:             $V = a[j + k]$
11:             $a[j] = U + V \bmod q$
12:             $a[j + k] = (U - V) \cdot S \bmod q$
13:         end for
14:         $j_1 = j_1 + 2k$
15:     end for
16:     $k = 2k$
17: end for
18: for ($j = 0;\ j < n;\ j{+}{+}$) do
19:     $a[j] = a[j] \cdot n^{-1} \bmod q$
20: end for
21: return $a$
```

and relatively small moduli q, which means that moduli satisfying the congruence have the form $q = k \cdot 2^m + 1$, where $2n \mid 2^m$ and $k \geq 3$ is a very small integer.

Modular reduction. In this section, we introduce a new modular reduction method for moduli of this special shape. We note that it works similarly for any modulus of the form $k \cdot 2^m \pm l$, where k and l are small positive integers such that $k \geq 3$ and $l \geq 1$. However, for ease of exposition and to focus on the case most relevant in the context of the NTT, we only treat the case $q = k \cdot 2^m + 1$.

Let $0 \leq a, b < q$ be two integers modulo q and let $C = a \cdot b$ be their integer product. Then $0 \leq C < q^2 = k^2 2^{2m} + k 2^{m+1} + 1$. The goal is to reduce C modulo q using the special shape of q, namely using the fact that $k2^m \equiv -1 \pmod{q}$. Write $C = C_0 + 2^m C_1$, where $0 \leq C_0 < 2^m$. Then $0 \leq C_1 = (C - C_0)/2^m < k^2 2^m + 2k + 1/2^m = kq + k + 1/2^m$. We have that $kC \equiv kC_0 - C_1 \pmod{q}$, and given the above bounds for C_0 and C_1, it follows that the integer $kC_0 - C_1$ has absolute value bounded by $|kC_0 - C_1| < (k + 1/2^m)q$. As k is a small integer, the value $kC_0 - C_1$ can be brought into the range $[0, q)$ by adding or subtracting a small multiple of q. The maximal value for C is $(q-1)^2 = k^2 2^{2m}$, in which case $C_0 = 0$ and $C_1 = k^2 2^m = k(q-1)$, meaning that $(k-1)q$ must be added to $kC_0 - C_1$ to fully reduce the result. In our application to the NTT, however, we do not intend to perform this final reduction into $[0, q)$ throughout the computation, but rather only at the very end of the algorithm. We are therefore content with the output of the function `K-RED` defined as follows:

```
function K-RED($C$)
    $C_0 \leftarrow C \bmod 2^m$
    $C_1 \leftarrow C/2^m$
    return $kC_0 - C_1$
end function
```

The function `K-RED` can take any integer C as input. It then returns an integer D such that $D \equiv kC \pmod{q}$ and $|D| < q + |C|/2^m$. Although this function alone does not properly reduce the value C modulo q, we still call it a reduction because it brings D close to the desired range; note that for $|C| > (2^m/(2^m - 1))q$, we have $|D| < |\texttt{K-RED}(C)|$, i.e. it reduces the size of C. As a specific example, take $q = 12289 = 3 \cdot 2^{12} + 1$. Then $k = 3$ and `K-RED` returns $3C_0 - C_1 \equiv 3C \pmod{q}$ using the equivalence $3 \cdot 2^{12} \equiv -1 \pmod{q}$.

In the context of a specific, longer computation, and depending on the parameter n and the target platform, we note that additional reductions might need to be applied to a limited number of intermediate values, for which overflow may occur. In this case, as an optimization, two successive reductions can be merged as follows. Let the input operand C be decomposed as $C = C_0 + C_1 \cdot 2^m + C_2 2^{2m}$ with $0 \leq C_0, C_1 < 2^m$. Then we can reduce C via the following function `K-RED-2x`.

```
function K-RED-2x($C$)
    $C_0 \leftarrow C \bmod 2^m$
    $C_1 \leftarrow C/2^m \bmod 2^m$
    $C_2 \leftarrow C/2^{2m}$
    return $k^2 C_0 - kC_1 + C_2$
end function
```

A remark on residue classes. Note that `K-RED` outputs a value that is congruent to a multiple of the input, i.e., it changes the input's residue class modulo q. This is reminiscent of Montgomery reduction [22]. Indeed, one way to deal with this fact is to use a system of representatives for the residue classes, similar to the Montgomery representation. Namely, define a system of representatives by setting $\tilde{i} = i \cdot k^{-1} \bmod q$ for $0 \leq i < q$. Note that k is coprime to q by construction. Then, modular multiplication proceeds as follows: given two integers $x, y \in [0, p)$, let $\tilde{x} = x \cdot k^{-1} \bmod p$ and $\tilde{y} = y \cdot k^{-1} \bmod p$. After applying the reduction procedure to the product of $\tilde{x}$ and $\tilde{y}$, we obtain $\texttt{K-RED}(\tilde{x} \cdot \tilde{y}) \equiv \tilde{x}\tilde{y}k \equiv xy \cdot k^{-1} \pmod{q}$, which is a representative of the product of x and y.

Speeding up the NTT. In the context of the NTT algorithm, we use a redundant representation of integers modulo q by allowing them to grow up to 32 bits and, when necessary, apply the reduction function `K-RED` to reduce the sizes of coefficients. We keep track of the factors of k that are implicitly multiplied to the result by an invocation of `K-RED`. For the sake of illustration, consider Algorithm 1. The main idea is to apply the function `K-RED` only after multiplications, i.e., one reduction per iteration in the inner loop, letting intermediate coefficient values grow such that the final coefficient values become congruent to $K \cdot a[\cdot] \bmod q$ for a fixed factor K. This factor can then be used at the end of the NTT-based polynomial multiplication to correct the result to the desired value.

Next, we specify the details of the method for $n \in \{256, 512, 1024\}$ for the prime $q = 12289$. We limit the analysis to platforms with native 32 (or higher)-bit multipliers, but note that the presented algorithms can be easily modified to cover other settings.

The case $q = 12289$. The modified NTT algorithms using `K-RED` and `K-RED-2x` are shown in Algorithm 3 and Algorithm 4 for the modulus $q = 12289$, which in practice is used with $n = 512$ (for BLISS signatures [9]) or 1024 (for key exchange [2]). In Step 7 of both algorithms, we are using the precomputed values scaled by k^{-1}, i.e. we use precomputed tables $\Psi_{rev,k^{-1}}[\cdot] = k^{-1} \cdot \Psi_{rev}[\cdot]$ and $\Psi^{-1}_{rev,k^{-1}}[\cdot] = k^{-1} \cdot \Psi^{-1}_{rev}[\cdot]$. We denote these modified algorithms by $\texttt{NTT}^K := \text{NTT}^K_{CT,\psi_{rev,k^{-1}}}$ and $\texttt{INTT}^K := \text{INTT}^K_{GS,\Psi^{-1}_{rev,k^{-1}}}$, respectively.

Given two input vectors a and b, let $c = \texttt{INTT}(\texttt{NTT}(a) \circ \texttt{NTT}(b))$ be computed using Algorithms 1 and 2. It is easy to see that the resulting coefficients after applying Algorithms 3 and 4, i.e., after computing $\texttt{INTT}^K(\texttt{NTT}^K(a) \circ \texttt{NTT}^K(b))$, are congruent to $K \cdot c[\cdot]$ modulo q for a certain fixed integer $K = k^s$ and an integer s. Note that by scaling the precomputed twiddle factors by $k^{-1} \bmod q$, we can limit the growth of the power of k introduced by the reduction steps. For example in Line 10 of Algorithm 3 the value S carries a factor k^{-1} which then cancels with the factor k introduced by `K-RED` in Step 15. Only additional reductions such as those in Steps 12 and 13 increase the power of k in the final result.

At the end of the computation, the final results can be converted back to the standard representation by multiplying with the inverse of the factor K. Moreover, this conversion can be obtained for free if the computation is merged with the scaling by n^{-1} during the inverse transformation, that is, if scaling is performed by multiplying the resulting vector with the value $n^{-1} \cdot K^{-1}$. However, we can do even better: by merging the second entry of the table $\Psi_{rev,k^{-1}}$ with the fixed value $n^{-1} \cdot K^{-1}$, we eliminate an additional $n/2$ multiplications and modular reductions. This is shown in Steps 24–29 of Algorithm 4.

4 Case Study: R-LWE Key Exchange

This section explains how we apply our new modular reduction and the improved NTT algorithms, together with a simplified message encoding, to the instantiation of Peikert's key exchange [24] that was proposed by Alkim, Ducas, Pöppelmann and Schwabe in [2]; the protocol is depicted in Figure 1. Accordingly, from hereon we fix $n = 1024$ and $q = 12289$

Algorithm 3 Modified function $\mathtt{NTT}^K$ using `K-RED` and `K-RED-2x` for reduction modulo $q = 12289$ (32 or 64-bit platform).

Input: A vector $a = (a[0], a[1], ..., a[n-1]) \in \mathbb{Z}_q^n$ in standard ordering, where $q = 12289$ and $n \in \{256, 512, 1024\}$, and a precomputed table $\Psi_{rev,k^{-1}} \in \mathbb{Z}_q^n$ storing scaled powers of ψ in bit-reversed order.
Output: $a \leftarrow \mathtt{NTT}^K(a)$ in bit-reversed ordering.

```
1: $k = n$
2: for ($m = 1$; $m < n$; $m = 2m$) do
3:     $k = k/2$
4:     for ($i = 0$; $i < m$; $i{+}{+}$) do
5:         $j_1 = 2 \cdot i \cdot k$
6:         $j_2 = j_1 + k - 1$
7:         $S = \Psi_{rev,k^{-1}}[m + i]$
8:         for ($j = j_1$; $j \leq j_2$; $j{+}{+}$) do
9:             $U = a[j]$
10:            $V = a[j + k] \cdot S$
11:            if $m = 128$ then
12:                $U = \mathtt{K\text{-}RED}(U)$
13:                $V = \mathtt{K\text{-}RED\text{-}2x}(V)$
14:            else
15:                $V = \mathtt{K\text{-}RED}(V)$
16:            end if
17:            $a[j] = U + V$
18:            $a[j + k] = U - V$
19:        end for
20:    end for
21: end for
22: return $a$
```

and the error distribution is defined to be the centered binomial distribution ψ_{12}, from which one samples by computing $\sum_{i=1}^{12}(b_i - b'_i)$, where the $b_i, b'_i \in \{0, 1\}$ are uniform independent bits. The functions `HelpRec` and `Rec` are modified instantiations of Peikert's reconciliation functions [24, §3] that essentially turn approximate key agreement into *exact* key agreement – see [2]. The function `SHAKE-128` is the extended output function (XOF) based on Keccak [4], which is also used to derive the 256-bit shared secret key in both Alice and Bob's final steps. Following [2], the random value a is generated directly in the NTT domain.

Viewing Figure 1, we identify the following NTT-based computations:

Alice	**Bob**
$b \leftarrow a \circ \mathrm{NTT}(s) + \mathrm{NTT}(e)$	$u \leftarrow a \circ \mathrm{NTT}(s') + \mathrm{NTT}(e')$
$v' \leftarrow \mathrm{INTT}\,(u \circ \mathrm{NTT}(s))$	$v \leftarrow \mathrm{INTT}\,(b \circ \mathrm{NTT}(s') + \mathrm{NTT}(e''))$

The sequence of NTT and INTT operations above are used to determine the value of K that results from our target parameters; note that $q = 3 \cdot 2^{12} + 1$ and thus $k = 3$. For determining K, Alice's and Bob's NTT/INTT computations can be seen as *two* polynomial operations: (1) the first operation begins with the computation of b on Alice's side, who then transmits it in the NTT domain to Bob for computing v and giving the result back in the standard domain; and similarly (2) the second operation consists of the computation of u on Bob's side followed by the computation of v' on Alice's side.

We first point out that if we include two extra reductions at stage $m = 128$ and $m = 32$ of the NTT and INTT algorithms, respectively, then intermediate values never grow beyond 32

Algorithm 4 Modified function $\texttt{INTT}^K$ using `K-RED` and `K-RED-2x` for reduction modulo $q = 12289$ (32 or 64-bit platform).

Input: A vector $a = (a[0], a[1], ..., a[n-1]) \in \mathbb{Z}_q^n$ in bit-reversed ordering, where $q = 12289$ and $n \in \{256, 512, 1024\}$, a precomputed table $\Psi^{-1}_{rev,k^{-1}} \in \mathbb{Z}_q^n$ storing scaled powers of ψ^{-1} in bit-reversed order, and the constants $n_K^{-1} = n^{-1} \cdot k^{-11}$, $\Psi_K^{-1} = n^{-1} \cdot k^{-10} \cdot \Psi^{-1}_{rev,k^{-1}}[1] \in \mathbb{Z}_q$, where $k = 3$.
Output: $a \leftarrow \texttt{INTT}^K(a)$ in standard ordering.

```
 1: $k = 1$
 2: for ($m = n$; $m > 2$; $m = m/2$) do
 3:     $j_1 = 0$
 4:     $h = m/2$
 5:     for ($i = 0$; $i < h$; $i$++) do
 6:         $j_2 = j_1 + k - 1$
 7:         $S = \Psi^{-1}_{rev,k^{-1}}[h + i]$
 8:         for ($j = j_1$; $j \leq j_2$; $j$++) do
 9:             $U = a[j]$
10:             $V = a[j + k]$
11:             $a[j] = U + V$
12:             $a[j + k] = (U - V) \cdot S$
13:             if $m = 32$ then
14:                 $a[j] = \texttt{K-RED}(a[j])$
15:                 $a[j + k] = \texttt{K-RED-2x}(a[j + k])$
16:             else
17:                 $a[j + k] = \texttt{K-RED}(a[j + k])$
18:             end if
19:         end for
20:         $j_1 = j_1 + 2k$
21:     end for
22:     $k = 2k$
23: end for
24: for ($j = 0$; $j < k$; $j$++) do
25:     $U = a[j]$
26:     $V = a[j + k]$
27:     $a[j] = \texttt{K-RED}((U + V) \cdot n_K^{-1})$
28:     $a[j + k] = \texttt{K-RED}((U - V) \cdot \Psi_K^{-1})$
29: end for
30: return $a$
```

bits during a full NTT or INTT computation (see steps 11–13 of Algorithm 3 and steps 13–15 of Algorithm 4). Following Section 3, the factor k introduced by every invocation of `K-RED` is canceled out by the corresponding multiplication with an entry from the $\Psi_{\text{rev},k^{-1}}$ and $\Psi^{-1}_{\text{rev},k^{-1}}$ tables. Hence, only the extra reductions above introduce a factor k to the intermediate results of the NTT and INTT.

Secondly, we point out that after performing component-wise multiplications of polynomials in the NTT domain, the individual factors get compounded. The results after these multiplications require two additional reductions and a conditional subtraction per coefficient to fully reduce them modulo q (this is required to avoid overflows and, when applicable, to transmit messages and derive shared keys in fully reduced form). It is important to keep track of these factors and to (i) ensure that they are balanced (i.e., the same) before, e.g., adding two summands that are the result of different NTT operations, and (ii) ensure that they are corrected at the end of the computation. Careful analysis of the above sequence of

Public parameters		
$n = 1024$, $q = 12289$, error distribution ψ_{12}		
Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$a \leftarrow \texttt{SHAKE-128}(seed)$		
$s, e \xleftarrow{\$} \psi_{12}^n$		$s', e', e'' \xleftarrow{\$} \psi_{12}^n$
$b \leftarrow as + e$	$\xrightarrow{m_A=(b,seed)}$	$a \leftarrow \texttt{SHAKE-128}(seed)$
		$u \leftarrow as' + e'$
		$v \leftarrow bs' + e''$
$v' \leftarrow us$	$\xleftarrow{m_B=(u,r)}$	$r \xleftarrow{\$} \texttt{HelpRec}(v)$
$\nu \leftarrow \texttt{Rec}(v', r)$		$\nu \leftarrow \texttt{Rec}(v, r)$
$\mu \leftarrow \texttt{SHA3-256}(\nu)$		$\mu \leftarrow \texttt{SHA3-256}(\nu)$

Fig. 1. The instantiation of Peikert's key exchange [24] from [2].

NTT operations reveals that the final factor is $K = k^{10} = 3^{10}$ for the two full polynomial operations mentioned before.

Message encoding and decoding. The messages exchanged between Alice and Bob need to be encoded efficiently to minimize encoding/decoding costs and network delays. In particular, we want to reduce the size of the messages in order to alleviate the network overhead, which can often be expensive. Internally, polynomials are encoded as 1024-element little-endian arrays, where each element or coefficient is represented either by a 32-bit signed integer (for secret keys and error polynomials) or a 32-bit unsigned integer (for everything else). Each coefficient that is part of a message is fully reduced modulo q before transmission and therefore only uses a fraction of the integer size (i.e., 14 bits). We simply encode messages in little endian format as a concatenation of these 1024 14-bit coefficients (for b and u; see Figure 1) immediately followed by the 256-bit *seed* in Alice's message and the 1024 2-bit array r in Bob's message. Accordingly, m_A and m_B consist of 1824 and 2048 bytes, respectively[2].

5 Cost analysis of the new NTT and Implementation Results

In this section, we carry out a theoretical analysis of the expected performance of the proposed reduction method inside the NTT and compare it with other efficient methods in the literature: Harvey's modified Shoup butterfly algorithm [13], the method by Aguilar-Melchor et al. [1], which adapts Harvey's variant of the Shoup butterfly for certain special-form primes, and the method by Alkim et al. [2], which adapts Harvey's variant of the Montgomery butterfly to $q = 12289$. We then present implementation results showcasing the performance of the new NTT algorithms and, in particular, benchmark them in the context of the Ring-LWE key exchange by Alkim et al. [2].

First, we carry out a basic operation count for the different approaches for the NTT.

[2] Adam Langley also mentions a compact encoding without giving the exact details in a blog post, see `https://www.imperialviolet.org/2015/12/24/rlwe.html`. We assume that the implied encoding is similar to the method described here, but seems to differ for Bob's message.

5.1 Basic operation count

Let us assume an x64 CPU as the targeted platform, and the use of a 14-bit modulus. Accordingly, the operations in the following analysis can be interpreted as 32 or 64-bit instructions.

A basic operation count reveals that the reduction method by Aguilar-Melchor et al. requires 3 multiplications, 4 additions/subtractions, 3 shifts, and 1 conditional subtraction in the inner loop of the NTT computation (see [1, Alg. 2]). It appears that the conditional subtraction is implemented in *variable-time* in the NFLlib software that accompanies [1]. A constant-time implementation of a conditional subtraction can be implemented with 1 subtraction, 1 shift, and 3 logical `and/xor` operations, which brings the total to 3 multiplications, 5 additions/subtractions, 4 shifts, and 3 logical `and/xor` operations. Harvey's modified Shoup butterfly appears to have a slightly lower cost with a count of 3 multiplications, 5 additions/subtractions, 2 shifts, and 3 logical `and/xor` operations to perform the same computation [13, Alg. 3] (also considering the same conversion for the variable-time conditional subtraction). Alkim et al.'s approach improves the efficiency of the previous methods by requiring 3 multiplications, 4 additions/subtractions, 1 shift, and 1 logical `and` operation. In contrast, the method proposed here only requires 1 multiplication, 4 additions/subtractions, 2 shifts and 1 logical `and` operation. In some cases (e.g., for $q = 12289$), the technique allows to trade 1 addition and 1 shift for 1 multiplication. Then, the cost would be 2 multiplications, 3 additions/subtractions, 1 shift and 1 logical `and` operation. In either case, the new method introduces reductions in the number of multiplications and other operations in comparison with all of the approaches above.

Next, we carry out an analysis of the proposed method on an Intel Haswell CPU. Following the analysis above, we include Alkim et al.'s approach in our comparisons since it appears to offer the lowest cost among the previous methods.

5.2 Analysis on a Haswell processor

We focus the analysis on the case of $n = 1024$ with $q = 12289$ and assume an Intel Haswell processor as the targeted platform. We use notation as defined in Section 3.

As hinted above, given a prime $q = k \cdot 2^m + 1$, there are two ways to compute kC_0 in the function `K-RED`: using additions (if the value k is relatively small) or 1 multiplication. We begin by focusing on the first option.

The main loop of Algorithm 3 consists of 1 multiplication, 1 addition and 1 subtraction (outside of the reduction) and 1 logical `and`, 2 shifts, 1 subtraction and 1 addition (during the reduction). On Haswell, the execution of these operations has a theoretical latency of approximately 7 cycles[3]. With $n = 1024$ and assuming that the software is not unrolled (e.g., it is based on a compact C implementation), then we can expect a theoretical performance of approximately $10 \times 512 \times 7 = 35840$ cycles for computing the forward NTT. The inverse NTT should have a slightly higher latency since it requires a scaling at the end (see Algorithm 4).

In comparison, the Montgomery-based method used in [2] requires more multiplications which are expected to increase the computing cost on many platforms (see [2, §7.2] for details). A similar analysis to the above yields that an inner loop using Montgomery reductions achieves a theoretical latency of approximately 14 cycles on a Haswell processor. This translates to a

[3] We assume that integer multiplications have a latency of 3 cycles [10] while other operations have one-cycle latencies. Note that with the exception of the multiplication and one of the additions, instructions can be paired and executed in one single cycle.

total estimate of $10 \times 512 \times 14 = 71680$ cycles for the full NTT. Thus, in theory the proposed method provides a factor-2 speedup.

On some modern CPUs, implementers can exploit powerful SIMD instructions to gain an extra performance boost. For example, the AVX2 instruction set permits to operate on 256-bit wide vectors. In this case a cost estimate for the NTT's inner loop using the new reduction method with vector integer instructions is about $10 \times 512/4 \times 9 = 11520$ cycles (the division by 4 is because each vector instruction is capable of operating on four 32-bit values per instruction). It is possible to further reduce the cost by applying optimizations such as loop unrolling, which facilitates hiding latencies arising from multiplications (sometimes via the use of assembly). In this case, if we assume that the processor is able to hide most latencies, the theoretical lower bound is about $10 \times 512/4 \times 5 = 6400$ cycles.

In contrast, Alkim et al.'s AVX2 implementation [2] uses vector floating-point arithmetic. They mention that, in their case, the use of these instructions provides better performance than with the use of vector integer instructions. This complies with the expected performance on Haswell; using Montgomery reduction requires many more multiplication instructions, and vector floating-point multiplications have the same latency (5 cycles) than their vector integer counterparts, but have twice the reciprocal throughput (0.5 cycles versus 1 cycle). Using floating-point arithmetic, the Montgomery approach achieves an estimate of $10 \times 512/4 \times 23 = 29440$ cycles. This can be reduced significantly by carrying out an aggressive unrolling and instruction scheduling in assembly to hide most multiplication and addition latencies. A lower bound in this case is given by $10 \times 512/4 \times (5.5 + 6.5)/2 = 8320$ cycles (we estimate 6.5 cycles for iterations containing Barrett reductions and 5.5 cycles for the other case). Thus, in theory the proposed NTT provides a 1.3x speedup over the NTT from [2]. Furthermore, we note that even when employing floating-point arithmetic in our approach, the expected performance is still better than that in [2]: $10 \times 512/4 \times 18 = 23040$ cycles, or $10 \times 512/4 \times 5.5 = 7040$ cycles (assuming the use of unrolling and optimal instruction scheduling).

We now summarize the second option mentioned at the beginning of this section: the use of a multiplication to compute kC_0. In the case of $q = 12289$, one substitutes 1 multiplication by k for 1 shift and 1 addition in the main loop of Algorithm 3. Under this assumption, the lower bound using vector integer instructions is given by $10 \times 512/4 \times 13 = 16640$ cycles, or $10 \times 512/4 \times 5 = 6400$ cycles, if we assume unrolling and optimal instruction scheduling (the same result is also achieved if we assume the use of vector floating-point arithmetic). Thus, without considering other practical factors, the analysis suggests that this approach provides similar performance to the case based on additions.

This analysis highlights the flexibility and the expected performance of the proposed method, which can be accommodated to different requirements and design goals.

5.3 Performance benchmarks

To ease the comparison with the state-of-the-art NTT implementation, we followed [2] and implemented *two* versions of the proposed NTT algorithms [17]: a portable and compact implementation written in the C language, and a high-speed implementation written in x64 assembly and exploiting AVX2 instructions. Based on the analysis from the previous section, for the AVX2 implementation we decided to use vector integer instructions, which are easier to work with and are expected to provide similar performance to a version using vector floating-point instructions.

The benchmarking results of our implementations are shown at the top of Table 1. These results were obtained by running the implementations on a 3.4GHz Intel Core i7-4770 Haswell processor with TurboBoost disabled. For compilation we used gcc v4.9.2 for the C implementation and clang v3.6.0 for the AVX2 implementation.

As one can see, for the C version, the new forward and inverse NTT implementations are 1.86 and 1.90 times faster than the corresponding implementations from Alkim et al. [2]. Our results confirm the cost estimates in §5.2. We achieve slightly better performance than expected, and this is probably due to some clever rescheduling done by the compiler (or the out-of-order processor) that hides the latency of the integer multiplier. Our prediction of the speedup over Alkim et al.'s implementation is also precise. The small error in the estimate might be due to Alkim et al.'s partitioning of the computation in "even" and "odd" iterations, which could facilitate the compiler's optimization work. The same approach could also be applied to our implementations to gain an additional speedup.

For the AVX2 version, the new algorithms achieve factor-1.21 and 1.25 speedups in comparison to Alkim et al.'s results. These results closely follow the cost estimates in the analysis from §5.2, meaning that the results are close to optimal. In this case, the extra overhead is mainly due to the cost of the additional instructions that are required to manipulate data (this relates to those instructions which the compiler/processor is not able to schedule perfectly). We also checked that implementing the two versions of the reduction (i.e., using additions or multiplying by k) achieve similar performance, as predicted by our analysis.

The faster performance of our algorithms is mainly due to the cheaper reduction technique, which in turn eliminates *all* of the corrections required after the NTT's internal additions and enables the efficient use of signed arithmetic (this has an overall positive effect at the protocol level as we discuss below). For example, using signed arithmetic avoids the correction of intermediate values to positive before reductions. In addition, we eliminate the need for the bit-reversal step by applying an efficient combination of NTT algorithms, and minimize the cost of additional overhead by merging operations; e.g., we merge the scaling by n^{-1} with the twiddle factors and then absorb this computation into the last stage of the inverse NTT (see §3). We note that the direct comparison of NTT implementations in Table 1 does not account for the additional benefits of our technique that are not observable at the NTT level. This includes the elimination of costly conversion routines required by the Montgomery arithmetic (as used in [2]) that are performed outside of the NTT.

To illustrate the *overall* performance benefits of the new reduction and NTT algorithms, we implemented the full key-exchange instantiation proposed by Alkim et al. [2]. To ease the comparison, we reuse the same routines for `HelpRec` and `Rec` (for error reconciliation) available in the software accompanying [2], and also reuse the same implementations of ChaCha20 and `SHAKE-128` used in Alkim et al.'s software for the seed expansion during the generation of a and for the polynomial error sampling, respectively.

Our results for the key exchange are summarized in Table 1. The C and AVX2 implementations are roughly 1.40x and 1.34x faster, respectively, than the corresponding C and AVX2 implementations by Alkim et al. These improvements are due to the the new NTT algorithms with faster reduction and the faster and compact message encodings. Part of the speedup is also the result of avoiding costly conversions when working with Montgomery arithmetic. In addition, using signed arithmetic makes computations more efficient because corrections from negative to positive values are not required in several of the key exchange routines. We remark that these performance improvements are obtained with significantly simpler integer arithmetic.

Table 1. Benchmarking results (in terms of 10^3 cycles) of our C and AVX2 implementations of the NTT and the key-exchange instantiation proposed by Alkim et al. [2] on a 3.4GHz Intel Core i7-4770 Haswell processor with TurboBoost disabled. Results are compared with Alkim et al.'s implementation results. At the bottom of the table, we show the total cost of a key-exchange, including Alice's and Bob's computations.

	C implementation		AVX2 implementation	
	ADPS [2]	This work	ADPS [2]	This work
NTT	55.9	**30.1**	11.0	**9.1**
INTT	60.3	**31.8**	12.1	**9.7**
Generating a	57.0	39.5	57.3	37.8
Error sampling	33.8	31.4	6.2	4.8
`HelpRec`	14.9	12.9	3.4	2.4
`Rec`	10.1	7.2	2.8	1.2
Key gen (server)	271.7	**194.8**	115.4	**92.1**
Key gen + shared key (client)	402.1	**298.9**	144.8	**105.0**
Shared key (server)	86.6	**49.7**	24.0	**15.7**
Total (key exchange)	760.4	**543.4**	284.2	**212.8**

A different Ring-LWE based kex-exchange implementation has been recently reported by Aguilar-Melchor et al. [1]. Direct comparisons with this work are especially difficult because they use different parameters and the most recent version of their implementation appears not to be protected against timing and cache attacks. As a point of reference, we mention that [1, Table2] reports that their NTT implementation using $N = 512$ and a 30-bit modulus runs in 13K cycles on a 2.9GHz Intel Haswell machine (scaled from 4.5 microseconds). This is more than 1.4x slower than our NTT using $N = 1024$ and a 14-bit modulus.

6 Conclusion

We describe a new modular reduction technique and improved FFT algorithms to compute the NTT. The improved NTT algorithms were applied to a recent key exchange proposal and showed significant improvements in performance using both a plain C implementation and a vectorized implementation that does not require floating-point arithmetic.

Although both the modular reduction and the improved NTT were motivated by (and are somewhat tailored towards) applications in R-LWE cryptography that use power-of-2 cyclotomic fields, our improvements should be of independent interest and might be applicable to other scenarios. Our method offers flexibility for implementations with different design goals without sacrificing performance.

References

1. C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint. NFLlib: NTT-based fast lattice library. In *Topics in Cryptology - CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2016. Software available at: https://github.com/quarkslab/NFLlib, accessed on May 20, 2016.
2. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - a new hope. *IACR Cryptology ePrint Archive, Report 2015/1092*, 2015. To appear at Usenix Security 2016. Software available at: https://github.com/tpoeppelmann/newhope, accessed on May 20, 2016.
3. S. Bai, A. Langlois, T. Lepoint, D. Stehlé, and R. Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. In Tetsu Iwata and Jung Hee

Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015*, volume 9452 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2015.

4. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak. In T.Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.
5. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015*, pages 553–570. IEEE Computer Society, 2015.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):13:1–13:36, 2014.
7. E. Chu and A. George. *Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms.* CRC Press, Boca Raton, FL, USA, 2000.
8. J. W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
9. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal Gaussians. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.
10. A. Fog. Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, accessed on 03/04/2016. `http://www.agner.org/optimize/`.
11. C. F. Gauss. Nachlass, Theoria Interpolationis Methodo Nova Tractata. *Carl Friedrich Gauss Werke, Band 3*, pages 265–330, 1866.
12. W. M. Gentleman and G. Sande. Fast Fourier transforms: For fun and profit. In *Fall Joint Computer Conference*, AFIPS '66, pages 563–578, New York, NY, USA, 1966. ACM.
13. D. Harvey. Faster arithmetic for number-theoretic transforms. *J. Symb. Comput.*, 60:113–119, 2014.
14. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory, ANTS-III*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.
15. D. E. Knuth. *Seminumerical Algorithms.* The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.
16. Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient Ring-LWE encryption on 8-bit AVR processors. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 663–682. Springer, 2015.
17. P. Longa and M. Naehrig. LatticeCrypto. `http://research.microsoft.com/en-us/downloads/2b241c5c-0972-4c76-bf59-6ef5b1389de7/default.aspx`, 2016.
18. V. Lyubashevsky. Lattice signatures without trapdoors. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.
19. V. Lyubashevsky, D. Micciancio, C. Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In K. Nyberg, editor, *Fast Software Encryption - FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2008.
20. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.
21. D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2013.
22. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
23. H. J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(2):205–215, 1980.
24. C. Peikert. Lattice cryptography for the internet. In M. Mosca, editor, *Post-Quantum Cryptography - PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 197–219. Springer, 2014.
25. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2014.
26. T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in*

Cryptology - LATINCRYPT 2015, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2015.

27. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 84–93, 2005.
28. S. Sinha Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.
29. V. Shoup. Number Theory Library (NTL). 1996-2016. `http://www.shoup.net/ntl`.
30. D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2011.
31. J. Zhang, Z. Zhang, J., M. Snook, and Özgür Dagdelen. Authenticated key exchange from ideal lattices. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 719–751. Springer, 2015.