# When NTT Meets Karatsuba: Preprocess-then-NTT Technique Revisited

Yiming Zhu[1,2], Zhen Liu[1,2], and Yanbin Pan[1(✉)]

[1] Key Laboratory of Mathematics Mechanization, NCMIS,
Academy of Mathematics and Systems Science, Chinese Academy of Sciences
Beijing, 100190, China
panyanbin@amss.ac.cn
[2] School of Mathematical Sciences, University of Chinese Academy of Sciences,
Beijing 100049, China
liuzhen16@mails.ucas.ac.cn
zhuyiming17@mails.ucas.ac.cn

**Abstract.** The Number Theoretic Transform (NTT) technique is widely used in the implementation of the cryptographic schemes based on the Ring Learning With Errors problem(RLWE), since it provides efficient algorithm for multiplication of polynomials over the finite field. However, to employ NTT, the finite field is required to have some special root of unity, such as $n$-th root, which makes the module $q$ in RLWE big since we need $q \equiv 1 \mod 2n$ to ensure such root exits. At Inscrypt 2018, Zhou *et al.* proposed a technique called preprocess-then-NTT to reduce the value of modulus $q$ while the NTT still works, and the time complexity is just a constant ($> 1$) multiple of the original NTT algorithm asymptotically. In this paper, we revisit the preprocess-then-NTT technique and point out it can be improved such that its time complexity is as the same as the original NTT algorithm asymptotically. What's more, through experiments we find that even compared with the original NTT our improved algorithm may have some advantages in efficiency.

**Keywords:** NTT, Ring Learning With Errors, preprocess-then-NTT

## 1 Introduction

In recent years, there have been a substantial amount of research in quantum computers. If quantum computers are ever builted, the public-key cryptosystems currently in use, based on the hardness of solving (elliptic curve) discrete logarithm or factoring large integers, will be broken easily [10]. To avoid this problem, the concept of post-quantum cryptography has been proposed. Lattice-based cryptography is widely believed to be a promising candidate in post-quantum cryptography. Compared with the classic hard lattice problems such as the shortest vector problem and the closest vector problem, the learning with errors problem (LWE) [9] appears to be much more versatile in the construction of cryptographic construction, especially the Ring-Learning With Errors problem (RLWE) [8].

Simply speaking, RLWE works on some polynomial ring, usually $\mathbb{Z}_q[x]/(x^n+1)$, so a polynomial can be used in RLWE instead of a matrix in the original LWE, which leads to a mush smaller cost to represent an RLWE sample than an LWE sample. RLWE is widely used in the construction of public-key encryption, digital signatures, key exchange and so on, such as [8, 6, 1]. To further improve the efficiency of RLWE-based schemes, the time-consuming discrete Gauss distribution is usually replaced with some other discrete distribution easy to sample. Hence, the most time-consuming operation in the implementation of RLWE-based schemes becomes the polynomial multiplication in the ring.

Many algorithms have been proposed to improve the efficiency of the polynomial multiplication, such as Karatsuba algorithm [5, 4] and Fast Fourier Transform (FFT) algorithm [3]. For RLWE, since the coefficient ring of the polynomial ring is usually a residue class ring, such as $\mathbb{Z}_q$, an FFT-analogy called Number Theoretic Transform (NTT) was proposed by using the root of unity in the residue class ring instead of the field of complexity number. For example, considering the polynomial multiplication in the widely used ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ where $q$ is a prime and $n$ is some power of 2, the product can be efficiently computed by NTT when there exists $2n$-th root of unity in $\mathbb{Z}_q$, that is, $2n|(q-1)$ [7]. As a result, applying the NTT algorithm significantly improves the efficiency of these RLWE-based schemes.

Note that the modulus $q$ is required to satisfy that $2n|(q-1)$ in NTT. A large enough $q$ should be chosen for the RLWE-based cryptographic schemes, which obviously enlarges the key size and decreases the efficiency. To overcome the issue, at Inscrypt 2018 Zhou et al.[11] presented a technique called preprocess-then-NTT which just requires $n|(q-1)$ or $\frac{n}{2}|(q-1)$ to weaken the restriction of modulus $q$. Their main idea is dividing the polynomial into new low-dimentional polynomials according to the parity of index and then applying the NTT to the new low-dimensional polynomials respectively. Moreover, they showed that the time complexity of their 1-Round Preprocess-then-NTT (1PtNTT) in the case that $n|(q-1)$ would be $\frac{7}{6}$ times of the original NTT and the time complexity of their 2-Round Preprocess-then-NTT (2PtNTT) in the case that $\frac{n}{2}|(q-1)$ would be $\frac{5}{4}$ times of the original NTT.

In this paper, we revisit the preprocess-then-NTT technique.

At first, we present a method with divide-and-conquer strategy to reduce the value of modulus $q$, which can be seen as the hybrid of the Karatsuba algorithm and NTT algorithm. In analogy with the idea of Karatsuba algorithm [5, 4], we simply divide the polynomial into several polynomials of lower degree, then apply the NTT on the multiplications of low-degree polynomials, and at last get the multiplication by the Karatsuba algorithm. With the technique, modulus $q$ is only required to satisfy that $\frac{n}{2^{\alpha-1}}|(q-1)$ when we divide the polynomial into $2^\alpha$ parts. The technique is quite easy to be understood but cost $\frac{5}{3}$ times of the original NTT algorithm.

Then we turn to improve the preprocess-then-NTT technique, and find that we can save one NTT computation in Zhou *et al.*'s algorithm, and then we prove that the time complexity of the improved algorithm is as the same as the origi-

nal NTT asymptotically. Furthermore, we save several point-wise multiplication computations to improve the efficiency of our improved algorithm.

**Roadmap.** The remainder of the paper is organized as follows. In Section 2 we present some premilinaries. In Section 3 we present our first algorithms based on Karatsuba algorithm and NTT.In Section 4 we present an improved version of the preprocess-then-NTT technique. In section 5 we give the experiment results of the techniques. In Section 6 we give the conclusion.

## 2 Preliminaries

### 2.1 Ring-Learning With Errors (RLWE) problem

The Learning with Errors (LWE) problem was raised by Regev [9] who proved that, solving a random LWE instance is as hard as solving worst-case instances of certain lattice problems under a quantum reduction. Later on, Lyubashevsky, Peikert and Regev[8] proposed a variant of the LWE problem, the Ring-LWE problem, whose hardness is related to the worst case hardness of finding short vectors in ideal lattices.

**The Ring Learning with Errors problem** The decisional version of the Ring Learning with Errors problem, with parameters $m$, modulo $q$ and error distribution $\chi$, is defined as follows: given $m$ samples either all of them are in the form $(a, b = a \cdot s + e \mod q)$ where for a secret $s \in R_q$ is fixed, $a$ is uniformly randomly chosen from $R_q$ and $e$ is chosen according to the distribution $\chi$, or all of them are in the form $(a, b)$ where $(a, b)$ is uniformly randomly chosen from the uniform distribution over $R_q \times R_q$, decide whether the samples come from the former or the latter case.

### 2.2 Karatsuba algorithm

Karatsuba [5, 4] introduced a method to perform multiplication of large numbers in fewer operations than the usual direct multiplication. The idea can also be used in computing the product of two polynomials.

Denote by $f(x)$, $g(x)$ two polynomials of degree bounded by n. Karatsuba algorithm first divides them into polynomials of lower degrees as follows:

$$f = f_0 + x^{\frac{n}{2}} f_1 \ , \ g = g_0 + x^{\frac{n}{2}} g_1.$$

Then the product can be computed as follows:

$$\begin{aligned} h &= f \cdot g \\ &= (f_0 + x^{\frac{n}{2}} f_1) \cdot (g_0 + x^{\frac{n}{2}} g_1) \\ &= f_0 \cdot g_0 + x^n f_1 \cdot g_1 + x^{\frac{n}{2}} (f_1 \cdot g_0 + f_0 \cdot g_1) \\ &= f_0 \cdot g_0 - f_1 \cdot g_1 + x^{\frac{n}{2}} (f_1 \cdot g_0 + f_0 \cdot g_1) \\ &= f_0 \cdot g_0 - f_1 \cdot g_1 + x^{\frac{n}{2}} ((f_0 + f_1) \cdot (g_0 + g_1) - f_0 \cdot g_0 - f_1 \cdot g_1) \end{aligned}$$

Thus by using Karatsuba algorithm we only need to compute products of polynomials of lower degrees for 3times to compute the original product. Repeating the process until the degree of the polynomial is zero, the complexity of multiplication operations for the final algorithm is shown to be $O(n^{\log 3})$.

### 2.3 Number-theoretic transform

The Number Theoretic Transform (NTT)[3] is a specialized version of the Fast Fourier Transform (FFT) over a finite field.

Considering the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ where $n$ is some power of 2, and $q$ is a prime satisfying that $2n|(q-1)$, let $\omega$ be an $n$-th primitive root of unity in $\mathbb{Z}_q$, and $f$ be any element of $R_q$. Then for $0 \leq i \leq n-1$, we define the forward transformation $\hat{f} = NTT(f)$ by $\hat{f}_i = \sum_{j=0}^{n-1} f_j \cdot \omega^{ij} \mod q$,and the inverse transformation $f = NTT^{-1}(\hat{f})$ is given by $f_i = n^{-1} \sum_{j=0}^{n-1} \hat{f}_j \cdot \omega^{-ij} \mod q$. Notice that $f = NTT^{-1}(NTT(f))$ holds.

Let $f$, $g$ be elements of $R_q$. Computing $h = f \cdot g \mod x^n+1$ would require applying the NTT of length $2n$ at the first glance. Then $h$ is given by $NTT^{-1}(NTT(\hat{f} \circ \hat{g})) \mod x^n+1$, where $\circ$ is the component-wise product. However, this method will require the computation of a reduction modulo $x^n+1$ and it seems not necessary to use a $2n$-point NTT since the degree of $h$ is at most $n-2$. In order to avoid this, an algorithm was introduced in [7], which behaves as follows. We denote $\gamma$ the $2n$-th primitive root of unity in $\mathbb{Z}_q$ such that $\gamma = \sqrt{\omega}$, and let $\tilde{f} = (f_0, \gamma f_1, \cdots, \gamma^{n-1} f_{n-1})$ and $\tilde{g} = (g_0, \gamma g_1, \cdots, \gamma^{n-1} g_{n-1})$, then $h$ is given by $(1, \gamma^{-1}, \cdots, \gamma^{1-n}) \circ NTT^{-1}(NTT(\tilde{f} \circ \tilde{g}))$.

Notice here we require that q satisfies that $2n|(q-1)$,which guarantees the existence of a 2n-th primitive root of unity in $\mathbb{Z}_q$.

To analyze the time complexity, we just count the number of multiplications. The time complexity of the operations for a forward transformation $NTT$ is determined by a function $T(n) = n \log n$ in [2], and same for the inverse transformation $NTT^{-1}$. The time complexity of the point-wise multiplications, computing $(1, \omega, \cdots, \omega^{n-1})$ are all bounded by $n$. Since the multiplication of two polynomials contains two forward transformations, three point-wise multiplication and one inverse transformation, the total time complexity is given by

$$T(n) = 3n \log n + O(n).$$

## 3 A hybrid algorithm of Karatsuba and NTT for the multiplication in $R_q$

### 3.1 2-Part-Sepration when $n|(q-1)$

To decrease the value of $q$, we will not have the $2n$-th roots in the finite field and then NTT will not work. However, if we compute the multiplication of several lower-degree polynomials as what we do in the Karatsuba algorithm, then we do not need the exitance of the $2n$-th roots. Hence we can first divide the

polynomials into several polynomials of lower degrees similar to the Karatsuba algorithm, and then apply NTT algorithm on multiplications of polynomials with lower degrees, which can decrease the value of modulus q. This is our idea.

We first consider to divide each polynomial into 2 parts as follows:

$$f = f_0 + x^{\frac{n}{2}} f_1 \ , \ g = g_0 + x^{\frac{n}{2}} g_1.$$

In this case the degree of product of $f_i$ and $g_j$ is bounded by $n$, which means we only need $n|(q-1)$. Thus it dose decrease the value of modulus q. We denote $\omega$ the n-th primitive root of unity in $\mathbb{Z}_q$, $f$, $g$ elements of $R_q$ and we can compute the product $h$ as follows:

$$\begin{aligned} h &= f \cdot g \\ &= (f_0 + x^{\frac{n}{2}} f_1) \cdot (g_0 + x^{\frac{n}{2}} g_1) \\ &= f_0 \cdot g_0 - f_1 \cdot g_1 + x^{\frac{n}{2}}((f_0 + f_1) \cdot (g_0 + g_1) - f_0 \cdot g_0 - f_1 \cdot g_1) \\ &= NTT^{-1}(\hat{f}_0 \circ \hat{g}_0 - \hat{f}_1 \circ \hat{g}_1 + \widehat{x^{\frac{n}{2}}} \circ ((\hat{f}_0 + \hat{f}_1) \circ (\hat{g}_0 + \hat{g}_1) - \hat{f}_0 \circ \hat{g}_0 - \hat{f}_1 \circ \hat{g}_1)) \end{aligned}$$

Note that we do not need to compute $\widehat{x^{\frac{n}{2}}} = NTT(x^{\frac{n}{2}})$ by the NTT algorithm, but just compute it as $(\gamma^{\frac{n}{2}}, \gamma^{\frac{3n}{2}}, \cdots, \gamma^{\frac{(2n-1)n}{2}})$ directly. The most time-consuming operations of the algorithm are NTT transformations and point-wise multiplications. Actually, it consists of 4 forward NTT transformations, 1 inverse NTT transformation and 4 point-wise multiplications. Then the computational cost is given by

$$T(n) = 5n \log n + O(n).$$

### 3.2 $2^\alpha$-Part-Separation when $\frac{n}{2^{\alpha-1}}|(q-1)$

In this section, we are going to generalize the algorithm above. Similarly, we first divide the original polynomial into $2^\alpha$ parts as follows:

$$f = \textstyle\sum_{i=0}^{2^\alpha - 1}(x^{\frac{in}{2^\alpha}} \cdot f_i).$$

Notice that modulus $q$ is only required to satisfy that $\frac{n}{2^{\alpha-1}}|(q-1)$ when we divide the original polynomial into $2^\alpha$ parts. At last, we can compute the product $h$ as

follows:

$$\begin{aligned}
h &= f \cdot g \\
&= \sum_{i=0}^{2^\alpha-1} (x^{\frac{in}{2^\alpha}} f_i) \cdot \sum_{j=0}^{2^\alpha-1} (x^{\frac{jn}{2^\alpha}} g_j) \\
&= \sum_{i=0}^{2^\alpha-1} \sum_{j=0}^{2^\alpha-1} (x^{\frac{(i+j)n}{2^\alpha}} f_i \cdot g_j) \\
&= NTT^{-1}(\sum_{i=0}^{2^\alpha-1} \sum_{j=0}^{2^\alpha-1} (NTT(x^{\frac{(i+j)n}{2^\alpha}}) \circ \hat{f}_i \circ \hat{g}_j)) \\
&= NTT^{-1}(\sum_{i=0}^{2^\alpha-1} (NTT(x^{\frac{in}{2^{\alpha-1}}}) \circ \hat{f}_i \circ \hat{g}_i) \\
&+ \sum_{0 \le i < j \le 2^\alpha-1} (NTT(x^{\frac{(i+j)n}{2^\alpha}}) \circ ((\hat{f}_i + \hat{f}_j) \circ (\hat{g}_j + \hat{g}_i) - \hat{f}_i \circ \hat{g}_i - \hat{f}_j \circ \hat{g}_j)))
\end{aligned}$$

The most costly operations of the algorithm are NTT transformations and point-wise multiplications. Actually, it consists of $2^{\alpha+1}$ forward NTT transformations, 1 inverse NTT transformation and $(2^{2\alpha} + 2^{\alpha+1} - 4)$ point-wise multiplications. Then the computational cost is given by

$$T(n) = 2^{\alpha+1} \cdot \frac{n}{2^{\alpha-1}} \log \frac{n}{2^{\alpha-1}} + n \log n + O(n) = 5n \log n + O(n).$$

## 4 Revisit the Preprocess-then-NTT technique

Note that a more efficient way to compute the multiplication $f \cdot g$ in $R_q$ is described in Section 2.3, by computing $(1, \omega, \cdots, \omega^{n-1}) \circ NTT^{-1}(NTT(\tilde{f} \circ \tilde{g}))$ where $\tilde{f} = (f_0, \omega f_1, \cdots, \omega^{n-1} f_{n-1})$ and $\tilde{g} = (g_0, \omega g_1, \cdots, \omega^{n-1} g_{n-1})$. For simplicity, for any polynomial $f \in R_q$, we denote by $\widehat{NTT}(f) = NTT(\tilde{f})$ and $\widehat{NTT}^{-1}(\cdot) = (1, \omega, \cdots, \omega^{n-1}) \circ NTT^{-1}(\cdot)$

### 4.1 Zhou et al.'s Algorithms

First, we racall the algorithms proposed in [11]. Denote $x^2$ by $y$. Their 1-round preprocess-then-NTT algorithm used the even-indexed and odd-indexed coefficients of $f(x) \in R_q$ separately to define two new polynomials $f_{even}(y)$ and $f_{odd}(y)$ whose degrees are bounded by $\frac{n}{2}$:

$$\begin{aligned}
f_{even}(y) &= f_0 + f_2 \cdot y + f_4 \cdot y^2 + \cdots + f_{n-2} \cdot y^{n/2-1} \in \mathbb{Z}_q[y]/(y^{n/2}+1), \\
f_{odd}(y) &= f_1 + f_3 \cdot y + f_5 \cdot y^2 + \cdots + f_{n-1} \cdot y^{n/2-1} \in \mathbb{Z}_q[y]/(y^{n/2}+1).
\end{aligned}$$

What's more, denote $\boldsymbol{f}_{odd}(y) = y \cdot f_{odd}(y) \in \mathbb{Z}_q[y]/(y^{n/2}+1)$. Their 1PtNTT algorithm is presented as follows:

Step 1: (Sepration) $f(x) = f_{even}(x^2) + x \cdot f_{odd}(x^2)$ , $g(x) = g_{even}(x^2) + x \cdot g_{odd}(x^2)$, where the degrees of $f_{even}$,$f_{odd}$,$g_{even}$,$g_{odd}$ are bounded by $\frac{n}{2}$.

Step 2: (Multiplication) Compute $h_{even}(y)$, $h_{odd}(y) \in \mathbb{Z}_q[y]/(y^{n/2}+1)$ as in the following:

$$\begin{aligned} h_{even}(y) &= f_{even}(y) \cdot g_{even}(y) + f_{odd}(y) \cdot \boldsymbol{g}_{odd}(y) \\ &= \widehat{NTT}^{-1}(\widehat{NTT}(f_{even}(y)) \circ \widehat{NTT}(g_{even}(y)) \\ &\quad + \widehat{NTT}(f_{odd}(y)) \circ \widehat{NTT}(\boldsymbol{g}_{odd}(y))) \end{aligned}$$

$$\begin{aligned} h_{odd}(y) &= f_{even}(y) \cdot g_{odd}(y) + f_{odd}(y) \cdot g_{even}(y) \\ &= \widehat{NTT}^{-1}(\widehat{NTT}(f_{odd}(y)) \circ \widehat{NTT}(g_{even}(y)) \\ &\quad + \widehat{NTT}(f_{even}(y)) \circ \widehat{NTT}(g_{odd}(y))). \end{aligned}$$

Step 3: (Gatheration) Compute $h$ in $R_q$ as following:

$$h(x) = h_{even}(x^2) + x \cdot h_{odd}(x^2).$$

By their analysis, 1PtNTT algorithm consists of 5 forward NTT transformations, 2 inverse NTT transformations and 4 point-wise multiplications. As a result, we have the following property.

*Property 1.* The computational cost of computing product of two polynomials whose degrees are bounded by $n$ by using 1PtNTT algorithm is

$$T(n) = \tfrac{7n}{2} \log n - \tfrac{3}{2} n.$$

Similarly, their 2-round preprocess-then-NTT(2PtNTT) algorithm divides each polynomial into 4 parts, and then transform the original multiplication into several multiplications of polynomials of lower degrees. They have the following result of complexity of 2PtNTT.

*Property 2.* The computational cost of computing product of two polynomials whose degrees are bounded by $n$ by using 2PtNTT algorithm is

$$T(n) = \tfrac{15n}{4} \log n - \tfrac{7}{2} n.$$

Notice that the complexity of the algorithm grows while the number of round increases. It seems meaningless to consider the general case(i.e. $\alpha$-round) for the PtNTT algorithm, since the efficiency loss can not be tolerated when the number of round is somehow large. However, we present an improved algorithm which makes it diffrent.

### 4.2 1-Round Improved-Preprocess-then-NTT(1IPtNTT)

The improved algorithm of 1PtNTT we present here is different in the step 2 of 1PtNTT algorithm. Actually, we find that there is no need for us to compute $\widehat{NTT}(\boldsymbol{g}_{odd}(y))$. However, we only need to compute $\widehat{NTT}(y) = (\gamma, \gamma^3, \cdots, \gamma^{n-1})$ instead. Then there is one more point-wise multiplication at a cost but one less forward NTT transformation. What's more, in analogy with karatsuba's idea we can reduce the number of point-wise multiplications to improve the efficiency further. We present the difference from 1PtNTT algorithm here:

$$\begin{aligned} h_{even}(y) &= f_{even}(y) \cdot g_{even}(y) + f_{odd}(y) \cdot (y \cdot g_{odd}(y)) \\ &= \widehat{NTT}^{-1}(\widehat{NTT}(f_{even}(y)) \circ \widehat{NTT}(g_{even}(y)) \\ &\quad + \widehat{NTT}(y) \circ \widehat{NTT}(f_{odd}(y)) \circ \widehat{NTT}(g_{odd}(y))) \end{aligned}$$

$$\begin{aligned} h_{odd}(y) &= f_{even}(y) \cdot g_{odd}(y) + f_{odd}(y) \cdot g_{even}(y) \\ &= \widehat{NTT}^{-1}(\widehat{NTT}(f_{odd}(y)) \circ \widehat{NTT}(g_{even}(y)) \\ &\quad + \widehat{NTT}(f_{even}(y)) \circ \widehat{NTT}(g_{odd}(y))) \\ &= \widehat{NTT}^{-1}((\widehat{NTT}(f_{even}(y)) + \widehat{NTT}(f_{odd}(y))) \circ \\ &\quad (\widehat{NTT}(g_{even}(y)) + \widehat{NTT}(g_{odd}(y))) \\ &\quad - \widehat{NTT}(f_{even}(y)) \circ \widehat{NTT}(g_{even}(y)) - \widehat{NTT}(f_{odd}(y)) \circ \widehat{NTT}(g_{odd}(y))) \end{aligned}$$

Note that $\widehat{NTT}(y) = (\gamma, \gamma^3, \cdots, \gamma^{n-1})$. Thus, 1IPtNTT algorithm consists of 4 forward NTT transformations, 2 inverse NTT transformations and 4 point-wise multiplications. Then the computational cost is given by

$$T(n) = 4 \cdot \tfrac{n}{2} \log \tfrac{n}{2} + 2 \cdot \tfrac{n}{2} \log \tfrac{n}{2} + 4 \cdot \tfrac{n}{2} = 3n \log n - n.$$

As a result, we have the following property.

*Property 3.* The computational cost of computing product of two polynomials whose degrees are bounded by $n$ by using 1PtNTT algorithm is

$$T(n) = 3n \log n - n.$$

### 4.3 $\alpha$-Round Improved-Preprocess-then-NTT($\alpha$IPtNTT)

In this section, we focus on the general case of the Preprocess-then-NTT algorithm and denote $x^{2^\alpha}$ by $z$. We can similarly present the improved algorithm for the 2-round case, and the complexity of it is also given by $T(n) = \mathcal{O}(3n \log n)$. Actually, we will present the improved algorithm for the general case as follows:

Step 1: (Sepration) $f(x) = \sum_{i=0}^{2^\alpha - 1} x^i \cdot \overline{f}_i(x^{2^\alpha})$ , $g(x) = \sum_{j=0}^{2^\alpha - 1} x^j \cdot \overline{g}_j(x^{2^\alpha})$, where the degrees of $\overline{f}_i(z)$,$\overline{g}_j(z)$ are bounded by $\frac{n}{2^\alpha}$.

Step 2: (Multiplication) Compute $\overline{h}_i(y) \in \mathbb{Z}_q[y]/(y^{n/2^\alpha}+1)$ as following:

$$\begin{aligned}\overline{h}_i(z) &= \sum_{l=0}^{i} \overline{f}_l(z)\cdot\overline{g}_{i-l}(z) + \sum_{l=i+1}^{2^\alpha-1} z\cdot\overline{f}_l(z)\cdot\overline{g}_{2^\alpha+i-l}(z)\\ &= \widehat{NTT}^{-1}\Big(\sum_{l=0}^{i}\widehat{NTT}(\overline{f}_l(z))\circ\widehat{NTT}(\overline{g}_{i-l}(z))\\ &\quad + \sum_{l=i+1}^{2^\alpha-1}\widehat{NTT}(z)\circ\widehat{NTT}(\overline{f}_l(z))\circ\widehat{NTT}(\overline{g}_{2^\alpha+i-l}(z))\Big)\end{aligned}$$

where $0 \le i < 2^\alpha$.

Step 3: (Gatheration) Compute $h$ in $R_q$ as following:

$$h(x) = \textstyle\sum_{i=0}^{2^\alpha-1} x^i\cdot\overline{h}_i(x^{2^\alpha}).$$

Similarly, $\widehat{NTT}(z)$ can be computed directly instead of by the NTT algorithm. And In the step 2, when we compute the each part $\widehat{NTT}(\overline{f}_i)\circ\widehat{NTT}(\overline{g}_j)$, for $i=j$ we do it as usual while for $i\neq j$ we will compute two corresponding parts together as follows in practice:

$$\begin{aligned}&\widehat{NTT}(\overline{f}_i)\circ\widehat{NTT}(\overline{g}_j) + \widehat{NTT}(\overline{f}_j)\circ\widehat{NTT}(\overline{g}_i) = (\widehat{NTT}(\overline{f}_i)+\widehat{NTT}(\overline{f}_j))\circ\\ &(\widehat{NTT}(\overline{f}_i)+\widehat{NTT}(\overline{f}_j)) - \widehat{NTT}(\overline{f}_i)\circ\widehat{NTT}(\overline{g}_i) - \widehat{NTT}(\overline{f}_j)\circ\widehat{NTT}(\overline{g}_j)\end{aligned}$$

Through this technique we can reduce the number of point-wise multiplications to 1 from 2 when compute the each part for $i\neq j$.

Thus by simple analysis, $\alpha$IPtNTT algorithm consists of $2^{\alpha+1}$ forward NTT transformations, $2^\alpha$ inverse NTT transformations and $(3\cdot 2^{2\alpha-2}+2^{\alpha-1})$ point-wise multiplications. Then the computational cost is given by

$$\begin{aligned}T(n) &= 2^{\alpha+1}\cdot\tfrac{n}{2^\alpha}\log\tfrac{n}{2^\alpha} + 2^\alpha\cdot\tfrac{n}{2^\alpha}\log\tfrac{n}{2^\alpha} + (3\cdot 2^{2\alpha-2}+2^{\alpha-1})\cdot\tfrac{n}{2^\alpha}\\ &= 3n\log n + (3\cdot 2^{\alpha-2} - 3\alpha + \tfrac{1}{2})n.\end{aligned}$$

Thus, we have the following property.

*Property 4.* The computational cost of computing product of two polynomials whose degrees are bounded by $n$ by using $\alpha$IPtNTT algorithm is

$$T(n) = 3n\log n + (3\cdot 2^{\alpha-2} - 3\alpha + \tfrac{1}{2})n.$$

From the above property, we find that the complexity of 2-Round Improved-Preprocess-then-NTT(2IPtNTT) algorithm is given by $T(n) = 3n\log n - \frac{5}{2}n$, same for the 3-Round case. Actually, it is the most efficient one among all cases while the complexity of 1-Round case is given by $T(n) = 3n\log n - n$. What's more, in the next section we will give some experimental results to further confirm this.

*Remark 1.* Actually, we can also apply the Karatsuba's idea to the polynomial multiplication in $\mathbb{Z}[x]$ using FFT. As a consequence, we will improve the efficiency of the algorithm too.

## 5 Experiment Results

We have the C implemrntation for our improved NTT algorithm, and we present the exprimental results in this section. It is worth mentioning that our algorithm just simply call the original NTT algorithm rather than change it essentially, so if there is any optimization of the original NTT algorithm, it can also be applied to our algorithm.

Although 1PtNTT and $\alpha$IPtNTT($\alpha = 1, 2, 3$) can use some parameters that are not suitable for NTT, we analyze and compare the computational cost of NTT, 1PtNTT and $\alpha$IPtNTT for the same parameters by runing a C implementation compiled with gcc-8.2.0 on a 3.70GHZ Inter(R) Core(TM) i3-4170 processor, so that we can make it easy to demonstrate the improvement of $\alpha$IPtNTT on the efficiency. The results are reported in Table 1 as follows:

**Table 1.** Results of our C implementation(average of 10000 runs)

| Time(ms) Parameters / Algorithm | n=256,q=7681 | n=512,q=12289 | n=1024,q=12289 |
|---|---|---|---|
| NTT | 0.045 | 0.101 | 0.2283 |
| 1PtNTT | 0.04725 | 0.1138 | 0.2582 |
| 1IPtNTT | 0.039 | 0.0972 | 0.2261 |
| 2IPtNTT | 0.03525 | 0.0879 | 0.2087 |
| 3IPtNTT | 0.035 | 0.0836 | 0.1996 |

To make it more intuitive, we present the ratio of each algorithm to NTT algorithm in Table 2 as follows:

**Table 2.** the ratio of each algorithm to NTT algorithm

| Parameters | n=256,q=7681 | n=512,q=12289 | n=1024,q=12289 |
|---|---|---|---|
| 1PtNTT-theoretical-ratio | 1.0600 | 1.0714 | 1.0806 |
| 1PtNTT-experimental-ratio | 1.0500 | 1.1267 | 1.1310 |
| 1IPtNTT-theoretical-ratio | 0.9200 | 0.9286 | 0.9355 |
| 1IPtNTT-experimental-ratio | 0.8667 | 0.9624 | 0.9904 |
| 2IPtNTT-theoretical-ratio | 0.8600 | 0.8750 | 0.8871 |
| 2IPtNTT-experimental-ratio | 0.7833 | 0.8703 | 0.9141 |
| 3IPtNTT-theoretical-ratio | 0.8600 | 0.8750 | 0.8871 |
| 3IPtNTT-experimental-ratio | 0.7778 | 0.8277 | 0.8743 |

## 6 Conclusion

We have presented a new technique and an improved version of previous work to decrease the value of modulus $q$. The former one is simpler and easier to understand, while the latter one is more efficient even than the original NTT algorithm.

## References


1. Joppe W Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE, 2015.
2. Eleanor Chu and Alan George. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC Press, 1999.
3. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
4. Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.
5. Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.
6. Vadim Lyubashevsky. Lattice signatures without trapdoors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–755. Springer, 2012.
7. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *International Workshop on Fast Software Encryption*, pages 54–72. Springer, 2008.
8. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.
9. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.
10. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science - FOCS*, pages 124–134, 1994.
11. Shuai Zhou, Haiyang Xue, Daode Zhang, Kunpeng Wang, Xianhui Lu, Bao Li, and Jingnan He. Preprocess-then-ntt technique and its applications to k yber and n ew h ope. In *International Conference on Information Security and Cryptology*, pages 117–137. Springer, 2018.