Optimizing polynomial convolution for NTRUEncrypt

Wei Dai[1], William Whyte[2], and Zhenfei Zhang[2]

[1] Worcester Polytechnic Institute, Worcester, Massachusetts, USA
wdai@wpi.edu
[2] Onboard Security, Wilmington, Massachusetts, USA
{wwhyte,zzhang}@onboardsecurity.com

Abstract. NTRUEncrypt is one of the most promising candidates for quantum-safe cryptography. In this paper, we focus on the NTRU743 paramter set. We give a report on all known attacks against this parameter set and show that it delivers 256 bits of security against classical attackers and 128 bits of security against quantum attackers. We then present a parameter-dependent optimization using a tailored hierarchy of multiplication algorithms as well as the Intel AVX2 instructions, and show that this optimization is constant-time. Our implementation is two to three times faster than the reference implementation of NTRUEncrypt.
Keywords: Quantum-safe cryptography, NTRUEncrypt, security estimation, constant-time implementation, AVX2.

1 Introduction

Quantum computers pose a significant threat to modern cryptography. The two most widely adopted public key cryptosystems, namely, RSA [42] and Elliptic Curve Cryptography (ECC), will be trivially breakable by sufficiently large general purpose quantum computers using Shor's algorithm [44]. This is sometimes referred to as the "Quantum Apocalypse".

Standardization groups all over the world: NIST [16], ETSI [37], etc. are preparing us against this quantum threat by migrating towards quantum-safe cryptosystems (also known as post-quantum cryptograhy), i.e., cryptography built upon problems that remain hard even with the presence of quantum computers.

Among all known candidates, the NTRUEncrypt public key encryption [33] algorithm is one of the most promising ones. It was firstly published in 1996. In 2008 and 2010, it was standardized by IEEE [6] and ASC X9 [47]. The most recent specification of how to implement NTRUEncrypt is available at [23]. To date, it is the only quantum-safe public key encryption algorithm that has ever been standardized. There are proposals to secure our day-to-day communication using NTRUEncrypt for handshake protocols (in a hybrid mode) within TLS [5], IKEv2 [3] and Tor network [43].

NTRUEncrypt is a lattice based encryption scheme. It bases its security on finding unique shortest vectors for an ideal lattice with a special structure (see

section 3.2.1 for more details). This is known as the *NTRU assumption*. The idea of NTRUEncrypt has inspired many important notions/constructions, such as ideal lattices [41], a vital improvement that makes lattice-based cryptography practical; and fully homomorphic encryption [26], a "holy grail" for cryptographers for 30 years. In particular, there are many cryptosystems built directly upon the NTRU assumption other than NTRUEncrypt, such as the BLISS signature scheme [21], the Falcon signature scheme [2] and the LTV fully homomorphic encryption scheme [40].

Although the NTRU assumption remains unbroken for over 20 years of time, the parameters derived for NTRUEncrypt in its early days are less conservative. During the last two decades, there has been several attacks against NTRUEncrypt (such as [20,34]). Among them, the best known attack is the hybrid attack due to Howgrave-Graham in 2007 [35] (although recent study suggests that we may have over-estimated the power of a hybrid attack [46]), which resulted in a parameter revision in 2008 [30]. For the last decade, those parameters from [30] are somewhat stable (albeit some minor changes, for example, switching from SHA-1 to SHA-256) and resist all cryptanalysis. The latest versions of the parameter sets, together with the most up-to-date security considerations, are provided in [32], and are going through NIST post-quantum standardization process [16]. Those parameter sets are available in a reference library `libntruencrypt` [4] developed by Security Innovation Inc., the owner of the patent to NTRUEncrypt. In March 2017, Security Innovation Inc. released its NTRUEncrypt patent to the public domain (see license statement of [4]).

In this paper, we focus on the parameter set of NTRU743 (also known as ntruees743-ep1 in [23]) with a minor difference. NTRU743 is reported to deliver over 256 bits of security against classical attackers and 128 bits of security against quantum attackers. Our parameter set is identical to NTRU743 except that we do not use product form polynomials (see section 3).

For this parameter set, we firstly give a thorough review of all known cryptanalysis, including hybrid attacks [36,46], lattice reduction attacks through BKZ 2.0 [17,9], subfield attacks [13,7,19,39] etc.; evaluate its security level against those attacks, and re-affirm the mentioned security level. We note that the analysis of hybrid attacks was reported in [32]; the analysis of lattice attacks was briefly mentioned in [31], an extend version of [32]. In this article, we re-apply the analysis from [31], taking into account that the secret keys are in flat form vs. product form in [32,23]; we also include an analysis of subfield attacks mentioned above.

Our major contribution is to provide a tailored optimization for this parameter set. We use a combination of a vectorized index-based multiplication and a hierarchy of fast multiplication algorithms such as Karatsuba and Toom-Cook. Our implementation improves the polynomial multiplication by a factor of 2.23 on modern processors that support AVX2, and a factor of 1.6 on processors that do not support AVX2. Our analysis shows that the proposed algorithm is constant-time (whereas it is not the case for [4]). We conjecture that our implementation is also constant-time (see section 5.4).

In [28], the authors provided an AVX2/AVX-512 based optimization for NTRUEncrypt. It has two main contributions, both due to the usage of new instructions: firstly, it replaces an SSE based Karatsuba algorithm with an AVX2/AVX-512 based Karatsuba, and secondly it uses AES-NI instructions to accelerate AES operations, which is used as the random number generation function in NTRUEncrypt. It does not provide algorithmic optimizations as to be shown in this work.

We release our software to the public domain under GPL. It is available from [48], and will be merged into the reference implementation [4].

2 Background

NTRUEncrypt works over a polynomial ring $\mathcal{R} = \mathbb{Z}_q[x]/(x^N - 1)$ for N a prime and q a power of 2. Other rings have also been proposed, such as NTRU-prime [14] and NTRU over an R-LWE type ring [45]. In this paper we stick to the original design from [33].

Elements in this ring are truncated polynomials, i.e., polynomials whose degrees are less than N and all their coefficients are in the range between 0 and $q - 1$. Arithmetic operations between integers, and between an integer and a polynomial, are carried out modulo q. Multiplications (and divisions) between polynomials are carried out modulo $x^N - 1$ modulo q.

For a polynomial $a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}$, denote $\langle a_0, a_1, a_2, \ldots, a_{N-1} \rangle$ its vector form. The i-th cyclic rotation of this vector is $\langle a_{N-i}, a_{N-i+1}, \ldots, a_N, a_0, a_1, \ldots, a_{N-i-1} \rangle$. The matrix associated with $a(x)$ over the $\mathcal{R}$ is an $N \times N$ matrix whose row vectors are a and its rotations.

For two polynomials $a(x)$ and $b(x)$, their product over $\mathcal{R}$ is $c(x) = a(x)b(x) \bmod (x^N - 1) \bmod q$. This is also known as polynomial convolution. It can be computed either via polynomial multiplication, or by vector-matrix multiplication as a times the cyclic matrix associated with b.

2.1 NTRUEncrypt

For completeness, we recall the following three algorithms (Algorithms 1-3) in NTRUEncrypt. Here, a $\cdot$ operation is an integer-polynomial multiplication modulo q. A $\times$ operation is a polynomial convolution over $\mathcal{R}$, i.e., polynomial multiplication modulo $x^N - 1$ modulo q.

2.2 Operations in NTRUEncrypt

As one can see from the above 3 algorithms, the most costly parts in an NTRUEncrypt cryptosystem are

- Polynomial arithmetics
- Hash functions

Table 1: A Cost Breakdown of Reference Code of NTRUEncrypt [4]

Function	Total	Polynomial arithmetics		Hash functions		Rest	
		k cycles	Percentage	k cycles	Percentage	k cycles	Percentage
Key generation	5424	5082	93.6%	247	4.5%	95	1.7%
Encryption	1008	780	77.4%	160	15.9%	68	6.7%
Decryption	1757	1560	88.8%	104	5.9%	93	5.3%

Algorithm 1 Key generation

Require: a parameter set, i.e., NTRU743
1: $F :=$ a random trinary polynomial with df number of 1s and df number of -1s.
2: $f = p \cdot F + 1$.
3: **if** f is not invertible **then**
4: goto 1
5: **end if**
6: $g :=$ a random trinary polynomial with dg number of 1s and df number of -1s.
7: $h = f^{-1} \times g$.
8: **return** public key h, secret key f, g

Polynomial operations are the core operation for NTRUEncrypt. Hash functions are used to expand random seeds (during key generation and encryption), as well as the binding method between public key/message and the random polynomial (during encryption and decryption). This binding process provides unmalleability of the ciphertext which is crucial to provide CCA-2 security.

Table 1 shows a breakdown of how much those two operations eat up the whole computation cost for three algorithms: key generation, encryption and decryption. It is easy to see that polynomial arithmetics take up almost 90% of the computing resources. To this end, in order to boost the efficiency, we aim to improve polynomial multiplications for NTRUEncrypt.

Algorithm 2 Encryption

Require: a parameter set, i.e., NTRU743
Require: message m, public key h
1: $M = (m|\text{salt})$
2: $seed = \text{Hash}_1(M|h)$
3: $r :=$ a random trinary polynomial with dr number of 1s and dr number of -1s generated using $seed$
4: $\text{mask} = \text{Hash}_2(p \cdot r \times h)$
5: **return** $e = p \cdot r \times h + m \oplus \text{mask}$

Algorithm 3 Decryption

Require: a parameter set, i.e., NTRU743
Require: ciphertext e, secret key f
1: $M_1 = e \times f \bmod p$
2: $\text{mask}_1 = \text{Hash}_2(e - M_1)$
3: $m_1 = M_1 \oplus \text{mask}$
4: $seed_1 = \text{Hash}_1(m_1|h)$
5: $r_1 :=$ a random trinary polynomial with dr number of 1s and dr number of -1s generated using $seed_1$
6: **if** $e == p \cdot r_1 \times h + M_1$ **then**
7: **return** m_1
8: **else**
9: **return** error
10: **end if**

2.3 AVX2

Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture for microprocessors which enable *single instruction, multiple data*. AVX2 and AVX-512 expand most integer instructions to support 256-bit and 512-bit registers, respectively.

To date, AVX2 is available to almost all mainstream processors. AVX-512, although provides more parallelism, is not as widely available. For this reason, we use AVX2 for this implementation.

We remark that our scheme can be improved further using AVX-512 when it becomes more available. As one shall see later in section 4.2, a smallest data unit in our setting is a degree 31 polynomial (32 coefficients) where each coefficient is a `uint16_t` unit. We currently use 2 `__mm256i` units from AVX2 to store this data. However, it is easy to see that this polynomial fits in a single 512-bit data unit. Therefore, the adaptation to AVX-512 is straightforward.

Arithmetic operations on `__mm256i` are carried out on each slot without any interactions among the slots. We make use of the following instructions:

- `_mm256_add_epi16`: this instruction adds the packed `uint16_t` integers using saturation;
- `_mm256_mullo_epi16`: this instruction multiplies the packed `uint16_t` integers, produces intermediate 32-bit integers, stores the 16 least significant bits and discards the other 16 bits.

as well as a few instructions that convert between a pack of `uint16_t` data and a `__mm256i` data.

2.4 Constant time implementation and side channel attacks

Aside from being efficient, a major concern when designing implementations for a cryptosystem is whether this program is constant time, i.e., the code runs

in time independent of any secret input. This is particularly important if the algorithm will potentially be exposed to side-channel attacks. Having a constant time implementation means that an attacker is not likely to gain any information on the secret input by simply observing the running time of the algorithm.

There are a few lattice-based cryptosystems that claim to have constant time implementations, such as [10] and [14]. In the meantime, attacks have also been proposed on some other cryptosystems (see, for example, [15] for an attack against [21] exploiting the Gaussian look-up table) indicating that those attacks can be practical, under some realistic assumptions.

3 Parameter set

3.1 Parameter overview

We consider a variant of the NTRU743 parameter set.

N	q	p	df, dg, dr
743	2048	3	247

The only difference we made is switching to flat form polynomials. A flat form trinary polynomial is a trinary polynomial f that has a fixed number of $+1$ and -1 coefficients; while a product form polynomial is computed from a set of three sparse trinary polynomials as $f_1 \times f_2 + f_3$ where f_1, f_2 and f_3 are flat form trinary polynomials.

The use of product form parameter sets was originally intended to provide improved performance by allowing a specialized multiplication algorithm that used knowledge of the indices of the non-zero coefficients of f; product-form polynomials allow the same level of security with about a third as many non-zero indices and so promised to reduce polynomial multiplication times significantly. However, this index-based multiplication proves to be very hard to implement in a constant-time fashion without losing the speed benefits, so in this paper we concentrate on other approaches of multiplication.

We work on a polynomial ring of $\mathbb{Z}_q[x]/(x^N - 1)$ for $q = 2048$ and $N = 743$. This choice of q is motivated by fast arithmetic operations (see section 4.2.1), as well as a low decryption failure probability.

We consider two types of polynomials over this ring, namely, flat form trinary polynomials and random "mod q" polynomials. Multiplications between those polynomials take up the majority of the cost of NTRUEncrypt. The goal of this paper is to improve those multiplications.

3.2 Security estimation for NTRU743

This parameter set delivers 256 bits of security against classical computers and 128 bits of security against quantum computers. In particular we consider the following attacks. We summarize its security estimates in Table 2.

Table 2: A summary of security estimates

Classical security estimation	256
Quantum security estimation	128
Lattice reduction attack	N/A
Search attack cost	$> 2^{289}$
Hybrid attack cost	2^{272}
Decryption failure probability	$< 2^{-112}$

3.2.1 Lattice attacks For an NTRUEncrypt public key polynomial h, let H be the matrix whose row vectors are the cyclic rotation of h. Then the NTRU lattice associated with h uses a basis

$$\begin{bmatrix} qI_N & 0 \\ H & I_N \end{bmatrix}$$

where I_N is an N-dimensional identity matrix. With in this NTRU lattice, there exist unique shortest vectors, namely, the vector form of $\langle f, g \rangle$ and its cyclic rotations.

This attack was firstly presented in the original NTRUEncrypt paper [33] circulated during the rump session of Crypto'96. It was later observed in [20] that one does not necessarily need to find the exact secret key to be able to decrypt. An attack is successful if the attacker can locate any vectors in this lattice that are sufficiently small (such as a cyclic rotation of the secret key).

It has been shown in [24] that the ability to locate a unique shortest vector in a lattice depends on the root Hermite factor of the lattice, which is the n-th root of

$$\frac{\text{Gaussian expected length}}{l_2 \text{ norm of the target vector}}$$

where n is the dimension of the lattice.

Here, we give an estimation of the root Hermite factor for the proposed parameter set. This lattice has a dimension of $2N$. The Gaussian expected length of this lattice is $\sqrt{\frac{Nq}{\pi e}}$, while the l_2 norm of the target vectors are $\|f, g\|_2 \approx \sqrt{4N/3}$. This gives a root Hermite factor of the lattice as

$$\left(\frac{\sqrt{Nq/\pi e}}{\|f, g\|_2} \right)^{\frac{1}{2N}} \approx 1.0025$$

It is believed that current technique of BKZ 2.0 [18] is only able to find a short vector with a root Hermite factor of 1.005 (except for the analysis from [10], see section 3.2.4). It is safe to conclude that BKZ 2.0 is not applicable against the proposed parameter set.

3.2.2 Search attack In the meantime, since the secret keys are trinary polynomials with *df* number of 1s and -1s, the search space for the secret key is

$\binom{N}{df,df})/N > 2^{1158}$. (The factor $1/N$ comes from the fact that an attacker can guess any of N cyclic rotations of the secret key, rather than just the secret key itself.) This key space for our parameter set is considerably larger than that in [32] due to the switch from product form polynomials to flat form polynomials. This is sufficient even with the presence of meet-in-the-middle attacks [34] and quantum attacks using Grover's algorithm [27].

3.2.3 Hybrid attack The best known attack against NTRUEncrypt is the aforementioned hybrid attack [35] which is a hybrid of a lattice attack and a meet-in-the-middle search attack.

According to a recent evaluation [32], this attack requires 2^{272} operations against NTRU743 parameter set.

3.2.4 Lattice strength analysis from "NewHope" In [10], the authors give a conservative analysis on the cost of BKZ 2.0 reduction. As pointed out by the authors themselves, those estimations are very optimistic about the abilities of an attacker. In particular, unlike the analysis of BKZ 2.0 [18], where the cost of shortest vector subroutines is estimated via the cost of enumeration with extremely pruning [25], this analysis assumes that for large dimensional lattices shortest vector problems can be solved very efficiently using heuristic sieving algorithms, ignoring the sub-exponential to exponential requirement of space.

In a bit details, the best known classical and quantum sieving algorithms have time costs of $2^{0.292n}$ and $2^{0.265n}$, respectively [11]. The best plausible quantum short vector problem solver costs more than $2^{0.2075n}$. In practice, sieving tends to process much slower than enumeration techniques. Moreover, sieving algorithms require a similar level of space complexity (exponential in n), while the space requirement of enumeration techniques is polynomial. For this reason, we stick to the original BKZ 2.0 analysis [18].

For the sake of completeness, in Table 3 we provide the cost of the attack against proposed parameter sets following their analysis. It is also worth pointing out that even with their extremely conservative analysis, the parameter set NTRU743 still provides roughly 128 bits of quantum security as intended.

3.2.5 Subfield attack Subfield attacks against NTRUEncrypt have been considered in [13]. It was reported in [8] that for certain "over-stretched" NTRU parameters, one can exploit the subfield. This attack was only applicable to the NTRU lattices that are used to instantiate a (fully) homomorphic encryption scheme. The author also showed that for our parameters the subfield attack will not be successful.

3.3 Other parameter sets

It was reported in [32] that the parameter set NTRU443 also provides 128 bits of security against quantum computers. This parameter set provides best performance among all lattice-based public key cryptography (such as NewHope [10])

Table 3: Lattice strength following analysis in [10]

Attack	m[c]	b[d]	Known Classical[e]	Known Quantum[f]	Best Plausible[g]
Primal[a]	613	603	176	159	> 125
Dual[b]	635	600	175	159	> 124

[a] find the unique shortest vector in NTRU lattice.
[b] find the unique shortest vector in the dual lattice.
[c] the number of used samples.
[d] block size for BKZ 2.0.
[e] using the best known classical SVP solver.
[f] using the best known quantum SVP solver.
[g] using the best plausible quantum SVP solver.

in terms of the package size. This is crucial to modern public key infrastructures as we tend to have plenty computation power but rather limited communication bandwidth.

Nonetheless, we pick NTRU743 over NTRU443 to build more security margin against quantum computers. Quantum attacks, unlike classical attacks, are much more mysterious. Although a lot of effort has been put into quantum cryptanalysis, such as [22], much less is known about them compared to classical attacks. It is possible that there exist quantum attacks that substantially reduce the bit complexity, as predicted in NewHope [10]. To this end, we pick NTRU743 in this work, although we must note that NTRU443 is sufficient for the target security level, and our optimization can be adapted to NTRU443 with minor modifications.

4 Polynomial Multiplication for NTRU743

4.1 Algorithms under consideration

There exists several algorithms that handle polynomial multiplications.

- Schoolbook multiplication.
- Karatsuba/Toom-Cook multiplications.
- Index based multiplication.
- Fast Fourier Transform (FFT)/ Number Theoretic Transform (NTT).

In general, schoolbook multiplication is asymptotically worse than Karatsuba or Toom-Cook multiplications. However, Karatsuba or Toom-Cook multiplications use additional additions/subtractions to combine the results of multiplications. Those operations cost more than the multiplications that are saved, when the

degree is small. So for small polynomials, schoolbook multiplication usually outperforms Karatsuba/Toom-Cook. The exact threshold varies for different processors, but in general, for polynomial with degree less than around 32 it is okay to use schoolbook multiplication.

The index-based multiplication is slower than Karatsuba. It is ideal for sparse polynomials (polynomials with a lot of 0 coefficients). It also performs well when SIMD is available.

The last option is FFT/NTT transform [29]. It is widely used for R-LWE based schemes such as BLISS [21] and NewHope [10]. This technique requires a special polynomial ring and is not applicable to our parameter set.

To enable NTT, one needs to work on a ring that "splits" completely - i.e., the modulus polynomial $x^N - 1$ (or $x^N + 1$ as in [10,**?**], for instance) has N distinct roots modulo q. Typically, for a prime N in our parameter set, q has to be 1487, 19319 or larger. In general, having q being a power of 2 with less than 14 bits means the modulo operation comes free due to the modern computer structure.

To be more detailed, the choice of $q = 1487$ is too small which results in a higher decryption error; while the choice of $q = 19319$ will reduce the overall performance compared to our current choice $q = 2048$: for one side it doesn't fit in a single `uint16_t` data type, taking into account the guarding bits; on the other side, a large q means a larger determinant of the lattice and overall a lower lattice security/hybrid security.

4.2 Our method

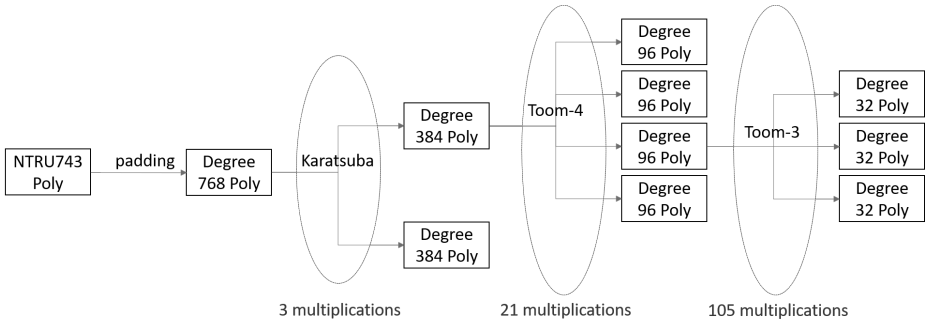


Fig. 1: The structure of our tailored multiplication.

The improvement (compared to the reference code [4]) comes from (1) a vectorized index-based multiplication for small polynomials (deg < 32) and (2) a tailored break down using Karatsuba $\rightarrow$ Toom-4 $\rightarrow$ Toom-3 (see Figure 1). For polynomials of degrees close to but less than 768, we reduce the degrees of polynomial multiplication operands using:

1. Karatsuba to break them into smaller polynomials (deg < 384),
2. Toom-4 to break polynomials (deg < 384) into polynomials (deg < 96),
3. Toom-3 to break polynomials (deg < 96) into polynomials (deg < 32),
4. vectorized index-based multiplication to perform multiplications of polynomials (deg < 32).

Overall, the choice for the above selections is obtained as follows. First, we have assumed that polynomial multiplications of degree less than 32 shares a similar cost using AVX2, regardless of the degree. In theory it is sound because of vectorized processing; in practice, this is also confirmed experimentally as shown in Figure 4. As a consequence we know that for polynomials of degrees close to but less than 768, each polynomial can be presented by 24 small polynomials of degree < 32.

Next we need to determine the hierarchy to process those 24 small polynomials. Karatsuba splits one polynomial into two polynomials using 3 multiplications; Toom-3 splits each polynomial into 3 polynomials using 5 multiplications; and Toom-4 splits each polynomial into 4 polynomials using 7 multiplications. Hence, it is natural that we use a combination of Karatsuba→Toom-4→Toom-3 to achieve a total of 24 polynomials with 105 multiplications.

We have also tested the following cases:

- Karatsuba only: this is basically the reference implementation in [4]. we provide more details on comparison in subsection 5.3.1.
- Toom-4→Toom-4→Karatsuba: in this option each polynomial is broken into 32 small polynomials of degree < 24, and result into 147 multiplications of small polynomials.
- Toom-3 only: in this option each polynomial is broken into 27 small polynomials of degree < 29, and result into 125 multiplications of small polynomials.

Our suggested hierarchy out-performs all three cases in terms of the number of multiplications.

Nevertheless, we note that we did not test the case of Karatsuba→Toom-3→Toom-4; in theory there shouldn't be any performance difference between this option and our choice as the underlying number of multiplications (105) remains the same.

In the following, we will give more details of our proposed method.

4.2.1 Data types and instructions We make use of two types of data, namely `uint16_t` and `__mm256i`. We use `__mm256i` to handle 16 slots of `uint16_t` data. A degree $N-1$ polynomial can be stored with N `uint16_t` units or $\lceil \frac{N}{16} \rceil$ `__mm256i` units. In the following we briefly describe arithmetic operations on `uint16_t` data. We omit the details for `__mm256i` correspondents, as they are merely a parallel version of those for `uint16_t` data.

Recall that all coefficients of our polynomials are integers modulo $q = 2^{11}$. That is, we can store each coefficient with a `uint16_t` type. For intermediate values during computations, we do not care for overflows during additions, subtractions and multiplications, as overflown bits will be "*mod out*" once we lift the

polynomial back to $\mathcal{R}_q$. In particular, when multiplying two `uint16_t` elements, we only need to compute the lower 16 bits of the product; the higher 16 bits will be mod out, thus, have no effect on the final result.

In a bit more details, for two 16-bit integers a and b, suppose their product c is expressed by $c_0 + 2^{16}c_1$, i.e., c_0 and c_1 are the lower/higher 16 bits of c, respectively, then, from $a \times b = c_0 + 2^{16}c_1$ we obtain $a \times b \equiv c_0 \bmod q$ for $q = 2^{11}$. This also holds for polynomial multiplications over the ring: $a(x) \times b(x) \equiv c_0(x) \bmod q$.

Notice that this may not be true when a division is performed. It may not be very straightforward to see why division is required in a polynomial multiplication. This is due to the Karatsuba/Toom algorithms (see section 4.2.3 for more details).

Considering three 16-bit integers a, b and d, suppose we need to compute $(a \times b)/d \bmod q$. For simplicity, assumes that d divides $ab = c_0 + 2^{16}c_1$. Then we only have $ab/d \equiv c_0/d \bmod q$ for d equal a small power of 2. For other choices of d this equation no longer holds.

Since our q has only one prime factor, 2, any odd integer is co-prime with q. Therefore, when divided by an odd integer d, one can always multiply its inverse modulo q (or mod 2^{16} as we are working with 16-bit integers). And from previous argument we know that this multiplication can be carried out error free.

For a division with a divisor in the form of $2^{d_1} * d_2$ where d_2 is an odd integer, we perform the above strategy for the division-by-d_2 operation. We only carry out a division when the divisor is a power of 2. Since in our algorithm divisions are always exact, i.e., they do not result into any reminders. We can perform those exact divisions by simply shifting each `uint16_t` to the right.

As such, we are only concerned with the lower 16 bits of the product, which implies that we can always store (intermediate) polynomials with `uint16_t` and `__mm256i` types.

4.2.2 Index based multiplication with AVX2 We use an index-based multiplication for polynomials of degrees less than 32. This algorithm inputs a base polynomial and an index polynomial, and outputs a result polynomial whose degree is less than 64 (see Algorithm 4).

That is, the base input polynomial a can be stored within 2 `__mm256i` units; during the execution, we need 3 `__mm256i` units to store $\mathbf{t}$; the result polynomial $\mathbf{r}$ can be stored within 4 `__mm256i` units. For each coefficient in the index polynomial, we shift the base polynomial; multiply it by this coefficient (and discard the higher 16 bits) and add the product to the result.

Note that we have used 3 `__mm256i` units for $\mathbf{t}$. An alternative solution is to use a buffer of $4N$ `uint16_t` units, and dynamically load/write the buffer from and to 2 `__mm256i` units that hold this intermediate results. This can reduce the number of `__mm256i` units from 3 to 2 (and consequently reduces the number of vectorized multiplications and additions required, see below). However, as we have observed in our implementation, reading and writing from memory for

Algorithm 4 Index based multiplication

Require: $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{31}x^{31}$
Require: $b(x) = b_0 + b_1x + b_2x + \cdots + b_{31}x^{31}$
1: $\mathbf{t} = \langle 0, \ldots, 0, a_0, a_1, a_2, \ldots, a_{31} \rangle$ with dimension 63;
2: $\mathbf{r} = b_0\mathbf{t}$;
3: **for** i in $(1, 31)$ **do**
4: $\quad \mathbf{t} =$ Left shift($\mathbf{t}$) ;
5: $\quad \mathbf{r} += b_i\mathbf{t}$
6: **end for**
7: **return r**

`__mm256i` units is in fact more costly than vectorized multiplication and additions. Therefore we use 3 `__mm256i` units for the intermediate polynomials.

Now we are ready to analyze the cost of this algorithm. When the index of coefficients for $b(x)$ is either 0 or 16, we require 2 `__mm256i` units to store (the shift of) $\mathbf{t}$. This results into 2 vectorized multiplications and additions for those two indexes. For the reminder of the indexes, the intermediate polynomial $\mathbf{t}$ spans over 3 `__mm256i` units, and therefore, we need 3 vectorized multiplications and additions for each index. In summary, if the degree of the index polynomial is 31, then our algorithm use 94 vectorized multiplications and additions. We also note that this algorithm is constant-time: i.e., it requires a constant number of operations regardless of the value of $a(x)$ and $b(x)$, so long as the degrees of those polynomials stay the same.

There are two alternative approaches to handle polynomial multiplications with small degrees, namely, using a pure schoolbook multiplication, or using Karatsuba to split into polynomials of degree less than 16, then use schoolbook multiplications. As will be shown in subsection 5.2.1, on the machines that we have tested, Karatsuba outperforms schoolbook for multiplications of degree greater than 16. However, due to the fact that schoolbook multiplication is sequential therefore does not benefit from vectorization, neither algorithm is as fast as the one that we adopt.

4.2.3 Toom-3 multiplication This algorithm handles polynomial multiplications of degree less than 96. It splits each input polynomial into 3 smaller polynomials of degree less than 32 and use index-based multiplication method to handle multiplications between those small polynomials.

In a bit more details, each input polynomial is split into 3 parts, i.e., $a(X) = a_0 + a_1X + a_2X^2$ and $b(X) = b_0 + b_1X + b_2X^2$ while the result polynomial is denoted by $c(X) = c_0 + c_1X + c_2X^2 + c_3X^3 + c_4X^4$ for $X = x^{32}$. Note that the degrees of a_i and b_i are less than 32 while the degrees of c_i are less than 64.

Then, we evaluate the polynomials at -1, 0, 1, 2 and ∞, and perform multiplications for each interpolation. That is

$$\begin{aligned}
c(-1) = \quad & a(-1)b(-1) = & (a_0 - a_1 + a_2)(b_0 - b_1 + b_2) \\
c(0) = \quad & a(0)b(0) = & a_0 b_0 \\
c(1) = \quad & a(1)b(1) = & (a_0 + a_1 + a_2)(b_0 + b_1 + b_2) \\
c(2) = \quad & a(2)b(2) = & (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2) \\
c(\infty) = \quad & a(\infty)b(\infty) = & a_2 b_2
\end{aligned}$$

Lastly we re-composite $c(X)$ from $c(-1), c(0), c(1), c(2)$ and $c(\infty)$ as

$$\begin{aligned}
c_0 = \quad & c(0) \\
c_4 = \quad & c(\infty) \\
c_2 = \quad & (c(1) + c(-1))/2 - c_0 - c_4 \\
c_3 = \quad & (c(2) - c_0 - 4c_2 - 16c_4 - (c(1) - c(-1))) \times (3^{-1})/2 \\
c_1 = \quad & c(1) - c_0 - c_2 - c_3 - c_4
\end{aligned}$$

and obtain $c(x) = c_0 + c_1 x^{32} + c_2 x^{64} + c_3 x^{96} + c_4 x^{128}$.

When implementing this algorithm, we use vectorized instructions to handle polynomial additions, subtractions, as well as multiplications and divisions by integers. These operations take up a small part of the whole computation. The most consuming part of this algorithm is those 5 multiplications between polynomials of degrees less than 32. We use the mentioned index-based multiplication to handle those multiplications.

4.2.4 Toom-4 multiplication This algorithm handles polynomial multiplications of degree less than 384. It splits each input polynomial into 4 small polynomials with degree less than 96. It then requires 7 multiplications between those small polynomials for which we apply Toom-3. Those consist the majority of computation of this algorithm.

This algorithm is very similar to Toom-3. We evaluate the polynomials at -2, -1, 0, 1, 2, 3 and ∞. For simplicity we omit the details of this multiplication.

4.2.5 Karatsuba multiplication This algorithm handles polynomial multiplications of degree less than 768. It provides the interface to the actual NTRUEncrypt code, where the inputs have fixed degrees of 743. The algorithm splits each input polynomial into 2 small polynomials with degree less than 384 and use Toom-4 to further multiply those small polynomials. It can be seen as a variant of Toom-2 with evaluations at 0, 1 and ∞.

In a bit more details, each input polynomial is split into 2 parts, i.e., $a(X) = a_0 + a_1 X$ and $b(X) = b_0 + b_1 X$ while the result polynomial is denoted by $c(X) = c_0 + c_1 X + c_2 X^2$ for $X = x^{384}$. Note that the degrees of a_i and b_i are less than 384 while degree of c_i is less than 768.

Then, we perform the interpolation as

$$\begin{aligned} c(0) &= a(0)b(0) &&= a_0b_0 \\ c(1) &= a(1)b(1) &&= (a_0+a_1)(b_0+b_1) \\ c(\infty) &= a(\infty)b(\infty) &&= a_1b_1 \end{aligned}$$

and obtain $c(x) = c_0 + c_1x^{384} + c_2x^{768}$.

5 Performance and Implementation

Table 4: Overall performance comparison

	Reference	Optimization 1+2		Optimization 1 w/ AVX2		Optimization 1 w/o AVX2	
	implementation[4]	cycles	improvement	cycles	improvement	cycles	improvement
With `-O3`	290,304	**130,031**	**2.23×**	166,014	1.74×	180,871	1.6×
Without `-O3`	1,350,080	335,223	4×	988,373	1.36×	1,158,010	1.16×

5.1 Test environment

We tested our implementation with a dual core Intel i7-6600U processor @ 2.60GHz. Our operation system was Linux Ubuntu 16.04. We used gcc version 5.4.0. For simplicity, `rand()` is used as the source of randomness to generate random polynomials in this test only, while a cryptographically secure random number generator is adopted in NTRUEncrypt.

For each test shown in the rest of this section (except for the profiling result) we repeated the test for 8,000 times. For some reasons the code ran a few times slower than average at the first several tests, and then quickly convergent into a stable state. We suspect it is due to the overhead to initialize AVX2 (see, for example, [1], for AVX/SSE transition penalty). For consistency purpose, and also for the ease of analysis, we have carefully removed those small amounts of data.

5.2 Performance

Now we are ready to present an overview of the performance. Karatsuba uses 3 calls to Toom-4. Each Toom-4 uses 7 calls to Toom-3. Each Toom-3 uses 5 calls to index-based multiplication. In the end, we require 105 calls to the index-based multiplication algorithm. This is displayed in Figure **??** [1]. Our profiling tool shows that those index-based multiplications takes up almost 90% of the total cost of computation. So it is crucial to show that those vectorized index-based multiplications are faster than schoolbook multiplications.

[1] The cycles shown in the profiling tool is not consistent with the implementation result shown in Table 4 or Figure 6, etc. This is because the profiling tool is essentially a virtual machine, and therefore incurs a different running time.

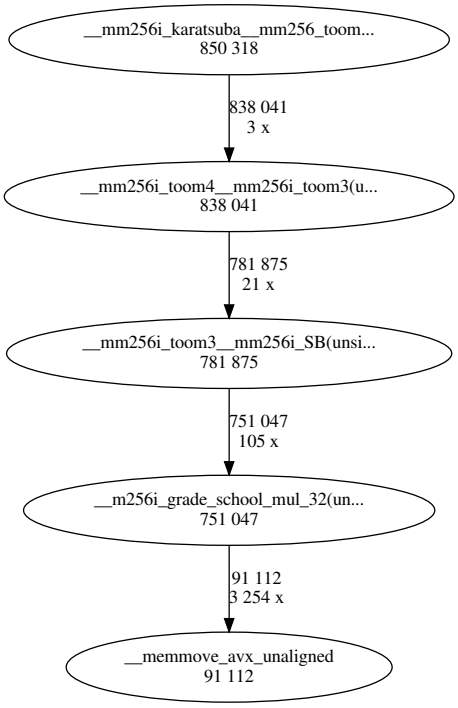


Fig. 2: Call graph. First row, i.e, 838 041, is the cycles; second row, i.e., 3 x, is the number of calls to the subroutine.

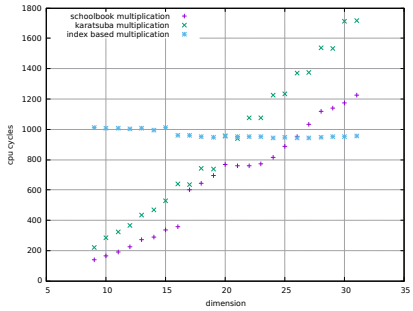
5.2.1 Performance of index-based multiplications In this section we show the performance of our vectorized index-based multiplications algorithm, and compare it with the following two methods:

- schoolbook multiplication
- Karatsuba then schoolbook multiplication

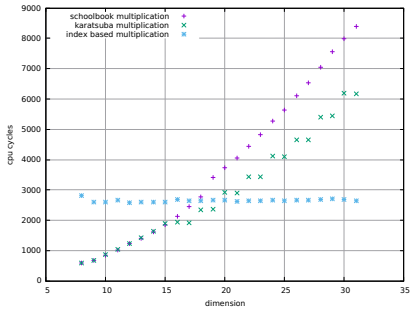
It is interesting to see that our algorithm starts to perform better than Karatsuba or schoolbook multiplication when the degree of input polynomial is around 18 to 20. For our target degree 31, our algorithm is almost twice faster than Karatsuba plus schoolbook multiplication, almost three times faster than a pure schoolbook multiplication.

5.2.2 Overall performance We have presented two optimizations in this paper, namely,

- a tailored breakdown of multiplications, and
- an index-based multiplication for small degree polynomials.



(a) Schoolbook vs Karatsuba vs Index-based for small degrees, with level 3 optimization (`-O3`).



(b) Schoolbook vs Karatsuba vs Index-based for small degrees, without level 3 optimization (`-O3`).

We show in Table 4 a performance comparison of our algorithm compared to the reference implementation of [4].

"*optimization 1+2*" uses both optimizations mentioned.

"*optimization 1 with AVX2*" is a method where only the first optimization is used. We use AVX2 to accelerate the breaking down. This result gives a overall performance of the scheme when constant-time implementation is required.

"*optimization 1 without AVX2*" is a method where only the first optimization is used and no vectorized operations are implemented for this optimization. This gives a best comparison to the original Karatsuba implementation as how good our tailored breaking down is.

In summery, on a modern CPU with AVX2 instructions, our implementation is 2.23 times better than the reference implementation in [4]. On a CPU without AVX2 instructions, our implementation is 1.6 times faster than [4].

5.3 Comparison

5.3.1 Comparison with reference code[4] The reference implementation uses Karatsuba to break a degree 743 multiplication into many polynomial multiplications whose degrees are no greater than $\lceil 743/32 \rceil - 1 = 23$. That is, there are $\log_2(32) = 5$ recursive calls, where each loop incurs 3 multiplications. Overall there are $3^5 = 243$ multiplications using schoolbook multiplications.

In comparison, we use 105 multiplications of degree 31 polynomials. Assuming a degree 31 polynomial multiplication takes roughly same time using index-based multiplication or schoolbook multiplication (indeed, our index-based solution is a bit faster than schoolbook, see Figure 4), we gain an improvement of around $243/105 \approx 2.3$. This is correctly observed in Table 4.

5.3.2 Comparison with other lattice-based cryptosystems We also give a rough comparison with other lattice based cryptosystems, namely the New Hope key exchange algorithm [9] and the NTRU-prime key encapsulation algorithm [14].

Note that Algorithm 2 and 3 in NTRUEncrypt provide a CCA-2 secure encryption algorithm which is stronger than the CPA secure key exchange as in New Hope [9]. Therefore, to deliver a fair comparison we present the cost of polynomial multiplications for all three algorithms.

It was reported in [14] that the multiplications takes $50k$ cycles for NTRU-prime with Toom-7 and $40k$ for NewHope with NTT; both are quite better than $130k$ cycles reported in this paper. We believe that the advantage is due to the fact that both implementations use assembly code for AVX2; while our code is written in C for more compatibility. Knowing that our code already improves 2.3 times over the reference implementation of [4]; and that we adopted a similar approach as [14] in terms of optimization, we are confident that our code could deliver a similar performance with assembly.

Nevertheless, although it is not directly related to the contribution of this work, it is worth mentioning that NTRUEncrypt with parameter set NTRU743 delivers a smaller throughput when used in key exchanges/encapsulations.

5.4 Constant-time algorithm/implementation

We have carefully designed our polynomial multiplication algorithm. Our algorithm does not have conditional branches, memory accesses or compiler optimizations regarding secret data. And all secret data are only used as operands of constant-time arithmetic operations.

For NTRU743 polynomials, our hierarchical multiplication uses same number of index-based multiplications. For AVX2 enabled processors, our index-based multiplications are also constant-time. For processors that does not support AVX2 instructions, our algorithm can still be made constant-time, so long as the subroutine that handles small polynomial multiplication is constant-time.

Thus we claim that our algorithm is a constant-time algorithm. If the processor (and the compiler) always carries out the same operation with the same latency, we will achieve a constant-time implementation. We remark that this assumption may not always be true. As a counter-example, there are cache missing attack, which exploit the timing difference of fetching data from memory and cache. See [38] for another counter-example of a non-constant implementation (and subsequently, a timing attack) for Curve25519 [12], which is a constant-time algorithm.

In order to show that our implementation is constant-time in practice, we tested out algorithm for the following three scenarios:

1. Fix same input polynomials a and b, and record the running time for our algorithm;
2. Fix one of the input polynomials, polynomial a, and record the running time for different b-s (the degree of b remains the same)
3. Choose different polynomial a and b, and record the running time.

The following tests are performed on an Intel Xeon E5-2640 v3 processor. For a more reliable computing power, we did the following system configurations: 1) we disabled power management and Intel SpeedStep Technology, and

Table 5: Statistical Analysis of 10,000 Measurements (in cycles)

Implementation	Mean	SD	SD/Mean
C -O3	187,834	605	0.0032
	187,825	603	0.0032
	187,817	600	0.0032
C -O0	1,500,030	5,547	0.0037
	1,500,094	5,607	0.0037
	1,500,135	5,400	0.0036
AVX2 -O3	115,434	2,123	0.0184
	115,392	1,347	0.0117
	115,503	2,939	0.0254
AVX2 -O0	313,507	2,912	0.0093
	313,461	2,055	0.0066
	313,547	3,074	0.0098

configured the CPU cores to run at a fixed 2.6 GHz frequency; 2) we disabled hyper-threading of the core on which we took measurements, 3) we configured the system to handle all interrupts but timer interrupts on the other cores. We did not adjust any settings concerning data or instruction cache behaviors. Therefore, the only differences between our testing environment and a typical/default one are CPU frequency, interruptions and interference from the other thread on the same core. None of these differences affect the constant-time behavior since they do not make an attacker capable of distinguishing the secret data from the other operand in a polynomial multiplication.

We repeat 10,000 measurements (in cycles) of all three scenarios for our C and AVX2 implementations with `-O3` and plot them in Figure 6 and 8, respectively. We observed: 1) two identical executions do not have the same execution time; and 2) there are periodic high measurements causing approximately 3,000 cycles delay, in Figure 6 around 192,000 cycles and in Figure 8 around 118,000 cycles. We believe that the former is caused by conditional branches (based on non-secret data) and that the latter is caused by system timer interrupts which cannot be disabled. We zoom-in to examine the first 1,000 measurements of each implementation and provide Figure 7 and 9. By observation, none of the plots shows any difference among three scenarios. We also provide statistical values (mean and standard deviation (SD)) for a wider range of implementations. In Table 5 we analyze 10,000 measurements of four implementations: C and AVX2 with `-O0` and `-O3`. The only noticeable difference is among the SDs of three scenarios in AVX2 implementations. The largest difference of SD-over-mean ratios is only 1.5%.

When we fully unroll the `for` loops in schoolbook multiplications (since we know the their number of iterations is 32 for $N = 743$), the variance/SD decreases. From Table 5, for the C implementation with `-O3`, the compiler optimize our code with aggressive loop unrolling, we can see a decrease in variance. We do not observe the same influence of optimization in AVX2 implementations because variance/SD is largely affected by timer interrupts, see Figure 8.

After all, the performance of our polynomial multiplications are quite consistent among all three scenarios. None of the issues mentioned above that cause non-constant execution time are related to secret date.

A minor concern one may have is on the constant offset for a minor portion of the data, eg. $192k$ cycles vs $188k$ cycles as in Figure 6. We argue that this phenomenon indeed suggests that the extra delay is due to the processor delay rather than secret-key related operations. Since we used different secret key for each new tests, intuitively speaking, if the delay was due to the secret key, then it should vary and appear pseudorandom or center-limited, as opposite to our result.

Hence, we conclude that our algorithm/implementation is constant-time.

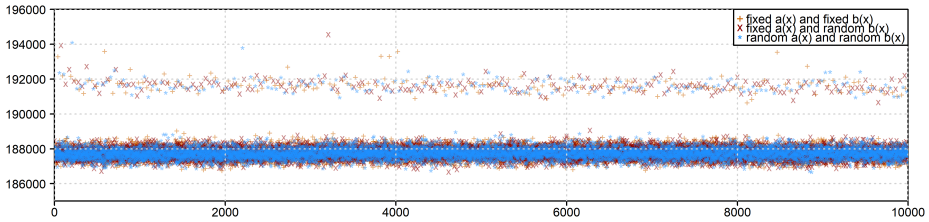


Fig. 4: 10,000 measurements of C implementation with level 3 optimization (`-O3`)

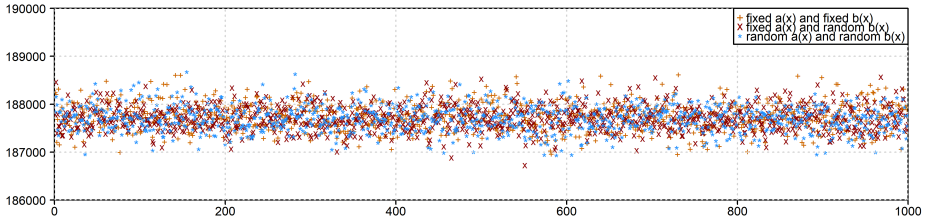


Fig. 5: 1,000 measurements of C implementation with level 3 optimization (`-O3`)

6 Future work

We conclude this paper by identifying a few future works. In this paper we presented a tailored optimization for NTRUEncrypt with parameter set NTRU743. A potential future work is to use the same design methodology to optimize NTRUEncrypt with other parameter sets, such as NTRU443.

Another potential future work is to perform a more through analysis of the implementation at the assembly level, to identify the reason for the small potion of constant delay in our testing result.

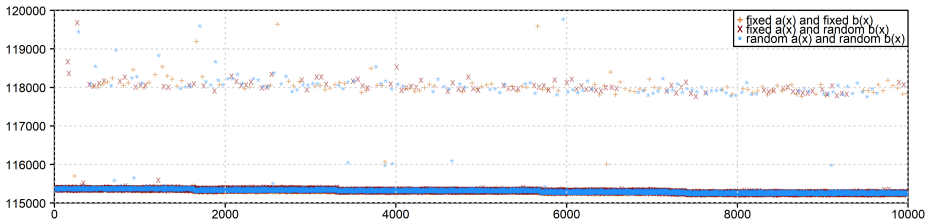


Fig. 6: 10,000 measurements of AVX2 implementation with level 3 optimization (`-O3`)

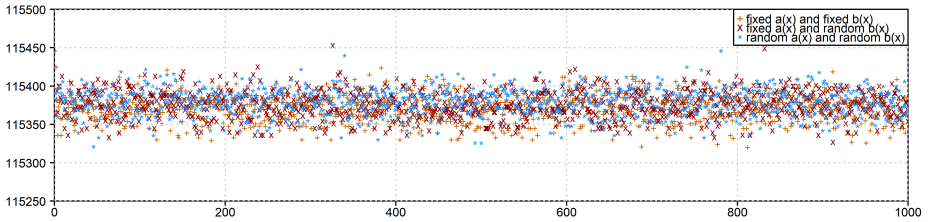


Fig. 7: 1,000 measurements of AVX2 implementation with level 3 optimization (`-O3`)

In addition, due to NIST's recent call for proposal of quantum-safe cryptography [16], we expect to see more lattice based cryptography implementations. It is interesting to investigate if the method in this paper is compatible with those schemes, especially those whose modulus is a power of 2.

Last but not least, as we mentioned before, our implementation can be migrant to AVX-512 with minimum modification. It remains interesting to see how much our implementation can be improved with such a migration.

References

1. Avoiding AVX-SSE transition penalties.
2. Falcon: Fast-fourier lattice-based compact signatures over NTRU.
3. Framework to integrate post-quantum key exchanges into internet key exchange protocol version 2 (ikev2).
4. NTRU OpenSource Project. online. available from `https://github.com/NTRUOpenSourceProject/ntru-crypto`.
5. Quantum-safe hybrid (QSH) key exchange for transport layer security (TLS) version 1.3.
6. IEEE Std 1363.1-2008. IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices, 2008.
7. Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 153–178, 2016.

8. Martin R. Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions - cryptanalysis of some FHE and graded encoding schemes. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 153–178, 2016.
9. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343, 2016.
10. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343, 2016.
11. Shi Bai, Thijs Laarhoven, and Damien Stehlé. Tuple lattice sieving. *IACR Cryptology ePrint Archive*, 2016:713, 2016.
12. Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 207–228, 2006.
13. Daniel J. Bernstein. A subfield-logarithm attack against ideal lattices, 2014. available from `https://blog.cr.yp.to/20140213-ideal.html`.
14. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime. *IACR Cryptology ePrint Archive*, 2016:461, 2016.
15. Leon Groot Bruinderink, Andreas Hlsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. Cryptology ePrint Archive, Report 2016/300, 2016. `http://eprint.iacr.org/2016/300`.
16. Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. National Institute of Standards and Technology Internal Report 8105, February 2016.
17. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, pages 1–20, 2011.
18. Yuanmi Chen and Phong Q Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT 2011*, pages 1–20. Springer, 2011.
19. Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without an encoding of zero. *IACR Cryptology ePrint Archive*, 2016:139, 2016.
20. Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In *EUROCRYPT*, pages 52–61, 1997.
21. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.
22. Scott R. Fluhrer. Quantum cryptanalysis of NTRU. *IACR Cryptology ePrint Archive*, 2015:676, 2015.
23. Consortium for Efficient Embedded Security. Efficient Embedded Security Standard (EESS) #1 version 3.0, 2015. available from `https://github.com/NTRUOpenSourceProject/ntru-crypto`.
24. Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, pages 31–51, Berlin, Heidelberg, 2008. Springer-Verlag.

25. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010.
26. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
27. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.
28. Shay Gueron and Fabian Schlieker. *Software Optimizations of NTRUEncrypt for Modern Processor Architectures*, pages 189–199. Springer International Publishing, Cham, 2016.
29. David Harvey. Faster arithmetic for number-theoretic transforms. *CoRR*, abs/1205.2926, 2012.
30. Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing ntruencrypt parameters in light of combined lattice reduction and MITM approaches. In *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, pages 437–455, 2009.
31. Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing Parameters for NTRUEncrypt (full version). *IACR Cryptology ePrint Archive*, 2015:708, 2015.
32. Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing Parameters for NTRUEncrypt. In *Topics in Cryptology - CT-RSA 2017, The Cryptographers' Track at the RSA Conference 2017*, 2017.
33. Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, pages 267–288, 1998.
34. Jeffrey Hoffstein and Joseph H. Silverman. Meet-in-the-middle Attack on an NTRU private key, 2006. available from `http://www.ntru.com`.
35. Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In *CRYPTO*, pages 150–169, 2007.
36. Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In *CRYPTO*, pages 150–169, 2007.
37. European Telecommunications Standards Institute. Quantum safe cryptography and security; an introduction, benefits, enablers and challenges.
38. Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *CANS*, 2016.
39. Paul Kirchner and Pierre-Alain Fouque. Comparison between subfield and straightforward attacks on NTRU. *IACR Cryptology ePrint Archive*, 2016:717, 2016.
40. Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1219–1234, 2012.
41. Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007.
42. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

43. John M. Schanck, William Whyte, and Zhenfei Zhang. Circuit-extension handshakes for tor achieving forward secrecy in a quantum world. *PoPETs*, 2016(4):219–236, 2016.
44. Peter W. Shor. Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In *ANTS*, page 289, 1994.
45. Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 27–47, 2011.
46. Thomas Wunderer. Revisiting the hybrid attack: Improved analysis and refined security estimates. Cryptology ePrint Archive, Report 2016/733, 2016. `http://eprint.iacr.org/2016/733`.
47. Accredited Standards Committee X9. Lattice-Based Polynomial Public Key Establishment Algorithm for the Financial Services Industry, 201.
48. Zhenfei Zhang. NTRU polynomial multiplication code. online. available from `https://github.com/zhenfeizhang/polynomial_mul`.