Implementation and Analysis of the NTRU Algorithm in Java

by

Tatiana Linardopoulou

Supervised by Dr. Paulo S. L. M. Barreto

A senior thesis submitted in partial fulfillment of the departmental honors requirements for the degree of

Bachelor of Science
Computer Science & Systems
University of Washington Tacoma

December 2020

Presentation of work given on **12/14/2020**

The student has satisfactorily completed the Senior Thesis, presentation and senior elective course requirements for CSS Departmental Honors.

Faculty advisor: _____ Date 12/16/2020

CSS Program Chair: _____ Date 12/16/2020

Abstract

Implementation and Analysis of the NTRU Algorithm in Java

Tatiana Linardopoulou

This research thesis describes in detail a Java implementation of the latest version of the NTRU algorithm, which is an important candidate for post-quantum cryptography standardization. Multiple supporting functions and classes were built for the Java implementation, including methods for integer addition, integer multiplication, polynomial addition, polynomial multiplication, and modular reduction. This background work allowed for the Java implementation of the entire NTRU algorithm. Detailed explanations for which algorithms are best-suited for the implementation of arithmetic operations are provided. Additionally, this thesis clarifies which algorithms are the most robust for a cryptographically secure implementation of the NTRU algorithm. Finally, there is a discussion on the caveats and potential security issues, specific to a Java implementation of NTRU. These contributions will be useful in practice and beneficial to future implementations of the NTRU algorithm.

Contents

Introduction

An important issue in modern cryptography is the ability of current cryptosystems to withstand attacks by quantum computers. Various cryptographic algorithms have been proposed to defend against such attacks. Currently, the National Institute of Standards and Technology (NIST) is going through a post-quantum cryptography standardization process, in which NTRU is one of the candidates for selection [1]. NTRU is an excellent candidate for standardization as it is efficient, compact, and has a long history of cryptanalysis [1]. Additionally, versions of NTRU have already been standardized by other organizations [1].

The version of NTRU that is the focus of this paper is the NTRU algorithm proposed in the second round of the NIST standardization process [2]. The proposed algorithm is deceptively simple and short, fitting in only two presentation slides [3]. However, it is quite challenging to implement. In Java, as well as implementations in other languages, there are many supporting functions and classes which need to be built before one can actually create the methods described by the authors of the algorithm [2]. In fact, they explicitly point out the lack of these methods: “Algorithms for integer addition, integer multiplication, polynomial addition, polynomial multiplication, modular reduction (Rq, S2, S3, Sq), and canonical representatives (Rq, S3, Sq) are omitted [2]”.

These background algorithms, however, are not only essential to the functionality of the implementation of this version of NTRU, but are also notoriously challenging to implement because of the amount of bit-manipulation involved. This project consists of a Java implementation of the entire NTRU algorithm, including these previously omitted sections, which will be described in a straight-forward manner and will hopefully make future implementations easier.

Contributions

This work provides several contributions to the field of cryptography, in general, and to the implementation of the NTRU algorithm, in particular. Firstly, detailed explanations for which algorithms are best-suited for the implementation of arithmetic operations are provided. Secondly, this paper clarifies which algorithms are the most robust for a cryptographically secure implementation of the NTRU algorithm. Finally, there is a discussion on the caveats and potential security issues, specific to a Java implementation of the NTRU algorithm. These contributions will be useful in practice and beneficial to future implementations of the NTRU algorithm.

The remainder of this paper is structured as follows: a background of the NTRU algorithm, a discussion on secure and efficient background algorithms, definitions and parameters, a description of the Java implementation, results and conclusions, acknowledgements, references, and an Appendix with code examples from the implementation.

Background

NTRU Evolution

The NTRU algorithm was originally proposed by J. Hoffstein, J. Pipher, J.H. Silverman [4] in an informal rump session of the Crypto ’96 conference [5]. It was, and remains, a lattice-based alternative to the Rivest-Shamir-Adleman (RSA) algorithm [6] and to Elliptic Curve Cryptography (ECC) [7]. NTRU is based on the shortest vector problem in a lattice—a problem that theoretically is not breakable by quantum computers [8]. Early versions of the algorithm were problematic, largely in that, much like early versions of the RSA cryptosystem, the chosen parameters were too small [9]. NTRU has gone through many evolutions, though mainly in regards to parameters—the structure of the algorithm itself has remained largely unchanged [1]. The proposed NTRU algorithm that serves as the basis for this implementation is a merger of two different algorithms that appeared in the first round of the NIST post-quantum standardization process [10]. These algorithms were the NTRUEncrypt (which included NTRU-HPS) [10] and the NTRU-HRSS-KEM [10]. In this implementation, the focus is specifically on NTRU-HPS, as it was previously shown to be both more efficient and more secure than NTRU-HRSS [2].

NTRU Synopsis

A synopsis of the version of the NTRU algorithm [3] that serves as the basis for our implementation, as well as its requirements, are shown in Figure 1:

Please note: symbol definitions can be located in the Definitions subsection of this thesis.

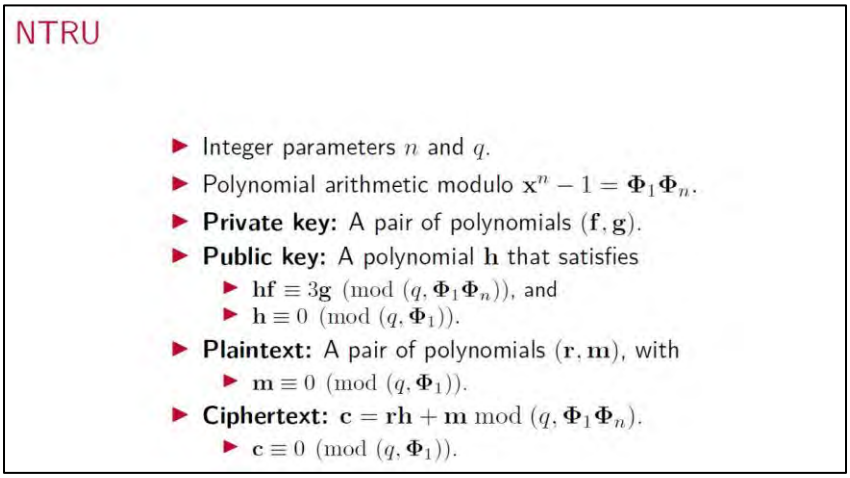

Figure 1: NTRU Algorithm, as presented within the NIST Round 2 Presentation [3]

A synopsis of the decryption algorithm for this version of NTRU [3] is shown in Figure 2:

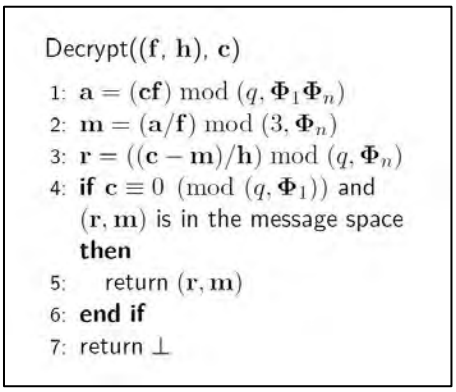

Figure 2: Decryption Algorithm for NTRU, as presented within the NIST Round 2 Presentation [3]

Algorithms

The implementation of cryptographic algorithms must be done with extra care because careless implementation can lead to leakage of private information (i.e., information about private keys). This is particularly problematic in conditional statements whose results depend on secret information. These types of statements can lead to a specific type of side-channel attack [11] called a timing attack. This is a type of attack which involves analyzing how long the execution of an algorithm takes and depends on the fact that the time of a conditional statement can vary based on different input [12]. The attacker can use this timing information to work backwards and discover the private input, such as a private key. Moreover, this type of attack is computationally inexpensive and only requires knowledge of the ciphertext [12].

As a very simple example of such an attack, with the inclusion of conditional statements, consider a theoretical method that checks an administrator password [13] (Example adapted for Java by the author of this thesis):

```
boolean admin_pass(char[] input_pass) {
    for (int i = 0; i < input_pass.length; i++) {
        if(input_pass[i] != password[i]) {
            return false;
        }
    }
    return true;
}
```

Figure 3: Java code example of conditional statements that allow for timing attacks

In this example, each character of an input password is checked against the character of the stored password and, if any character in the input password is incorrect, returns false. Note that if the first character in the input password is incorrect, this method will return false immediately. An attacker can record this increment of response time. However, if they then try a different input password and the first character is correct, the method will take a longer amount of time to execute because it will go on to the next iteration of the loop. The attacker can then record that increment of response time as well. Thus, much like a safecracker listening

for the click of the correct code combination, character by character, the attacker can discover the entire secret password.

Any programming algorithm that includes conditional statements can be vulnerable to timing attacks. For example, the Extended Euclidian Algorithm is efficient in achieving inversion, so it may seem like a good algorithm choice for the inversion required in our Java implementation of NTRU. However, this algorithm is exquisitely difficult to implement without the use of conditional statements and the subsequent leaking of private information [14]. For this reason, this implementation uses alternative, more secure algorithms to achieve inversion.

Inversion

The Itoh-Tsujii inversion algorithm [15] is used for binary polynomial inversion. This algorithm requires a fixed number of squaring and multiplication operations, which result in the same number of steps, regardless of input parameters. In C implementations, this algorithm also benefits from the existence of machine-level instructions, making it a secure and efficient choice for binary polynomial inversion [15].

However, for ternary polynomial inversion, the Itoh-Tsujii algorithm does not appear to provide an advantage, as there are no machine-level instructions for ternary operations. Moreover, there is no reason to believe that such instructions will be available in the future. Instead, the Almost Inverse Algorithm [16] can be implemented in a way that does not leak information. The authors of the NTRU algorithm claim that the Bernstein-Yang inversion method is more efficient [2], however the evidence provided does not seem to support this claim [2]. Additionally, the Bernstein-Yang algorithm requires floating-point arithmetic [17], which can lead to decryption issues due to rounding errors. To improve on this algorithm, we created a workaround using a combination of bit-shifting and integer arithmetic. This involved implementing ternary arithmetic in which we use Boolean expressions, thus mimicking hardware operations, and resulting in a constant time implementation.

For q-ary polynomial inversion, we implemented Hensel's lifting algorithm, as per the NTRU algorithm specification [2]. This algorithm is constant time, does not leak any information about private keys, and is efficient [2].

Multiplication

The plain multiplication algorithm for polynomials could be used [18], and would be secure. However, for large dimensions of hundreds of coefficients, such as those in the parameters required for NTRU [2], this algorithm is highly inefficient. Subsequently, the Karatsuba and ternary Karatsuba algorithms were implemented for both binary and ternary polynomial multiplication [18]. They were chosen due to their efficiency and direct application to polynomials. Ternary Karatsuba, specifically, was implemented because in Python and C implementations and tests of the NTRU Algorithm, performed by Dr. Paulo S. L. M. Barreto, this algorithm was shown to significantly improve execution time.

An alternative one might consider for polynomial multiplication is the Fast-Fourier Transform (FFT), but the dimensions would have to be doubled to be large enough to hold two polynomials, and even tripled in the case of larger inputs. Additionally, operations involve modulo a prime number, instead of modulo two, so the FFT quickly becomes an expensive algorithm. It is unclear that there would be any gain from using the FFT. In practice, for parameters the size of those used for NTRU, it seems highly unlikely that the FFT would be more efficient than the Karatsuba-based algorithms.

Definitions and Parameters

The selected definitions, derived constants, and recommended parameters used in this Java implementation were originally described in the NTRU Algorithm Specifications and Supporting Documentation [2] that was submitted in the second round of the NIST standardization process. They are listed in the subsections that follow.

Definitions

All definitions are with respect to a fixed odd prime n [2].

1. $(\mathbb{Z}/n)^x$ is the multiplicative group of integers modulo n.
2. Φ_1 is the polynomial $(x - 1)$.
3. Φ_n is the polynomial $(x^n - 1) = (x - 1) = x^{n-1} + x^{n-2} + \ldots + 1$.
4. R is the quotient ring $\mathbb{Z}[x]/(\Phi_1 \Phi_n)$.
5. S is the quotient ring $\mathbb{Z}[x]/(\Phi_n)$.
6. R/3 is the quotient ring $\mathbb{Z}[x]/(3, \Phi_1 \Phi_n)$.
7. R/q is the quotient ring $\mathbb{Z}[x]/(q, \Phi_1 \Phi_n)$. The canonical R/q-representative of $a \in \mathbb{Z}[x]$ is the unique polynomial $b \in \mathbb{Z}[x]$ of degree at most $n - 1$ with coefficients in $\{-q/2, -q/2 + 1, \ldots, q/2 - 1\}$ such that $a \equiv b \ (mod \ (q, \Phi_1 \Phi_n))$.
8. S/2 is the quotient ring $\mathbb{Z}[x]/(2, \Phi_n)$. The canonical S/2-representative of $a \in \mathbb{Z}[x]$ is the unique polynomial $b \in \mathbb{Z}[x]$ of degree at most $n - 2$ with coefficients in $\{0, 1\}$ such that $a \equiv b \ (mod \ (2, \Phi_n))$.
9. S/3 is the quotient ring $\mathbb{Z}[x]/(3, \Phi_n)$. The canonical S/3-representative of $a \in \mathbb{Z}[x]$ is the Unique polynomial $b \in \mathbb{Z}[x]$ of degree at most $n - 2$ with coefficients in $\{-1, 0, 1\}$ such that $a \equiv b \ (mod \ (3, \Phi_n))$.
10. S/q is the quotient ring $\mathbb{Z}[x]/(q, \Phi_n)$. The canonical S=q-representative of $a \in \mathbb{Z}[x]$ is the unique polynomial $b \in \mathbb{Z}[x]$ of degree at most $n - 2$ with coefficients in $\{-q/2, -q/2 + 1, \ldots, q/2 - 1\}$ such that $a \equiv b \ (mod \ (q, \Phi_n))$.
11. A polynomial is ternary if its coefficients are in $\{-1, 0, 1\}$.

12. A ternary polynomial $v = \sum_i v_i x^i$ has the non-negative correlation property if $\sum_i v_i v_{i+1} \geq 0$.
13. T is the set of non-zero ternary polynomials of degree at most $n - 2$. Equivalently, T is the set of canonical S/3-representatives.
14. T_+ is the subset of T consisting of polynomials with the non-negative correlation property.
15. $\mathrm{T}(\mathrm{d})$, for an even positive integer d, is the subset of T consisting of polynomials that have Exactly $d/2$ coefficients equal to $+1$ and $d/2$ coefficients equal to -1.
16. $\perp$ is the logical 'false'.

Derived Constants

logq

Formula: $\log_2(q)$

sample_iid_bits

Formula: $8 * (n - 1)$

sample_fixed_type_bits

Formula: $30 * (n - 1)$

sample_key_bits

Formula: sample_iid_bits + sample_fixed_type_bits

sample_plaintext_bits

Formula: sample_iid_bits + sample_fixed_type_bits

packed_sq_bytes

Formula: $\lceil (n - 1) * logq/8 \rceil$

packed_rq0_bytes

Formula: $\lceil (n - 1) * logq/8 \rceil$

dpke_public_key_bytes

Formula: packed_rq0_bytes

dpke_private_key_bytes

Formula: 2 * packed_s3_bytes + packed_sq_bytes

dpke_plaintext_bytes

Formula: 2 * packed_s3_bytes

dpke_ciphertext_bytes

Formula: packed_rq0_bytes

Recommended Parameters

	ntruhps2048509	ntruhps2048677	ntruhps4096821
n	509	677	821
q	2048	2048	4096
Sample_fixed_type_bits	15240	20280	24630
sample_iid_bits	4064	5408	6560
sample_key_bits	19304	25688	31190
sample_plaintext_bits	19304	25688	31190
packed_rq0_bytes	699	930	1230
packed_sq_bytes	699	930	1230
dpke_public_key_bytes	699	930	1230
dpke_private_key_bytes	903	1202	1558
dpke_plaintext_bytes	204	272	328
dpke_ciphertext_bytes	699	930	1230

Implementation

Implementing the RS2 Class

Most of the methods described below have been implemented as static to avoid creating objects that may contain sensitive information [19]. The user is advised to pass arrays representing ring elements as parameters. In Java, unlike in C, which allows the use of pointers, an index offset must be used on the arrays to specify where data should be written. In order to make the implementation efficient, bit shifting was used to simulate multiplication.

Acceptable:

This Boolean method is used to check that ring degree 'n', represented as an integer, is an acceptable ring degree to define R/2 and S/2. It returns true if n is acceptable and false otherwise. A ring degree is acceptable if n is a prime such that 2 is primitive mod n.

RS2_nbits:

This method is necessary to compute the actual number of bits needed to represent u. The input parameter is a natural number u, represented as a long array, integer 'c' representing the word count of 'u', and the method returns the integer number of bits needed to represent u.

Display:

This method returns a String representation of the input binary polynomial u. The input is a binary polynomial, represented as long array 'u'. For simplicity on coding, this implementation calls the toString() method, from the Java Arrays library, on u.

Inv:

This method computes u^{-1} in the finite field $(Z/2)[x]/(\Phi_n(x))$ for 0 <= degree of u < n. This is the Itoh-Tsujii inversion algorithm. The input parameters are 'n', a prime with

$(x^n - 1)/(x - 1)$, which is irreducible in $(Z/2)[x]$, output array 'w', and long array 'u', a non-zero element from $(Z/2)[x]/(\Phi_n(x))$.

Clsqr:

This method computes carryless u^2 for a 64-bit (1-word) polynomial u, represented as a long. The input parameters are w_o, an integer representing the index offset in output array 'w' where u^2 is to be written, and u, the long representation of the 1-word ring element to be squared.

Clmul:

This method computes the caryless multiplication of two binary, 64-bit, polynomials u and v, represented as longs. The input parameters include output array 'w', w_o, an integer representing the index offset in output array 'w' where u*v is to be written, as well as u and v, the long representations of the 1-word ring elements to be multiplied.

Kar:

This method computes u*v in (Z/2)[x] with the help of Karatsuba's algorithm for multiplication, using 2-way block sizes. Input parameters include integer 'k', which represents lg(m) where m is the 2-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the long array representations of the binary polynomials to be multiplied, as well as offsets u_o and v_o.

Kar3:

This method computes u*v in (Z/2)[x] by applying an initial 3-way Karatsuba layer, followed by recursive 2-way Karatsuba layers. Input parameters include integer 't', which represents lg(m) where m is the 3-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the long array representations of the binary polynomials to be multiplied, as well as offsets u_o and v_o.

Redc:

This method is used to reduce rings mod $x^n - 1$ or mod $\Phi_n(x)$. Input parameters include 'n', the ring degree, long array 'u' representing the ring element to reduce (with degree of u < 2*n), and integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both). The method returns the number of words left on 'u', or, if no reduction is performed, returns -1.

Mul:

This method computes u*v in (Z/2)[x] using Karatsuba multiplication. An initial 3-way Karatsuba layer is automatically applied when n is closer to $2^{\lfloor lg(n) \rfloor}$ than $2^{\lceil lg(n) \rceil}$. Input parameters include 'n', representing the ring degree, output array 'w', u and v, the long array representations of the binary polynomials to be multiplied (and later reduced using the redc method). The final parameter is integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both).

Sqr:

This method computes $u^{(2^k)}$ in $(Z/2)[x]/(x^n - 1)$. Input parameters include 'n', representing the ring degree, long array 'u', an element from $(Z/2)[x]/(x^n - 1)$, and integer 'k' representing the number of times to square 'u'.

Implementing the RS3 Class

Only the most consequential methods have been included in this description. Most of the methods described below have been implemented as static to avoid creating objects that may contain sensitive information [19]. The user is advised to pass arrays representing ring elements as parameters. In Java, unlike in C, which allows the use of pointers, an index offset must be used on the arrays to specify where data should be written. Additionally, helper methods to convert Booleans to longs, as well as integers to Booleans, were needed since casting of those types is not allowed in Java. Addition, subtraction, and negation methods have been omitted from this description, but the methods have been included in the implementation and perform those operations in GF(3).

Acceptable:

This Boolean method is used to check that ring degree 'n', represented as an integer, is an acceptable ring degree to define R/3 and S/3. It returns true if n is acceptable and false otherwise. A ring degree is acceptable if n is a prime such that 3 is primitive mod n and $\lceil (n + 3)/64 \rceil$ is equal to $\lceil n/64 \rceil$.

RS3_ntrits:

This method is necessary to compute the actual number of trits needed to represent u. The input parameter is array 'u', integer 'c' representing the word count of u[0] and/or u[1], and the method returns the integer number of bits needed to represent u.

RS3_checksum:

This method computes u mod(q, x -1). Input parameters include integer 'lgq', representing lg(q), ring degree 'n, and 'u', and element of $(Z/3)[x]/(x^n - 1)$.

RS3_inv:

This method computes u^{-1} in the finite field $(Z/3)[x]/(\Phi_n(x))$ for 0 <= degree of u < n. This algorithm combines parts of Schroeppel et. al.'s Almost Inverse algorithm with a modification of the Bernstein and Yang method. The input parameters are 'n', a prime with $(x^n - 1)/(x - 1)$, which is irreducible in $(Z/2)[x]$, output array 'v', and 'u', a non-zero element from $(Z/3)[x]/(\Phi_n(x))$.

_redc:

This method is used to reduce rings mod $x^n - 1$ or mod $\Phi_n(x)$. Input parameters include 'n', the ring degree, int 'm' representing the upper bound for the degree of 'u', array 'u' representing the ring element to reduce, and integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both).

S3:

This method reduces all coefficients of 'u' mod 3 to create an element of $(Z/3)[x]$, $(Z/3)[x]/(x^n - 1)$, or $(Z/3)[x]/(\Phi_n(x))$. Input parameters include 'n', the ring degree, output array 'w', a list of integers called 'u', and integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both).

_kar:

This method computes u*v in $(Z/3)[x]$, with the help of Karatsuba's algorithm for multiplication, using 2-way block sizes. Input parameters include integer 'k', which represents lg(m) where m is the 2-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the array representations of the ternary polynomials to be multiplied, as well as offsets u_o and v_o.

_kar3:

This method computes u*v in $(Z/3)[x]$ by applying an initial 3-way Karatsuba layer, followed by recursive 2-way Karatsuba layers. Input parameters include integer 't', which represents lg(m) where m is the 3-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the array representations of the ternary polynomials to be multiplied, as well as offsets u_o and v_o.

RS3_mul:

This method computes u*v in $(Z/3)[x]$ using Karatsuba multiplication. An initial 3-way Karatsuba layer is automatically applied when n is closer to $2^{\lfloor lg(n) \rfloor}$ than $2^{\lceil lg(n) \rceil}$. Input parameters include 'n', representing the ring degree, output array 'w', u and v, the array representations of the ternary polynomials to be multiplied (and later reduced using the _redc method). The final parameter is integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both).

Implementing the RSq Class

Only the most consequential methods have been included in this description. Most of the methods described below have been implemented as static to avoid creating objects that may contain sensitive information [19]. The user is advised to pass arrays representing ring elements as parameters. In Java, unlike in C, which allows the use of pointers, an index offset must be used on the arrays to specify where data should be written.

RSq_acceptable:

This Boolean method is used to check that ring degree 'n', represented as an integer, is an acceptable ring degree to define R/q and S/q. It returns true if n is acceptable and false otherwise. A ring degree is acceptable if n is a prime, where $n < 2^{11}$, and such that 2 and 3 are primitive mod n and $\lceil (n + 3)/64 \rceil$ is equal to $\lceil n/64 \rceil$.

RSq_checksum:

This method computes u mod(q, x -1). Input parameters include integer lgq, representing lg(q), ring degree 'n, and 'u', and element of $(Z/q)[x]/(x^n - 1)$.

Sq:

This method maps 'u' from $(Z/3)[x]/(x^n - 1)$ to $(Z/q)[x]/(x^n - 1)$. Input parameters include ring degree 'n, output short array 'v', and 'u', and element of $(Z/3)[x]/(x^n - 1)$.

_kar:

This method computes u*v in $(Z/q)[x]$, with the help of Karatsuba's algorithm for multiplication, using 2-way block sizes. Input parameters include integer 'k', which represents lg(f) where f is the full 2-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the array representations of the q-ary polynomials to be multiplied, as well as offsets u_o and v_o.

kar3:

This method computes u*v in $(Z/q)[x]$ by applying an initial 3-way Karatsuba layer, followed by recursive 2-way Karatsuba layers. Input parameters include integer 't', which represents lg(m) where m is the 3-way Karatsuba block size, output array 'w', and w_o, an integer representing the index offset in output array 'w' where u*v is to be written. Additional inputs include u and v, the array representations of the q-ary polynomials to be multiplied, as well as offsets u_o and v_o.

mod2:

This method reduces 'v' from $(Z/q)[x]/(\Phi_n(x))$, where q is a power of 2, to $(Z/2)[x]/(\Phi_n(x))$. Input parameters include ring degree 'n, array 'v', the ring element to reduce, and output array 'u'.

RSq_mul:

This method computes u*v in $(Z/q)[x]$ using Karatsuba multiplication. An initial 3-way Karatsuba layer is automatically applied when n is closer to $2^{\lfloor lg(n) \rfloor}$ than $2^{\lceil lg(n) \rceil}$. Input parameters include integer 'lgq', representing lg(q), 'n', representing the ring degree, output array 'w', u and v, the array representations of the q-ary polynomials to be multiplied (and later reduced). An additional parameter is integer red, which indicates the modular reduction option (where an input of 0 means no reduction, 1 means reduce $x^n - 1$, 2 means reduce mod $\Phi_n(x)$, and 3 means reduce mod both). The final input parameter is Boolean 'cen', which specifies whether the coefficients of the product should be centered around zero.

RSq_inv:

This method computes u^{-1} in the finite field $(Z/q)[x]/(\Phi_n(x))$ using the Hensel Lift. The input parameters are integer 'lgq', representing lg(q), integer 'n', representing the ring degree, output array 's', and 'u', the ring element to invert, and Boolean 'cen', which specifies whether the coefficients of the product should be centered around zero.

RSq_pack:

This method packs u in canonical form, either from $R/q = (Z/q)[x]/(x^n - 1)$, satisfying $u = 0(mod\ q, x - 1)$, or from $S/q = (Z/q)[x]/(\Phi_n(x))$, into a byte array. The input parameters are integer 'lgq', representing lg(q), integer 'n', representing the ring degree, output byte array 'p', an integer offset, 'u' the ring element to pack, and Boolean 'sq', which specifies whether to encode Sq(u) or Rq0(u).

RSq_unpack:

This method unpacks byte array 'p' into 'u' in $S/q = (Z/q)[x]/(\Phi_n(x))$ or $R/q = (Z/q)[x]/(x^n - 1)$, in canonical form. The input parameters are integer 'lgq', representing lg(q), integer 'n', representing the ring degree, output array 'u' a ring element decoded from 'p' with centered coefficients, 'p' the byte array to unpack, an integer offset, and Boolean 'sq', which specifies whether to unpack into S/q, if sq is not equal to 0, or R/q, if sq is equal to 0.

Implementing the NTRU Class

This class was implemented as per the method descriptions within the NTRU Algorithm Specifications and Supporting Documentation [2]. The RS2, RS3, and RSq Classes must be implemented prior to the implementation of this class.

NTRU_param:

This method includes all NTRU parameter pairs (n, lgq) with $n < 2^{11}$ and $q < 2^{12}$, such that 2 and 3 are primitive mod n and $\lceil (n + 3)/64 \rceil$ is equal to $\lceil n/64 \rceil$ and $(3/8) * n \leq q/8 - 2 \leq (2/3) * n$.

NTRU_acceptable:

This Boolean method is used to check that ring degree 'n', represented as an integer, is an acceptable ring degree to define R/q and S/q. It returns true if n is acceptable and false otherwise. A ring degree is acceptable if n is a prime, where $n < 2^{11}$, and such that 2 and 3 are primitive mod n and $\lceil (n + 3)/64 \rceil$ is equal to $\lceil n/64 \rceil$ and $(3/8) * n \leq q/8 - 2 \leq (2/3) * n$.

NTRU_parse:

This method finds 'n' and 'q' such that packed_sq_bytes is the corresponding S/q element size in bytes. Input parameters include packed_sq_bytes, representing the ring element size in bytes, and output array lgq_ptr. The method returns the corresponding ring degree n, or 0 if not suitable ring exists for the size.

NTRU_sample_hps:

This method generates a pair of NTRU-HPS ternary vectors 'u' and 'v', each of degree at most $n - 2$, meaning they are in $(Z/3)[x]/(\Phi_n(x))$. This corresponds to Sample_fg and Sample_rm in the Round 2 written specification. The input parameters are integer 'lgq', representing lg(q), integer 'n', representing the ring degree, output arrays 'u' and 'v'

representing ternary vectors, output array ‘z’, and byte array uv_bits, which is a byte array of bit length sample_key_bits $= (8 + 30) * (n - 1)$ bits.

DPKE_Sample_HPS:

This method creates an NTRU-hps short lattice element (u, v) in packed form packed_uv, packed from a secret bit string uv_bits. The input parameters are integer ‘lgq’, representing lg(q), integer ‘n’, representing the ring degree, output byte array packed_uv of length dpke_plaintext_bytes, and byte array uv_bits, a bit string of bit length sample_key_bits $= (8 + 30) * (n - 1)$ bits.

DPKE_Public_Key:

This method computes ‘h’, satisfying $Rq(h * f) = 3 * g$, and ‘h_q’ satisfying $Sq(h * h\_q) = 1$. The input parameters are integer ‘lgq’, representing lg(q), integer ‘n’, representing the ring degree, _f and _g, elements of $(Z/3)[x]/(\Phi_n(x))$, and output arrays h and h_q, which are elements of $(Z/q)[x]/(x^n - 1)$.

DPKE_Key_Pair:

This method creates an NTRU-hps key-pair in packed form (packed_private_key, packed_public_key), where the packed_private_key is a byte array of length dpke_private_key_bytes and the packed_public_key is a byte array of length dpke_public_key_bytes. The input parameters are integer ‘lgq’ representing lg(q), integer ‘n’ representing the ring degree, byte array fg_bits, a bit string of bit length sample_key_bits $= (8 + 30) * (n - 1)$ bits, and output byte arrays packed_private_key and packed_public_key.

DPKE_Encrypt:

This method encrypts an NTRU encapsulation pair with the NTRU deterministic public key encryption (DPKE) scheme. The input parameters are integer ‘lgq’ representing lg(q), integer ‘n’ representing the ring degree, packed_public_key, which is the target public key in packed form,

packed_rm, which is the NTRU encapsulation pair in standard packed form, and output byte array ‘packed_ciphertext of length dpke_ciphertext_bytes $= \lceil (n - 1) * lgq/8 \rceil$ bytes.

DPKE_Decrypt:

This method decrypts an NTRU cryptogram with the NTRU deterministic public key encryption (DPKE) scheme. The input parameters are integer ‘lgq’ representing lg(q), integer ‘n’ representing the ring degree, packed_private_key, which is the target private key in standard packed form, output byte array packed_rm of length dpke_plaintext_bytes, which is the NTRU encapsulation pair in standard packed form, and byte array packed_ciphertext, an NTRU cryptogram in standard packed form.

Results/Conclusion

This research project resulted in a Java implementation of the latest version of the NTRU algorithm, an important candidate for post-quantum cryptography standardization. Multiple supporting functions and classes were built and described in detail, including methods for integer addition, integer multiplication, polynomial addition, polynomial multiplication, and modular reduction. Additionally, algorithm choices for the inversion and multiplication of polynomials, as well as those for achieving a cryptographically secure implementation were discussed. This background work allowed for the Java implementation of the NTRU algorithm in its entirety, which was described in a straight-forward manner and will be useful in future implementations.

In building the Java implementation, one important security issue arose. Java does not allow for implicit conversion of certain types. For example, Java does not provide a way to cast Booleans to integers and vice versa. Subsequently, due to necessary conditional statements involved in these conversions, a secure implementation could be undermined. An attacker might potentially be alerted to the presence of secret information based on the time it takes to run one condition versus another [11] . Another complicating issue that arose in the implementation of this algorithm is the lack of pointers in Java. This means that one must be vigilant in keeping track of array offsets, which can be time-consuming and may introduce a lot of potential for errors.

There are many opportunities to build on this implementation. To ensure further security, future implementations should attempt to clear all arrays and stored variables at the end of each function. Additionally, implementing the Toom-Cook algorithm for polynomial multiplication, though more complicated than implementing Karatsuba's algorithm, could improve efficiency. This algorithm was not implemented due to time constraints of the project and because in previous Python and C implementations it was shown to only improve performance slightly.

Acknowledgements

This project was based on previously performed work on the NTRU algorithm, specifically that of Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang [2]. The Java implementation was also based on previous C and Python implementations of NTRU, created by Dr. Paulo S. L. M. Barreto. The author would especially like to thank Dr. Barreto for his advice and support throughout this research project.

References

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone and Y.-K. Liu, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," July 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf. [Accessed 2020].

[2] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte and Z. Zhang, "NTRU Algorithm Specifications And Supporting Documentation," 30 March 2019. [Online]. Available: https://ntru.org/f/ntru-20190330.pdf. [Accessed 2020].

[3] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte and Z. Zhang, "NTRU Round 2 Presentation," 24 August 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Presentations/ntru-round-2-presentation/images-media/ntru-schanck.pdf. [Accessed 2020].

[4] J. Hoffstein, J. Pipher and J. H. Silverman, "NTRU: A New High Speed Public Key Cryptosystem," 13 August 1996. [Online]. Available: https://web.securityinnovation.com/hubfs/files/ntru-orig.pdf. [Accessed 2020].

[5] "Crypto '96 Rump Session Presentations," International Association for Cryptologic Research, 1996. [Online]. Available: https://www.iacr.org/conferences/c96/c96rump.html. [Accessed 2000].

[6] R. L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications ACM,* vol. 21, no. 2, 1978.

[7] D. V. Bailey and C. Paar, "Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography," 6 September 2000. [Online]. Available: https://link.springer.com/article/10.1007/s001450010012. [Accessed 2020].

[8] "Quantum Safe Cryptography and Security," European Telecommunications Standards Institute, June 2015. [Online]. Available: https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf. [Accessed 2020].

[9] P. Kirchner and P.-A. Fouque, "Revisiting Lattice Attacks on Overstretched NTRU Parameters," 01 April 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-56620-7_1. [Accessed 2020].

[10] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, R. Angela and D. Smith-Tone, "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process," January 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf. [Accessed 2020].

[11] P. A. Grassi, M. E. Garcia and J. L. Fenton, "Digital Identity Guidelines," June 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf. [Accessed 2020].

[12] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," 1996. [Online]. Available: https://paulkocher.com/doc/TimingAttacks.pdf. [Accessed 2020].

[13] M. Hutle and M. Kammerstetter, "Resilience Against Physical Attacks," 2015. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/side-channel-attack. [Accessed 2020].

[14] D. Moghimi, B. Sunar, T. Eisenbarth and N. Heninger, "TPM-FAIL: TPM meets Timing and Lattice Attacks," August 2020. [Online]. Available: https://www.usenix.org/system/files/sec20-moghimi-tpm.pdf.

[15] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in GF(2m) Using Normal Bases," 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540188900247?via%3Dihub. [Accessed 2020].

[16] R. Schroeppel, H. Orman, S. O'Malley and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," 1995. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-44750-4_4. [Accessed 2020].

[17] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," 09 May 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8298/7648. [Accessed 2020].

[18] A. Weimerskirch and C. Paar, "Generalizations of the Karatsuba Algorithm for Efficient Implementations," January 2006. [Online]. Available: https://www.researchgate.net/publication/220333483_Generalizations_of_the_Karatsuba_Algorithm_for_Efficient_Implementations. [Accessed 2020].

[19] "Secure Coding Guidelines for Java," Oracle, September 2020. [Online]. Available: https://www.oracle.com/java/technologies/javase/seccodeguide.html. [Accessed 2020].

Appendix

Code example 1: Karatsuba multiplication in RS2 Class

Example 1 is taken out of the RS2 Class, which focuses on binary polynomials. This example presents the simplest implementation of the Karatsuba algorithm within this project. Despite its comparative simplicity, it is important to note that one must be sure to calculate and implement the correct offsets in order for the overall cryptosystem to be effective. An incorrect offset in only one calculation would compromise the integrity (and readability) of the input message, causing the overall cryptosystem to become essentially useless.

```
/**
 * Compute u*v in (Z/2)[x] via Karatsuba multiplication.
 *
 * @param k: lg(m) where m is the 2-way Karatsuba block size
 * @param w: u*v
 * @param w_o: offset on w where u*v is to be written
 * @param u: a binary polynomial
 * @param u_o: offset on u
 * @param v: a binary polynomial
 * @param v_o: offset on v
 */
private static void kar(int k, long[] w, int w_o, long[] u, int u_o, long[] v, int v_o) {
    // f := 1 << (k - 6), i.e. full word count
    // 0 <= u < 1 << k, i.e. f 64-bit words
    // 0 <= v < 1 << k, i.e. f 64-bit words
    // 0 <= w < 1 << (k + 1), i.e. 2*f 64-bit words
    // use Karatsuba only when above this threshold (k = 6 --> 2^6 = one 64-bit word)
    if (k > 6) {
        int f = 1 << (k - 6);  // full word count
        int h = f >> 1;  // half word count
        // d0: w from offset w_o
        // d2: w from offset w_o + f
        long[] d1 = new long[f];
        // xor the two halves of u and v
        for (int i = 0; i < h; i++) {
            //d0[i] = u0[i] ^ u1[i];
            w[w_o     + i] = u[u_o + i] ^ u[u_o + h + i];
            //d2[i] = v0[i] ^ v1[i];
            w[w_o + f + i] = v[v_o + i] ^ v[v_o + h + i];
        }
        // (u0 + u1)*(v0 + v1)
        kar(k - 1, d1,      0, w, w_o,     w, w_o + f);
        // u0*v0
        kar(k - 1, w, w_o + 0, u, u_o,     v, v_o);
        // u1*v1
        kar(k - 1, w, w_o + f, u, u_o + h, v, v_o + h);


        for (int i = 0; i < f; i++) {
            // (u0 + u1)*(v0 + v1) - u0*v0 - u1*v1
            d1[i] ^= w[w_o + i] ^ w[w_o + f + i];
        }
        for (int i = 0; i < f; i++) {
            w[w_o + i + h] ^= d1[i];
        }
    } else {
        //If k <= 6, use plain mult
        // only get here when 1 word left
        clmul(w, w_o, u[u_o], v[v_o]);
    }
}
```

Code example 2: Ternary Karatsuba multiplication in RSq Class

This example offers an even deeper look into the challenge of implementing array offsets in Java. The example is taken from the RSq class because this class utilized the most offset-involving operations within the entire project. This method epitomizes the importance of paying vigilant attention to the offsets, as a small error could result in making the entire cryptosystem essentially unusable, as the encryption and decryption will not function properly.

```
/**
 * Compute u*v in (Z/q)[x] by applying an initial 3-way Karatsuba layer, then recursive 2-way Karatsuba layers.
 *
 * @param t: lg(m) where m is the 3-way Karatsuba block size (bitlen(u) <= 3*2^t, bitlen(v) <= 3*2^t)
 * @param w: u*v
 * @param w_o: offset on w where u*v is to be written
 * @param u: a q-ary polynomial
 * @param u_o: offset on w where u*v is to be written
 * @param v: a q-ary polynomial
 * @param v_o: offset on w where u*v is to be written
 */
private static void _kar3(int t, int[/*6*m*/] w, int w_o, int[/*3*m*/] u, int u_o, int[/*3*m*/] v, int v_o) {
    int m = 1 << (t - 6);  // e.g. t = 8, m = 2^(8-6) = 4 64-bit words (256 bits): bitlen(u), bitlen(v) <= 3*256 = 768
    int _2m = m << 1;
    int _3m = _2m + m;
    int _4m = _2m << 1;
    // d0: w from offset w_o
    // d2: w from offset w_o + _4m
    // d4: w from offset w_o + _2m
    int[] d1 = new int[_2m];
    int[] d3 = new int[_2m];
    int[] d5 = new int[_2m];

    // add the 1st and 2nd thirds of u and v
    _add(m, w, w_o + _4m, u, u_o, u, u_o + m);
    _add(m, w, w_o + _4m + m, v, v_o, v, v_o + m);
    // add the 1st and 3rd thirds of u and v
    _add(m, d1, 0, u, u_o, u, u_o + _2m);
    _add(m, d1, m, v, v_o, v, v_o + _2m);
    // add the 2nd and 3rd thirds of u and v
    _add(m, w, w_o, u, u_o + m, u, u_o + _2m);
    _add(m, w, w_o + m, v, v_o + m, u, v_o + _2m);


    _kar(t, d3, 0, w, w_o + _4m, w, w_o + _4m + m);
    _kar(t, w, w_o + _2m, d1, 0,  d1, m);
    _kar(t, d5, 0, w, w_o, w, w_o + m);
    _kar(t, w, w_o, u, u_o, v, v_o);
    _kar(t, d1, 0, u, u_o + m, v, v_o + m);
    _kar(t, w, w_o + _4m, u, u_o + _2m, v, v_o + _2m);

    //      c0 = w[w_o];
    //      c1 = d3[0];
    //      c2 = w[w_o + _2m];
    //      c3 = d5[0];
    //      c4 = w[w_o + _4m];

    // |========4========|  c4 = d2
    //                   |========2========|  c2 = d4
    //                                       |========0========|  c0 = d0
    int[] c1 = new int[d3.length];
    int[] c3 = new int[d5.length];

    for (int i = 0; i < d3.length; i++) {
        c1[i] = d3[i];
    }

    for (int i = 0; i < d5.length; i++) {
        c3[i] = d5[i];
    }

    //          |========3========|  c3 = c3 - d2 - d1  // d5 = d5 - c4 - d1
    //                   |========2========|  c2 = c2 - d2 + d1 - d0  // = c2 - c4 + d1 - c0
    //                            |========1========|  c1 = c1 - d1 - c0  // d3 = d3 - d1 - c0
    _sub(_2m, c1, 0, c1, 0, d1, 0);
    _sub(_2m, c1, 0, c1, 0, w, w_o);
```

(…continued from previous page)

```
    _sub(_2m, w, w_o + _2m, w, w_o + _2m, w, w_o + _4m);
    _add(_2m, w, w_o + _2m, w, w_o + _2m, d1, 0);
    _sub(_2m, w, w_o + _2m, w, w_o + _2m, w, w_o);

    _sub(_2m, c3, 0, c3, 0, w, w_o + _4m);
    _sub(_2m, c3, 0, c3, 0, d1, 0);

    //      |--------3--------|  w34 += c3
    //                          |--------1--------|  w12 += c1
    // |--------|========4========|========2========|--------|  w
    //          5        4        3        2        1        0
    _add(_2m, w, w_o + m, w, w_o + m, c1, 0);
    _add(_2m, w, w_o + _3m, w, w_o + _3m, c3, 0);
}
```