Implementing and Testing Lattice-Based Cryptosystems in Rust

A Project Report

Presented to

Professor Chris Pollett

Department of Computer Science

San Josè State University

In Partial Fulfillment

Of the Requirements for the Class

CS 297

By

Michaela Molina

May 2021

ABSTRACT

In this project I investigated lattice-based cryptography. I first looked into the Rust programming language to learn about the language's features and strengths. I wrote a small program which demonstrates the use of two Rust libraries. I incorporated these libraries by including them as dependencies in the Cargo configuration file. I used this knowledge to implement the md5 hash function. This implementation was tested by hashing 2000 different strings of varying lengths and comparing each hash to the hash produced by the md5 function from a Rust library. Following this I implemented the Impagliazzo-Naor hash function, which is a hash function whose security is based on difficulty of a hard lattice problem. I verified the distribution of this hash function by generating fifteen instances of it. To do this I generated fifteen random matrices and for each matrix I hashed each possible vector of the specified length. I verified that each possible hash value has an equal number of occurrences. Finally I implemented polynomial arithmetic in a ring. I did this by implementing arithmetic, multiplication, and division combined with modular arithmetic. I used these operations to implement the Extended Euclidean algorithm which finds the polynomial inverse. These operations are used for key generation in the NTRU system. This project report follows my investigation into lattice-based cryptography and shows the tests that were run on each component implemented.

Index terms--**Lattice-based cryptography, quantum computing, hash functions, worse-to-average case reduction**

I. INTRODUCTION

Lattice-based cryptography has implications for the future in that it is believed to be quantum-resistant. This means that where other cryptographic schemes whose security holds today will fail against the attack of a quantum computer, a system whose security rests on a hard lattice problem will remain uncompromised. This is because there is no known polynomial time algorithm which can solve these problems [1]. In this project I investigated cryptosystems based on these problems and implemented components of them. Before implementing these components I developed proficiency in the Rust programming language. To do this I learned about the features and strengths of Rust and wrote a small program which demonstrated the use of the Cargo projecy manager and crates. I used this knowledge to help me implement the md5 hash function using Rust. By doing this I expanded my knowledge further and developed my understanding of language constructs such as mutability, slices, moving, borrowing, modules, and type conversions. After this I implemented a hash function based on a hard lattice problem, the Impagliazzo-Naor hash function. This implementation included generating random matrices and matrix multiplication. To test the implementation I generated fifteen different instances of the hash function by generating fifteen random matrices, and I hashed each possible input to the matrix. I showed the output by for each number of occurrences printing out the list of hash values that occurred that many times. Finally I implemented polynomial arithmetic in a ring. This arithmetic is used in the NTRU public-key system to generate keys. I did this by implementing polynomial addition, multiplication, and division and taking the remainder modulo NTRU's polynomial. I then used these operations to implement the Extended Euclidean algorithm in order to find the polynomial inverse. These implementations were tested against other calculators.

The rest of the project report proceeds as follows. First I discuss my initial steps to get familiar with Rust. Then I discuss how I implemented the md5 hash function. Following this I describe how I implemented the Imagliazzo-Naor hash function, and finally I describe how polynomial arithmetic is used in the NTRU system and how I implemented the arithmetic operations.

II. GETTING FAMILIAR WITH RUST

My first goal was to get started with the Rust programming language. To do this I investigated Rust's strengths and features and wrote a short program which uses Rust libraries. In my small program I make use of the `rand` library and the `abbreviator` library, with results shown below.

```
    Updating crates.io index
  Downloaded libc v0.2.94
  Downloaded abbreviator v0.1.4
  Downloaded 2 crates (513.7 KB) in 1.23s
   Compiling libc v0.2.94
   Compiling rand_core v0.4.2
   Compiling abbreviator v0.1.4
   Compiling rand_core v0.3.1
   Compiling rand v0.5.6
   Compiling intro v0.1.0 (/mnt/c/Users/Michaela/Documents/SJSU/Spring2021/Del1/intro)
    Finished dev [unoptimized + debuginfo] target(s) in 10.30s
michaela@DESKTOP-IIVM95D:/mnt/c/Users/Michaela/Documents/SJSU/Spring2021/Del1/intro/src$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `/mnt/c/Users/Michaela/Documents/SJSU/Spring2021/Del1/intro/target/debug/intro`


Use of abbreviate() from abbreviator crate:
Abbreviation of "hello" is h3o
Abbreviation of "san jose state university" is s23y
Abbreviation of "This is a sentence." is T17.

Use of thread_rng() from rand crate:
The random number is: 55
```

Doing this initial research on Rust prepared me for completing more involved programs in the next deliverables.

III. IMPLEMENTING THE MD5 HASH FUNCTION

Md5 was invented in 1992 and appears in RFC 1321 [2]. It became insecure in 2004 when it was shown how to find collisions efficiently in less than an hour [3]. Despite this, implementing this hash function provided a good opportunity to expand my proficiency with the Rust programming language.

In the code I converted an arbitrary length string of ascii characters into a binary string. Each character was converted into its ascii code and had a zero appended to the beginning to form a byte. These bytes made up the message string. The string was padded so that its length was congruent to 448 modulo 512. The string was then put into a vector of 32-bit words, where each group of 4 bytes were reversed so that the byte appearing last in the group appeared first. The remaining 64 bits were filled with a binary

representation of the length of the original message, where the lower-order 32-bit word was appended first. The message was processed in 16-word blocks. Four registers held initial values, and four different rounds of sixteen operations were performed on each 16-word block. These operations used auxiliary functions, addition, multiplication, and bitshifting to modify the contents of one register at each step. For the four registers which held the hash values at each step, I used four `u32` data types. I passed in mutable references to these registers to the four round functions which altered their values as the algorithm specifies. By storing each register value as a `u32` I utilized Rust's built-in math operations to do the operations of the algorithm with the exception of a circular left bitshift which I implemented by converting the `u32` to a binary string, shifting it, and converting it back to a `u32`. At the end of the rounds, the original content of the registers was appended to the resulting content of the registers, and the values of the registers represented the value of the hash, starting with the low-order word. The final value was 128 bits.

Random strings were generated using the Rust library `random_string`. This library allowed a random string of a chosen length to be generated over a specific character set. Here I used a set of 94 ASCII characters as the character set. For each of the lengths of 50, 100, 300, and 500 I generated 500 random strings and computed their hash value using the implemented md5 function. I also computed their hash value using Rust's built-in md5 function. I compared these two values, and used a hashmap to record the results, where the keys were `true` and `false` and the values were the number of occurrences of that key. A printout of this hashmap is shown below.

```
    Finished dev [unoptimized + debuginfo] target(s) in 1.45s
     Running `/mnt/c/Users/Michaela/Documents/SJSU/Spring2021/CS_297/Del2/md5/target/debug/md5`
true => 2000
```

The implemented hash function was shown to be correct on 2000 different strings.

IV. IMPLEMENTING THE IMPAGLIAZZO-NAOR HASH FUNCTION

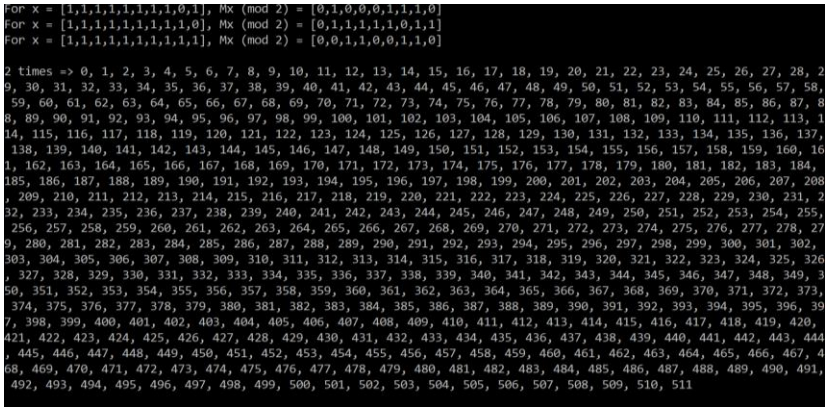
In [4] a hash function is described which is supposed to be as difficult to invert as a worst-case lattice problem. I implemented this hash function using Rust and ran experiments to test the randomness of the hash values. A tool to generate a random matrix was written which takes from the command line the variable *m* which is the number of vectors in the matrix, and the variable *c* which sets the dimension of the vectors as the floor of $lm = (1-c)m$. The vectors were represented as column vectors, while the dimension represented the matrix height. Internally, these matrices were represented as two-dimensional `Vec`s where each vector in the matrix was a `Vec` contained in a `Vec` of other vectors. To generate the random vectors, Rust's library `rand_chacha` was loaded and a random number generator from this library, `ChaCha20Rng`, was used. I initialized this random number generator with the line `let mut rng = ChaCha20Rng::from_entropy()`. This generator used a fresh seed from the operating system.

The code generated *m* `Vec`s of length *lm* one at a time by pushing a random number in the range [0..2) to each `Vec` *lm* times and pushing the resulting `Vec` to the containing `Vec`. This random matrix generator was included as one of two main programs in the Cargo's project directory and was used as `cargo run --bin generate_matrix m c > m.txt` where *m* is the number of vectors or the width of the random matrix, *c* is the constant used to calculate the dimension of the vectors, and *m.txt* is the file which contains a text representation of the random matrix. Another main program used this random matrix as `cargo run --bin hash m.txt` where *m.txt* is the file containing the representation of the random matrix. This program opened this file, read it line by line, and created its own copy of the matrix. Each line of the file represented a column of the random matrix. Once the random matrix was obtained, the code checked the number of vectors in the matrix, which will be equal to *m* or the `len()` of the outer Vec, and generated all vectors of this length with elements from the set {0,1}. The number of vectors generated was equal to two to the power of *m*. The code generated the vectors by counting in decimal from zero to this number minus one and using a method to convert each number first to a binary string representation and then to a `Vec` of zeros and ones.

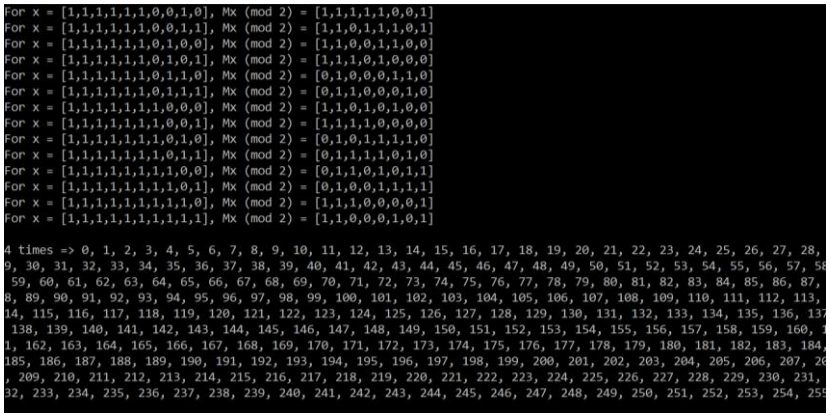
Finally, the code iterated through all of these vectors and used a matrix multiplication method to multiply the random matrix by the vector. By the rules of matrix multiplication, the output of each multiplication should be a `Vec` of the same length as the input value. The code created two data structures as it calculated the hash of each vector. One structure was an array of size two to the power of *lm*, or the length of the output vector. Since each hash value consisted of only ones and zeros and was of length *lm*, each value corresponded to a decimal number in this array's index range. The code converted each hash value to a decimal number and recorded the number of occurrences of that value in the corresponding index. The second data structure created was a hash table where a key was a number that represents a count of occurrences of a hash value. The value of this key was a list of hash values that occurred the number of times represented by the key. This allowed a one-screen visual of the distribution of hash values.

If the hash function was effective, each possible hash value should have about the same number of occurrences. To test this, I generated five random matrices, consisting of ten, eight, six, and three vectors. For each of these matrices I ran the hash program with *c* at .1, .3, and .8. The results are shown below.

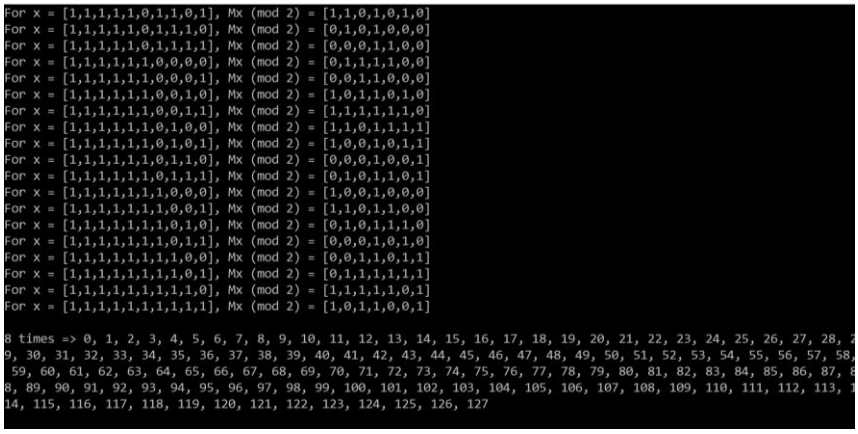
m = 10, c = .1



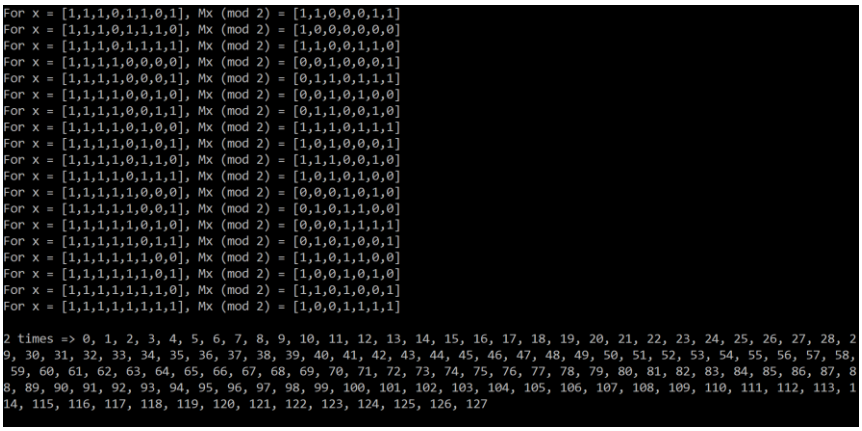
m = 10, c = .2



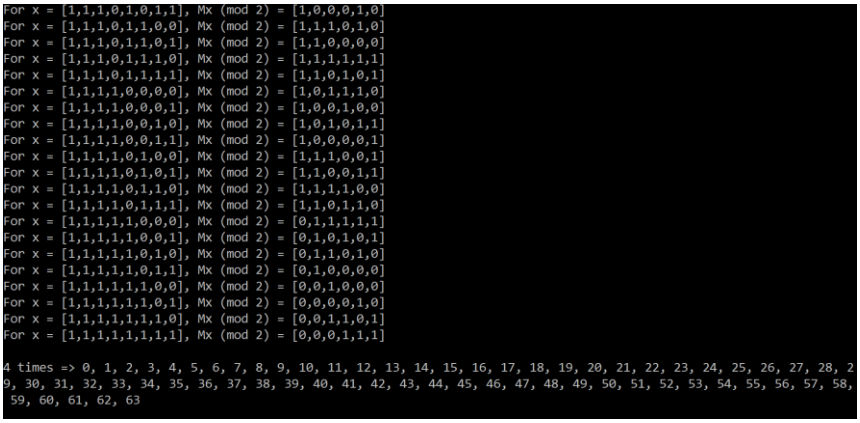
m = 10, c = .3



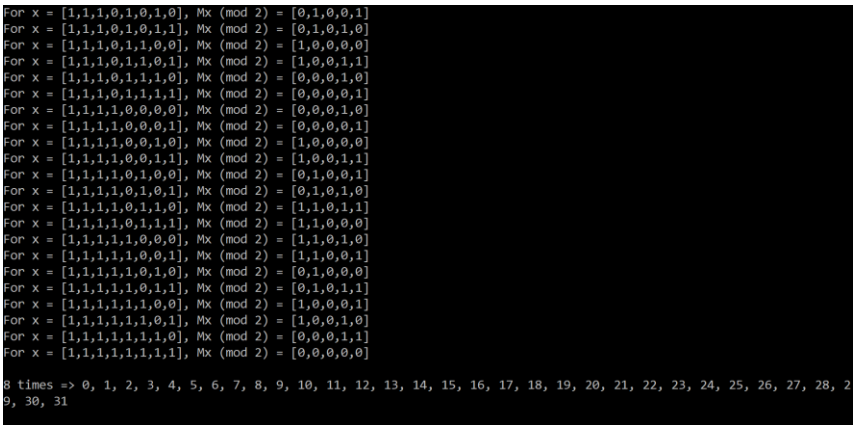
m = 8, c = .1



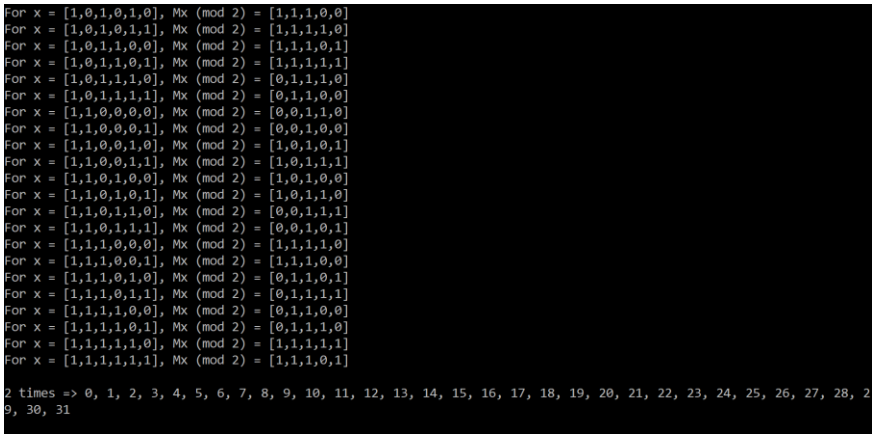
m = 8, c = .2



m = 8, c = .3



m = 6, c = .1



m = 6, c = .2

```
For x = [1,0,1,0,0,1], Mx (mod 2) = [1,0,0,0]
For x = [1,0,1,0,1,0], Mx (mod 2) = [1,0,1,0]
For x = [1,0,1,0,1,1], Mx (mod 2) = [1,1,1,1]
For x = [1,0,1,1,0,0], Mx (mod 2) = [0,1,0,0]
For x = [1,0,1,1,0,1], Mx (mod 2) = [0,0,0,1]
For x = [1,0,1,1,1,0], Mx (mod 2) = [0,0,1,1]
For x = [1,0,1,1,1,1], Mx (mod 2) = [0,1,1,0]
For x = [1,1,0,0,0,0], Mx (mod 2) = [0,0,1,0]
For x = [1,1,0,0,0,1], Mx (mod 2) = [0,1,1,1]
For x = [1,1,0,0,1,0], Mx (mod 2) = [0,1,0,1]
For x = [1,1,0,0,1,1], Mx (mod 2) = [0,0,0,0]
For x = [1,1,0,1,0,0], Mx (mod 2) = [1,0,1,1]
For x = [1,1,0,1,0,1], Mx (mod 2) = [1,1,1,0]
For x = [1,1,0,1,1,0], Mx (mod 2) = [1,1,0,0]
For x = [1,1,0,1,1,1], Mx (mod 2) = [1,0,0,1]
For x = [1,1,1,0,0,0], Mx (mod 2) = [1,1,0,1]
For x = [1,1,1,0,0,1], Mx (mod 2) = [1,0,0,0]
For x = [1,1,1,0,1,0], Mx (mod 2) = [1,0,1,0]
For x = [1,1,1,0,1,1], Mx (mod 2) = [1,1,1,1]
For x = [1,1,1,1,0,0], Mx (mod 2) = [0,1,0,0]
For x = [1,1,1,1,0,1], Mx (mod 2) = [0,0,0,1]
For x = [1,1,1,1,1,0], Mx (mod 2) = [0,0,1,1]
For x = [1,1,1,1,1,1], Mx (mod 2) = [0,1,1,0]

4 times => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

m = 6, c = .3

```
For x = [1,0,1,0,0,1], Mx (mod 2) = [0,1,0,1]
For x = [1,0,1,0,1,0], Mx (mod 2) = [1,0,0,0]
For x = [1,0,1,0,1,1], Mx (mod 2) = [0,1,0,0]
For x = [1,0,1,1,0,0], Mx (mod 2) = [0,0,1,1]
For x = [1,0,1,1,0,1], Mx (mod 2) = [1,1,1,1]
For x = [1,0,1,1,1,0], Mx (mod 2) = [0,0,1,0]
For x = [1,0,1,1,1,1], Mx (mod 2) = [1,1,1,0]
For x = [1,1,0,0,0,0], Mx (mod 2) = [0,1,1,1]
For x = [1,1,0,0,0,1], Mx (mod 2) = [1,0,1,1]
For x = [1,1,0,0,1,0], Mx (mod 2) = [0,1,1,0]
For x = [1,1,0,0,1,1], Mx (mod 2) = [1,0,1,0]
For x = [1,1,0,1,0,0], Mx (mod 2) = [1,1,0,1]
For x = [1,1,0,1,0,1], Mx (mod 2) = [0,0,0,1]
For x = [1,1,0,1,1,0], Mx (mod 2) = [1,1,0,0]
For x = [1,1,0,1,1,1], Mx (mod 2) = [0,0,0,0]
For x = [1,1,1,0,0,0], Mx (mod 2) = [0,1,0,0]
For x = [1,1,1,0,0,1], Mx (mod 2) = [1,0,0,0]
For x = [1,1,1,0,1,0], Mx (mod 2) = [0,1,0,1]
For x = [1,1,1,0,1,1], Mx (mod 2) = [1,0,0,1]
For x = [1,1,1,1,0,0], Mx (mod 2) = [1,1,1,0]
For x = [1,1,1,1,0,1], Mx (mod 2) = [0,0,1,0]
For x = [1,1,1,1,1,0], Mx (mod 2) = [1,1,1,1]
For x = [1,1,1,1,1,1], Mx (mod 2) = [0,0,1,1]

4 times => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

m = 3, c = .1

```
M:
a0 a1 a2
-- -- --
0  0  1
0  1  1

For x = [0,0,0], Mx (mod 2) = [0,0]
For x = [0,0,1], Mx (mod 2) = [1,1]
For x = [0,1,0], Mx (mod 2) = [0,1]
For x = [0,1,1], Mx (mod 2) = [1,0]
For x = [1,0,0], Mx (mod 2) = [0,0]
For x = [1,0,1], Mx (mod 2) = [1,1]
For x = [1,1,0], Mx (mod 2) = [0,1]
For x = [1,1,1], Mx (mod 2) = [1,0]

2 times => 0, 1, 2, 3
```

m = 3, c = .2

```
M:
a0 a1 a2
-- -- --
1  1  0
1  0  0

For x = [0,0,0], Mx (mod 2) = [0,0]
For x = [0,0,1], Mx (mod 2) = [0,0]
For x = [0,1,0], Mx (mod 2) = [1,0]
For x = [0,1,1], Mx (mod 2) = [1,0]
For x = [1,0,0], Mx (mod 2) = [1,1]
For x = [1,0,1], Mx (mod 2) = [1,1]
For x = [1,1,0], Mx (mod 2) = [0,1]
For x = [1,1,1], Mx (mod 2) = [0,1]

2 times => 0, 1, 2, 3
```

m = 3, c = .3

```
M:
a0 a1 a2
-- -- --
1  1  0
1  0  0

For x = [0,0,0], Mx (mod 2) = [0,0]
For x = [0,0,1], Mx (mod 2) = [0,0]
For x = [0,1,0], Mx (mod 2) = [1,0]
For x = [0,1,1], Mx (mod 2) = [1,0]
For x = [1,0,0], Mx (mod 2) = [1,1]
For x = [1,0,1], Mx (mod 2) = [1,1]
For x = [1,1,0], Mx (mod 2) = [0,1]
For x = [1,1,1], Mx (mod 2) = [0,1]

2 times => 0, 1, 2, 3
```

The results show that each possible output of the hash function is hash value an equal number of times.

V. IMPLEMENTING ARITHMETIC FOR POLYNOMIALS IN A RING

The NTRU public key system is a practical system whose security depends on a lattice problem. It features short keys, easy key generation, and efficient encryption and decryption [5]. To generate a public key for this system, finding a polynomial inverse in necessary. I implemented a function to find the polynomial inverse by implementing polynomial arithmetic in a ring.

Polynomial arithmetic in this ring was implemented using Rust. Polynomials were represented as `Vec`s of integers, where the first integer represented the coefficient with the highest degree. First a method was implemented which added coefficients of polynomials from low index to high index and returned a vector of these sums, also from low index to high index. Polynomial addition was implemented by reversing two polynomial vectors so that their low order coefficients were in low order indices, calling the previous method to add coefficients, and then reversing the result. To assist in later methods, a method which added many polynomial vectors at the same time `polynomial_add_multiple` was implemented using this basic polynomial addition method. To get a better visual representation of results, a function `get_polynomial_string` was implemented which returned a string representation of a polynomial, including plus signs, coefficients, and degrees. Polynomial multiplication was implemented following the methods of polynomial multiplication by hand. First I found which polynomial was the shortest. Then, for each coefficient in the shortest vector, I multiplied it by each coefficient in the longest

vector. I ended up with a two dimensional vector of coefficients and called `polynomial_add_multiple` to get the final result. Similarly, a polynomial multiplication method which reduced the coefficients modulo some integer used this method.

Polynomial division was implemented in the same way as polynomial long division by hand with the only difference in the coefficients. For use in finding a polynomial inverse modulo an integer q, described below, the division algorithm used coefficients in the range [0, q). At each step, coefficients were reduced modulo this number. To find the coefficient of some leading quotient, I found an integer so that when it was multiplied with the leading coefficient of the divisor it formed a coefficient which when taken modulo q was equal to the leading coefficient of the dividend. If the integer does not exist the division could not continue. Outside of this step division proceeded like long division. I found the quotient degree and the quotient coefficient and multiplied this polynomial by the divisor. I subtracted from the dividend and checked to see if the remainder had degree greater or equal to that of the divisor. If it did, the algorithm stopped. Otherwise, it continued by adding another term to the quotient.

A method to find the inverse of a polynomial in this ring modulo some integer was implemented using the Extended Euclidean Algorithm [6]. This method is important because key generation depends on finding these inverses. In the ring defined by NTRU, finding the inverse of some polynomial modulo some integer p means finding a second polynomial so that if it is multiplied by the first polynomial, reduced modulo the polynomial of degree N defined by NTRU, and reduced modulo the integer p, the result should be equal to one. If ax is the N^{th} degree polynomial, and bx is the polynomial I want to find the inverse of, step one was to divide ax by bx and produce the remainder r_1x. Step two was to divide bx by r_1x and produce the remainder r_2x. Step three was to divide r_1x by r_2x and produce remainder r_3x. These steps were continued until the remainder was a constant. If this remainder was co-prime to the integer p then an inverse to the polynomial existed. If the two numbers were not co-prime or the remainder never reaches a constant then there was no inverse. At each of the division steps, I recorded the quotient and remainder in an array of quotients and an array of remainders. The index of each quotient

and remainder corresponded to the step where was produced. After each array is filled, I created two new arrays, called `v` and `w`. I initialized the first two elements of `v` to 0 and 1, and the first two elements of `w` to 1 and $-q1$. Then, I filled the remaining elements of `v` and `w` with the following rules: $v_i = v_{i-2} - q_i \cdot v_{i-1}$ and $w_i = w_{i-2} - q_i \cdot w_{i-1}$. At each step i the remainder r_i was equal to $ax \cdot v_i + bx \cdot w_i$. Once all the indices were filled, the last element of `w`, corresponding to the last constant remainder, contained the inverse of the polynomial which still must be multiplied by a number if the first constant remainder was not 1 [7].

To test my results I generated the a polynomial inverse with five different parameter sets of the type used in NTRU. Specifically a user must first find the inverse of some polynomial containing coefficients from the set {-1,0,1} and where in these coefficients there appears one more positive ones than negative ones, and any number of zeros [13]. The results are shown below.

N = 11, p = 7, f = [-1,1,0,0,1,0,-1,0,1,1,-1]

```
v_7 = 3x^6 +0x^5 +0x^4 +4x^3 +0x^2 +2x^1 +6
w_7 = 3x^7 +3x^6 +3x^5 +0x^4 +3x^3 +1x^2 +0x^1 +4
q_7 = 4x^1 +0
r1_7 = 1x^3 +3x^2 +2x^1 +4
r2_7 = 1x^3 +3x^2 +2x^1 +4

v_8 = 1x^7 +3x^6 +1x^5 +6x^4 +3x^3 +5x^2 +0x^1 +6
w_8 = 1x^8 +4x^7 +5x^6 +4x^5 +1x^4 +3x^3 +0x^2 +6x^1 +3
q_8 = 2x^1 +6
r1_8 = 4x^2 +4x^1 +5
r2_8 = 4x^2 +4x^1 +5

v_9 = 5x^8 +4x^7 +3x^6 +5x^5 +5x^4 +3x^3 +1x^2 +4x^1 +3
w_9 = 5x^9 +2x^8 +5x^7 +3x^6 +6x^5 +4x^4 +5x^3 +3x^2 +5x^1 +6
q_9 = 2x^1 +4
r1_9 = 4x^1 +5
r2_9 = 4x^1 +5

v_10 = 2x^9 +6x^8 +6x^7 +4x^6 +6x^5 +6x^4 +1x^3 +3x^2 +5x^1 +5
w_10 = 2x^10 +1x^9 +0x^8 +4x^7 +5x^6 +5x^5 +4x^4 +3x^3 +1x^2 +3x^1 +1
q_10 = 1x^1 +5
r1_10 = 1
r2_10 = 1

inverse1 = 2x^10 +1x^9 +0x^8 +4x^7 +5x^6 +5x^5 +4x^4 +3x^3 +1x^2 +3x^1 +1
```

N = 11, p = 97, f = [-1,1,0,0,1,0,-1,0,1,1,-1]

```
v_7 = 7x^6 +0x^5 +0x^4 +4x^3 +0x^2 +5x^1 +3
w_7 = 7x^7 +7x^6 +7x^5 +0x^4 +7x^3 +8x^2 +0x^1 +4
q_7 = 10x^1 +0
r1_7 = 8x^3 +7x^2 +10x^1 +4
r2_7 = 8x^3 +7x^2 +10x^1 +4

v_8 = 7x^7 +2x^6 +7x^5 +4x^4 +2x^3 +8x^2 +0x^1 +4
w_8 = 7x^8 +9x^7 +5x^6 +9x^5 +7x^4 +2x^3 +0x^2 +4x^1 +2
q_8 = 10x^1 +6
r1_8 = 9x^2 +9x^1 +5
r2_8 = 9x^2 +9x^1 +5

v_9 = 6x^8 +10x^7 +1x^6 +7x^5 +6x^4 +2x^3 +7x^2 +10x^1 +1
w_9 = 6x^9 +5x^8 +6x^7 +2x^6 +3x^5 +10x^4 +6x^3 +2x^2 +6x^1 +3
q_9 = 7x^1 +6
r1_9 = 9x^1 +7
r2_9 = 9x^1 +7

v_10 = 5x^9 +7x^8 +5x^7 +7x^6 +8x^5 +8x^4 +8x^3 +5x^2 +9x^1 +5
w_10 = 5x^10 +1x^9 +6x^8 +2x^7 +4x^6 +2x^5 +0x^4 +6x^3 +7x^2 +7x^1 +5
q_10 = 1x^1 +10
r1_10 = 1
r2_10 = 1

inverse2 = 5x^10 +1x^9 +6x^8 +2x^7 +4x^6 +2x^5 +0x^4 +6x^3 +7x^2 +7x^1 +5
```

N = 7, p = 37, f = [1,0,-1,-1,1,0,1,0]

```
v_4 = 1x^5 +35x^4 +2x^3 +36x^2 +36x^1 +1
w_4 = 0x^5 +0x^4 +0x^3 +1x^2 +36x^1 +1
q_4 = 1x^1 +36
r1_4 = 2x^4 +35x^3 +0x^2 +3x^1 +35
r2_4 = 2x^4 +35x^3 +0x^2 +3x^1 +35

v_5 = 18x^6 +0x^5 +0x^4 +18x^3 +20x^2 +19x^1 +0
w_5 = 0x^6 +0x^5 +0x^4 +18x^3 +18x^2 +18x^1 +36
q_5 = 19x^1 +1
r1_5 = 16x^2 +36x^1 +1
r2_5 = 16x^2 +36x^1 +1

v_6 = 7x^8 +5x^7 +23x^6 +8x^5 +19x^4 +3x^3 +36x^2 +13x^1 +1
w_6 = 0x^8 +0x^7 +0x^6 +7x^5 +12x^4 +35x^3 +6x^2 +32x^1 +10
q_6 = 14x^2 +10x^1 +9
r1_6 = 2x^1 +26
r2_6 = 2x^1 +26

v_7 = 18x^9 +7x^8 +24x^7 +8x^6 +18x^5 +19x^4 +25x^3 +15x^2 +19x^1 +12
w_7 = 0x^9 +0x^8 +0x^7 +18x^6 +25x^5 +12x^4 +20x^3 +19x^2 +26x^1 +8
q_7 = 8x^1 +25
r1_7 = 17
r2_7 = 17

inverse3 = 25x^6 +8x^5 +29x^4 +36x^3 +12x^2 +32x^1 +7
```

N = 12, p = 73, f = [-1,1,-1,0,0,1,0,-1,1,0,1]

```
v_4 = 29x^6 +57x^5 +37x^4 +9x^3 +23x^2 +67x^1 +25
w_4 = 29x^7 +13x^6 +21x^5 +17x^4 +19x^3 +25x^2 +44x^1 +11
q_4 = 15x^1 +3
r1_4 = 5x^3 +36x^2 +35x^1 +37
r2_4 = 5x^3 +36x^2 +35x^1 +37

v_5 = 29x^7 +25x^6 +69x^5 +34x^4 +31x^3 +47x^2 +37x^1 +49
w_5 = 29x^8 +54x^7 +21x^6 +1x^5 +11x^4 +13x^3 +20x^2 +48x^1 +4
q_5 = 72x^1 +59
r1_5 = 10x^2 +61x^1 +20
r2_5 = 10x^2 +61x^1 +20

v_6 = 22x^8 +19x^7 +72x^6 +13x^5 +32x^4 +67x^3 +48x^2 +55x^1 +9
w_6 = 22x^9 +41x^8 +18x^7 +63x^6 +4x^5 +30x^4 +42x^3 +63x^2 +1x^1 +38
q_6 = 37x^1 +48
r1_6 = 17x^1 +26
r2_6 = 17x^1 +26

v_7 = 30x^9 +10x^8 +51x^7 +16x^6 +42x^5 +25x^4 +29x^3 +71x^2 +46x^1 +44
w_7 = 30x^10 +40x^9 +61x^8 +37x^7 +18x^6 +36x^5 +14x^4 +7x^3 +66x^2 +27x^1 +64
q_7 = 65x^1 +33
r1_7 = 38
r2_7 = 38

inverse4 = 20x^10 +51x^9 +65x^8 +49x^7 +12x^6 +24x^5 +58x^4 +29x^3 +44x^2 +18x^1 +67
```

N = 15, p = 11, f = [1,1,1,1,-1-1-1-1,1,0,0,0,0];

```
v_7 = 10x^7 +2x^6 +9x^5 +0x^4 +7x^3 +0x^2 +8x^1 +8
w_7 = 1x^9 +8x^8 +4x^7 +9x^6 +6x^5 +1x^4 +0x^3 +7x^2 +7x^1 +9
q_7 = 3x^1 +8
r1_7 = 9x^4 +4x^3 +0x^2 +3x^1 +3
r2_7 = 9x^4 +4x^3 +0x^2 +3x^1 +3

v_8 = 6x^8 +7x^7 +0x^6 +1x^5 +9x^4 +7x^3 +9x^2 +6x^1 +3
w_8 = 5x^10 +10x^9 +7x^8 +10x^7 +2x^6 +0x^5 +1x^4 +1x^3 +8x^2 +10x^1 +9
q_8 = 6x^1 +8
r1_8 = 4x^3 +2x^2 +5x^1 +8
r2_8 = 4x^3 +2x^2 +5x^1 +8

v_9 = 3x^9 +7x^8 +4x^7 +8x^6 +4x^5 +6x^4 +0x^3 +0x^2 +2x^1 +7
w_9 = 8x^11 +7x^10 +3x^9 +7x^8 +9x^7 +1x^6 +1x^5 +3x^4 +0x^3 +2x^2 +10x^1 +6
q_9 = 5x^1 +4
r1_9 = 9x^1 +4
r2_9 = 9x^1 +4

v_10 = 6x^11 +7x^10 +9x^9 +3x^8 +6x^7 +0x^6 +4x^5 +7x^4 +0x^3 +0x^2 +0x^1 +8
w_10 = 5x^13 +10x^12 +9x^11 +6x^10 +7x^9 +4x^8 +3x^7 +9x^6 +0x^5 +4x^4 +9x^3 +7x^2 +0x^1 +7
q_10 = 9x^2 +6x^1 +4
r1_10 = 3
r2_10 = 3

inverse5 = 9x^13 +7x^12 +3x^11 +2x^10 +6x^9 +5x^8 +1x^7 +3x^6 +0x^5 +5x^4 +3x^3 +6x^2 +0x^1 +6
```

The correctness of these results is checked with [8] and [9].

Finally, I show below the results of one iteration of generating a public key. This result is checked with [8]. Since this step was not the focus of this deliverable, I only complete one example. The public key is *h* at the bottom of the screen.

```
q_7 = 32x^1 +0
r1_7 = 8x^3 +93x^2 +85x^1 +4
r2_7 = 8x^3 +93x^2 +85x^1 +4

v_8 = 85x^7 +6x^6 +85x^5 +12x^4 +6x^3 +24x^2 +0x^1 +12
w_8 = 85x^8 +91x^7 +79x^6 +91x^5 +85x^4 +6x^3 +0x^2 +12x^1 +6
q_8 = 94x^1 +50
r1_8 = 91x^2 +91x^1 +79
r2_8 = 91x^2 +91x^1 +79

v_9 = 81x^8 +32x^7 +65x^6 +40x^5 +81x^4 +24x^3 +49x^2 +32x^1 +65
w_9 = 81x^9 +16x^8 +81x^7 +24x^6 +89x^5 +32x^4 +81x^3 +24x^2 +81x^1 +89
q_9 = 31x^1 +2
r1_9 = 73x^1 +40
r2_9 = 73x^1 +40

v_10 = 4x^9 +35x^8 +7x^7 +82x^6 +30x^5 +49x^4 +2x^3 +48x^2 +19x^1 +1
w_10 = 4x^10 +39x^9 +46x^8 +31x^7 +65x^6 +56x^5 +3x^4 +43x^3 +85x^2 +57x^1 +76
q_10 = 73x^1 +33
r1_10 = 20
r2_10 = 20

inverse1 = 2x^10 +1x^9 +0x^8 +4x^7 +5x^6 +5x^5 +4x^4 +3x^3 +1x^2 +3x^1 +1
inverse2 = 39x^10 +65x^9 +12x^8 +84x^7 +76x^6 +61x^5 +5x^4 +7x^3 +77x^2 +95x^1 +62
h = 76x^10 +63x^9 +5x^8 +8x^7 +76x^6 +94x^5 +61x^4 +37x^3 +65x^2 +11x^1 +86
```

These examples demonstrate my implementation of polynomial arithmetic in the NTRU ring.

VI. CONCLUSION

This semester I started out with little knowledge of Rust or lattice-based cryptography. However, by reading publications on this topic I developed my awareness and I used these publications to prompt investigation of topics unfamiliar to me. I explored topics such as worse-to-average case reductions, hard lattice problems, the history of research in the field, and linear algebra concepts. I asked questions in order to clarify these topics. Furthermore, I gained proficiency in programming in Rust by implementing components of three systems. Next semester, I hope to continue expanding my understanding of lattice-based cryptography by learning about the LWE and McEliece problems and implementing related components. Lattice-based cryptography is promising for post-quantum information security. Further investigation into this topic will be both practical and engaging.

REFERENCES

[1] J. Hoffstein, J. Pipher, and J.H. Silverman. An Introduction to Mathematical Cryptography, Undergraduate Texts in Mathematics. 2nd Ed. New York, NY: Springer New York, 2014, ch. 7, pp. 373.

[2] R. Rivest. "RFC1321: The MD5 message-digest algorithm." (1992).

[3] Wang, Xiaoyun, and Y. Hongbo. "How to break MD5 and other hash functions." In Annual international conference on the theory and applications of cryptographic techniques, pp. 19-35. Springer, Berlin, Heidelberg, 2005.

[4] C. Dwork. "Positive applications of lattices to cryptography." In International Symposium on Mathematical Foundations of Computer Science, pp. 44-51. Springer, Berlin, Heidelberg, 1997.

[5] J. Hoffstein, J. Pipher, and J.H. Silverman. "NTRU: A ring-based public key cryptosystem." In International Algorithmic Number Theory Symposium, pp. 267-288. Springer, Berlin, Heidelberg, 1998.

[6] W. Stallings. Cryptography and Network Security: Principles and Practice. Sixth ed. Boston: Pearson, 2014.

[7] Q. Ma. "The LTV Homomorphic Encryption Scheme and Implementation in Sage," B.S. thesis, Worcester Polytechnic Institute, April 25, 2013. Available: https://web.wpi.edu/Pubs/E-project/Available/E-project-042613-101713/unrestricted/MQP_Report.pdf

[8] Asecuritysite.com. 2021. Lattice Encryption: NTRU Key Generation. [online] Available at: <https://asecuritysite.com/encryption/ntru_key> [Accessed 11 May 2021].

[9] Mathsci2.appstate.edu. 2021. The Extended Euclidean Algorithm. [online] Available at: <https://mathsci2.appstate.edu/~cookwj/sage/algebra/Euclidean_algorithm-poly.html> [Accessed 11 May 2021].