POQR: Post-Onion Quantum-Routing

(almost) a basic post-quantum safe onion-routing setup

Alex Khosrowshahi
alexander_khosrowshahi@brown.edu
akhosrow

Tanish Makadia
tanish_makadia@brown.edu
tmakadia

Abstract

Quantum computers are coming, whether we like it or not. Governments are storing vast amounts of general internet traffic for retroactive decryption in anticipation of having these machines perfected (breaking current encryption schemes like RSA, AES, ECDH, etc.). If we are to retain our anonymity and privacy, then we must bolster the security of existing onion-routing protocols and utilize post-quantum encryption schemes. In this project, we sought to implement our own scaled-down version of The Onion Router (TOR) protocol using an original implementation of the NTRU Lattice-based cryptosystem. Though we ultimately did not succeed in finishing the onion-routing aspect of our project within the given timeframe, we succeeded in creating the first ever pure Rust NTRU implementation and learned a significant amount about network security, cryptography, and TOR's design.

1 Preface/Demo Video

We tried our best to get a full onion-routing implementation working on top of the NTRU cryptosystem, but ran into issues with being able to send a full encrypted public key. After this realization, we attempted to implement onion-routing with RSA that is then encrypted with NTRU identity keys, but ran out of time before we got it all working. All our work in is the GitHub repo, so you can see the stubs of what could have blossomed into a beautiful onion.

In its current state, though, our project features the first ever pure Rust implementation of the NTRU cryptosystem, among relatively few implementations in the first place. Given the goal of the final project to learn and explore a new topic not covered in depth within the course, we feel extremely successful in interrogating the topic of network security, encryption, and how new schemes are designed.

This is explained later on in more detail, but for the time being we can demonstrate our test suite for NTRU passing and also being able to send messages naively over a single channel following an asymmetric key handshake. Here is the Google Drive link. Thank you again for a wonderful semester; we both learned so much from *TCP* and *IP* :).

2 Onion-Routing

In this section, we will discuss what we *intended* to implement (and what we will likely try to implement after the due date) with regards to onion-routing.

2.1 Hosts and Relays

Our virtual network (all of which exists on localhost) consists of two types of nodes: Hosts and Relays. A Host is what a user interacts with; it has a user-facing REPL that can be used to send commands to other hosts (or servers in the real-world). In order to actually send data to the final destination, though, we must first create a pathway of intermediate hops that our messages take in an attempt to mask the true origin source and final destinations of packets.

2.2 Circuits

In our implementation, this pathway is called a *Circuit*. A circuit contains three intermediate hops/relays. Hosts are fully aware of their intended circuit path before circuit creation, picking three relays at random through the network. During circuit creation, a host sends a message using a special packet type (called an Onion packet in our implementation). Onion packets have a header telling the relay that receives them what to do with them, but are otherwise fairly simple. We'll describe how our circuits use these packets and get their keys in section 2.4.

2.3 Channels

Channels in TOR networks are bidirectional, but have the issue of encryption acting differently in different directions. In the forward direction (Host $\rightarrow$ Endpoint), each relay *strips* a layer of encryption *off* the packet before passing it along the circuit. In the backward direction (Endpoint $\rightarrow$ Host), each relay *applies* a layer of encryption*to* the packet before passing it along the circuit backwards. To deal with this, we use two different sets of keys. The forward keys, all private, are used to decrypt the packets as they're passed through the network. The backward keys, all public, are used to encrypt packets as they're passed back. One thing of note in our actual implementation is that channels in their struct form see their forward and backward directions as relative to the direction a packet is being sent. This means that *forwards* can be backwards, and *backwards* can be forwards, at least on a granular per-channel basis. We will not expound on this further as to not bore the grader.

2.4 Handshake

The TOR handshake is quite complex (evidently, the developers making a completely anonymous browser are paranoid types), so we planned on implementing a limited version, described as follows:

- A host establishes a secure connection to the first relay in its predetermined circuit
- The first message a host sends is called a CREATE message, which the host sends to the first hop in its predetermined path. A CREATE message's data contains a public key used to encrypt the packet on the way *back* through the network. The host retains the private part of the keypair.
- The host receives a CREATED message confirming the establishment of a link to the relay and an agreement to exchange information. This also contains the relay's public key, which the host can use to encrypt future sent packets.
- Now one connection has been established, and the entry node of the circuit is established.

This is continued further to establish the next hops in the circuit, as follows:

- The host sends the last node in its current circuit (provided the circuit does not yet have three hops) an EXTEND message. This message is wrapped in a RELAY message and encrypted using the first host's NTRU public key, along with a layer of RSA encryption under that. This contains the same data as a CREATE message and an address of the next relay determined in the circuit.

- Upon receiving an EXTEND message, the relay checks if the determined next relay belongs to another circuit *in the same direction*. If it does, it sends back a response indicating that an EXTEND has failed, and the host should recompute the circuit path. If it doesn't, the relay goes through the CREATE/CREATED process described above, and upon success returns an EXTENDED packet with the new relay's public key.
- This process continues until the circuit grows to three members, upon which the host sends a BEGIN message to the last relay, which establishes a connection to the endpoint and opens up data transfer.

One important note is that as more nodes are added to the circuit, EXTENDS are wrapped in more and more layers of encryption. This somewhat paranoid policy ensures outside actors cannot decipher the path of circuits during creation.

2.5 Application of NTRU: Pitfalls and Plans

Our initial plan was to wrap all packets in multiple layers of NTRU encryption, but this came to a major pitfall when we realized that, in TOR, packet structure is nested.

That is to say, every packet after the CREATE/CREATED handshake is wrapped in a RELAY message, and those payloads contain more than just the data transferred in a message. NTRU's main disadvantage is that the larger the size of messages sent, and the larger parameters are made to allow larger messages, the more performance degrades. Another issue is that all messages in NTRU, when encrypted, are of the same size. This means that we can't really multilayer messages with any extra information, such as command headers. To combat this, we planned doing the actual onion-encryption part of TOR (that is, the three applied layers of encryption), with RSA, and then applying one final concrete skin of NTRU on top of the whole packet.

Unfortunately, by the time of this revelation, we were relatively close to the due date. We decided to instead improve our NTRU implementation and establish a naive demonstration of encrypting and transferring packets using the scheme. In the future, we hope to use this NTRU-wrapped onion approach to re-implement the project.

3 NTRU Cryptosystem

3.1 Lattice Based Cryptography

NTRU, like many other post-quantum cryptosystems, relies on the *Shortest Vector Problem* (SVP) and *Closest Vector Problem* (CVP) with lattices to make brute-force decryption the only known avenue for an attacker. You can think of a lattice like a vector space from linear algebra, except with discrete points arranged in an N-dimensional grid rather than a continuous span of vectors. In NTRU, our messages, keys, and ciphers are all polynomials that are represented as vectors based on their coefficients. When we encrypt a message, we "perturb" it by adding a random polynomial with very small coefficients to it.

These polynomials exist within an algebraic structure called a *ring* which we represent as R. You can think of a ring as a structure in which addition, subtraction, and multiplication exist between elements like normal, but division only works for special elements with multiplicative inverses called *units*.

3.2 Convolution Polynomial Rings

Choose a positive integer N. We define the *ring of convolution polynomials of rank* N as the quotient ring:

$$R = \frac{\mathbb{Z}[x]}{\langle x^N - 1 \rangle}.$$

Likewise, we define the *ring of convolution polynomials modulo* q as the quotient ring:

$$R_q = \frac{(\mathbb{Z}/q\mathbb{Z})[x]}{\langle x^N - 1 \rangle}.$$

In the first ring, the polynomial coefficients are integers $c_i \in \mathbb{Z}$. In the second ring, R_q, we consider coefficients as integers modulo q (that is, $c_i \in \mathbb{Z}/q\mathbb{Z}$). Both rings are quotient rings (notice the denominator $\frac{\dots}{\langle x^N-1\rangle}$); without going too far into the math, think of this division as something that allows us to replace any occurrence of x^N with the multiplicative identity 1, so that we may reduce exponents modulo N.

From this point forward, it will be simpler to refer to polynomials in R or R_q as vectors based on their coefficients in $\mathbb{Z}$ or $\mathbb{Z}/q\mathbb{Z}$ respectively:

$$r_0 + r_1 x + \cdots + r_{N-1}x^{N-1} \in R \text{ or } R_q \overset{\text{bij}}{\longleftrightarrow} (r_0, r_1, \ldots, r_{N-1}) \in \mathbb{Z}^N \text{ or } (\mathbb{Z}/q\mathbb{Z})^N.$$

Now, we are ready to define addition and multiplication in R and R_q.

3.3 Operations

3.3.1 Addition and Subtraction

Let $\mathbf{a}(x), \mathbf{b}(x) \in R$ or R_q. We define the *sum* of $\mathbf{a}(x)$ and $\mathbf{b}(x)$ as:

$$\mathbf{a}(x) + \mathbf{b}(x) \longleftrightarrow (a_0 + b_0, a_1 + b_1, \ldots, a_{N-1} + b_{N-1}).$$

Subtraction is the exact same, except that we take the difference of matching coefficients.

3.3.2 Multiplication

Let $\mathbf{a}(x), \mathbf{b}(x) \in R$ or R_q. We define the *product* of $\mathbf{a}(x)$ and $\mathbf{b}(x)$ as:

$$\mathbf{a}(x) \star \mathbf{b}(x) \longleftrightarrow (c_0, c_1, \ldots, c_{N-1}),$$

where

$$c_k = \sum_{i+j \equiv k \bmod N} a_i b_{k-i} \qquad \text{for all } i, j \text{ between } 0 \text{ and } N-1.$$

This definition for multiplication can be obtained by taking the product of both polynomials in the standard sense and substituting $x^N = 1$ as mentioned earlier.

3.3.3 Division

We can use regular old polynomial long division to compute the quotient $q(x)$ and remainder $r(x)$ when dividing two polynomials $a(x)$ and $b(x)$. This yields $a(x) = b(x)q(x) + r(x)$.

$$\begin{array}{r} \mathbf{q}(x) \\ \mathbf{b}(x) \overline{)\ \mathbf{a}(x)} \\ -\ \ \vdots \\ \hline \mathbf{r}(x) \end{array}$$

3.3.4 GCD and Inverses

The Euclidean Algorithm can be applied to polynomials in the ring R_q. The process allows us to compute the greatest common divisor (GCD) of two polynomials, as well as their multiplicative inverses when they exist.

Euclidean Algorithm. Given two polynomials $a(x)$ and $b(x) \in R_q$, the GCD can be found using polynomial division. Let

$$a(x) = q(x)b(x) + r(x),$$

where $q(x)$ is the quotient and $r(x)$ is the remainder. Notice that $r(x) = a(x) - q(x)b(x)$, which means that any polynomial which divides both $a(x)$ and $b(x)$ also divides $r(x)$ (since it is a combination of the two). Thus, $\gcd(a(x), b(x)) = \gcd(b(x), r(x))$, and we can successively reduce $a(x)$ and $b(x)$ until the remainder is zero. The last non-zero remainder is the GCD.

```rust
pub fn gcd(a: &ConvPoly, b: &ConvPoly, m: i32, n: usize) -> Result<ConvPoly, String> {
    let (mut old_r, mut r) = (a.clone(), b.clone());

    while !r.is_zero() {
        let (_, new_r) = old_r.div_mod(&r, m, n)?;
        (old_r, r) = (r, new_r);
    }

    // Normalize the gcd by dividing by its leading coefficient, if possible
    if let Ok(inverse) = inverse(old_r.lc(), m) {
        let inverse_poly = ConvPoly::constant(inverse);
        old_r = old_r.mul(&inverse_poly, n).modulo(m);
    }

    Ok(old_r)
}
```

Extended Euclidean Algorithm. The Extended Euclidean Algorithm computes not only the GCD of two polynomials $a(x)$ and $b(x)$, but also coefficients $s(x)$ and $t(x)$ such that:

$$a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x)).$$

This is particularly useful for finding inverses of polynomials in $R_q = (\mathbb{Z}/m\mathbb{Z})[x]/(x^N - 1)$ because if $b(x)$ is set to $x^N - 1$ (which is technically equal to zero) and the computed GCD ends up being 1, then we obtain

$$a(x)s(x) = 1.$$

In other words, $a(x)^{-1} = s(x)$ so that the first returned coefficient is the multiplicative inverse of our inputted polynomial $a(x)$.

We initialize:

$$r_{old}(x) = a(x), \quad s_{old}(x) = 1, \quad t_{old}(x) = 0$$
$$r(x) = b(x), \quad s(x) = 0, \quad t(x) = 1$$

These represent the initial linear combinations:

$$a(x) = 1 \cdot a(x) + 0 \cdot b(x), \quad b(x) = 0 \cdot a(x) + 1 \cdot b(x).$$

At each iteration of the algorithm, we compute the quotient $q(x)$ and the remainder $r_{new}(x)$ from the polynomial division:

$$r_{old}(x) = q(x)r(x) + r_{new}(x).$$

We update the coefficients $s_{old}(x)$, $t_{old}(x)$ as:

$$s_{new}(x) = s_{old}(x) - s(x)q(x), \quad t_{new}(x) = t_{old}(x) - t(x)q(x).$$

By induction, these remain linear combinations of $a(x)$ and $b(x)$. The process continues until $r(x) = 0$, at which point $r_{old}(x)$ is the GCD of $a(x)$ and $b(x)$, and $s_{old}(x)$, $t_{old}(x)$ satisfy Bézout's identity. Finally, we normalize the solution by dividing through the leading coefficient of the GCD, $\text{lc}(r_{old})$, if it is invertible in $\mathbb{Z}/m\mathbb{Z}$.

```rust
/// The Extended Euclidean Algorithm for polynomials. Returns (gcd, s(x), t(x)) such that
/// a(x)s(x) + b(x)t(x) = gcd(a(x), b(x)) within the ring (Z/mZ)[x]/(x^n - 1). Returns an error
/// if division fails at any point (which occurs when the leading coefficient of the divisor isn't
/// a unit in the ring Z/mZ).
pub fn extended_gcd(
    a: &ConvPoly,
    b: &ConvPoly,
    m: i32,
    n: usize,
) -> Result<(ConvPoly, ConvPoly, ConvPoly), String> {
    // Initial state
    let (mut old_r, mut old_s, mut old_t) =
        (a.clone(), ConvPoly::constant(1), ConvPoly::constant(0));
    let (mut r, mut s, mut t) = (b.clone(), ConvPoly::constant(0), ConvPoly::constant(1));

    while !r.is_zero() {
        let (q, new_r) = old_r.div_mod(&r, m, n)?;
        (old_r, r) = (r, new_r);
        (old_s, s) = (s.clone(), old_s.sub(&s.mul(&q, n)).modulo(m));
        (old_t, t) = (t.clone(), old_t.sub(&t.mul(&q, n)).modulo(m));
    }

    // Normalize the solution by dividing by the gcd's leading coefficient, if possible
    if let Ok(inverse) = inverse(old_r.lc(), m) {
        let inverse_poly = ConvPoly::constant(inverse);
        old_r = old_r.mul(&inverse_poly, n).modulo(m);
        old_s = old_s.mul(&inverse_poly, n).modulo(m);
        old_t = old_t.mul(&inverse_poly, n).modulo(m);
    }

    Ok((old_r, old_s, old_t))
}
```

3.3.5 Center Lifts

How do we lift a polynomial out of R_q and place it back into R? There are many equivalent ways of doing this, but the one NTRU chooses is called a center-lift. Let $\mathbf{a}(x) \in R_q$. The *center lift* of $\mathbf{a}(x)$ is the unique polynomial $\mathbf{a}^*(x) \in R$ that satisfies:

$$\mathbf{a}^*(x) \equiv \mathbf{a}(x) \pmod{q}$$

where the coefficients of $\mathbf{a}^*(x)$ are in the range $-\frac{1}{2}q < a_i^* \leq \frac{1}{2}q$.

3.3.6 Ternary Polynomials

And last but not least, we must define a special kind of polynomial with only $c_i \in \{-1, 0, 1\}$ as possible coefficients. Let d_1 and d_2 be positive integers. Define

$$\mathfrak{T}(d_1, d_2) = \left\{ \mathbf{a}(x) \in R : \begin{array}{l} \mathbf{a}(x) \text{ has } d_1 \text{ coefficients equal to } 1, \\ \mathbf{a}(x) \text{ has } d_2 \text{ coefficients equal to } -1, \\ \mathbf{a}(x) \text{ has all other coefficients equal to } 0 \end{array} \right\}.$$

3.4 Encryption & Decryption

Phew! That was a lot of math. Now we can finally get to the heart of our implementation and talk about NTRU encryption and decryption.

3.4.1 NTRU Parameters

To get started, Alice (or some other trusted 3rd-party) chooses the public parameters (N, p, q, d) with N and p prime, and $gcd(N, q) = gcd(p, q) = 1$. Additionally, $q > (6d + 1)p$.

3.4.2 NTRU Rings

These parameters allow us to fully define the following three convolution polynomial rings.

$$R = \frac{\mathbb{Z}[x]}{\langle x^N - 1 \rangle}, \qquad R_p = \frac{(\mathbb{Z}/p\mathbb{Z})[x]}{\langle x^N - 1 \rangle}, \qquad R_q = \frac{(\mathbb{Z}/q\mathbb{Z})[x]}{\langle x^N - 1 \rangle}.$$

3.4.3 Private Key

Alice can then generate a private key comprised of two randomly chosen ternary polynomials.

$$\mathbf{f}(x) \in \mathfrak{T}(d + 1, d) \qquad \text{and} \qquad \mathbf{g}(x) \in \mathfrak{T}(d, d)$$

Here, it's important to note that $\mathbf{f}(x)$ must have a multiplicative inverse in both R_p and R_q. If this is not the case, then Alice must choose a new $\mathbf{f}(x)$ and try again until she finds a suitable one. Thus, assuming Alice has already gone through this process, let

$$\mathbf{F_p}(x) = \mathbf{f}(x)^{-1} \in R_p \qquad \text{and} \qquad \mathbf{F_q}(x) = \mathbf{f}(x)^{-1} \in R_q.$$

3.4.4 Public Key

At this point, Alice can compute her public key:

$$\mathbf{h}(x) = \mathbf{F_q}(x) \star \mathbf{g}(x) \text{ in } R_q$$

3.4.5 Messages

A message $\mathbf{m}(x) \in R$ is a polynomial with coefficients in the range $-\frac{1}{2}p < m_i \leq \frac{1}{2}p$.

3.4.6 Ciphertext

Let $\mathbf{r}(x) \in \mathfrak{T}(d, d)$ be random. The ciphertext $\mathbf{e}(x) \in R_q$ is given by

$$\mathbf{e}(x) \equiv p\mathbf{h}(x) \star \mathbf{r}(x) + \mathbf{m}(x) \pmod{q}.$$

3.4.7 Decryption

First, compute

$$\mathbf{a}(x) \equiv \mathbf{f}(x) \star \mathbf{e}(x) \pmod{q}.$$

Take the *center lift* of this value to obtain $\mathbf{a}^*(x)$. The message can be retrieved by taking

$$\mathbf{m}(x) \equiv \mathbf{F_p}(x) \star \mathbf{a}^*(x) \pmod{p}.$$

4 Implementation of the NTRU Cryptosystem

Note: Much of this section is dedicated to minutia of implementing NTRU, and can be skimmed.

4.1 Representing Keys and Key Generation

In developing our keys, we had to define a struct to represent convolution polynomials in the rings described in Section 2.

```
pub struct ConvPoly {
    // Coefficients of the polynomial such that coeffs[i] is the coefficient of x^i
    pub coeffs: Vec<i32>,
}
```

ConvPolys form the core of our entire NTRU encryption implementation, as they are used both as keys and to encode encrypted messages. Our keys, then, take the following form:

```
/// An NTRU key pair
pub struct NtruKeyPair {
    pub public: NtruPublicKey,
    pub private: NtruPrivateKey,
}

// A public key used in the NTRU encryption scheme
pub struct NtruPublicKey {
    //..ConvPolys here
}

/// A private key used in the NTRU encryption scheme
pub struct NtruPrivateKey {
    //..ConvPolys here
}
```

To generate keys, we use the function ternary_polynomial(), which generates a balanced ternary ConvPoly (that is to say, a polynomial with -1, 0, and 1 coefficients) with a bounded number of ones and negative ones, with the indices of those being shuffled. All keys have N coefficients, where N is defined as a parameter in params.rs.

4.2 Encryption in the NTRU Crate

Encryption in the NTRU crate uses rules laid out in Section 3, but has some additional complexity provided by the need to serialize messages into valid convolution polynomials before encryption.

Messages are encrypted using encrypt_bytes(). This function calls the serialize() function, located in ntru_util.rs, which takes a Vec<u8> to a balanced ternary convolution polynomial.

```
pub fn serialize(plain_msg: Vec<u8>) -> ConvPoly {
    // Convert the message to a vector of ternary digits
    let mut digit_vec = Vec::with_capacity(plain_msg.len() * 5);
    for c in plain_msg {
        let arr = ternary(c.into());
        digit_vec.extend_from_slice(&arr);
    }

    ConvPoly { coeffs: digit_vec }
}
```

Of interest in this function is the ternary() function, which converts a character in a given byte vector to a 5-digit balanced ternary representation.

```
fn ternary(mut c: i32) -> [i32; 5] {
    assert!(c < 242 && c >= 0);
    if c == 0 {
        return [0; 5];
    }

    let mut digits = [0; 5];
    for i in (0..5).rev() {
        if c == 0 {
            break;
        }
        let rem_temp = c % 3;
        c /= 3;
        digits[i] = if rem_temp == 2 { -1 } else { rem_temp };
    }
    digits
}
```

Due to the use of only 5 digits in our ternary representations of inputs, we are only able to serialize messages encoding characters with values under 242, but this can fit in the ASCII character set, so we don't mind. This can be increased, but this would mean we would have to either set a smaller limit on encrypted message size or increase N (thereby decreasing efficiency).

Once the message is in the form of a valid convolution polynomial, we can perform our encryption as described in Section 3. This can also be found in the ntru_key.rs file, where operations are noted.

4.3 Decryption in the NTRU Crate

Decryption also follows operations outlined in Section 3, but has some added complexity from deserialization out of convolution polynomials into normal byte vectors.

```
//..This function is called by decrypt_to_bytes, which adds in the step of deserializing
// out of convolution polynomial coefficients into an actual vector!
pub fn decrypt_to_poly(&self, enc_msg: ConvPoly) -> ConvPoly {
        // a(x)  e(x) * f(x) (mod q)
        let a = enc_msg.mul(&self.f, N).center_lift(Q);
        // m(x)  a(x) * Fp(x) (mod p)
        let msg_poly = a.mul(&self.f_p, N).modulo(P);
        msg_poly
    }
```

After decrypting an encrypted polynomial to a message polynomial using the operations described in Section 3, we obtain a balanced ternary message polynomial, with each character of the unencrypted message represented by 5 balanced ternary digits. We'll discuss this functionality, along with some other considerations, when we discuss performance.

4.4 Performance & Implementation

Performance of the NTRU cryptosystem is widely seen as one of its main blockers and was a major issue in our initial implementation before extensive measures were made to optimize the crate.

4.4.1 Those are some big polynomials, man

The main blocker to our implementation's performance is the size of NTRU polynomials—which have the potential to increase massively depending on the size of N. Many operations, such as mul(), add(), and sub(), have terrible worst-case speed in cases where both convolution polynomials have many nonzero coefficients. Worst of all is inverse() due to the extended euclidean algorithm's inefficiency on large polynomials. We considered multiple ways to deal with this, such as abridging repeated coefficients, but did not have time to correctly implement some of these space-saving operations.

One massive speedup came between our first naive implementation which ran correctly, but made all convolution polynomials have exactly N coefficients (even with leading zeros), and the introduction of operations between differing length polynomials, plus the addition of trim() to truncate leading zeros of polynomials. Between these updates, we saw a speedup of roughly 160x, from key generation, encryption, and decryption taking 17520 ms $\rightarrow$ 110 ms (on Alex's laptop).

4.4.2 The issue with Parameters

The main issue with NTRU is that in order to send larger messages, we need to increase N. As many functions increase in cost proportional to N, this is a problem. Additionally, a major barrier seems to be q, which we have set much lower than the usual 256-bit NTRU parameter value of 8192.

4.4.3 Optimization of Deserialization

Deserialization saw much of Alex's focus when it came to our implementation. We don't have much time to talk about it, and deserialize() itself is rather boring, but Alex would like you to marvel at his work on the most unnecessarily optimized helper function of all time.

```rust
fn out_of_ternary(ser_ch: &[i32]) -> Option<u8> {
    if ser_ch == [0; 5] {
        return None;
    }
    // Neurotic optimization
    const POWERS: [i32; 5] = [1, 3, 9, 27, 81];
    // AHA! CONSTANT TIME ESCAPE TERNARY!
    let mut ans = 0;
    ans += bal_tern_esc(ser_ch[4], POWERS[0]);
    ans += bal_tern_esc(ser_ch[3], POWERS[1]);
    ans += bal_tern_esc(ser_ch[2], POWERS[2]);
    ans += bal_tern_esc(ser_ch[1], POWERS[3]);
    ans += bal_tern_esc(ser_ch[0], POWERS[4]);

    //..Converts ans to u8, returns None if not a u8
}

fn bal_tern_esc(n: i32, exp: i32) -> i32 {
    if n == -1 {
        2 * exp
    } else {
        n * exp
    }
}
```

out_of_ternary is a great example of the ~~neurotic~~ great optimization we tried to implement throughout our NTRU implementation, abridging iterating through the ternary 5-digits into a constant-time balanced ternary escape.

4.4.4 Future Optimizations

We have decided that over Winter break, we will likely try and release our NTRU implementation publicly on crates.rs. As such, we have considered many opportunities for efficiency (as we would not want to release our code publicly at its current fast, but not fast enough, state.

Our main considerations currently are to truncate repeated coefficients with some sort of embedded counter (perhaps allowing larger/smaller coefficients to represent repetition), integrating multithreading in index-safe functions like add() and sub(), and potentially using some bit magic in places that could use it.

4.5 Challenges, Extensibility

As discussed above, the main challenge with extending NTRU beyond the scope of our project so far is performance. Ideally, we would be able to encrypt messages of some reasonable size with 256-bit security, but that would require even more optimization, whatever form that may take. That said, in the words of Professor Hoffstein himself, "in order to even break 128-bit NTRU, you'd need every atom on Earth to be a computer, and all those computers would have to try and decode your messsage." Don't tell him we quoted him on that.

5 Acknowledgments

We would like to point our sincere appreciation toward Professor Jeffrey Hoffstein, one of the original creators of the NTRU cryptosystem. He personally met with us during the project and supported us with mathematical insight on some points of confusion.

We would also like to thank our professor, Nick DeMarinis, for teaching an incredible course. We both count CSCI 1680 as our favorite course of the semester (and, in Alex's case, his favorite course at Brown).