Verifying Correctness of the Number Theoretic Transform and Fast Number Theoretic Transform in F*

by

Ryuta R. Ono

S.B. in Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by: Ryuta R. Ono
Department of Electrical Engineering and Computer Science
August 9, 2024

Certified by: Anish Athalye
Doctoral Candidate, Thesis Supervisor

Certified by: Nickolai Zeldovich
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair
Master of Engineering Thesis Committee

Verifying Correctness of the Number Theoretic Transform and Fast Number Theoretic Transform in F*

by

Ryuta R. Ono

Submitted to the Department of Electrical Engineering and Computer Science
on August 9, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

As engineers continue to develop more sophisticated algorithms to optimize cryptographic algorithms, their often simple mathematical specifications become convoluted in the algorithms, from which a class of correctness bugs arise. Because cryptographic algorithms often secure sensitive information, their correctness, and in turn their security is a top priority. The Number Theoretic Transform (NTT) is an algorithm that enables efficient polynomial multiplication and has recently gained importance in post-quantum cryptography. This thesis presents a proof of correctness of the NTT in F*, a proof-oriented programming language that extracts to OCaml, and shows that we can use the NTT to perform polynomial multiplications. We provide an implementation of the Cooley-Tukey fast NTT algorithm and a proof that it matches the original NTT specification. This thesis also presents a representation of polynomials in the F* subset Low*, which extracts to performant C code.

Thesis supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis supervisor: Nickolai Zeldovich

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Thank you to Anish Athalye for being a remarkable mentor to me. Anish was always there when I needed guidance, and this thesis would not have been possible without him and I am extremely grateful. Thank you also to Professor Nickolai Zeldovich for the direction and thoughtful advice you gave to my thesis as it progressed.

Thank you to all of my professors at MIT who kindled my interests and enabled me to come to this point.

Thank you to my friends that made my MIT experience memorable, who I have gotten the pleasure of forming enduring relationships with over the past five years of ups and downs.

Finally, thank you to my family for their unwavering love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cryptographic algorithms are backed by assumed hardness of core mathematical problems. Cryptographic algorithms are proven to satisfy a property, for which people write optimized algorithms. Algorithm optimizations are complicated, increasing potential for bugs as the optimizations stray further from the specifications. Even widely used cryptographic libraries like OpenSSL have been found to have bugs, and even minor bugs could lead to critical vulnerabilities [1].

Formal verification is a technique that provides guarantees that an implementation satisfies a specification. Though writing verified code requires greater effort, in certain applications where the cost of writing buggy code is much greater, the effort can be well worth it. By writing specifications and mechanical proofs for cryptographic algorithms, we can mitigate security risks from this class of bugs.

Recent years have seen a substantial research effort in quantum computation, which can solve problems that are difficult or intractable on a conventional computer. Advancements in these technologies would break many cryptosystems that are currently in use. Post-quantum cryptography aims to develop cryptosystems that are secure against both conventional and

quantum computers. Although we do not yet know when the first physical quantum computers will be built, modern public key cryptography has taken almost two decades to deploy [2], so it is imperative to begin work on quantum-resistant cryptography well before a physical quantum computer is created.

The use of polynomials is widespread in both traditional and post-quantum cryptographic algorithms. The Number Theoretic Transform (NTT) is a powerful mathematical tool that has become important in many post-quantum cryptographic algorithms. A naive multiplication of two n degree polynomials requires multiplying every pair of coefficients, for a resulting $O(n^2)$ multiplications. Converting to the NTT domain allows us to multiply two polynomials elementwise with the property that the product polynomial will be equivalent to polynomial multiplication when we convert it back to the original domain [3]. While the naive conversions to and from the NTT domain are still $O(n^2)$, the Cooley-Tukey NTT and Gentleman-Sande INTT algorithms allow for $O(n \log n)$ conversions [4]. Thus, we can use the Cooley-Tukey NTT algorithm to convert two polynomials to the NTT domain in $O(n \log n)$, perform our coefficientwise multiplication in $O(n)$, and use the Gentleman-Sande INTT algorithm to convert the result back in $O(n \log n)$. In total, this yields a $O(n \log n) + O(n) = O(n \log n) < O(n^2)$ runtime.

The National Institute of Standards and Technology (NIST) have three post-quantum candidates for standardization, two of which use the NTT internally. ML-KEM/FIPS-203 is a lattice-based algorithm and one of the two candidates that uses the NTT [5].

F* is a proof-oriented language with a rich type system that allows users to restrict types to certain values, for example, as well as specify pre- and post-conditions of functions and prove that the implementation matches that specification [6]. HACL* is an F* library containing many cryptographic structures and is already used in production systems [7]. HACL* also has a verified implementation of FrodoKEM, a lattice-based post-quantum cryptographic algorithm that was previously a candidate for standardization [8], but does not use polynomial multiplication.

We provide an F* representation of polynomials along with proofs about their operations, including multiplication using the NTT, in hopes that this functionality will allow for a more streamlined experience for future engineers.

1.2 A Verified Number Theoretic Transform in F*

HACL* implements many cryptographic algorithms, but missing from its library are interfaces for modular integer arithmetic. To reason about polynomials and multiplication under the NTT, we require some way to represent sums of integers under a modulus, the modular inverse, and roots of unity modulo some modulus m [3].

The goal of this thesis is to provide a verified implementation of the NTT and a fast NTT algorithm in F*, which extracts to OCaml to yield an executable version of the NTT. ML-KEM uses a specific prime number q which we tailor our implementation of polynomials of integers modulo q to match, though a more general library requires minimal changes to the code. Changing to a different prime q requires no additional changes, given that the prime is sufficiently small. Note that for the NTT proofs, we require that there exists a $2n$-th root of unity modulo q.

The F* code can also act as a specification for a Low* implementation, which extracts to C for a performant polynomial multiplication algorithm ready for use in real world systems. This thesis provides a starting point for a Low* implementation, with a memory-safe representation of polynomials using memory safe fixed-length buffers equipped with addition and scalar multiplication.

1.3 Thesis contributions

This thesis makes the following contributions:

1. F* representations of mathematical sums of modular integers, primitive roots of unity,

and polynomials.

2. F* proofs about properties of the above representations.

3. A proof of correctness that component-wise multiplication in the NTT domain is equivalent to polynomial multiplication in the quotient ring, mechanized in F*.

4. An F* implementation of the Cooley-Tukey NTT algorithm, and an F* proof that it matches the NTT specification proven to be correct above.

5. A Low* representation of polynomials, as well as implementations for addition and scalar multiplication proven to match simple F* specifications.

The F* code for this thesis, as well as a Makefile to extract to OCaml can be found at https://github.com/rickono/hacl-star, in the `code/mlkem` directory.

1.4 Thesis outline

Chapter 2 of this thesis provides some background information on the F* proofs, the NTT, and mathematical properties needed to prove the NTT. Chapter 3 discusses the F* specification for the NTT and the mathematical structures required to represent it. Chapter 4 dives further into the proof details, including techniques to prove the mathematical lemmas and a sketch of the main correctness proof. Chapter 6 provides an evaluation of an extracted OCaml version of the fast NTT. Chapter 7 discusses some of the decisions made in proving the NTT. Chapters 8 and 9 presents some ideas for future work and related work in the field. Finally, chapter 10 concludes this thesis.

Chapter 2

Background

2.1 Formal Verification in F*

F* is a dependently-typed programming language and proof assistant. We use F* in a proof-oriented manner to simultaneously design programs and prove properties about them.

As a dependently-typed language, F* types can reason about their values. For example, we can say that a function `times_two` takes a natural number and returns an even natural number.

```
let times_two (n:nat): r:nat{r % 2 = 0} = 2 * n
```

This snippet initializes the function `times_two`, which has a parameter `n` that is a natural number. The return value `r`, also a natural number has a refinement type indicated by `{r % 2 = 0}`, restricting `r` to the elements in the set of natural numbers for which `r % 2 = 0` evaluates to `true`.

We can also introduce lemmas, which are F* functions which always return the `():unit` value. The type of the lemma contains pre- and post-conditions. If we want to prove that the output of factorial is greater than its argument, we could do that in a lemma.

```
let rec factorial (n:nat) =
  if n = 0 then 1
```

```
  else n * factorial (n - 1)
val factorial_gt_arg: (x:int)
    -> Lemma (requires x > 2) (ensures factorial x > x)
```

This property could be written in F* as an assertion.

```
assert (forall (x:int). x > 2 ==> factorial x > x)
```

However, F* is unable to prove the assertion, but not because the fact is untrue. We require a proof by induction to prove this fact, which F* is not able to do on its own. We can help F* by writing a lemma.

In a traditional mathematical proof, we might start with the base case 3, for which we know that the $3! = 6 > 3$. Then we can use weak induction to prove for all arguments greater than 3. F* allows us to apply the same logic by recursively applying our lemma.

```
let rec factorial_gt_arg x =
  if x = 3 then ()
  else factorial_gt_arg (x-1)
```

Invoking a lemma adds its post-condition to the proof context, given the pre-conditions are satisfied.

2.2 Fast polynomial multiplication

Cryptographic algorithms often require multiplying large polynomials, naively an $O(n^2)$ algorithm that we can speed up to $O(n \log n)$ using a fast NTT algorithm such as the Cooley-Tukey NTT. We break the proof of the fast NTT into two major steps:

1. Prove that the NTT and INTT allow for polynomial multiplication.
2. Prove that the fast NTT and fast INTT algorithms match the NTT and INTT that we specified.

2.2.1 Multiplication in the NTT domain

We first aim to prove that coefficientwise multiplication in the NTT domain corresponds to quotient-ring multiplication in the original domain. This allows for $O(n)$ multiplications, not including the conversion between domains.

$$\text{INTT}(\text{NTT}(f) \odot \text{NTT}(g)) = f \cdot g$$

where $\odot$ denotes componentwise multiplication.

The underlying algebraic proof depends on two main theorems, the reversibility of the NTT and the convolution theorem which relates coefficientwise multiplication to multiplication in the quotient-ring [3].

The NTT proof relies on sums of an arbitrary function $\sum_i f(i)$, along with various properties of these sums, primitive roots of unity mod m, that is an integer ψ where $\psi^n = 1 \mod m$, and modular inverse. Neither F* nor HACL* have these constructs so this thesis includes a representation for these mathematical tools as well as proofs to reason about them [3].

2.2.2 The fast NTT

While the NTT specification is indeed sufficient to prove correctness, the naive algorithm is $O(n^2)$, and requires a more efficient algorithm such as the Cooley-Tukey NTT algorithm to reap the performance benefits [4]. The Cooley-Tukey NTT algorithm is a divide and conquer algorithm that splits the polynomial on its even and odd terms, reducing runtime to $O(n \log n)$.

2.3 Notation

Our notation will match the mathematical symbols used in Kyber, an earlier version of ML-KEM, for simplicity. Kyber is being integrated into industry systems including Cloudflare and Amazon AWS [9]. The F* proofs in this thesis generalize to polynomials outside of the specific polynomials used in Kyber.

- $\mathbb{Z}_m$: The ring of integers modulo m.
- f_i: The coefficient of X^i of the polynomial f.
- n: The constant integer 256.
- q: The prime integer 7681.
- R_q: The ring $\mathbb{Z}_q[X]/(X^n + 1)$, equipped with addition and multiplication modulo $X^n + 1$.
- T_q: The image of R_q under the NTT.

Chapter 3

Approach

This chapter discusses the high-level approach for proving NTT in F*. This section will focus on the mathematical proofs required for proving NTT, and F* structures that needed to be implemented to support these proofs. Many proof details will be glossed over and will be explored further when we discuss the F* proof.

3.1 Proof goals

The goal of the NTT is to transform polynomials into a domain in which multiplications can be reduced to $O(n)$. While componentwise multiplication can trivially be done in $O(n)$, $O(n \log n)$ transformations in and out of the NTT domain are tricky, as a naive implementation of the specification is still $O(n^2)$. For our proof of the fast NTT, we break the NTT proof into two main goals:

1. Prove that componentwise multiplication in the NTT domain yields multiplication in the original domain:

$$f \cdot g = \text{INTT}(\text{NTT}(f) \odot \text{NTT}(g))$$

2. Prove correctness of $O(n \log n)$ implementations of the NTT and INTT.

With these two facts proven, we can use our $O(n \log n)$ algorithms to convert polynomials in and out of the NTT domain, and be confident that componentwise multiplication of the output polynomials yields multiplication of the original polynomials [3]. We can write these goals in F* as simple lemmas.

```
val mul_ntt_ok:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> g:lpoly n
  -> Lemma
      (ensures equal
        (mul_quotient_ring f g)
        (intt #n #psi
            (mul_componentwise (ntt #n #psi f) (ntt #n #psi g))))

val cooley_tukey_ok:
    #n:power_of_two{2 * n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2*n)
  -> f:lpoly n
  -> Lemma (equal (ntt #n #psi f) (ntt_ct #n #psi f))
```

3.1.1 Multiplication specification

Our top level specification is for multiplication in the quotient ring, and it is important we define it precisely.

We define multiplication in the quotient ring.

$$(f \cdot g)_k = \sum_{j=0}^{n-1} (-1)^{k-j \text{ div } n} f_j g_{k-j \bmod n} \tag{3.1}$$

```
let mul_quotient_ring_kth_term #n f g k =
  if k < 0 || k >= n then 0
  else sum_of_zqs 0 n
         (fun j
           -> (pow_mod_int_neg_one ((k - j) / n))
                %* ((poly_index f j) %* poly_index g ((k - j) % n)))
```

3.2 Proving NTT allows multiplication

We define the coefficient definitions of the NTT, INTT, and coefficient multiplication

$$(f \odot g)_k = f_k \cdot g_k$$
$$\mathrm{NTT}(f)_k = \sum_{j=0}^{n-1} f_j \psi^{j \cdot (2k+1)}$$
$$\mathrm{INTT}(f)_k = n^{-1} \cdot \sum_{i=0}^{n-1} f_i \psi^{-k \cdot (2i+1)}$$

where ψ is a $2n$-th root of unity mod q. It is important to note that all operations are done modulo q, and n^{-1} represents the modular inverse of n.

There are two main proofs that are necessary to prove that componentwise multiplication in T_q is equivalent to multiplication modulo $X^n + 1$ in R_q. These two proofs combine easily to prove our first of two large proof goals.

1. NTT is reversible: $\mathrm{INTT}(\mathrm{NTT}(f)) = f$
2. Convolution theorem: $\mathrm{NTT}(f \cdot g) = \mathrm{NTT}(f) \odot \mathrm{NTT}(g)$

Given both of these facts are true, applying INTT to both sides of the convolution theorem statement gives us:

$$\begin{aligned}\text{INTT}(\text{NTT}(f) \odot \text{NTT}(g)) &= \text{INTT}(\text{NTT}(f \cdot g)) \\ &= f \cdot g\end{aligned}$$

Therefore, if we have algorithms for INTT and NTT, we can use coefficient-wise multiplication instead of using naive multiplication in the quotient ring.

3.2.1 NTT is reversible

To prove that NTT is reversible, we prove the equality for each coefficient, from which the equality of the entire polynomial should immediately follow.

Unfolding NTT and INTT gives us the expression

$$\text{INTT}(\text{NTT}(f))_k = n^{-1} \left(\sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} f_j \psi^{j(2i+1)} \right) \psi^{-k(2i+1)} \right)$$

A series of basic sum and modular exponentiation properties allow us to rearrange this, which we cover in more detail in section 4.4.1.

$$\text{INTT}(\text{NTT}(f))_k = n^{-1} \sum_{j=0}^{n-1} f_j \psi^{j-k} \left(\sum_{i=0}^{n-1} \left(\psi^{2(j-k)} \right)^i \right)$$

Notice that the right summation is a geometric sum. If $j = k$, then this sum is simply a sum over 1s, resulting in n. Otherwise, the sum collapses to zero because ψ is a $2n$-th root of unity, and thus $\psi^{2n(j-k)} = 1^{j-k} = 1$.

$$\sum_{i=0}^{n-1} (\psi^{2(j-k)})^i = \frac{\psi^{2(j-k)n} - 1}{\psi^{2(j-k)} - 1} = 0$$

This allows us to evaluate the outer sum as well, as $j = k$ in only one instance, and we

end up with

$$n^{-1} f_k \psi^{k-k} n = n^{-1} n f_k = f_k$$

3.2.2 Convolution theorem

To prove the convolution theorem we again prove equality for each coefficient individually. Unfolding $\text{NTT}(f \cdot g)_k$ and rewriting the resulting expression gives us

$$\begin{aligned}
\text{NTT}(f \cdot g)_k &= \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} (-1)^{i-j \text{ div } n} f_j g_{i-j \bmod n} \right) \psi^{i(2k+1)} \\
&= \sum_{j=0}^{n-1} f_j \psi^{j(2k+1)} \left(\sum_{i=0}^{n-1} (-1)^{i-j \text{ div } n} \psi^{i(2k+1)} \psi^{-j(2k+1)} g_{i-j \bmod n} \right)
\end{aligned}$$

Using the definition of modulo and $\psi^n = -1$, we know that

$$(-1)^{i-j \text{ div } n} \psi^{i(2k+1)} \psi^{-j(2k+1)} = \psi^{(i-j \bmod n)(2k+1)}$$

which we can apply to our previous calculation

$$\begin{aligned}
&= \sum_{j=0}^{n-1} f_j \psi^{j(2k+1)} \left(\sum_{i=0}^{n-1} \psi^{(i-j \bmod n)(2k+1)} g_{i-j \bmod n} \right) \\
&= \sum_{j=0}^{n-1} f_j \psi^{f(2k+1)} \cdot \sum_{i'=0}^{n-1} \psi^{i'(2k+1) g_{i'}} \\
&= \text{NTT}(f)_k \cdot \text{NTT}(g)_k
\end{aligned}$$

While these two proofs allow us to convert between R_q and T_q and prove that we can use elementwise multiplications in T_q to compute multiplication in R_q, the simple coefficient specifications we gave for NTT and INTT still involve doing $O(n^2)$ operations. For a verified fast polynomial multiplication algorithm, we must implement more efficient NTT and INTT algorithms, and prove that they match our initial specifications.

3.3 The Cooley-Tukey NTT algorithm

The Cooley-Tukey NTT algorithm allows us to translate a polynomial into the NTT domain using a divide and conquer algorithm with a time complexity represented by the recurrence

$$T(n) = T(n/2) + \Theta(n) = O(n \log n)$$

If we then prove $\text{NTT}^{\text{CT}}(f) = \text{NTT}(f)$ for all polynomials f, we can use the Cooley-Tukey NTT with confidence that it behaves identically to our verified specification.

As with the NTT correctness proofs, we find it useful to reason about each of the terms in the Cooley-Tukey NTT and relate them to our original NTT specification and first prove $\text{NTT}^{\text{CT}}(f)_k = \text{NTT}(f)_k$. We define the kth term of the Cooley-Tukey NTT, $\text{NTT}^{\text{CT}}(f)_k$ as

$$\text{NTT}^{\text{CT}}(f)_k = \text{NTT}^{\text{CT}}(f_{even})_k + \psi^{2k+1} \cdot \text{NTT}^{\text{CT}}(f_{odd})_k$$

for $k < n/2$ and

$$\text{NTT}^{\text{CT}}(f)_k = \text{NTT}^{\text{CT}}(f_{even})_k - \psi^{2k+1} \cdot \text{NTT}^{\text{CT}}(f_{odd})_k$$

for $k \geq n/2$, where

$$f_{even} = f_0 + f_2X^2 + \cdots + f_nX^n \text{ and } f_{odd} = f_1X + f_3X^3 + \cdots + f_{n-1}X^{n-1}$$

allowing us to compute two terms of the NTT with just one additional operation.

For the proof of correctness it is again useful to look at individual coefficients. We can manipulate the specification for the kth term of the NTT to match our Cooley-Tukey specification.

$$\mathrm{NTT}(f)_k = \sum_{j=0}^{n-1} f_j \psi^{j\cdot(2k+1)} \tag{3.2}$$

$$= \sum_{j=0}^{n/2-1} f_{2j} \cdot \psi^{2j\cdot(2k+1)} + \sum_{j=0}^{n/2-1} f_{2j+1} \cdot \psi^{(2j+1)\cdot(2k+1)} \tag{3.3}$$

$$= \sum_{j=0}^{n/2-1} f_{2j} \cdot \psi^{2j\cdot(2k+1)} + \psi^{2k+1} \cdot \sum_{j=0}^{n/2-1} f_{2j+1} \cdot \psi^{(2j+1)} \tag{3.4}$$

$$= \mathrm{NTT}(f_{even})_k + \psi^{2k+1} \cdot \mathrm{NTT}(f_{odd})_k \tag{3.5}$$

Using strong induction, we can easily prove that $\mathrm{NTT}(f)_k = \mathrm{NTT}^{\mathrm{CT}}(f)_k$, for polynomials f with degree that is a power of two minus one. The base case for a degree 0 polynomial proceeds trivially as we use the original specification to dictate the behavior of $\mathrm{NTT}^{\mathrm{CT}}$ for the base case [4].

3.4 F⋆ interfaces and lemmas

There are several concepts necessary for these mathematical proofs for which F⋆ did not have a representation:

- Fixed-degree polynomials of integers modulo a modulus m
- Sums of integers modulo a modulus m
- Modular exponentiation with negative exponents (modular inverse)
- Roots of unity modulo a modulus m

3.4.1 F⋆ polynomial representation

For our proofs, we use a coefficient representation of polynomials. We use a natural representation in F⋆: `lpoly`, defined as a sequence of `zq`, integers modulo q. Our polynomials

are written using a fixed q, but can be modified to accommodate any modulus. We use the `nat_mod` from the HACL* library, defined as a natural number less than a given positive modulus `m`.

```
let nat_mod (m:pos) = n:nat{n < m}
...
let zq = nat_mod q
let lpoly deg = lseq zq deg
```

3.4.2 F* sum representation

We represent a sum of integers $\in \mathbb{Z}_q$ as a function that takes the bounds of the sum, and a function that takes an integer and returns an integer $\in \mathbb{Z}_q$. The sum is calculated recursively by adding the function applied to the current upper bound to the rest of the sum.

```
let rec sum_of_zqs (start:int) (stop:int) (f:(i:int -> zq))
    : Tot zq (decreases stop - start) =
  if start >= stop then 0
  else f (stop - 1) +% (sum_of_zqs start (stop - 1) f)
```

We discuss specific lemmas about sums along with details about their proofs in section 4.2.

3.4.3 Modular exponentiation and inverse

F* has a function `pow_mod: #m:pos{1 < m} -> a:nat_mod m -> b:nat -> nat_mod m`, which implments $a^b \mod m$, but requires b to be a natural number. NTT requires the concept of modular inverses. While one option would be to introduce one function, perhaps `mod_inverse` that takes a natural number modulo m, and returns its modular inverse, modular inverses also allow us to treat the exponent similarly to normal integer exponentiation, which simplifies working with the exponents. Thus we define a function `pow_mod_int`, which

allows `b` to be any integer, and treats a negative exponent as a modular inverse composed with the original `pow_mod`.

```
let pow_mod_int #m a b =
  if b >= 0 then
    pow_mod #m a b
  else
    pow_mod #m a ((-b) * (m - 2))
```

To tie the exponent properties with modular inverse, we need to prove the definition of modular inverse, $a^1 \cdot a^{-1} = 1$, and then prove that this new `pow_mod_int` still allows us to use the following exponent rules:

- $0^b = 0$
- $1^b = 1$
- $a^0 = 1$
- $a^1 = a$
- $a^b a^c = a^{b+c}$
- $(a^b)^c = a^{bc}$

These proofs are somewhat tedious, but simply involve converting `pow_mod_int` to the existing `pow_mod` and working through both positive and negative cases.

3.4.4 The F⋆ root of unity type

Our NTT and INTT proofs require ψ to be a primitive $2n$-th root of unity mod q, as we need to exploit some of the properties of the roots of unity. An n-th root of unity under some modulus m is defined as an integer ω such that $\omega^n = 1$ and $\omega \neq 0 \mod m$.

We can write a predicate for an n-th root of unity rather directly, and use it to define a generic n-th root of unity type.

```
let is_nth_root_of_unity_mod
    (#m:prime{m > 2}) (n:nat{n > 0}) (root:nat_mod m)
= pow_mod #m root n == 1 /\ root % m <> 0
let nth_root_of_unity_mod
    (#m:prime{m > 2}) (n:nat{n > 0})
= root:nat_mod m{is_nth_root_of_unity_mod #m n root}
```

For ω to be a primitive n-th root of unity, it is required that ω is the smallest n-th root of unity. We can compose the two predicates to make a type for primitive n-th roots of unity.

```
let is_primitive
    (#m:prime{m > 2}) (#n:nat{n > 0}) (a:nat_mod m)
= (forall (k:nat{k < n}). pow_mod #m a k <> 1)

let is_primitive_nth_root_of_unity_mod
    (#m:prime{m > 2}) (n:nat{n > 0}) (root:nat_mod m)
= is_nth_root_of_unity_mod #m n root /\ is_primitive #m #n root

let primitive_nth_root_of_unity_mod
    (#m:prime{m > 2}) (n:nat{n > 0})
= root:nat_mod m{is_primitive_nth_root_of_unity_mod #m n root}
```

Chapter 4

Proving NTT correctness in F*

4.1 General proof development

F* proofs rely on Z3, an SMT solver that relies on specific heuristics. Small changes in a query can greatly impact the solver's performance. To keep the search space small, it is often beneficial to break F* proofs into the smallest pieces of logic possible to later be composed into larger proofs. For example the algebraic proof for the reversibility of NTT can be roughly broken into ten small steps that manipulate the definition of NTT. Attempting to bundle the entire proof into one F* function has far too large of a search space and will fail to verify, while breaking the ten steps into separate lemmas, and invoking each allows F* to verify the theorem.

For this reason it is useful in F* to have a vision of the entire proof before diving into proof details, as this makes it much easier to break the proof into components. This approach also allows the engineer to write the types for each of the lemmas, admitting the proof for each step, to increase confidence that the proof will verify before spending time on implementation details.

4.2 Lemmas about sums

Instead of working with function extensionality `f = g`, we explicitly state in the preconditions of the lemmas our definition of function equality: that two functions are to be considered equivalent if they produce the same value for inputs between `start` and `stop` in a sum. If we know that two functions are equal by this definition, we can safely swap the function to be evaluated in our sum, allowing us to manipulate sums.

```
let rec sum_rewrite_lemma (start stop:int) (f g:int -> zq): Lemma
    (requires (forall (i:int).{:pattern (f i)} start <= i /\
        i < stop ==> f i == g i))
    (ensures (sum_of_zqs start stop f) == (sum_of_zqs start stop g))
    (decreases (stop - start))
    [SMTPat (sum_of_zqs start stop f)]
    =
  if start < stop then sum_rewrite_lemma start (stop - 1) f g
```

We attach an `SMTPat` to this lemma, so if Z3 sees a sum, and sees the precondition satisfied, it tries to apply this lemma. Many proofs about sum equality followed the same general process:

1. Prove an auxiliary lemma `aux` that the two function bodies are equal for some unrestricted integer input i.

2. Invoke `Classical.forall_intro aux`, which introduces `forall i. aux i` into the context. Given an unpolluted local context, the aforementioned SMT pattern allows the equality of sums to go through. If this fails to verify, it could be a sign that the proof has become too large.

4.2.1 Inductive proofs about sums

$$\sum_{i=m}^{n} a \cdot f(i) = a \sum_{i=m}^{n} f(i) \tag{4.1}$$

On paper, it appears that the proof $\sum_{i=m}^{n} a \cdot f(i) = a \sum_{i=m}^{n} f(i)$ should follow rather directly from the definitions of sum and the distributive property of addition. However this would require us to convert the sum into an n-ary addition operation and proving an n-ary distributive property. Written recursively, the proof lends itself to induction, in which we can apply the distributive property of addition to a binomial addition. Many of our proofs about sums follow an inductive structure.

The base case $m = n$ goes through with a lemma `lemma_sum_none` that simply states that $\sum_{i=n}^{n} = 0$, to which we attach an SMT pattern.

```
let rec sum_mul_lemma (a:zq) (start stop:int) (f:int -> zq): Lemma
    (ensures mul_zq a (sum_of_zqs start stop f) == sum_of_zqs start
       stop (fun i -> mul_zq a (f i)))
    (decreases (stop - start))
    [SMTPat (sum_of_zqs start stop (fun i -> a %* (f i)))] =
  if start < stop then (
    sum_mul_lemma a start (stop - 1) f;
    calc (==) {
      mul_zq a (sum_of_zqs start stop f);
      (==) {}
      a %* ((sum_of_zqs start (stop - 1) f) +% f (stop - 1));
      (==) {lemma_mod_distributivity_add_right #q a (sum_of_zqs start
          (stop - 1) f) (f (stop - 1))}
      (a %* (sum_of_zqs start (stop - 1) f)) +% (a %* f (stop - 1));
    }
  )
```

Here, in our proof body we invoke the inductive hypothesis, which introduces

$$\sum_{i=start}^{stop-2} a \cdot f(i) = a \sum_{i=start}^{stop-2} f(i)$$

to the context. `calc (==)` allows us to chain equivalent terms. We unfold the original sum, apply the distributivity of addition lemma, and the inductive hypothesis allows us to finish the rest of the proof.

The lemma for the linearity property of sums

$$\sum_{i=n}^{m} f(i) + g(i) = \sum_{i=n}^{m} f(i) + \sum_{i=n}^{m} g(i) \tag{4.2}$$

proceeds similarly by induction by unfolding the sum, and applying the definition of sums.

```
1 let rec lemma_sum_linearity (start stop:int) (f:int -> zq) (g:int ->
    zq): Lemma
2   (ensures sum_of_zqs start stop (fun i -> f i +% g i) ==
       sum_of_zqs start stop f +% sum_of_zqs start stop g)
3   (decreases stop - start)
```

We also require interchanging the order of summation

$$\sum_i \sum_j f(i,j) = \sum_j \sum_i f(i,j) \tag{4.3}$$

Most proofs rely on a matrix representation of a double sum, in which entries in the matrix are added together, claiming that swapping the order of the sums does not affect the resulting sum, per the commutative property of addition. In our representation of the sum this kind of proof is not possible, and we use an inductive proof here as well. Unfolding both sums reveals four terms, to which when we invoke the inductive hypothesis it becomes obvious

that the sums are equivalent.

$$\begin{aligned}\sum_{i=n}^{m}\sum_{j=n'}^{m'} f(i,j) &= \sum_{i=n}^{m-1}\sum_{j=n'}^{m'-1} f(i,j) + \sum_{i=n}^{m-1} f(i,m'-1) + \sum_{j=n}^{m'-1} f(m-1,j) + f(m-1,m'-1) \\ &= \sum_{j=n'}^{m'-1}\sum_{i=n}^{m-1} f(i,j) + \sum_{i=n}^{m-1} f(i,m'-1) + \sum_{j=n}^{m'-1} f(m-1,j) + f(m-1,m'-1) \\ &= \sum_{j=n'}^{m'}\sum_{i=n}^{m} f(i,j)\end{aligned}$$

Our inductive hypothesis allows to flip the sum order for the first term, and the remaining terms are equivalent if start with the reversed sum.

```
let rec swap_sum_order (start1 stop1 start2 stop2:int) (f:int -> int
   -> zq): Lemma
    (requires stop1 > start1 /\ stop2 > start2)
    (ensures
      sum_of_zqs start1 stop1
        (fun i -> sum_of_zqs start2 stop2 (fun j -> f i j)) ==
      sum_of_zqs start2 stop2
        (fun j -> sum_of_zqs start1 stop1 (fun i -> f i j)))
    (decreases (stop1 - start1))
```

The following lemmas are necessary to prove the NTT and proceed similarly by induction.

Additivity of summation

$$\sum_{i=a}^{b-1} f(i) + \sum_{i=b}^{c} f(i) = \sum_{i=a}^{c} f(i) \tag{4.4}$$

```
let rec lemma_sum_join (i j k:int) (f:int -> zq): Lemma
  (requires i <= j /\ j <= k)
  (ensures sum_of_zqs i k f == sum_of_zqs i j f +% sum_of_zqs j k f)
  (decreases (k - j))
```

Shift property

$$\sum_{i=n}^{m} f(i) = \sum_{i=n+k}^{m+k} f(i-k) \tag{4.5}$$

The F* specification is more general than the above equation, and we can capture the shifted equation by using $g(i) = f(i-k)$.

```
let rec lemma_sum_shift (start stop:int) (shift:int) (f g:int -> zq):
    Lemma
  (requires (forall (i:int).
    start <= i /\ i < stop ==> f i == g (i + shift)))
  (ensures sum_of_zqs start stop f ==
    sum_of_zqs (start + shift) (stop + shift) g)
  (decreases (stop - start))
```

Geometric sum

$$\sum_{i=0}^{n-1} a^i = (a^n - 1) \cdot (a-1)^{-1} \tag{4.6}$$

```
let rec lemma_geometric_sum (stop:pos) (a:zq): Lemma
  (requires a % q <> 1)
  (ensures sum_of_zqs 0 stop (fun i -> pow_mod_int #q a i) ==
    ((pow_mod_int #q a stop) -% 1)
      %* (pow_mod_int #q ((a - 1) % q) (-1)))
  (decreases stop)
```

We also need to split the sum into odd and even terms for the fast NTT algorithm.

$$\sum_{i=0}^{n-1} f(i) = \sum_{i=0}^{n/2-1} f(2i) + \sum_{i=0}^{n/2-1} f(2i+1) \tag{4.7}$$

```
let rec lemma_sum_split_parity (stop:nat{stop % 2 = 0}) (f:int -> zq)
   : Lemma
  (requires stop >= 0)
```

```
3   (ensures sum_of_zqs 0 stop f == sum_of_zqs 0 (stop/2) (fun i -> f
       (2 * i)) +% sum_of_zqs 0 (stop/2) (fun i -> f (2 * i + 1)))
```

Unfolding and both sums allows this proof to go through with induction.

4.2.2 Other proofs about sums

In the event that we have a geometric sum on ones from 0 to some `stop`, the sum evaluates to `stop`.

$$\sum_{i=0}^{n-1} 1^i = n \mod m \tag{4.8}$$

```
1 let lemma_geometric_sum_ones (stop:pos) (a:zq): Lemma
2   (requires a % q = 1)
3   (ensures sum_of_zqs 0 stop (fun i -> pow_mod_int #q a i) == stop %
       q)
```

A sum of any number of zeros is always zero.

$$\sum_{i=0}^{n-1} 0 = 0 \tag{4.9}$$

```
1 let rec lemma_sum_zeros (start stop:nat): Lemma
2   (requires start <= stop)
3   (ensures sum_of_zqs start stop (fun i -> 0) == 0)
```

To prove the convolution theorem, we need to reindex a sum.

$$\sum_{i=0}^{n-1} f(i-j \mod n) = \sum_{i'=0}^{n} f(i') \tag{4.10}$$

The above property is intuitively true because of the cyclic nature of modulo. For any shift j, the sum still cycles through all integers 0 to n. If F* we prove this lemma by breaking

the sum into two sums at $j \mod n$, manipulating them separately, and joining them back together.

```
let lemma_sum_shift_mod (stop:nat{stop > 0}) (j:int) (f:int -> zq):
Lemma
  (sum_of_zqs 0 stop f == sum_of_zqs 0 stop (fun i -> f ((i - j) %
     stop)))
```

4.3 Lemmas about modular arithmetic and roots of unity

4.3.1 Modular arithmetic

The NTT requires the concept of a modular inverse, which the F* `nat_mod` does not support. We use Fermat's Little Theorem to prove our definition of the modular inverse. Here, we give implement a new `pow_mod_int` function that allows the exponent to be any arbitary integer, and in our lemma define what the negative exponent means.

```
val pow_mod_int:
    #(m:prime)
  -> (a:nat_mod m)
  -> (b:int)
  -> nat_mod m
let pow_mod_int #m a b =
  if b >= 0 then
    pow_mod #m a b
  else
    pow_mod #m a ((-b) * (m - 2))

val lemma_pow_mod_inv_def_nat:
    #m:prime
```

```
14   -> a:nat_mod m{a % m <> 0}
15   -> b:nat
16   -> Lemma (pow_mod_int #m a b * pow_mod_int #m a (-b) % m == 1)
```

This is sufficient to define `pow_mod_int`, but we can no longer apply the same algebraic rules to these exponents. For instance, we can no longer say that $a^b \cdot a^c = a^{b+c}$. For most of these lemmas, we reprove these algebraic properties by dividing into three cases:

- Both exponents are positive, in which case we can use the corresponding lemma for `pow_mod`
- Both exponents are negative.
- One exponent is positive and one is negative.

4.3.2 Roots of unity

There are several lemmas about roots of unity that we require for our proofs of the NTT. For a $2n$-th root of unity ψ, we require

- $\psi^{n/2} = -1$
- ψ^2 is a primitive n-th root of unity

The first lemma takes a prime number `m` and natural number n, and says that for a primitive $2n$th root of unity ψ, $\psi^{n/2} = -1 \mod m$ for all even n.

```
1  val lemma_primitive_unity_half_n:
2      #m:prime{m > 2}
3    -> #n:nat{n > 0}
4    -> a:primitive_nth_root_of_unity_mod #m n
5    -> Lemma (requires n % 2 == 0)
6             (ensures pow_mod_int #m a (n / 2) == (-1) % m)
```

The mathematical proof for this proceeds as follows. Let $k = n/2$.

$$\begin{aligned} 1 &= a^{2k} \mod m \\ &= (a^k)^2 \mod m \end{aligned}$$

We know that for any x, $x^2 = 1 \mod m \implies x = 1$ or $x = -1$. Since we know that $k < n$, we know that $a^k = -1$ by the definition of a *primitive* root of unity (that there is no smaller root of unity). The F* proof follows these same steps:

```
val lemma_unit_roots:
    #m:prime{m > 2}
  -> x:nat_mod m
  -> Lemma
     (pow_mod_int #m x 2 == 1 ==> x == (1 % m) \/ x == ((-1) % m))

let lemma_primitive_unity_half_n #m #n a =
  let k = n / 2 in
  calc (==) {
    1;
    (==) {}
    pow_mod_int #m a (k * 2);
    (==) {lemma_pow_mod_int_mul #m a k 2}
    pow_mod_int #m (pow_mod_int #m a k) 2;
  };
  lemma_unit_roots #m (pow_mod_int #m a k);
  assert (pow_mod_int #m a k == (1 % m)
    \/ pow_mod_int #m a k == ((-1) % m))
```

We also require the fact that ψ^2 is a primitive nth roof of unity so that we can acquire a primitive root of unity for an NTT for a polynomial with half as many terms. Let $\omega = \psi^2$.

To prove that ω is a primitive nth root of unity, we start by proving that ω is a root of unity, then prove that it is a primitive root of unity.

Proving that ω is an nth root of unity follows directly from the definition of roots of unity

$$\omega^n = (\psi^2)^n = \psi^{2n} = 1$$

We prove that this is indeed a primitive root of unity by contradiction. If we do have $\omega^k = 1$ for some $k < n$, this necessarily means that we also have some $k' < 2n$ where $\psi^{k'} = 1$.

4.4 The NTT proof

For the NTT and INTT function, we define a function that calculates the k-th coefficient of the NTT representation of a polynomial, and create an `lpoly_n` using this function. The NTT specification is shown below, and the INTT is defined analogously.

```
val ntt_kth_term:
    #n:power_of_two
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> k:int
  -> zq
let ntt_kth_term #n #psi f k =
  if k < 0 || k >= n then 0
  else sum_of_zqs 0 n (fun j ->
    mul_zq (poly_index f j) (pow_mod_int #q psi (j * (2 * k + 1))))

let ntt (#n:power_of_two)
        (#psi:primitive_nth_root_of_unity_mod #q (2 * n)) (f:lpoly n)
=
  createi n (fun k -> ntt_kth_term #n #psi f k)
```

Recall that to prove correctness of the NTT we need to prove that NTT is reversible and the convolution theorem. Both of these proofs rely on relatively simple rewriting of sums and application of properties of modular exponentiation and roots of unity but require careful compartmentalization and lemmas about sums and modular arithmetic.

The top-level theorem that verifies that our NTT and INTT definitions are sound should simply state that component-wise multiplication in the NTT domain results in multiplication in the quotient ring when we transform using the INTT, that is

$$f \cdot g = \mathrm{INTT}(\mathrm{NTT}(f) \odot \mathrm{NTT}(g))$$

We can state this as a simple F* lemma, using sequence equality for polynomial equality:

```
val mul_ntt_ok:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> g:lpoly n
  -> Lemma
      (ensures Seq.equal
        (mul_quotient_ring f g)
        (intt #n #psi
            (mul_componentwise (ntt #n #psi f) (ntt #n #psi g))))
```

Defining the two lemmas that we need allow this proof to go through easily. The lemma for reversibility simply states that $f = \mathrm{INTT}(\mathrm{NTT}(f))$.

```
val intt_ntt_is_id:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> Lemma (ensures equal f (intt #n #psi (ntt #n #psi f)))
```

The convolution theorem relates multiplication in the quotient ring to component-wise multiplication in the NTT domain, $\text{NTT}(f \cdot g) = \text{NTT}(f) \odot \text{NTT}(g)$.

```
val convolution_theorem:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> g:lpoly n
  -> Lemma
      (ensures equal
        (ntt #n #psi (mul_quotient_ring f g))
        (mul_componentwise (ntt #n #psi f) (ntt #n #psi g)))
```

Given these two lemmas, it is easy to prove our original correctness theorem.

```
let mul_ntt_ok #n #psi f g =
  convolution_theorem #n #psi f g;
  assert (Seq.equal
            (intt #n #psi (ntt #n #psi (mul_quotient_ring f g)))
            (intt #n #psi
              (mul_componentwise (ntt #n #psi f) (ntt #n #psi g))));
  intt_ntt_is_id #n #psi (mul_quotient_ring f g);
  assert (mul_quotient_ring f g == intt #n #psi (mul_componentwise (
     ntt #n #psi f) (ntt #n #psi g)))
```

Invoking our two lemmas allows the proof to verify without much added work. The next two sections discuss the proof strategy for these two lemmas.

4.4.1 NTT is reversible

To prove that NTT is reversible, we prove equality each term k. We split the proof for `intt_ntt_is_id_kth_term` into steps that are easy to reason about in F* individually. Writ-

ing each step as a lemma allows us to tie them all together to prove the overall property. All of the operations are done $\mod q$. We first start with a function `intt_ntt_kth_term`, which yields $\text{INTT}(\text{NTT}(f))_k$, and manipulate the definition.

In the majority of this proof we work with a double summation. Recall that we represent sums of integers mod q in F* as a function that takes a function `f: int -> zq`, and evaluates it with inputs between two natural numbers `start` and `stop`, summing the results. Our `sum_rewrite_lemma` allows us to rewrite a sum using a function `f` to a function `g` as long as we satisfy the predicate

```
forall (i:int). start <= i /\ i < stop ==> f i = g i
```

To manipulate our sums, we make an auxiliary lemma of the form

```
let aux (f g:int -> zq) (i:int): Lemma (f i = g i)}
```

Then we can use `Classical.forall_intro (aux f g)` which introduces `forall (i:int). f i = g i` to the proof context, which is picked up by the `SMTPat` on `sum_rewrite_lemma`.

$$\begin{aligned}\texttt{intt\_ntt\_kth\_term}\ \#\texttt{n}\ \#\texttt{psi}\ \texttt{f}\ \texttt{k} &= n^{-1} \cdot \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} f_j \cdot \psi^{j \cdot (2i+1)} \right) \cdot \psi^{-k \cdot (2i+1)} \\ &= n^{-1} \cdot \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_j \cdot \psi^{j \cdot (2i+1)} \cdot \psi^{-k \cdot (2i+1)}\end{aligned}$$

In this first step in our proof, we unfold the definition of `intt_ntt_kth_term` to expose the underlying double sum. The first manipulation simply uses the property from equation 4.1, because the term $\psi^{-k \cdot (2i+1)}$ does not contain j. In F*, we define the auxiliary lemma that allows us to reason about the function inside the sum, by using a fixed `i`. This auxiliary lemma easily verifies with an invocation of `sum_mul_lemma` which is our F* proof of equation 4.1. Then, we use `Classical.forall_intro` which introduces the auxiliary lemma with a forall quantifier to our proof context, and allows us to relate the original double sums.

```
val intt_ntt_is_id_kth_term_1_aux
```

```
  ...
  -> Lemma
      ((sum_of_zqs 0 n (fun j -> (poly_index f j) %* (pow_mod_int #q
         psi (j * (2 * i + 1))))) %* (pow_mod_int #q psi (-k * (2 * i
          + 1)))
    == (sum_of_zqs 0 n (fun j -> (poly_index f j) %* (pow_mod_int #q
       psi (j * (2 * i + 1))) %* (pow_mod_int #q psi (-k * (2 * i +
       1))))))
let intt_ntt_is_id_kth_term_1_aux #n #psi f k i =
  sum_mul_lemma
    (pow_mod_int #q psi (-k * (2 * i + 1)))
    0 n
    (fun j ->
      (poly_index f j) %* (pow_mod_int #q psi (j * (2 * i + 1))))
let intt_ntt_is_id_kth_term_1 #n #psi f k = Classical.forall_intro (
   intt_ntt_is_id_kth_term_1_aux #n #psi f k)
```

Almost all of the steps in proofs involving sums use this approach, and will be omitted in future explanations for brevity.

$$= n^{-1} \cdot \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_j \cdot \psi^{(j-k)\cdot(2i+1)}$$
$$= n^{-1} \cdot \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_j \cdot \psi^{(j-k)\cdot 2i} \cdot \psi^{(j-k)}$$

These next two steps simply use properties of modular exponentiation, using the new `pow_mod_int` lemma `lemma_pow_mod_int_add` as well as `calc (==)` for the distributive property of integer multiplication over addition. You may notice that the code sometimes omits lemmas such as `distributivity_add_left` in a `calc (==)` block, though for lines of reasoning containing many algebraic manipulations, the prover can struggle without an

explicit statement of even these basic facts.

$$= n^{-1} \cdot \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} f_j \cdot \psi^{(j-k)\cdot 2i} \cdot \psi^{(j-k)}$$

This proceeds easily with the `swap_sum_order` lemma, without need of an auxiliary lemma as the lemma already reasons about double summations.

$$= n^{-1} \cdot \sum_{j=0}^{n-1} f_j \cdot \psi^{(j-k)} \sum_{i=0}^{n-1} \psi^{(j-k)\cdot 2i}$$
$$= n^{-1} \cdot \sum_{j=0}^{n-1} f_j \cdot \psi^{(j-k)} \sum_{i=0}^{n-1} (\psi^{2\cdot(j-k)})^i$$

These next two steps use equation 4.1's lemma as well as `lemma_pow_mod_int_mul` in a similar fashion to the previous steps.

$$= n^{-1} \cdot \left(\sum_{j=0}^{k-1} f_j \cdot \psi^{(j-k)} \sum_{i=0}^{n-1} (\psi^{2(j-k)})^i \right.$$
$$+ f_k \cdot \psi^{(k-k)} \sum_{i=0}^{n-1} (\psi^{2(k-k)})^i$$
$$\left. + \sum_{j=k+1}^{n-1} f_j \cdot \psi^{(j-k)} \sum_{i=0}^{n-1} (\psi^{2(j-k)})^i \right)$$

The next step in the lemma breaks the outer sum into three parts: a sum of $j < k$, the term $j = k$, and a sum of $j > k$. In F* we do this in two separate manipulations, where we split into two sums of $j \leq k$ and $j > k$ using `lemma_sum_join`, and unfold the first sum simply

using the definition of sums.

$$
\begin{aligned}
&= n^{-1} \cdot \left(\sum_{j=0}^{k-1} f_j \cdot \psi^{(j-k)} \cdot \left(\frac{(\psi^{2(j-k)})^n - 1}{\psi^{2(j-k)} - 1} \right) \right. \\
&\quad + f_k \cdot n \\
&\quad \left. + \sum_{j=k+1}^{n-1} f_j \cdot \psi^{(j-k)} \cdot \left(\frac{(\psi^{2(j-k)})^n - 1}{\psi^{2(j-k)} - 1} \right) \right)
\end{aligned}
$$

We notice that the inner sum is a geometric sum, which we can evaluate using our lemma `lemma_geometric_sum`. This step again requires us to use an auxiliary lemma, though we require just one as the geometric sum lemma reasons about the sum as a whole.

In this step we also rewrite the middle term. $\psi^{k-k} = 1$ by `lemma_pow_mod_int_pow0`. The term inside the sum, $(\psi^{2(k-k)})^i = (\psi^0)^i$ by simple arithmetic on F* integers, and $\psi^0 = 1$ by `lemma_pow_mod_int_pow0`. We invoke `lemma_pow_mod_int_one` to reduce the sum to a sum of ones, and finally use `lemma_sum_ones` which equates the sum of ones to `stop-start`, in this case n.

$$
\begin{aligned}
&= n^{-1} \cdot f_k \cdot n \\
&= f_k
\end{aligned}
$$

In these last steps, we see the two sums collapse to zero. If we examine the geometric sum term, we have the expression $(\psi^{2(j-k)})^n - 1$ in the numerator. We can rearrange the exponent to $(\psi^{2n})^{(j-k)}$, and by the definition of roots of unity $\psi^{2n} = 1$. The dividend then becomes $1 - 1 = 0$, and in turn the whole summed expression becomes zero. We finish off the proof with `lemma_sum_zeros` to take us to the first line above, and `lemma_pow_mod_int_add` brings us to the end of the proof $n^{-1} \cdot n^1 = n^{-1+1} = n^0 = 1$.

Now that we have proved the identity $\mathrm{INTT}(\mathrm{NTT}(f))_k = f_k$, `Classical.forall_intro` allows us to easily prove that the full polynomials $\mathrm{INTT}(\mathrm{NTT}(f))$ and f are equal.

```
val intt_ntt_is_id_kth_term:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> (f:lpoly n)
  -> (k:nat{k < n})
  -> Lemma (ensures (f.[k]) == ((intt #n #psi (ntt #n #psi f)).[k]))

let intt_ntt_is_id #n #psi f =
  Classical.forall_intro (intt_ntt_is_id_kth_term #n #psi f)
```

4.4.2 Convolution theorem

The other large lemma that we need to prove for the correctness of the NTT is the convolution theorem, which relates componentwise multiplication to multiplication in the quotient ring.

$$\mathrm{NTT}(f \cdot g) = \mathrm{NTT}(f) \odot \mathrm{NTT}(g)$$

Again, we find it easiest to reason about one term at a time. In F*, we write the convolution theorem as a sequence equality that relates the `ntt`, `mul_quotient_ring`, and `mul_componentwise` functions.

```
val convolution_theorem:
    #n:power_of_two{n < q}
  -> #psi:primitive_nth_root_of_unity_mod #q (2 * n)
  -> f:lpoly n
  -> g:lpoly n
  -> Lemma
      (requires True)
      (ensures equal
        (ntt #n #psi (mul_quotient_ring f g))
```

```
10        (mul_componentwise (ntt #n #psi f) (ntt #n #psi g)))
```

$$
\begin{aligned}
\mathrm{NTT}(f \cdot g) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \psi^{i(2k+1)} \cdot (-1)^{(i-j)/n} \cdot f_j \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} \psi^{i(2k+1)} \cdot (-1)^{(i-j)/n} \cdot f_j \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} f_j \cdot \sum_{i=0}^{n-1} \psi^{i(2k+1)} \cdot (-1)^{(i-j)/n} \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} f_j \cdot \sum_{i=0}^{n-1} \psi^{i(2k+1)} \cdot \psi^{j(2k+1)} \cdot \psi^{-j(2k+1)} \cdot (-1)^{(i-j)/n} \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} f_j \cdot \psi^{j(2k+1)} \cdot \sum_{i=0}^{n-1} \psi^{i(2k+1)} \cdot \psi^{-j(2k+1)} \cdot (-1)^{(i-j)/n} \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} f_j \cdot \psi^{j(2k+1)} \cdot \sum_{i=0}^{n-1} \psi^{(i-j)(2k+1)} \cdot (-1)^{(i-j)/n} \cdot g_{((i-j) \mod n)}
\end{aligned}
$$

Most of the convolution theorem uses very similar techniques to the reversibility lemma. The steps above use familiar lemmas including `lemma_pow_mod_int_add`, `sum_mul_lemma`, and `sum_swap_lemma` along with auxiliary lemmas and `Classical.forall_intro` to manipulate the expression to this point.

Using the fact that ψ is a $2n$th root of unity, so $\psi^n = -1$, along with the definition of modulo, we can deduce the following equality.

$$
(-1)^{(i-j)} \psi^{(i-j)(2k+1)} = \psi^{(i-j \mod n)(2k+1)}
$$

The proof details can be found in `convolution_theorem_kth_term_ok_6_rewrite`, but

involve just simple arithmetic manipulations. Using this fact we continue our proof.

$$
\begin{aligned}
&= \sum_{j=0}^{n-1} f_j \cdot \psi^{j(2k+1)} \cdot \sum_{i=0}^{n-1} \psi^{((i-j) \mod n)(2k+1)} \cdot g_{((i-j) \mod n)} \\
&= \sum_{j=0}^{n-1} f_j \cdot \psi^{j(2k+1)} \cdot \sum_{i'=0}^{n-1} \psi^{i'(2k+1)} \cdot g_{i'}
\end{aligned}
$$

The penultimate step involves reindexing our inner sum, using a new variable $i' = (i - j) \mod n$. We use `lemma_sum_shift_mod` which proves equation 4.10 to reindex.

$$
\begin{aligned}
&= \left(\sum_{j=0}^{n-1} f_j \cdot \psi^{j(2k+1)} \right) \cdot \left(\sum_{i'=0}^{n-1} \psi^{i'(2k+1)} \cdot g_{i'} \right) \\
&= \mathrm{NTT}(f)_k \cdot \mathrm{NTT}(g)_k
\end{aligned}
$$

Finally we use `sum_mul_lemma` to split the expression from a nested double sum into two sums, which we realize to be equivalent to expressions for $\mathrm{NTT}(f)_k$ and $\mathrm{NTT}(g)_k$. Since this is true for any k this completes our proof of the convolution theorem, and thus the correctness of NTT as well.

4.5 Proving the Cooley-Tukey fast NTT algorithm

Though we have proven that coefficientwise multiplication in the NTT domain yields multiplication in the quotient ring, our NTT algorithm is still $O(n^2)$, making it useful only in theory. The Cooley-Tukey algorithm allows us to convert from R_q to T_q in $O(n \log n)$ time to reap the benefits of the theorem we have proven.

To do this, we prove that for a polynomial f, $\mathrm{NTT}(f)_k = \mathrm{NTT}_{\mathrm{CT}}(f)_k$. Given this property, we are confident that using the Cooley-Tukey NTT still yields a valid NTT transformation, and the NTT correctness proof still applies.

Since many of the steps of the Cooley-Tukey proof mimic steps in the NTT reversibility and convolution theorem proofs, we do not go into as much depth about the arithmetic details of this proof. However, one major difference in our proof of the Cooley-Tukey algorithm is that we now care about performance in our code. The expression for the kth term of the Cooley-Tukey NTT is given by

$$\mathrm{NTT}_{\mathrm{CT}}(f)_k = \begin{cases} 0 & k < 0 \text{ or } k \geq n \\ \mathrm{NTT}_{\mathrm{CT}}(f_{even})_k + \psi^{2k+1} \cdot \mathrm{NTT}_{\mathrm{CT}}(f_{odd})_k & k < n/2 \\ \mathrm{NTT}_{\mathrm{CT}}(f_{even})_{(k-n/2)} & \\ \quad -\psi^{2(k-n/2)+1} \cdot \mathrm{NTT}_{\mathrm{CT}}(f_{odd})_{(k-n/2)} & k \geq n/2 \end{cases}$$

We write this in F* directly, and use the arithmetic outlined in section 3.3 to prove our desired property. However, constructing a polynomial by using this for every coefficient k is not efficient. Notice that two coefficients a and b where $b = a + n/2$ only differ in the sign of the second term, a property we can take advantage of. We write a new F* function `ntt_ct` which calculates the NTT that takes advantage of this property.

```
let rec ntt_ct
    (#n:power_of_two{2 * n < q})
    (#psi:primitive_nth_root_of_unity_mod #q (2 * n))
    (f:lpoly n)
  : lpoly n
=
  if n = 1 then f
  else begin
    power_of_two_div_two n;
    nth_root_squared_is_primitive_root #q (2 * n) psi;
    let half_n:power_of_two = n / 2 in
    assert (half_n * 2 < q);
```

```
    let odds:lpoly (n/2) = poly_odd f in
    let evens:lpoly (n/2) = poly_even f in
    let omega:primitive_nth_root_of_unity_mod #q n
      = pow_mod #q psi 2 in
    let odd_ntt:lpoly (n/2) = ntt_ct #(half_n) #omega odds in
    let even_ntt:lpoly (n/2) = ntt_ct #(n/2) #omega evens in
    let term_twos:lpoly (n/2) =
      createi (n/2) (fun k ->
        pow_mod_int #q psi (2 * k + 1) %* (poly_index #(n/2) odd_ntt
           k)) in
    createi n (fun k ->
      if k < n / 2 then poly_index #(n/2) even_ntt k +% poly_index
         term_twos k
      else poly_index #(n/2) even_ntt (k - n / 2) -% poly_index
         term_twos (k - n / 2)
    )
  end
```

This `ntt_ct` allows us to take advantage of the fact that the $n/2$-length NTTs need only be computed once, and not k times as we would have done by using the coefficient definition, which is still $O(n \log n)$ asymptotically.

Then, we prove a lemma that relates the coefficient definition to the F* implementation.

```
val ntt_ct_lemma
  (#n:power_of_two{2 * n < q})
  (#psi:primitive_nth_root_of_unity_mod #q (2 * n))
  (f:lpoly n)
  (i:int)
  : Lemma (poly_index (ntt_ct #n #psi f) i
        == ntt_ct_kth_term #n #psi f i)
```

This gives us everything we need to prove the following string of logic, which allows us to relate the Cooley-Tukey NTT to the NTT specification we proved correctness for. Note that some arguments to these functions are omitted for brevity and clarity.

$$\begin{aligned}
\texttt{poly_index}\ (\texttt{ntt_ct}\ \texttt{f})\ \texttt{k} &= \texttt{ntt_ct_kth_term}\ \texttt{f}\ \texttt{k} \\
&= \texttt{ntt_kth_term}\ \texttt{f} \\
&= \texttt{poly_index}\ (\texttt{ntt}\ \texttt{f})\ \texttt{k}
\end{aligned}$$

The Gentleman-Sande INTT algorithm is similar to the Cooley-Tukey NTT algorithm, but splits the NTT based on the first and last terms rather than by parity. The proof for this algorithm is almost complete, though there is one algebraic property remaining to prove the last part of the algorithm. The proof can be found in `Hacl.Spec.NTT.Fast.fst` at the end of the file, commented out.

Chapter 5

Implementation

The verified NTT and fast NTT are implemented on top of HACL*.

5.1 F* implementation code

We implement our representation of polynomials, along with addition and scalar multiplication in 25 lines of F* code.

The `ntt_ct`, `ntt_gs`, and `mul_componentwise` functions are all that is necessary to perform polynomial multiplication, and are implemented in 60 lines of F* code.

5.2 Proof code

The majority of the code lies in the proofs. The root of unity proofs consist of 250 lines, the summation proofs 650 lines, and the `pow_mod_int` proofs 850 lines of F* code. The F* NTT proof, including the reversibility and convolution theorems involved 1092 lines of code. The Cooley-Tukey fast NTT was proven in 726 lines of code.

5.3 Extracted OCaml

The extracted OCaml `mul_componentwise` is 14 lines of OCaml code, and the NTT and INTT code is 100 lines of OCaml code.

Chapter 6

Evaluation

A major result of this thesis is the correctness of the NTT, which we prove in F*. Since this result is motivated by performance, we measure performance of using this alternate method of multiplcation.

6.1 Performance metrics

Verified code in F* can be extracted to runnable programs in languages including OCaml and F#. When we run `fstar` and pass options to extract to OCaml, for instance, F* verifies all of the code passed to it and produces OCaml code. F* generates code only for definitions that correspond to executable code, while lemmas and proof-only code is erased after being checked. Executable code that exists only for proof purposes, such as the naive `ntt`, can be annotated to be omitted in the extracted OCaml code. F* types are converted into OCaml types, though because F* types are more expressive than OCaml, there is a loss of precision in many types. Extracting our NTT code to OCaml allows us to have code whose behavior is verified to match the specification.

There is one last proof that we require before extracting to OCaml, which is a proof of a root of unity modulo q. In our F* functions and proofs, we simply use a refinement type to represent the properties of the root of unity, but never specify a concrete integer which

satisfies those properties. In our case, 62 happens to be a 512-th root of unity of 7681, which we prove using a proof by reflection.

6.1.1 Proving a concrete primitive root of unity

The general strategy for a proof by reflection is to

1. Write a function that tests a property, returning a boolean. For example, `is_odd` or in our case, `test_primitive_root_of_unity`.

2. Prove that if the test function returns `true`, then the input does indeed exhibit that property. In our case

   `test_primitive_root_of_unity psi = true` ⇒ `is_primitive_root_of_unity psi`

Now, using the function gives us a guarantee that a true output from the function yields a primitive root of unity. In F⋆, we can use `assert_norm` which simplifies the form inside the assertion by running the function, and invoke the lemma proving the fact from step 2.

We construct a function `test_primitive_root_of_unity` from AND-ing the output from two functions `test_root_of_unity` and `test_primitive`. Then we prove a lemma to show that our test is valid.

```
let lemma_test_primitive_root_of_unity_ok (#m:prime{m > 2}) (#n:nat{n
    > 0}) (root:nat_mod m):
  Lemma (requires test_primitive_root_of_unity #m #n root)
        (ensures is_primitive_nth_root_of_unity_mod #m n root)
```

Finally, we are able to use the test along with the lemma above to prove that 62 is a 512-th root of unity mod q.

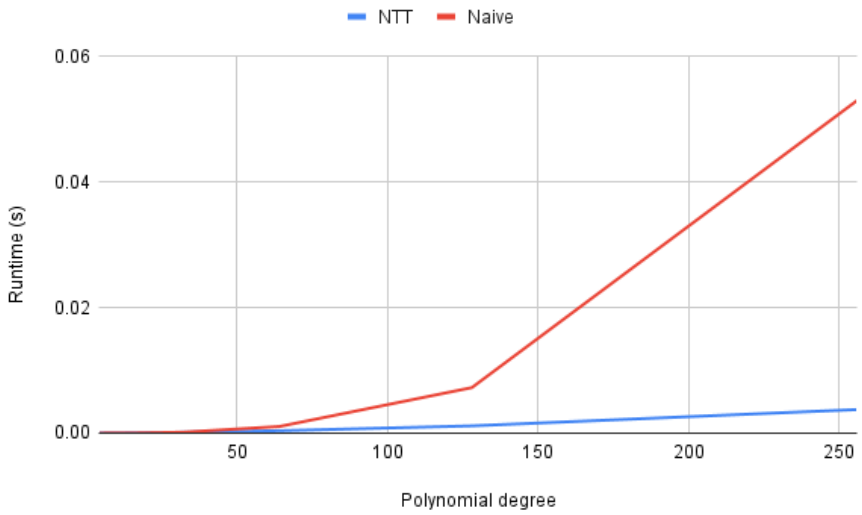
```
let root_of_unity_kyber: primitive_nth_root_of_unity_mod #q 512 =
  assert_norm (test_primitive_root_of_unity #q #512 62);
```

```
3  lemma_test_primitive_root_of_unity_ok #q #512 62;
4  62
```

6.2 Timing verified code

Polynomial multiplications using the NTT transformations were considerably faster than multiplying polynomials naively. We time multiplication using the NTT, which involves two NTT transformations, a coefficientwise multiplication on this result, and finally an INTT transformation. We compare this to the naive multiplication in the quotient ring using the coefficient definition from eq. (3.1). We ran the multiplication $10 \cdot (2^{13-n})$ times at each degree 2^n, and we report the average runtime.

Figure 6.1: Runtime of multiplying various degree polynomials using the naive definition against using the NTT. The extracted OCaml code was benchmarked on a 2021 Apple MacBook Pro with M1 Pro.



The NTT multiplication yields a much faster runtime than the naive algorithm, and a larger speed up when multiplying higher degreed polynomials. Because our NTT proof requires a $2n$-th root of unity, we cannot multiply higher degree polynomials than 256 with our chosen prime number q. While at small degrees, it is better to use the naive multiplication,

Table 6.1: Speedup of the $O(n \log n)$ NTT multiplication over the naive $O(n^2)$ quotient ring multiplication.

Degree	NTT rt (s)	Naive rt (s)	NTT / Naive NTT speedup factor
256	3.75	52.99	14.12
128	1.16	7.25	6.27
64	0.38	1.06	2.81
32	0.13	0.14	1.11
16	0.045	0.026	0.57
8	0.015	0.005	0.34
4	0.005	0.001	0.24

we get very large speedup at higher degrees. The comparatively poor performance of the NTT algorithm for small polynomials suggests that we can further optimize our algorithm by coarsening our base case.

Multiple componentwise multiplications in the NTT domain also correspond to quotient-ring multiplications. The ML-KEM/Kyber algorithm performs multiple multiplications before converting, so we can reap even more performance benefit than this initial benchmarking suggests. Note that although the Gentlemen-Sande INTT algorithm is not verified, we include proofs for most of its properties and use an unverified version.

Chapter 7

Discussion

This chapter discusses some decisions made in verifying the NTT in F*.

7.1 Prioritizing F* vs. Low*

Ultimately, the goal in a verified implementation of the NTT is to be able to use the verified code in production with guarantees about its correctness. An implementation in Low*, the F* subset that compiles to low level code such as C or Rust, would provide a more direct application to real world systems. Low* proofs often involve several layers to modularlize the proof process.

1. A high-level specification in F*, working with arbitrary precision integers. In this case, our `ntt` function along with structures like sums and roots of unity.

2. Proofs and lemmas about this specification in F*. This includes all of the correctness proofs.

3. A specification in Low*.

4. A proof that the Low* specification behaves in correspondence with the high-level F* specification.

It is not necessary to follow this order exactly. While proofs are dependent on specifications, the Low* proof does not necessarily depend on F* proofs. For our proof of the NTT, we could have written a Low* implementation of the NTT and proved that it matches the F* NTT before proving the necessary correctness theorems.

We prioritized the proof of correctness for the NTT over the Low* implementation. While we are not able to run code that is as performant, proving that we can write a function using buffers that matches one using F* sequences is not as interesting of a result. The memory safety that Low* provides has similar guarantees to carefully written code in a language like Rust. As a proof of concept, we have completed a Low* implementation of polynomials using fixed-length buffers, along with a proof of correctness for addition and scalar-multiplication.

Chapter 8

Future work

8.1 The Gentleman-Sande INTT algorithm

The first obvious piece of work is to finish the proof of the Gentleman-Sande INTT algorithm. While the majority of the algebraic proof is complete, there is one unverified property about the symmetry and periodicity of the modular inverse of a root of unity required for the full proof.

Completing this proof allows us to perform efficient multiplications on polynomials with confidence.

8.2 A verified Low* implementation

A Low* implementation and proof provides a major step towards production-ready code with the ability to extract into C using the KaRaMeL compiler. As discussed in section 7.1, the proof involves a Low* specification along with a proof that the Low* code matches the F* code. The proofs of correctness done in pure F* provide guarantees that the Low* code is correct.

Often times, it is beneficial to introduce another intermediate layer between the high-level F* specification and the Low* specification. This layer is another F* specification that more

closely matches the Low* execution order. This allows the engineer to do more of the proof in F*, where it is often easier to reason about and manipulate expressions. For example, this may be particularly useful for the recursive fast-NTT algorithms. The performant approach is to repeatedly scan the polynomial, which is difficult to map directly to the recursive approach that we specified in F*.

We have started the Low* implementation, complete with representations of $\mathbb{Z}_q$ including constant time addition, subtraction, and multiplication, polynomials of integers mod q, and addition and scalar multiplication of these polynomials. We also include a verified precomputation table, which can be easily modified to accommodate any integer of the form a^b, which we use for values ψ^x without the need for modular exponentiation. Because of the property $\psi^{2n} = 1 \mod q$, we can perform any modular exponentiation of ψ very efficiently using this table.

Chapter 9

Related work

9.1 Verified implementations of NTT multiplications and Kyber

Hwang et al provide a verified implementation of NTT multiplications using CryptoLine, a tool and language for verifying low-level implementations of mathematical constructs [10]. Almeida et al. presented the first formally verified implementation of Kyber, including the NTT using EasyCrypt and Jasmin [11].

9.2 HACL*

HACL* is a verified cryptography library written in F*. HACL* primitives are as fast as the fastest C code in OpenSSL, and 1.1 to 5.7x slower than the fastest hand-vectorized assembly in SUPERCOP [7]. HACL* has a verified implementation of Frodo-KEM, a post-quantum lattice-based algorithm that was selected as a round 3 candidate in the NIST Post-Quantum Cryptography Standardization project, but was not selected for standardization. Frodo-KEM does not use polynomials or the NTT in its implementation.

While many cryptographic algorithms are implemented and verified in HACL*, the NTT

is not among them. We build our NTT and proofs using the HACL* library.

Chapter 10

Conclusion

This thesis provides an F* proof of correctness of the NTT to allow fast polynomial multiplication in the quotient ring. This thesis also implements several modular arithmetic concepts, including sums, roots of unity, and modular inverse, which have many applications in cryptography. We also present the start of the Low* NTT implementation, leaving a well-defined path towards the full implementation.

References

[1] National Institute of Standards and Technology (NIST), *Cve-2022-3602 detail*, https://nvd.nist.gov/vuln/detail/CVE-2022-3602, [Accessed: 2024-07-29], 2022.

[2] National Institute of Standards and Technology (NIST), *Post-quantum cryptography*, https://csrc.nist.gov/projects/post-quantum-cryptography, [Accessed: 2024-07-29], 2023.

[3] A. Satriawan, R. Mareta, and H. Lee, *A complete beginner guide to the number theoretic transform (NTT)*, Cryptology ePrint Archive, Paper 2024/585, https://eprint.iacr.org/2024/585, 2024. DOI: 10.1109/ACCESS.2023.3294446. [Online]. Available: https://eprint.iacr.org/2024/585.

[4] K. Kreuzer, *Verification of the (1–δ)-correctness proof of CRYSTALS-KYBER with number theoretic transform*, Cryptology ePrint Archive, Paper 2023/027, https://eprint.iacr.org/2023/027, 2023. [Online]. Available: https://eprint.iacr.org/2023/027.

[5] National Institute of Standards and Technology (NIST), "Module-lattice-based key-encapsulation mechanism standard," National Institute of Standards and Technology, Tech. Rep., 2023, [Accessed: 2024-07-29]. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf.

[6] N. Swamy, G. Martínez, and A. Rastogi, *Proof-oriented programming in F**, [Accessed: 2024-07-29], 2024. [Online]. Available: https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf.

[7] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, *HACL*: A verified modern cryptographic library*, Cryptology ePrint Archive, Paper 2017/536, https://eprint.iacr.org/2017/536, 2017. [Online]. Available: https://eprint.iacr.org/2017/536.

[8] P. Longa, *FrodoKEM round 3: A simple and conservative KEM from generic lattices*, https://csrc.nist.gov/CSRC/media/Presentations/frodokem-round-3-presentation/images-media/session-6-frodokem-longa.pdf, Presentation at the NIST PQC Standardization Process, [Accessed: 2024-07-29], 2021.

[9] P. Schwabe, *CRYSTALS - Kyber*, https://pq-crystals.org/kyber/index.shtml, Accessed: 2024-07-29.

[10] V. Hwang, J. Liu, G. Seiler, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Verified ntt multiplications for nistpqc kem lattice finalists: Kyber, saber, and ntru," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 718–750, Aug. 2022. DOI: 10.46586/tches.v2022.i4.718-750. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9838.

[11] J. B. Almeida, M. Barbosa, G. Barthe, *et al.*, *Formally verifying kyber episode IV: Implementation correctness*, Cryptology ePrint Archive, Paper 2023/215, https://eprint.iacr.org/2023/215, 2023. [Online]. Available: https://eprint.iacr.org/2023/215.