Asymmetric Cryptography from Number Theoretic Transformations

Knapsack Reduction Pre-print

Samuel Lavery

sam@trustlessprivacy.com
sam@quantumshield.us
Trustless Privacy Inc. (USA)

February 10, 2024

Abstract

In this work, we introduce a family of asymmetric cryptographic functions based on dynamic number theoretic transformations with multiple rounds of modular arithmetic to enhance diffusion and difficulty of inversion. This function acts as a basic cryptographic building block for a novel communication-efficient zero-knowledge crypto-system. The system as defined exhibits partial homomorphism and behaves as an additive positive accumulator. By using a novel technique to constructively embed lattice problems in a nested fashion, the dimensionality and overall complexity of the lattice structure is increased.

This linked lattice framework obscures internal structure and mitigates cryptanalysis by applying a novel 'noisy roots' technique. By relaxing the need for specifically correct nth ω roots in a given field, we apply offset values to create a framework of consisting of a set of uniquely transforming but arithmetically compatible NTTs. We provide specific parameters for conjectured NIST level V security. Communication costs are extremely low at 288-bytes per public key and 144-bytes per cipher-text or digital signature. Example protocols for key agreement, secure data exchange, additive accumulation, and digital signatures are provided.

Peer review is in preliminary stages at time of dissemination. Claims within have not undergone rigorous validation and likely contain inaccuracies, errors, flaws or incomplete analysis. Contents may see significant modification through later iterations.

Contents

1 Introduction

This paper introduces and explores a novel construction, examining its theoretical underpinnings, practical implementation considerations, and potential impact on the evolving field of post-quantum cryptography. Lattice cryptography and number theoretic transforms[11] (NTTs) are complex topics generally less understood than systems based on the multiplication of large primes or hash-based trees. For readers without a PhD in abstract algebra or a related field, a brief recommended review of lattice cryptography can be found in the appendix. For lack of a cooler name, we call this function ZKVolute.

1.1 Communication Cost Prevents Adoption

In the realm of cryptography, the emergence of quantum computing represents a significant problem[12], necessitating a re-evaluation of established cryptographic primitives and methods. Traditional cryptographic systems, robust against current computational threats, are going to fail in the face of quantum algorithms. Lattice-based cryptography [13], known for its potential quantum resilience, emerged as an early frontrunner candidate for next-generation solutions. However, broad adoption has been hindered by significantly increased communication overhead compared to existing implementations.

The issue is simply that modern systems and protocols were not designed to support the overhead required by these systems [1], with keys and cipher-texts orders of magnitude larger than those in current suites of cryptography. The scope of this problem is hard to overestimate, as any environment constrained by a fixed size hardware IO buffer, radio frequency link budget, acoustic limitations, or other immutable physical factors will not be able to leverage the current lattice-based cryptographic solutions under NIST consideration. Without alternative options, the cryptographic community has resorted to proposals that dramatically weaken secure primitives such as SPHINCS+ to reduce communication costs, but still fail to achieve practical parity.

This problem extends beyond just securing data in motion. Applying quantum-resilient digital signatures to ensure the authenticity of data at rest presents a second serious issue. While mitigated by additional storage and compute capacity, this remains a prohibitively expensive solution. Without an alternative to DILITHIUM [8] for machine learning and data warehousing scenarios, the authenticating signatures may vastly outsize the data being authenticated.

1.2 Comparison of Variable Sizes

This work introduces the $\mathcal{F}_{\text{ZKVolute}}$ family of cryptographic functions, characterized by compact variable sizes and efficient communication protocols presents a new option to consider for systems unable to adopt proposed NIST standards. This section details the variable size of $\mathcal{F}_{\text{ZKVolute}}$ against the draft alternatives.

Table 1: Post-Quantum Lattice-Based KEMs in Round 3 and Our Work

Scheme Name	Security Level	Public Key Size (bytes)	Ciphertext Size (bytes)
Kyber	I/III/V	800/1184/1568	768/1088/1568
NTRU	I/III/V	699/930/1230	699/930/1230
Saber	I/III/V	672/800/992	736/864/1088
FrodoKEM	I/III/V	976/1984/3120	1088/2096/3360
This Work	V	288	144

For constrained networks the future for secure communication is grim at best. The compact variable sizes of $\mathcal{F}_{\text{ZKVolute}}$, combined with its inherent zero-knowledge cryptographic properties present unexplored and potentially fertile terrain for future cryptographic research and development.

2 DRAFT - Initial Implementation Parameters

We describe a cryptographic system featuring a series of sequential NTT transformation, layers of modular addition and dynamic 'noisy' roots of unity. At a high level during each stage s, the uniquely transformed input representations are added mod p_s to a set of 'carry over' coefficients. This 'carry over' vector is transformed at each stage as well. At the end of the process this value is reduced by NTT inversions to the final output proof. A 'base' set of roots of unity are defined below and used for transforming the carry over polynomial vector coefficients at each forward stage and are used for each NTT inversion transform. While each stage s of the transformation uses the same finite field $\mathbb{F}_{ps}$ modulus parameters(defined for NTT1-7 in the example below), the roots of unity are not fixed. Dynamic 'noisy ω root' offset values are applied to create ωA,$\omega J, \omega J'$ during cryptographic proof generation. These dynamically parameterized NTTs create an opaque internal structure acting as an algebraic mask on the input values during proof π(also referred as O, σ or ψ) generation. These offset roots of unity obscure internal structural patterns and complicate attempts at algebraic and lattice basis reduction attacks. This process of expansion and reduction resembles the absorb and squeeze phases of the Keccak sponge construct. For the reference implementation 'base' ω values and prime p moduli values are defined below. These variables define the dimensionality and internal lattice structures over which the system is constructed. The reference implementation use 128 degree input/output polynomial vectors with 9-bit coefficients, increasing in size across 7 stages 'gradually' from 9-bit to 1023-bit, and then reduced back to 9-bit coefficients for output.

- NTT1: $p_1 = 257$
  $\omega 1 = 9$
- NTT2: $p_2 = 133733557800380723707235713$
  $\omega 2 = 64067737555402157637662896$
- NTT3: $p_3 = 3674486483990229966264985502154881$
  $\omega 3 = 1890847646321174403442558538216985$
- NTT4: $p_4 = 277607387889041068344634698360622849$
  $\omega 4 = 36911849899402943224960606644039613$
- NTT5: $p_5 = 46892497174283636007114170355627877121$
  $\omega 5 = 37253208155173144133957004832098928$
- NTT6: $p_6 = 3001732422913219376338819698366525492620301042048305923969$
  $\omega 6 = 398896133189425383704948920182858832658280341425750379731$
- NTT7: $p_7 = 118744335938510172901865837048012915675869744546731752058788813398407843194772622919696372887588295330577087836583448696260068152634969521002231254822337254541535132698518177689925659970852178836369012022087730057974652696089279343672884959353530300970377$

4970553525520293944982922003884945755629517640228056333

$\omega 7 =$ 360022047507013676821675038584319752886113651654286732753841764735446170119896949509051754276630785838077393016410591769135799910936396038902011815047992316316966350988653835178243395444905363387470757773009838675129340226509168574348475223341098160891231276673182681605412136139697986848728012980039972458 12

2.1 Noisy Roots of Unity

Normally, besides the modulus, NTTs are parameterized by a carefully chosen primal root of unity, ω. Specifically the primal *nth* root of unity for the NTT is required, where n is the degree of the polynomial the NTT is configured to transform. For example, in NTT1 we use $\omega = 9$ as a 128th root of unity. In a normal system, the *nth* root of unity is constrained such that $(\omega^n \mod p \equiv 1)$ and the primal root is the smallest ω value that satisfies this condition. The modulus, degree, and root of unity are what define the characteristics of the set of lattices.

Proposition 1. *In the context of the number theoretic transformation the existence and utilization of the correct primal nth root of unity is only a requirement for true NTT convolutions that perform point-wise multiplication over a given field. For transformations that do not perform multiplication such as those involving modular addition, the requirement that* $(\omega^n \mod p \equiv 1)$ *is relaxed. Inaccurate, or noisy values for* ω *can be used in such systems.*

In existing lattice based cryptographic systems the NTTs use carefully selected moduli and roots, and once set these values generally do not change. Typically, computational efficiency and ability to leverage algebraic optimizations and/or available vector math hardware accelerated opcodes are major driving factors in this selection. As these values generally do not change, the 'algebraic structure' of the crypto-system becomes an embedded constant. This constant internal structure enables certain algebraic and lattice reduction attacks as these forms of differential cryptanalysis require the ability to determine if one result is 'closer' to the target than another. This dithering process enables these algorithms to hone in on a solution by discarding solution branches resulting in more distant outputs.

Conjecture 1. *Algorithms such Lenstra–Lenstra–Lovász (LLL) lattice basis reduction uses Gram–Schmidt orthonormalization. This process assumes that the vectors have elements that exist in same inner product space to make a meaningful comparison. By not using constant roots of unity, vectors are projected over different lattices. Gram-Schmidt requires the ability to produce a mapping of common vector space before analysis. By not revealing the dynamic roots chosen for secret inputs, the common inner product mapping requirements are unknown. The inability to build a common map makes orthonormalization difficult, due to lack of information.*

It's important to note that offsets are not applied to all inputs and not for every mode of operation. Given the function ZKVolute(x,y) we have three possible offset modes to consider.

Like other lattice based systems, it is possible for an adversary to pass in a very sparse input (mostly 0 values) as an attempt to leak secret bits during internal addition or multiplication of values. This extra second representation functionality eliminates this attack vector by taking a potentially sparse input, extracting randomness and generating a second related input that is statistically indistinguishable from random and has an extremely low likelihood of also being considered dangerously sparse. First came One input, that gave way to Two inputs, combined with the secret Third input, we get everything. By ensuring adversarial inputs are internally mixed with a randomized representation, the ability to mount a polynomial attacks is essentially eliminated. However, not all inputs are expanded into whitened representations. To control this functionality, an integer flag w is set as the 3rd input to the core function. w=0 means no separate representations, w=1 means the second input is expanded and is the most common, w=2 means both inputs are expanded.

To keep the interface simple, we define three versions of our ZKVolute function below:

- ZKVolute0(x,y,w) - Offset values are not applied to either input, w is generally 0 in this case.

- ZKVolute1(x,y,w) - Offset values are only applied to second input y, w is generally set to 1 in this case.
- ZKVolute2(x,y,w) - Offset values are applied to both x and y inputs, w is mostly set to 1 in this case as well.

3 Algorithms and Definitions

3.1 NIZK PKE(M)NO Overview

The public key encryption with non-interactive opening (PKENO), defined by Damgård et al[5], refined by Galindo[9] is not widely known, nor is it a consistently defined concept. Our work continues this trend of not conforming to prior definitions to create a practical non-interactive zero knowledge (NIZK) variant. There is no single way to construct a PKENO, each is defined slightly differently depending on the internal primitives used. Systems based on Decisional Diffie-Hellman for core cryptographic strength inherit the probabilistic nature of DDH.

In contrast, this work is default deterministic with the possibility to introduce randomness if required. Another crucial difference between currently known PKENOs is their support of 'direct' encryption and decryption of payloads, enabling recovery of message m. This is also a feature of DDH based systems. The notion of decryption of a proof variable is not present in our system. As these proof values function as additive accumulators, the calculations only go in the forward direction. Similar to other lattice systems a form of key encapsulation and shared secret derivation is a required intermediate step for encryption/decryption. This shared secret is used to key an appropriate symmetric cipher such as AES. Due to this extra required step, this work can more accurately be described as a post-quantum Public Key Exchange Mechanism with Non-Interactive Zero-Knowledge Opening (PKEMNO), NIZK variant going forward.

3.2 Core Concepts and Properties

This number theoretic primitive is a convolution-like additive combiner of inputs. From the final convolutional output one can prove in polynomial time knowledge of a secret input value, while making recovery of that exact input challenging. This is also known as a proof of possession (PoP), or knowledge of secret key (KOSK) style system.

A conceptually easy way to view the π proof system is as a bottomless paint can. This is filled with a variety of separate pigments, some secret, some public, mixed together to create a unique shade. Each new addition changes what appears on the brush and the wall. If you know which specific pigments were used to create a color, it can be exactly reproduced. Without knowledge of the original inputs, reproduction of the final output is difficult.

This additive accumulator construct yields a primitive with a set of desirable properties:

- **Zero Knowledge** - $\pi \leftarrow\!\!\$\, f(x,y)$ where x is secret and y and π are both known should not enable an adversary to learn anything beyond the validity of the proof demonstrating knowledge of x.
- **Convolution-like** - A convolution in the mathematical sense of two functions g and h produces a third function $(g * h)$, and is defined as the product of the two functions after a specific series of transformations. While our work avoids the point-wise multiplication required for true convolution, the end result preserves the key convolutional property of commutativity.
- **Additive Accumulator** - The notion of the 'one-way accumulator' defined by Benaloh[2] has existed for some time. These primitives act as a compact and privacy preserving mechanism for set membership testing.
- **Commutativity as Equivariance Test** - Our construction exhibits the commutative property where $f(f(a,b),c) = f(f(a,c),b)$ for secret a and known values b and c. This can serve as an equivariance test to verify if the creator of two accumulated values knows the secret a used in the commutative accumulation function $f()$. By checking if $f()$ commutes properly using different known values, the secret holder can demonstrate knowledge of a in zero knowledge.
- **'Somewhat' Homomorphic** - As a lattice system, like many others, this work has inherited 'somewhat' or 'partial' homomorphism. Multiplication and division are not supported, arithmetic addition is.

- .
- **'Lossy' Compressor** - As each convolution of values results in an output with less than half of the information entropy of the inputs, it is considered a lossy compressor. From an information theoretic perspective this preserves the notion of the one way trapdoor function, as information is destroyed, lost during the internal modular reductions.
- **Intrinsically Authenticated** - Unlike most PKE systems, the PKEMNO construct enables the public key to support multiple operations including signature validation, key establishment via derivation of shared secrets, and public authentication of cipher-text proofs and payloads. Operations are performed 'under' a given public key or set of keys producing a separate authenticating proof ψ. This ψ value can be combined with the payload, π key exchange proof, and origin/destination public key values to validate integrity and authenticity for multiple configurations.

3.3 Key-pair Notation

In the function definitions and algorithmic descriptions below, a variable labeled pk represents the entire 288-byte public key, pk' represents the public homomorphic 144-byte convolution and C is the 144-byte common reference vector only. Notation:

- sk - Secret vector of coefficients
- C - Public reference vector - Specific to a given key-pair, similar to common reference strings found in other ZK systems.
- pk' - First half of the public key, contains convolution of sk and C.
- pk - Full public key with both public variables concatenated as $\mathsf{pk}'||C$

4 Modes of Operation - Functional Interfaces

4.1 Naked Key Agreement

The more complex functional use cases require protocol and API adaptations. Simple operations such as signature generation and validation do not. We start by defining a naked key agreement protocol. Alice and Bob need to derive the same shared secret for use at a later time. To derive shared secret SS between A and B the following functional interface is used:

- 1a. KeyGen(1^λ) $\rightarrow$ (sk, pk) - Returns a key-pair given a security parameter defined by 1^λ. We will use sk_A, pk_A, C_A and sk_B, pk_B, C_B as A and Bs keys and public reference vectors below.
- 2a. Encapsulate($\mathsf{sk}_a, \mathsf{pk}_b$, C_A) $\rightarrow \pi$ - Returns key establishment proof.
- 2b. Sender_Decapsulate($\mathsf{sk}_A, \mathsf{pk}_B, C_A$) $\rightarrow SS\_A$ - Tx-side algorithm
- 3a. Receiver_Decapsulate(sk_b, C_A, π) $\rightarrow SS\_B$ - Rx-side algorithm

4.2 Authenticated Key Agreement with Non-Interactive Opening

Next we define a scenario where the π proof value can be authenticated and validated by a third party. Alice and Bob need to derive the same shared secret that provably originates from Alice, for use at a later time. To derive shared secret SS between A and B the following functional interface is defined:

- 1a. KeyGen(1^λ) $\rightarrow$ (sk, pk) - Returns a key-pair given a security parameter defined by 1^λ. We will use sk_A, pk_A, C_A and sk_B, pk_B, C_B as A and Bs keys and public reference vectors below.
- 2a. Encapsulate($\mathsf{sk}_a, \mathsf{pk}_b$, C_A) $\rightarrow \pi$ - Returns key establishment proof.
- 2b. Sender_Decapsulate($\mathsf{sk}_A, \mathsf{pk}_B, C_A$) $\rightarrow SS\_A$- Tx-side algorithm
- 3a. Receiver_Decapsulate(sk_b, C_A, π) $\rightarrow SS\_B$ - Rx-side algorithm
- 4a. Prove(sk_A, π, C_B) $\rightarrow \psi$) - Returns opening value to validate proof authenticity.
- 5a. Verify($\mathsf{pk}'_A, \psi, \pi, C_B$) $\rightarrow$) (true or $\perp$))

4.3 Key Agreement with Encrypted Data and Non-Interactive Opening

Next is a key exchange with secure transmission of an encrypted plain-text m from A to B. Alice needs to send Bob a secret message now, not at a future time. Unfortunately Bob gets a suspicious number of secret messages, so he's hired Charlie to sort through them and only give him ones from Alice. Similar to a postal envelope, only information about the sender and receiver are available for Charlie to confirm. The message contents can only be decrypted using Bob's secret key. To send a secret message we define the PKEMNO functions below:

- 1a. KeyGen(1^λ) $\rightarrow$ (sk, pk) - Returns a key-pair given a security parameter defined by 1^λ. We will use sk_A, pk_A, C_A and sk_B, pk_B, C_B as A and Bs keys and public reference vectors below.
- 1b. Setup(pk_A, m) $\rightarrow H_m$ - Generates a compressed representation of m by application of keyed hash function binding to public variables.
- 2a. Encapsulate($\mathsf{sk}_a, \mathsf{pk}'_B, H_m$) $\rightarrow \pi$ - Returns key establishment proof.
- 2b. Sender_Decapsulate($\mathsf{sk}_A, \mathsf{pk}'_B, C_A, H_m$) $\rightarrow SS\_A$ - Sending side shared secret derivation.
- 2c. Encrypt(m, SS_A) $\rightarrow m_{enc}, H_{enc}$ - Returns encrypted payload and a binding hash value authenticating the encrypted payload.
- 3. Prove($\mathsf{sk}_A, \pi, C_B, H_{enc}$) $\rightarrow \psi$ - Returns opening value to validate payload authenticity.
- 4. Verify($\mathsf{pk}'_A, \pi, C_B, H_{enc}$) $\rightarrow (\psi')$ Charlie determines if Alice sent this payload or not by checking if the given ψ was created under pk_A using m_{enc} convolved with π. Charlie convolves (ψ, C_A, π,C_B, H_{enc}) as ψ_0 and tests equality by computing ψ' as the convolution of pk'_A, ψ, π,C_B, H_{enc}. If it matches, it's from Alice.
- 5a. Receiver_Decapsulate(sk_b, π) $\rightarrow SS\_B$ - Receiver side shared secret derivation.
- 5b. Decrypt(SS_B, m_{enc}) $\rightarrow (m, H_m)$ - Returns decrypted payload and validation hash(optional).

4.4 Digital Signatures

To generate a signature σ on plaintext message m using a given key-pair.

- 1 KeyGen(1^λ) $\rightarrow$ (sk, pk) - Returns a key-pair given a security parameter defined by 1^λ. As only one party is signing will use sk_A, pk_A, C_A below.
- 2 Sign(sk_A, m) $\rightarrow \sigma$ Returns a signature σ on message m.
- 3 Verify(pk_A, σ, m) $\rightarrow$ (true or $\perp$)

4.5 Authenticated Accumulator

- 1 KeyGen(1^λ) $\rightarrow$ (sk, pk)
- 2 Create($\mathsf{sk}, x_0, \ldots, x_n$) $\rightarrow (\pi, \psi)$ - returns new accumulator π
- 3 Update($\mathsf{sk}, \pi, \psi, x_0, \ldots, x_n$) $\rightarrow (\pi, \psi)$ - returns updated accumulator π
- 4 Authenticate(pk, π, ψ) $\rightarrow$ (true or $\perp$)
- 5 Validate($\mathsf{pk}, \pi_0, \psi, \pi_1, x_0, \ldots, x_n$) $\rightarrow$ (true or $\perp$) - validates π_0 was updated properly to create π_1.

5 Modes of Operation - Implementations

5.1 Helper Definitions

Algorithm 1 PolyVecToBin

Input:

- Y: Polynomial coefficient vector
- d: Degree of polynomial
- cs: Coefficient size in bits - assume reference is 9-bit

Output:

- B: Binary string

```
function PolyVecToBin($Y, |Y|, cs$)
    $B \leftarrow$ empty string
    for $i \leftarrow 0$ to $d$ do
        $b_i \leftarrow$ CoeffToBin($Y[i], cs$)
        $B \leftarrow B|b_i$
    end for
    return $B$
end function
```

Algorithm 2 CoeffToBin

```
function CoeffToBin($c, cs$)
    $b \leftarrow$ empty bit string
    for $i \leftarrow 0$ to $cs - 1$ do
        if $c \geq 2^{cs-i-1}$ then
            $b \leftarrow b|$ '1'
            $c \leftarrow c - 2^{cs-i-1}$
        else
            $b \leftarrow b|$ '0'
        end if
    end for
    return $b$
end function
```

Algorithm 3 ParseBits

```
function ParseBits($b, cs$)
    $c \leftarrow 0$
    for $i \leftarrow 0$ to $cs - 1$ do
        if $b[i] =$ '1' then
            $c \leftarrow c + 2^{cs-i-1}$
        end if
    end for
    return $c$
end function
```

Algorithm 4 BinToPolyVec

Input:

- B: Binary string of length $n * cs/8$ bytes
- cs: Coefficient size in bits - assume reference is 9-bit

Output:

- Y: Polynomial coefficient vector

```
1: function BinToPolyVec($B, cs$)
2:     for $i \leftarrow 0$ to $n * cs/8 - 1$ do
3:         $c_i \leftarrow$ ParseBits($B[i : i + cs - 1], cs$)
4:         Append $c_i$ to vector $Y$
5:     end for
6:     return $Y$
7: end function
```

This algorithm follows the convention of using the pk to key a secure cryptographic hash function H. For simplicity we assume an XOF hash function such as SHAKE256.

Algorithm 5 GenRelatedPoly

Input:

- Y - Polynomial coefficient vector
- b - Coefficient size in bits
- pk - Public key

Output:

- Y' - Related polynomial coefficient vector

```
1: function GenRelatedPoly($Y, pk, b$)
2:     Parse $pk$ as $(pk', c)$                                ▷ Get public key parts
3:     $B \leftarrow$PolyVecToBin($Y, |Y|, b$)                 ▷ Convert to binary
4:     $KH() \leftarrow$KeyedHash(pk$||c$)                      ▷ Key hash function
5:     $B' \leftarrow$KH($B$)                                   ▷ Hash binary string
6:     $Y' \leftarrow []$                                       ▷ Empty coefficient vector
7:     for $i \leftarrow 0$ to $n - 1$ do
8:         $c_i \leftarrow$ParseBits($B'[i : i + b - 1], b$)    ▷ Extract/parse next $b$ bits
9:         Append $c_i$ to $Y'$
10:    end for
11:    return $Y'$
12: end function
```

Algorithm 6 Roots Of Unity Offset Generation

Input:

- Y - Polynomial coefficient vector
- pk_a - Public key A
- l - Number of offsets
- b - coefficient size in bits
- r - Offset sample range

Output:

- offsets - Array of l offset values

```
1:  function RootGen($Y, \mathsf{pk}_a, l, r, b$)
2:      $B \leftarrow$PolyVecToBin($Y, |Y|, b$)                    ▷ Convert polynomial to binary
3:      $numBytes \leftarrow \lceil b/8 \rceil$                      ▷ Number of bytes per coeff
4:      $KH \leftarrow$KeyedHash($\mathsf{pk}_a || C_a || B, numBytes$)   ▷ Key hash XOF function, outputs $numBytes$
5:      offsets $\leftarrow$ empty array of size $l$
6:      for $i \leftarrow 0$ to $l - 1$ do
7:          $r_i \leftarrow$KH.Digest(())                           ▷ Get random bits from KH
8:          $o_i \leftarrow (r_i \bmod 2r) - r$                      ▷ Map to range $-r, ..., r$
9:          offsets[$i$] $\leftarrow o_i$
10:     end for
11:     return offsets
12: end function
```

Algorithm 7 Pointwise Addition

Input:

- $polyVec$ - input poly vector of n coefficients
- $polyVec2$ - input poly vector of n coefficients
- $modulus$ - NTT modulus defining the field

Output:

- $polyOut$ - point-wise sum of vectors coefficients, wrapped around modulus

```
1:  function PointwiseAdd(polyVec, polyVec2, modulus)
2:      $polyOut \leftarrow$ empty list
3:      for $i = 0$ to $n$-1 do
4:          $x \leftarrow polyVec([)i]$
5:          $y \leftarrow polyVec2([)i]$
6:          $sum \leftarrow (x + y)$ mod $modulus$
7:          append($polyOut$, $sum$)
8:      end for
9:      return $polyOut$
10: end function
```

5.2 ZKVolute Functions

The functions for computing the forward NTT and inverse NTT are standard Cooley-Tukey (CT) butterfly based and example python implementations are included in the appendix. The RNDS constant represents the number of rounds of modular addition performed after the first stages. This increases diffusion, but higher values tend to become proof incomplete as noise accumulates. Recommended value is is 3-10 rounds.

Algorithm 8 ZKVolute0 - No noisy roots

Constants:

- l - Number of stages of levels
- MODULI - l item array of NTT moduli
- ROOTS - l item array of NTT base roots of unity
- b - size of coefficients in bits
- pk - our public key
- RNDS - Number of rounds to perform - default 5

Input:

- x - First input poly
- y - Second input poly
- w - Whitening flag

Output:

- π - convolution like representation of inputs

```
function ZKVOLUTE0($x, y, w$)
    $\pi[] \leftarrow 0$                                          ▷ Define empty output polyvec
    If $w = 1$:                                                   ▷ Create representation for second input
        $y' \leftarrow$ GENRELATEDPOLY($y$, pk, $b$)
    ElseIf $w = 2$:                                               ▷ Create representation for both inputs
        $y' \leftarrow$ GENRELATEDPOLY($y$, pk, $b$)
        $x' \leftarrow$ GENRELATEDPOLY($x$, pk, $b$)
    EndIf
    for $i \leftarrow 0$ to $l - 1$ do                            ▷ Begin the transforms
        $x \leftarrow$ NTT($x$,MODULI[$i$],ROOTS[$i$])
        $y \leftarrow$ NTT($y$,MODULI[$i$],ROOTS[$i$])
        If $w = 1$:
            $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$])
        ElseIf $w = 2$
            $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$])
            $x' \leftarrow$ NTT($x'$,MODULI[$i$],ROOTS[$i$])
        If $i > 0$:                                               ▷ Need to transform the output
            $\pi \leftarrow$ NTT($\pi$,MODULI[$i$],ROOTS[$i$])
        If $i < 2$:
            $\pi \leftarrow$ POINTWISEADD($\pi, y$,MODULI[$i$])
            If $w \neq 0$:                                        ▷ If we have alternates, add them
              $\pi \leftarrow$ POINTWISEADD($\pi, y'$,MODULI[$i$])
        If $i \geq 2$:
        for $j \leftarrow 0$ to RNDS$-1$ do
            $pi \leftarrow$ POINTWISEADD($\pi, x$,MODULI[$i$])
            $pi \leftarrow$ POINTWISEADD($\pi, y$,MODULI[$i$])
            If $w = 1$:                                           ▷ If we have alternates, add them
              $pi \leftarrow$ POINTWISEADD($\pi, y'$,MODULI[$i$])
            If $w = 2$:                                           ▷ If we have alternates, add them
              $pi \leftarrow$ POINTWISEADD($\pi, y'$,MODULI[$i$])
              $pi \leftarrow$ POINTWISEADD($\pi, x'$,MODULI[$i$])
        end for
    end for
    for $i \leftarrow (l - 1)$ to $0$ do                          ▷ Begin inversions
        $\pi \leftarrow$ NTT_INV($\pi$,MODULI[$i$],ROOTS[$i$])
    end for
    return $\pi$
end function
```

Algorithm 9 ZKVolute1 - Noisy roots on second input

Constants:

- l - Number of stages of levels
- MODULI - l item array of NTT moduli
- ROOTS - l item array of NTT base roots of unity
- b - size of coefficients in bits
- pk - our public key
- RNDS - Number of rounds to perform - default 5

Input:

- x - First input poly
- y - Second input poly
- w - Whitening flag

Output:

- π - convolution like representation of inputs

```
function ZKVolute1($x, y, w$)
    $\pi[] \leftarrow 0$                                              ▷ Define empty output polyvec
    $y_{OFF} \leftarrow$ RootGen($y, \mathsf{pk}_a, l, r, b$)
    If $w = 1$:                                                   ▷ Create representation for second input
        $y' \leftarrow$ GenRelatedPoly($y, \mathsf{pk}, b$)
        $y'_{OFF} \leftarrow$ RootGen($y', \mathsf{pk}_a, l, r, b$)
    ElseIf $w = 2$:                                               ▷ Create representation for both inputs
        $y' \leftarrow$ GenRelatedPoly($y, \mathsf{pk}, b$)
        $x' \leftarrow$ GenRelatedPoly($x, \mathsf{pk}, b$)
        $y'_{OFF} \leftarrow$ RootGen($y', \mathsf{pk}_a, l, r, b$)
    EndIf
    for $i \leftarrow 0$ to $l - 1$ do                                  ▷ Begin the transforms
        $x \leftarrow$ NTT($x$,MODULI[$i$],ROOTS[$i$])
        $y \leftarrow$ NTT($y$,MODULI[$i$],ROOTS[$i$] $+ y_{OFF}[i]$)
        If $w = 1$:
            $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$] $+ y'_{OFF}[i]$)
        ElseIf $w = 2$
            $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$] $+ y'_{OFF}[i]$)
            $x' \leftarrow$ NTT($x'$,MODULI[$i$],ROOTS[$i$])
        If $i > 0$:                                               ▷ Need to transform the output
            $\pi \leftarrow$ NTT($\pi$,MODULI[$i$],ROOTS[$i$])
        If $i < 2$:
            $\pi \leftarrow$ PointwiseAdd($\pi, y$,MODULI[$i$])
            If $w \neq 0$:                                        ▷ If we have alternates, add them
                $\pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
        If $i \geq 2$:
        for $j \leftarrow 0$ to RNDS$-1$ do
            $pi \leftarrow$ PointwiseAdd($\pi, x$,MODULI[$i$])
            $pi \leftarrow$ PointwiseAdd($\pi, y$,MODULI[$i$])
            If $w = 1$:                                           ▷ If we have alternates, add them
                $pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
            If $w = 2$:                                           ▷ If we have alternates, add them
                $pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
                $pi \leftarrow$ PointwiseAdd($\pi, x'$,MODULI[$i$])
        end for
    end for
    for $i \leftarrow (l - 1)$ to 0 do                                  ▷ Begin inversions
        $\pi \leftarrow$ NTT_INV($\pi$,MODULI[$i$],ROOTS[$i$])
    end for
    return $\pi$
end function
```

Algorithm 10 ZKVolute2 - Noisy roots on both inputs

Constants:

- l - Number of stages of levels
- MODULI - l item array of NTT moduli
- ROOTS - l item array of NTT base roots of unity
- b - size of coefficients in bits
- pk - our public key
- RNDS - Number of rounds to perform - default 5

Input:

- x - First input poly
- y - Second input poly
- w - Whitening flag

Output:

- π - convolution like representation of inputs

```
1:  function ZKVolute2($x, y, w$)
2:      $\pi[] \leftarrow 0$                                            ▷ Define empty output polyvec
3:      $y_{OFF} \leftarrow$ RootGen($y, \mathsf{pk}_a, l, r, b$)
4:      $x_{OFF} \leftarrow$ RootGen($x, \mathsf{pk}_a, l, r, b$)
5:      If $w = 1$:                                      ▷ Create representation for second input
6:          $y' \leftarrow$ GenRelatedPoly($y, \mathsf{pk}, b$)
7:          $y'_{OFF} \leftarrow$ RootGen($y', \mathsf{pk}_a, l, r, b$)
8:      ElseIf $w = 2$:                                  ▷ Create representation for both inputs
9:          $y' \leftarrow$ GenRelatedPoly($y, \mathsf{pk}, b$)
10:         $x' \leftarrow$ GenRelatedPoly($x, \mathsf{pk}, b$)
11:         $y'_{OFF} \leftarrow$ RootGen($y', \mathsf{pk}_a, l, r, b$)
12:         $x'_{OFF} \leftarrow$ RootGen($x', \mathsf{pk}_a, l, r, b$)
13:     EndIf
14:     for $i \leftarrow 0$ to $l - 1$ do                                     ▷ Begin the transforms
15:         $x \leftarrow$ NTT($x$,MODULI[$i$],ROOTS[$i$] $+ x_{OFF}[i]$)
16:         $y \leftarrow$ NTT($y$,MODULI[$i$],ROOTS[$i$] $+ y_{OFF}[i]$)
17:         If $w = 1$:
18:             $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$] $+ y'_{OFF}[i]$)
19:         ElseIf $w = 2$
20:             $y' \leftarrow$ NTT($y'$,MODULI[$i$],ROOTS[$i$] $+ y'_{OFF}[i]$)
21:             $x' \leftarrow$ NTT($x'$,MODULI[$i$],ROOTS[$i$] $+ x'_{OFF}[i]$)
22:         If $i > 0$:                                        ▷ Need to transform the output
23:             $\pi \leftarrow$ NTT($\pi$,MODULI[$i$],ROOTS[$i$])
24:         If $i < 2$:
25:             $\pi \leftarrow$ PointwiseAdd($\pi, y$,MODULI[$i$])
26:             If $w \neq 0$:                               ▷ If we have alternates, add them
27:                 $\pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
28:         If $i \geq 2$:
29:         for $j \leftarrow 0$ to RNDS$-1$ do
30:             $pi \leftarrow$ PointwiseAdd($\pi, x$,MODULI[$i$])
31:             $pi \leftarrow$ PointwiseAdd($\pi, y$,MODULI[$i$])
32:             If $w = 1$:                                  ▷ If we have alternates, add them
33:                 $pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
34:             If $w = 2$:                                  ▷ If we have alternates, add them
35:                 $pi \leftarrow$ PointwiseAdd($\pi, y'$,MODULI[$i$])
36:                 $pi \leftarrow$ PointwiseAdd($\pi, x'$,MODULI[$i$])
37:         end for
38:     end for
39:     for $i \leftarrow (l - 1)$ to 0 do                                     ▷ Begin inversions
40:         $\pi \leftarrow$ NTT_INV($\pi$,MODULI[$i$],ROOTS[$i$])
41:     end for
42:     return $\pi$
43: end function
```

5.2.1 Main Algorithmic Definitions

Algorithm 11 KeyGen

Input:

- n - security parameter in bits

Output:

- pk_a - public key
- sk_a - secret key

```
1: function KeyGen(n)
2:     $\mathsf{sk}_a \leftarrow\!\!\$\ \mathbb{B}^n$
3:     $C_a \leftarrow\!\!\$\ \mathbb{B}^n$
4:     $pk'_a \leftarrow \text{ZKV}\textsc{olute}2(\mathsf{sk}_a, C_a, 1)$
5:     $\mathsf{pk}_a \leftarrow \mathsf{pk}'_a || C_a$
6:     return $\mathsf{sk}_a, \mathsf{pk}_a$
7: end function
```

Algorithm 12 Proof Generation (Signature Variant)

Input:

- sk_A - secret key
- pk_A - public key
- m - message digest

Output:

- σ - signature proof

```
1: function ProofGenSig($\mathsf{sk}_A, \mathsf{pk}_A, m$)
2:     $(\mathsf{pk}'_A, C_A) \leftarrow \textsc{SplitPK}(\mathsf{pk}_A)$
3:     FS_Chal $\leftarrow \textsc{Hash}(\mathsf{pk}'_A || m)$
4:     $\sigma \leftarrow \text{ZKV}\textsc{olute}2(\mathsf{sk}_A, \text{FS\_Chal}, 1)$
5:     return $\sigma$
6: end function
```

Algorithm 13 Proof Generation (Naked Key Exchange)

Input:

- sk_A - secret key
- pk_A - public key
- pk_B - public key

Output:

- π - key establishment proof
- SS - shared secret

```
1: function ProofNakedKexGen($\mathsf{sk}_A, \mathsf{pk}_B, \mathsf{pk}_A$)
2:     $(\mathsf{pk}'_A, C_A) \leftarrow \textsc{SplitPK}(\mathsf{pk}_A)$
3:     $(\mathsf{pk}'_B, C_B) \leftarrow \textsc{SplitPK}(\mathsf{pk}_B)$
4:     tmp $\leftarrow \text{ZKV}\textsc{olute}0(\mathsf{sk}_A, \mathsf{sk}_A, 0)$
5:     $tmp_\pi \leftarrow \text{ZKV}\textsc{olute}0(\text{tmp}, \mathsf{pk}'_B, 0)$
6:     SS $\leftarrow \text{ZKV}\textsc{olute}0(\text{tmp}, \mathsf{pk}'_B, 0)$
7:     $\pi \leftarrow \text{ZKV}\textsc{olute}0(tmp_\pi, C_B, 0)$
8:     return $\pi$, SS
9: end function
```

Algorithm 14 Key Agreement with Encrypted Payload

Input:

- sk_A, pk_A, C_A - A's keys
- pk'_B, C_B - B's public keys
- m - Payload

Output:

- π - Proof
- m_{enc} - Encrypted payload
- ψ - Opening value

```
1:  function KeyAgree(...)
2:      $H_m \leftarrow$ZKConvolute0($\mathsf{pk}'_B, m, 0$)          ▷ Bind payload
3:      $\pi \leftarrow$ZKConvolute0($\mathsf{sk}_A, \mathsf{pk}'_B, 0$)
4:      $\pi \leftarrow$ZKConvolute0($\pi, H_m, 0$)
5:      $SS_A \leftarrow$ZKConvolute0($\mathsf{sk}_A, \mathsf{pk}'_B, 0$)
6:      $SS_A \leftarrow$ZKConvolute0($SS_A, C_A, 0$)
7:      $SS_A \leftarrow$ZKConvolute0($SS_A, H_m, 0$)
8:      $m_{\text{enc}}, \leftarrow$Encrypt($m, SS_A$)          ▷ Encrypt payload
9:      $H_{\text{enc}} \leftarrow$Hash($m_{\text{enc}}, SS_A$)          ▷ Hash Encrypted payload
10:     $\psi \leftarrow$Prove($\mathsf{sk}_A, \pi, C_B, H_{\text{enc}}$)          ▷ Opening
11:     return $\pi, m_{\text{enc}}, \psi$
12: end function
```

Algorithm 15 Proof Generation (Accumulator Variant)

Input:

- sk_a - secret key
- pk_a - private key (optional, can be derived)
- π_0 - previous accumulator value
- $x_0, \ldots, x_n$ - variables to add to forward accumulator

Output:

- ψ - authenticating value
- π - new accumulation proof

```
1:  function ProofAccumulate($\mathsf{sk}_a, \mathsf{pk}_a, \pi_0, x_0, \ldots, x_n$)
2:      $\psi[] \leftarrow \pi_0$          ▷ Copy
3:      $\pi[] \leftarrow \pi_0$          ▷ Copy
4:      for all $x_i$ in inputs do          ▷ Each $X_n$ value provided
5:          $\psi \leftarrow$ ZKVolute0($\pi, x_i, 0$)
6:          $\pi \leftarrow$ ZKVolute0($\pi, x_i, 0$)
7:      end for
8:      $\psi \leftarrow$ ZKVolute0($\psi, \mathsf{sk}_a, 0$)          ▷ embed secret for equivariance text
9:      return $\psi, \pi$
10: end function
```

Algorithm 16 Proof Validation (Signature Variant)

Input:

- σ - signature proof value
- pk_a - Signer public key
- m - message

Output:

- TRUE or $\perp$

```
function ProofVerSig($\mathsf{pk}_a, m, \sigma$)
    FS_Chal ← Hash($\mathsf{pk}'_a || m$)
    $test_0 \leftarrow$ ZKVolute1($\sigma, C_a, 1$)
    $test_1 \leftarrow$ ZKVolute1($\mathsf{pk}'_a$,FS_CHAL, 1)
If $test_0 == test_1$
    return $TRUE$
Else
    return $\perp$
end function
```

Algorithm 17 Proof Validation and Payload Decryption

Input:

- pk_a, sk_b - Keys
- π - Encapsulation proof
- ψ - Opening value
- C_b - Public reference
- m_{enc} - Encrypted payload

Output:

- m - Decrypted payload
- $isValid$ - Validation result

```
function ValidateKexDecrypt($\mathsf{pk}_a, \mathsf{sk}_b, \pi, \psi$, $m_{\text{enc}}$)
    $H_{\text{enc}} \leftarrow$Hash($m_{\text{enc}}$)
    $\psi_0 \leftarrow$ZKVolute0($\psi, \pi, 0$)
    $\psi_0 \leftarrow$ZKVolute0($psi_0, C_b, 0$ )
    $\psi_0 \leftarrow$ZKVolute0($psi_0, H_{\text{enc}}, 0$)
    $\psi' \leftarrow$ZKVolute0($\mathsf{pk}'_a, \psi$)
    $\psi' \leftarrow$ZKVolute0($\pi, C_b, 0$)
    $\psi' \leftarrow$ZKVolute0($\pi, H_{\text{enc}}, 0$)
    if $\psi_0 = \psi'$ then
        $SS_b \leftarrow$ZKVolute0($\mathsf{sk}_b, \pi, 0$)
        $m \leftarrow$Decrypt($SS_b, m_{\text{enc}}$)
        return $m$, TRUE
    else
        return NULL, $\perp$
    end if
end function
```

Algorithm 18 Proof Validation (Accumulator Variant)

```
Input:
  • $\pi$ - New proof
  • $\mathsf{pk}_a$ - Signer public key
  • $\psi$ - Accumulator authenticator value
Output:
  • TRUE or $\perp$
function ProofVerAcc($\pi, \mathsf{pk}_a, \psi$)
    $(C_a, \mathsf{pk}'_a) \leftarrow$ PKSplit($\mathsf{pk}_a$)          ▷ Simply splits the key
    $test_0 \leftarrow$ ZKVolute0($\psi, C_a, 0$)
    $test_1 \leftarrow$ ZKVolute0($\pi, \mathsf{pk}'_a, 0$)
If $test_0 == test_1$
    return $TRUE$
Else
    return $\perp$
end function
```

6 Core Design - Connected Stages of NTT Transformations with Distinct Variables

Given n degree integer polynomials $A = A_0, A_1, \ldots, A_n$, $J = J_0, J_1, \ldots, J_n$, $J' = J'_0, J'_1, \ldots, J'_n$, and n degree output polynomial $O = O_0, O_1, \ldots, O_n$. J is either a random challenge/reference string in the case of KeyGen, or a value derived from a given m as a challenge polynomial. J' is a cryptographically whitened representation of J'. J' produced via standard randomness extraction from J. O (or π) is used to represent the output polynomial.

To generate a public reference output O_{pk} let f represent the 7 stage transform and inversion below, let A represent the private secret, let J represent a randomly selected challenge and J' as the cryptographically whitened representation of J. $f(A, J, J') = O_{pk}$. Let $\omega A, \omega J, \omega J'$ represent the pseudorandom input dependent offset roots of unity. These pseudorandom roots are derived using standard randomness extraction from J and J' to compute offset values that are applied to the 'base' set of ω roots. This offset technique is similar to the concept of adding noise or errors in other lattice cryptographic systems to thwart cryptanalysis.

To generate a proof of possession π of A using polynomial challenge D combined with a whitened representation D' let f represent the 7 stage transform and inversion below. $f(A, D, D') = \pi$.

To confirm the provers knowledge of A a verifier can leverage the convolution-like properties of the scheme (associativity and commutativity) to compute and verify the equivariance $f(O_{pk}, D, D') == f(\pi, J, J')$. If the results are the same, the verifier knows π was produced by convoluting A and D.

6.1 Transformation f - ZKVolute Definition

Let n be the degree polynomial, s the number of stages of forward and inverse NTT transformations defined by the set of $p_0, \ldots, p_s$ and nth roots of unity $\omega_0 \ldots, \omega_s$, using the parameters above. Additionally, the non-constant function variables are:

$$Inputs:$$

$$A = A_0, A_1, \ldots, A_n$$

$$J = J_0, J_1, \ldots, J_n$$

$$Derived\ Values:$$

$$J' = J'_0, J'_1, \ldots, J'_n$$

$$\omega A = \omega A_0, \ldots \omega A_s$$

$$\omega J = \omega J_0, \ldots \omega J_s$$

$$\omega J' = \omega J'_0, \ldots \omega J'_s$$

$$Output:$$

$$O = O_0, O_1, \ldots, O_n$$

6.2 Function Definitions:

6.2.1 Stage 1 in $\mathbb{F}_{p_1}$ with $\omega A_1, \omega J_1, \omega J'_1$

The first NTT transformation is:

$$B_k = \sum_{i=0}^{n} A_i (\omega A_1)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$K_k = \sum_{i=0}^{n} J_i (\omega J_1)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$K'_k = \sum_{i=0}^{n} J'_i (\omega J'_1)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = (K_k + K'_k) \pmod{p_1} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.2 Stage 2 in $\mathbb{F}_{p_2}$ with $\omega_2, \omega A_2, \omega J_2, \omega J'_2$

Using the outputs of Stage 1, the second NTT transformation is:

$$C_k = \sum_{i=0}^{n} B_i (\omega A_2)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$L_k = \sum_{i=0}^{n} K_i (\omega J_2)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$L'_k = \sum_{i=0}^{n} K'_i (\omega J'_2)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_2)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = (C_k + L_k + L'_k + O_k) \pmod{p_2} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.3 Stage 3 in $\mathbb{F}_{p_3}$ with $\omega_3, \omega A_3, \omega J_3, \omega J'_3$

Inputs are the outputs of Stage 2. The third NTT transformation:

$$D_k = \sum_{i=0}^{n} C_i (\omega A_3)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$M_k = \sum_{i=0}^{n} L_i (\omega J_3)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$M'_k = \sum_{i=0}^{n} L'_i (\omega J'_3)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_3)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = D_k + M_k + M'_k + O_k \pmod{p_3} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.4 Stage 4 in $\mathbb{F}_{p_4}$ with $\omega_4, \omega A_4, \omega J_4, \omega J'_4$

Inputs are the outputs of Stage 3. The NTT transformation:

$$E_k = \sum_{i=0}^{n} D_i (\omega A_4)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$N_k = \sum_{i=0}^{n} M_i (\omega J_4)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$N'_k = \sum_{i=0}^{n} M'_i (\omega J'_4)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_4)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = E_k + N_k + N'_k + O_k \pmod{p_4} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.5 Stage 5 in $\mathbb{F}_{p_5}$ with $\omega_5, \omega A_5, \omega J_5, \omega J'_5$

Inputs are the outputs of Stage 4. The NTT transformation:

$$F_k = \sum_{i=0}^{n} E_i (\omega A_5)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$P_k = \sum_{i=0}^{n} N_i (\omega J_5)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$P'_k = \sum_{i=0}^{n} N'_i (\omega J'_5)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_5)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = F_k + P_k + P'_k + O_k \pmod{p_5} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.6 Stage 6 in $\mathbb{F}_{p_6}$ with $\omega_6, \omega A_6, \omega J_6, \omega J'_6$

Inputs are the outputs of Stage 5. The NTT transformation:

$$G_k = \sum_{i=0}^{n} F_i (\omega A_6)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$Q_k = \sum_{i=0}^{n} P_i (\omega J_6)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$Q'_k = \sum_{i=0}^{n} P'_i (\omega J'_6)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_6)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = G_k + Q_k + Q'_k + O_k \pmod{p_6} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.7 Stage 7 in $\mathbb{F}_{p_7}$ with $\omega_7, \omega A_7, \omega J_7, \omega J'_7$

Inputs are the outputs of Stage 6. The NTT transformation:

$$H_k = \sum_{i=0}^{n} G_i (\omega A_7)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$R_k = \sum_{i=0}^{n} Q_i (\omega J_7)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$R'_k = \sum_{i=0}^{n} Q'_i (\omega J'_7)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = \sum_{i=0}^{n} O_i (\omega_7)^{ik} \quad \text{for } k = 0, 1, \ldots, n$$

$$O_k = H_k + R_k + R'_k + O_k \pmod{p_7} \quad \text{for } k = 0, 1, \ldots, n$$

6.2.8 Inversion in $\mathbb{F}_{ps}$ with Inverse of ω_s for s = 7,6,...,1

Given the output coefficients $O_0, O_1, \ldots, O_n$ from the final forward transformation, the NTT inversion steps are:

$$O_i = \frac{1}{n+1} \sum_{k=0}^{n} O_k (\omega_7)^{-ik} \pmod{p_7} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1} \sum_{k=0}^{n} O_k (\omega_6)^{-ik} \pmod{p_6} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1} \sum_{k=0}^{n} O_k (\omega_5)^{-ik} \pmod{p_5} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1} \sum_{k=0}^{n} O_k (\omega_4)^{-ik} \pmod{p_4} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1} \sum_{k=0}^{n} O_k (\omega_3)^{-ik} \pmod{p_3} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1}\sum_{k=0}^{n} O_k(\omega_2)^{-ik} \pmod{p_2} \quad \text{for } i = 0, 1, \ldots, n$$

$$O_i = \frac{1}{n+1}\sum_{k=0}^{n} O_k(\omega_1)^{-ik} \pmod{p_1} \quad \text{for } i = 0, 1, \ldots, n$$

The final O output polynomial or proof π is a convolution-like representation of the combination of inputs A, J, J' masked by a unique set of ω values. This π proof value can be used for equivariance evaluation.

6.3 Equivariance Test

Note that inputs are uniquely transformed by pseudorandom ω roots of unity during the forward transformation portion of the proof π generation. However, as the π value is inverted using the non-pseudorandom 'base' ω values, for equivariance tests the same non-pseudorandom 'base' ω roots are used for the forward transformation of the π and O_{pk} values. For the challenge values J and the related cryptographically whitened representation J' the pseudorandom roots are applied during proof generation and equivariance testing.

7 Knapsack Problem Representation

We show that solutions satisfying the constraints of our cryptographic system constructed from staged multidimensional number-theoretic transforms are equivalent to feasible solutions to a Multi-Objective Multidimensional Knapsack Problem with Carry Over(MOMKP-CO) formulation. The output polynomial O is initially calculated during the first transformation stage and carried over to each subsequent stage.

$$O_{ij} = \sum_{k=0}^{n}(A_k + J_k + J'_k)(\omega(A|J|J')_i)^{jk} \pmod{p_i}$$

where p_i defines stage-i modulus

7.1 MOMPK-CO Formulation

The MOMPK-CO problem is defined by:

- Multiple knapsacks i, with capacities p_i
- Items j with profit v_j and weight w_j
- Carry-over constraints between knapsacks

With variables x_{ij} indicating whether item j is assigned to knapsack i. These constraints couple the multiple knapsacks together to create interdependencies.

7.2 Equivalence Proof

Consider solution vectors:

$\vec{O} = [A; B; O_1; ...; O_n]$ values in system $\vec{x} \quad = [x_{11}; ...; x_{1m}; ...; x_{n1}; ...; x_{nm}]$ MOMPK-CO variable assignments

Any feasible $\vec{O}$ satisfies the MOMPK CO constraints for corresponding $\vec{x}$. Any feasible $\vec{x}$ yields a valid $\vec{O}$ under the transforms. Thus solutions are equivalent between the domains.

7.3 Dependencies

Let I be the set of items, K be the set of knapsacks, and D be the set of dependencies among items.

- Let $I = 1, 2, ..., n$ be the set of items
- Let $K = 1, 2, ..., m$ be the set of knapsacks
- Let $D \subseteq I \times I$ be the set of dependencies between items, where:
  - $(a, b) \in D$ means item a depends on item b
  - D encodes prerequisites constraints between items

For instance, a simple example could have:

- $I = 1, 2, 3$ (3 items)
- $K = 1$ (1 knapsack)
- $D = (2, 1), (3, 1)$ (item 2 depends on 1, item 3 depends on 1)

$$x_b \geq x_a \quad \forall (a, b) \in D$$

Where $x_i \in 0, 1$ indicates if item i is selected.
Let x_{ij} be the j^{th} item in knapsack i.

7.3.1 Intra-knapsack Dependencies

: The items within a knapsack are transformed NTT representations of the original inputs. For knapsack i:

$$x_{ij} = \sum_{k=0}^{n} a_{jk} (\omega(A|J|J')_i)^{jk} \pmod{p_i}$$

Where a_{jk} is the input vector, ω_i is the transform basis for knapsack i, and p_i is the knapsack capacity. This creates direct dependencies between items within a knapsack dictated by the transform.

7.3.2 Inter-knapsack Dependencies

: The output O_i from sack i is propagatively combined into knapsack $i + 1$:

$$O_{i+1} = O_i + \sum_{j=1}^{m} w_{ij} x_{ij}$$

Where w_{ij} are item weights. This carries dependencies between knapsacks - the outputs of one knapsack directly are inputs to the next.

7.4 Knapsack Parameters

- n = number of coefficients per polynomial
- K = number of transformation stages (7 in described in this work)
- p_i = modulus in stage i
- ω_i = standard transform basis in stage i
- ωA_i = input dependent transform basis in stage i where $1 < \omega A_i < p_i$
- ωJ_i = input dependent transform basis in stage i where $1 < \omega J_i < p_i$
- $\omega J'_i$ = input dependent transform basis in stage i where $1 < \omega J`_i < p_i$

7.5 Knapsack Variables

- A_j = Coefficients of input polynomial 1 ($j = 1...n$)
- B_j = Coefficients of input polynomial 2 ($j = 1...n$)
- O_{ij} = Transformed coefficients in stage i ($j = 1...n$)
- y_{ij} = Selection variables ($j = 1...n, i = 1...K$)

7.6 Dependencies

- Intra-stage: O_{ij} depends on input polynomials A_j, J_j, J'_j and $\omega(A|J|J')_i$ transforms
- Inter-stage: $O_{(i+1)j}$ depends on O_{ij} due to modular additions between stages

7.6.1 Objective Function

The objective is to maximize the total value of items subject to capacity constraints and interdependencies.

Assumption 1 (Multidimensional Knapsack)**.** *Solving a random multidimensional knapsack problem over K interdependent knapsacks with many specific capacity constraints p_K per stage, is computationally intractable. More formally:*

Given: Items x_{ij} with values v_{ij} and weights w_{ij}
Stage capacities p_i for $i = 1, ..., K$
Find: Selections y_{ij}

To maximize: $\sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij} y_{ij}$
Subject to: $\sum_{j=1}^{m} w_{ij} y_{ij} \leq p_i$

$y_{ij} \in 0, 1$ *is computationally infeasible to solve for large random n, p_i.*

8 Knapsack Algorithmic Representation

In this multidimensional interdependent knapsack system reduction, the finite field constraints defined in the NTT construction above are lifted to integer constraints in the knapsack representation.

8.1 Knapsack Constraints

Variables:

$$A_i, J_i, J'_i, B_k, K_k, K'_k, O_k, C_k, L_k, L'_k, D_k, M_k, M'_k, E_k, N_k, N'_k, F_k, P_k, P'_k, G_k, Q_k, Q'_k, H_k, R_k, R'_k$$

Stage 1 Knapsack Constraints:

$$\sum_{i=0}^{n} A_i(\omega A_1)^i \leq p_1 \quad \text{for } i = 0 \text{ to } n$$

$$\sum_{i=0}^{n} J_i(\omega J_1)^i \leq p_1 \quad \text{for } i = 0 \text{ to } n$$

$$\sum_{i=0}^{n} J'i(\omega J'_1)^i \leq p_1 \quad \text{for } i = 0 \text{ to } n$$

$$B_k = \sum_{i=0}^{n} A_i(\omega A_1)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$K_k = \sum_{i=0}^{n} J_i(\omega J_1)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$K'k = \sum_{i=0}^{n} J'_i(\omega J'_1)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$O_k = (K_k + K'_k) \pmod{p_1} \quad \text{for } k = 0 \text{ to } n$$

Stage 2 Knapsack Constraints:

$$\sum_{i=0}^{n} B_i(\omega A_2)^i \leq p_2 \quad \text{for } i = 0 \text{ to } n$$

$$\sum_{i=0}^{n} K_i(\omega J_2)^i \leq p_2 \quad \text{for } i = 0 \text{ to } n$$

$$\sum_{i=0}^{n} K'_i(\omega J'_2)^i \leq p_2 \quad \text{for } i = 0 \text{ to } n$$

$$\sum_{i=0}^{n} O_i(\omega_2)^i \leq p_2 \quad \text{for } i = 0 \text{ to } n$$

$$C_k = \sum_{i=0}^{n} B_i(\omega A_2)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$L_k = \sum_{i=0}^{n} K_i(\omega J_2)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$L'k = \sum_{i=0}^{n} K'_i(\omega J'_2)^{ik} \quad \text{for } k = 0 \text{ to } n$$

$$O_k = (C_k + L_k + L'_k + O_k) \pmod{p_2} \quad \text{for } k = 0 \text{ to } n$$

Stage 3 Knapsack Constraints:

$$\sum_{i=0}^{n} C_i(\omega A_3)^i \leq p_3 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} L_i(\omega J_3)^i \leq p_3 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} L'_i(\omega J'_3)^i \leq p_3 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} O_i(\omega_3)^i \leq p_3 \quad \text{for } i = 0 \text{ to } n$$

$$D_k = \sum_{i=0}^{n} C_i(\omega A_3)^{ik} \quad \text{for k = 0 to n}$$

$$M_k = \sum_{i=0}^{n} L_i(\omega J_3)^{ik} \quad \text{for k = 0 to n}$$

$$M'k = \sum_{i=0}^{n} L'_i(\omega J'_3)^{ik} \quad \text{for k = 0 to n}$$

$$O_k = D_k + M_k + M'_k + O_k \pmod{p_3} \quad \text{for k=0 to n}$$

Stage 4 Knapsack Constraints:

$$\sum_{i=0}^{n} D_i(\omega A_4)^i \leq p_4 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} M_i(\omega J_4)^i \leq p_4 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} M'i(\omega J')_4^i \leq p_4 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} O_i(\omega_4)^i \leq p_4 \quad \text{for } i = 0 \text{ to } n$$

$$E_k = \sum_{i=0}^{n} D_i(\omega A_4)^{ik} \quad \text{for k = 0 to n}$$

$$N_k = \sum_{i=0}^{n} M_i(\omega J_4)^{ik} \quad \text{for k = 0 to n}$$

$$N'k = \sum_{i=0}^{n} M'_i(\omega J'{}_4)^{ik} \quad \text{for k = 0 to n}$$

$$O_k = E_k + N_k + N'_k + O_k \pmod{p_4} \quad \text{for k=0 to n}$$

Stage 5 Knapsack Constraints:

$$\sum_{i=0}^{n} E_i(\omega A_5)^i \leq p_5 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} N_i(\omega J_5)^i \leq p_5 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} N'i(\omega J'{}_5)^i \leq p_5 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} O_i(\omega_5)^i \leq p_5 \quad \text{for } i = 0 \text{ to } n$$

$$F_k = \sum_{i=0}^{n} E_i(\omega A_5)^{ik} \quad \text{for k = 0 to n}$$

$$P_k = \sum_{i=0}^{n} N_i(\omega J_5)^{ik} \quad \text{for k = 0 to n}$$

$$P'k = \sum_{i=0}^{n} N'_i(\omega J'{}_5)^{ik} \quad \text{for k = 0 to n}$$

$$O_k = F_k + P_k + P'_k + O_k \pmod{p_5} \quad \text{for k=0 to n}$$

Stage 6 Knapsack Constraints:

$$\sum_{i=0}^{n} F_i(\omega A_6)^i \leq p_6 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} P_i(\omega J_6)^i \leq p_6 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} P'i(\omega J'{}_6)^i \leq p_6 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} O_i(\omega_6)^i \leq p_6 \quad \text{for } i = 0 \text{ to } n$$

$$G_k = \sum_{i=0}^{n} F_i(\omega A_6)^{ik} \quad \text{for k = 0 to n}$$

$$Q_k = \sum_{i=0}^{n} P_i(\omega A'_6)^{ik} \quad \text{for k = 0 to n}$$

$$Q'k = \sum_{i=0}^{n} P'_i(\omega_6)^{ik} \quad \text{for k = 0 to n}$$

$$O_k = G_k + Q_k + Q'_k + O_k \pmod{p_6} \quad \text{for k=0 to n}$$

Stage 7 Knapsack Constraints:

$$\sum_{i=0}^{n} G_i(\omega A_7)^i \le p_7 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} Q_i(\omega J_7)^i \le p_7 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} Q'i(\omega J'_7)^i \le p_7 \quad \text{for i = 0 to n}$$

$$\sum_{i=0}^{n} O_i(\omega_7)^i \le p_7 \quad \text{for } i = 0 \text{ to } n$$

$$H_k = \sum_{i=0}^{n} G_i(\omega A_7)^{ik} \quad \text{for k = 0 to n}$$

$$R_k = \sum_{i=0}^{n} Q_i(\omega J_7)^{ik} \quad \text{for k = 0 to n}$$

$$R'k = \sum_{i=0}^{n} Q'_i(\omega J'_7)^{ik} \quad \text{for k = 0 to n}$$

$$O_k = H_k + R_k + R'_k + O_k \pmod{p_7} \quad \text{for k=0 to n}$$

8.2 Knapsack Inversion Phase

Inversion Knapsack Constraints:

$$\sum_{k=0}^{n} O_{k,7}(\omega_7)^{-ik} \le p_7 \quad for \text{ i=0 to n}$$

$$O_{i,7} = \frac{1}{n+1} \sum_{k=0}^{n} O_{k,7}(\omega_7)^{-ik} \pmod{p_7} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,6}(\omega_6)^{-ik} \le p_6 \quad \text{for i=0 to n}$$

$$O_{i,6} = \frac{1}{n+1} \sum_{k=0}^{n} O_{k,6}(\omega_6)^{-ik} \pmod{p_6} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,5}(\omega_5)^{-ik} \le p_5 \quad \text{for i=0 to n}$$

$$O_{i,5} = \frac{1}{n+1} \sum_{k=0}^{n} O_{k,5}(\omega_5)^{-ik} \pmod{p_5} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,4}(\omega_4)^{-ik} \leq p_4 \quad \text{for i=0 to n}$$

$$O_{i,4} = \frac{1}{n+1}\sum_{k=0}^{n} O_{k,4}(\omega_4)^{-ik} \pmod{p_4} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,3}(\omega_3)^{-ik} \leq p_3 \quad \text{for i=0 to n}$$

$$O_{i,3} = \frac{1}{n+1}\sum_{k=0}^{n} O_{k,3}(\omega_3)^{-ik} \pmod{p_3} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,2}(\omega_2)^{-ik} \leq p_2 \quad \text{for i=0 to n}$$

$$O_{i,2} = \frac{1}{n+1}\sum_{k=0}^{n} O_{k,2}(\omega_2)^{-ik} \pmod{p_2} \quad \text{for i=0 to n}$$

$$\sum_{k=0}^{n} O_{k,4}(\omega_4)^{-ik} \leq p_1 \quad \text{for i=0 to n}$$

$$O_{i,1} = \frac{1}{n+1}\sum_{k=0}^{n} O_{k,1}(\omega_1)^{-ik} \pmod{p_1} \quad \text{for i=0 to n}$$

8.3 Solution Equivalence

The forward and inverse NTT transformations define an equivalence between the solutions to the knapsack problems. Specifically, let A, J, J', and O be solutions that satisfy the forward knapsack constraints. Then the inverse NTT will produce solutions $\hat{A}$, $\hat{J}$, $\hat{J}'$, and $\hat{O}$ that satisfy the inverse knapsack constraints. This can be seen by substituting the forward solutions into the inverse NTT summations:

$$\hat{O}i = \frac{1}{n+1}\sum k = 0^n O_k, (\omega)^{-ik} \quad = \frac{1}{n+1}\sum_{k=0}^{n}(H_k + R_k + R'_k + O_k), (\omega)^{-ik}, (\omega A)^{-ik}, (\omega J)^{-ik}, (\omega J')^{-ik}$$

Since H_k, R_k, R'_k, and O_k satisfy the forward constraints, their inverse NTTs will satisfy the inverse constraints. A similar argument follows for $\hat{A}_i$, $\hat{J}_i$, and $\hat{J}'_i$. Therefore, any solution to the forward knapsack problem can be transformed into a solution to the inverse problem via the inverse NTT.

Reduction: Let A be an algorithm for breaking the security of the cryptographic system based on the knapsack problem. We can express this as a reduction showing that breaking the cryptosystem is computationally equivalent to solving the hard knapsack problem:

$$A_{\text{break NTT scheme}} \leq_p A_{\text{solve knapsack}} \ \text{Time}(A_{\text{break NTT scheme}}(I)) \quad \leq \text{Time}(A_{\text{solve knapsack}}(f(I))) + c$$

Where:
- I is an instance of the cryptosystem (public keys, outputs, etc.)
- f is the function mapping cryptosystem instances to knapsack instances
- c is some constant representing overhead

The function f outputs a knapsack instance with:
- Items corresponding to the coefficients of A, J, J'
- Knapsacks corresponding to each NTT stage/modulus
- Capacities equal to the moduli $p_1, ..., p_n$
- Dependency constraints encoding the NTT transformations
- Objective function related to the output O

- Cryptosystem parameters: polynomials A, J, J', moduli $p_1, ..., p_n$, roots of unity $\omega_1, ..., \omega_n$
- Cryptosystem inputs/outputs: O (public output), π (proof)

This states that if we have an efficient algorithm for breaking the cryptographic system, we can use it to construct an efficient algorithm for solving the underlying knapsack problem. Specifically, given:

- An algorithm $A_{\text{break NTT scheme}}$ that breaks the cryptosystem in time $\text{Time}(A_{\text{break NTT scheme}}(I))$
- A mapping f from cryptosystem instances to equivalent knapsack instances

We can solve any knapsack instance K as:

```
$I = f^{-1}(K)$                        // Get equivalent cryptosystem instance
$A = A_{\text{break NTT scheme}}(I)$    // Break cryptosystem
**return**$(A)$                        // Solves knapsack instance
```

The overhead c accounts for the extra steps of computing f^{-1} and f. Therefore, if we can break the NTT based cryptosystem in polynomial time, we can then leverage this attack to solve the knapsack problem in polynomial time. This contradicts the knapsack problem's conjectured computational hardness.

8.4 Summary

The multidimensional interdependent knapsack problem formulated in this document encapsulates the constraints of the cryptographic system based on the Number Theoretic Transform. Solving this knapsack instance would allow an adversary to find solutions for the secret polynomial A and output polynomial O.

Furthermore, the knapsack problem is at least as hard as solving a system of multivariate quadratic equations over finite fields. The complexity of this underlying mathematical problem ensures the robust security of the cryptosystem against known attacks.

Therefore, an efficient algorithm to find optimal solutions for this knapsack formulation would completely break the security of the cryptographic scheme, and represent a substantial advancement in the field of cryptanalysis. The presumed difficulty of this problem is fundamental to the construction and security analysis.

In conclusion, this comprehensive knapsack model accurately represents the cryptographic transforms and constraints. Solving this hard optimization problem would directly compromise the secrecy of data protected by the system. Thus solving the knapsack is fundamentally equivalent to breaking the cryptosystem itself.

9 Known Limitations

9.1 Initial Parameter Choices

Prior to this work, there was no scientific basis for parameter selection. Parameters estimated to be secure were derived experimentally. Given that a brute-force search against the input space of $\mathcal{O}(2^{1152})$, in the 8-bit 144 value case, we consider brute force currently computationally infeasible.

Faster and more efficient instances targeting $\mathcal{O}(2^{128})$, $\mathcal{O}(2^{256})$, and $\mathcal{O}(2^{512})$ are forthcoming. We do not recommend implementing systems using the initial given parameters aside for test and validation purposes. Significantly more efficient instances already exist for instances of five levels, and hardware accelerated vector operations are dramatically more performant than the unoptimized implementation provided.

9.2 Current Computational Performance

While not as computationally intense as fully hash based digital signature schemes such as XMSS[10] or SPHINCS+[3], the current level of performance is too slow for widespread use. Currently, without optimization, the complete cycle of *keyGen*, *prove*, *validate* exceeds 16 seconds on an ARM M2 Max, as implemented in Python3. This is due to the initial choice of overly conservative parameters.

9.3 Vulnerability to Power Analysis Attacks in Algebraic Systems

In algebraic cryptosystems like PoSSoL and other lattice-based schemes, power analysis attacks present a significant risk. For example, the recent KyberSlash vulnerabilities target certain Kyber implementations. These attacks exploit variations in power use during cryptographic computations, allowing an adversary to extract secret key information. Due to the complexity of operations in algebraic constructs, distinct power consumption patterns can emerge that correlate with specific operations on secret data. Mitigating such attacks requires careful implementation and countermeasures like constant-time modular arithmetic and other power obfuscation techniques.

This attack vector is common in schemes using number theoretic transforms (NTTs). Fortunately, mitigation is achievable on some platforms through hardware acceleration like AVX or Neon instructions. Alternatively, proper constant-time software implementations of the vulnerable math, while challenging, can be created.

9.4 Future Advances in Cryptanalysis

Currently, algebraic attacks are thwarted by the complexity introduced by constructive embedding in complex lattices. Lattice reduction attacks are mitigated by the dimensionality and 'noisy' internal structure created by dynamic NTT configuration. It is possible, perhaps, that somehow a sufficient basis reduction/polynomial solving algorithm is developed. This could become an issue should quantum computers become extremely powerful, which is unlikely in the foreseeable future.

10 Future Work - Root Noise Tolerance

Initial experimental results have shown there to be some level of intolerance for offsets applied to roots. Not every single value between 1 and p for each stage maintains proof stability. There is likely a range of values distributed around 0 up to some currently undefined limit that will always successfully pass equivariance. This limit is yet to be defined. That said, it is likely that even small a perturburance in these scaling factors mitigates lattice reduction and direct algebraic cryptanalysis. Example working roots for random inputs using the same p fields using offset values for ω for the first 3 stages:
A:
$p = 257$ $\omega = 100$
$p = 1337335578003807237072357113$ $\omega = 2306773755540215763766289$5
$p = 367448648399022996626498550215488$1 $\omega = 1890847646321174403442558538216986$
J:
$p = 257$ $\omega = 250$
p= 1337335578003807237072357113 $\omega = 3406773755540215763766289$3
$p = 367448648399022996626498550215488$1 $\omega = 1108476463211744034425585382169855$
J':
$p = 257$ ω=256
$p = 1337335578003807237072357113$ $\omega = 1306773755540215763766289$7
p =3674486483990229966264985502154881 $\omega = 1790847646321174403442558538216985$
O:
$p = 257$ $\omega = 9$
$p = 1337335578003807237072357113$ $\omega = 6406773755540215763766289$6
$p = 367448648399022996626498550215488$1 $\omega = 1890847646321174403442558538216985$

10.1 MQ reduction

Initially the complex problem chosen to prove security was the well known MQ over a finite field. As this reduction became overly voluminous, a constraint based knapsack based proof was constructed instead. A complete and accurate MQ reduction that accurately represents all the finite fields and transformations remains an open research and analysis task.

10.2 Proof Soundness

Additionally, a value for proof soundness error probability needs to be rigorously calculated. This effort will likely require a moderate amount of highly specialized analysis and creation of a symbolic system. However, experimentally, no forged proof has ever been accepted. If the calculated soundness error probability is unacceptably high, a simple composite proof chaining technique can be used to mitigate this vector.

10.3 From Deterministic to Probabilistic

While probabilistic algorithms aren't a strict requirement for semantic security, and there are several lattice systems that are fully deterministic, a design that produces a unique output for the same inputs can be desirable compared to their deterministic counterparts. Probabilistic output is generally achieved by random noise/errors that are applied per output and error correcting bits of information that are included with the cipher-text. This future work will obscure the internal lattice structure further, making it more resilient against attack. As algebraic instability is fairly easy to reliably achieve using this type of NTT transformation framework, this instability can be leveraged as noise itself. An additional mode comprised of 'stable' stages of transforms below one or more 'unstable' transforms allows a form noise that is intrinsic and 'functional'. For a given set of polynomial coefficients convolving via 'unstable' NTT, one sees harmonic patterns emerge for correct vs. failed equivariance tests for honest provers. By applying small adjustments to the coefficients, an unstable representation can be modified to create a stable one. This noise can be extracted and sent as additional error correcting bits. Unstable configurations of NTTs can have a wide ratio of failures before a passing solution is found. Some configurations yield systems that pass 9 times out of 10, others, 1 out of 1,000. These unstable transformation stages can act as a filter, requiring 'noise' to be added to a given polynomial such that equivariance stability is maintained. This entropic noise be used to build a probabilistic system from a deterministic one, increasing cryptanalysis difficulty. Determining the optimal modifications needed to achieve this 'with errors' version is left as a future research item.

11 Proofs - Game Based

Security of PKEMNO NIZK Security of PKEMNO is defined by indistinguishability under chosen-cipher-text and prove attacks (IND-CCPA)[9] and proof soundness. We define both notions in the subsequent section.

11.1 Zero Knowledge - 1

Definition 1 (IND-CCPA security). Consider the following game between a challenger and an adversary A:

1. **Setup:** The challenger runs Gen(1^k) and gives pk to A.
2. **Phase 1:** The adversary issues queries of the form:
   (a) proof generation query to an oracle Prove($\mathsf{sk}, \cdot$).
   These may be asked adaptively in that they may depend on the answers to previous queries.
3. **Challenge:** At some point, A outputs a proof-message pair (π_0, m_0) and (π_1, m_1). The challenger chooses a random bit β and returns the pair (π_β, m_β).
4. **Phase 2:** As Phase 1, except that no queries on (π_β, m_β) are allowed.
5. **Guess:** The adversary A outputs a guess $\beta' \in \{0, 1\}$. The adversary wins the game if $\beta = \beta'$.

Define A's advantage as:

$$\mathrm{Adv}_{\text{ind-ccpa}}^{\text{PKEMNO},A}(1^k) = \Pr[\beta' = \beta] - \frac{1}{2} \tag{1}$$

A PKEMNO type scheme is called indistinguishable against chosen-cipher-text and prove attacks (IND-CCPA secure) if: A, $\mathrm{Adv}_{\text{ind-ccpa}}^{\text{PKEMNO},A}(\cdot)$ is negligible.

We define Game 1 between a challenger Challenger and adversary $\mathcal{A}$ to argue zero-knowledge property:

```
Setup: Challenger runs KeyGen to obtain $(\mathsf{pk}, \mathsf{sk})$ and gives $\mathsf{pk}$ to $\mathcal{A}$.
Commit:
  $\mathcal{A}$ selects two messages $m_0$, $m_1$ of equal length and sends them to Challenger
Challenge:
  Challenger flips a coin $b \leftarrow \{0, 1\}$
  Challenger computes proof $\pi \leftarrow \mathbf{F}(\mathsf{sk}, m_b)$
  Challenger sends $\pi$ to $\mathcal{A}$
Guess: $\mathcal{A}$ outputs a guess $b'$ for $b$.
   $\mathcal{A}$ wins the game if $b = b'$.
```

We argue that $\mathcal{A}$ can win this guessing game with only negligible advantage greater than $\frac{1}{2}$ based on the zero-knowledge property of the proof system.

11.2 Simulation Soundness - 2

Definition 2 (Proof Soundness in PKEMNO NIZK)
Consider the following game between a challenger and an adversary
Setup:
$\mathcal{C} \leftarrow \text{Gen}(1^\lambda)(pk, sk)$
$\mathcal{C}$ gives pk to adversary $\mathcal{A}$
Forgery Attempt:
$\mathcal{A}$ outputs $(\pi\text{'}x\text{'}y\text{'})$
Define advantage as:
$\text{Adv}_{\mathcal{A}}^{\text{proof-snd}}(\lambda) = \Pr\left[\text{Verify}(pk, x\text{'}y\text{'}\pi\text{'}) = 1 : \wedge : (x\text{'}y\text{'}) \notin R\right]$
Where:
$R = (x, y) : \exists w \text{ s.t. } (x, y, w) \in \mathcal{L}$
We say the scheme satisfies proof soundness if
$\text{Adv}_{\mathcal{A}}^{\text{proof-snd}}(\lambda)$ is negligible for all PPT $\mathcal{A}$

A PKEMNO NIZK scheme is proof sound if for every PPT adversary A its advantage is negligible.

We define Game 2 between a challenger Challenger and adversary $\mathcal{A}$ to argue simulation soundness:

```
Setup: Challenger runs KeyGen to obtain $(\mathsf{pk}, \mathsf{sk})$ and gives both to $\mathcal{A}$.
Simulation:
  Challenger generates $n$ simulated proofs $\pi_1, \ldots, \pi_n$
  Challenger sends the simulated proofs to $\mathcal{A}$
Forgery Attempt:
  $\mathcal{A}$ outputs a new proof $\pi^A$ message, ciphertext pair $(m^A, d^A)$
Verification:
  Challenger checks if $\mathbf{F}(\mathsf{pk}'\ d^A) == \mathbf{F}(\pi^A, c)$
Winning Condition:
  If the check passes, $\mathcal{A}$ wins the game.

```

We argue that $\mathcal{A}$ wins only with negligible probability based on the simulation soundness of the proofs.

11.3 Strong Proof Soundness - 3

Definition 3 (Strong Proof Soundness for PKEMNO NIZK) Consider the following game between a challenger and an adversary A:

1. **Stage 1:** $A(1^k)$ outputs a public key pk and a proof-message pair (π, m).
2. **Stage 2:** The challenger verifies the proof-message pair using pk and outputs a decision.

The adversary's advantage is defined as the probability

$$\mathrm{Adv}_{\text{s-proof-snd}}^{\text{PKEMNO-NIZK},A}(1^k) := \Pr[1 \leftarrow \mathrm{Ver}(pk, \pi, m') \wedge m' \neq m].$$

A PKEMNO NIZK scheme is strongly proof sound if any PPT adversary A has a negligible advantage. We define Game 3 between a challenger Chal and adversary $\mathcal{A}$ to show non-adaptive soundness:

```
Setup: Chal runs key generation algorithm to obtain $(\mathsf{pk}, \mathsf{sk})$ and gives $\mathsf{pk}$to $\mathcal{A}$.
Challenge:
  $\mathcal{A}$ selects challenge messages $m_1, \ldots, m_n$
  $\mathcal{A}$ sends message vector $\vec{m} = (m_1, \ldots, m_n)$ to Chal
Proof Generation:
  Chal computes proofs $\pi_1, \ldots, \pi_n$ where $\pi_i = \mathbf{Prove}(sk, m_i)$
  Chal sends proof vector $\vec{\pi} = (\pi_1, \ldots, \pi_n)$ to $\mathcal{A}$
Forgery Attempt:
  $\mathcal{A}$ outputs $m^{\mathcal{A}}\pi^{\mathcal{A}}$ where $m^*$ not in $\vec{m}$
Winning Condition:
  If $\mathbf{F}(\mathsf{pk}, m`\pi)$ true, then $\mathcal{A}$wins the game.

```

We argue that $\mathcal{A}$ wins only with negligible probability based on non-adaptive soundness of the NIZK proofs.

11.4 Committing Property - 4

Definition 4 (Committing Property for PKEMNO NIZK)
An alternative strong notion of soundness (with adversarially chosen keys) follows the idea that for any given proof π, one can only find one valid message m such that the pair (π, m) is valid under a public key pk. The scheme is strongly committing if, for any adversary A that outputs (pk, π, m, m') on input 1^k, the following probability is negligible:

$$\mathrm{Adv}_{\text{s-com}}^{\text{PKEMNO-NIZK},A}(1^k) := \Pr[1 \leftarrow \mathrm{Ver}(pk, \pi, m) \wedge 1 \leftarrow \mathrm{Ver}(pk, \pi, m') \wedge m \neq m']. \tag{2}$$

Given the unique characteristics of the PKEMNO NIZK Secure Messaging Protocol, particularly the absence of a traditional decryption function and the use of the *ZKVolute* operation, we adapt and introduce new security definitions suitable for this protocol. We define a version of proof soundness tailored to this protocol and introduce a new notion related to the integrity of the proof and message.

11.5 Proof-Message Pair - 5

Definition 5 (Integrity of Proof-Message Pair). This definition ensures that for a given proof π, there is a unique valid message m such that the pair (π, m) is valid under a public key pk.

$$\mathrm{Adv}_{\mathrm{integrity}}^{\mathrm{PKEMNO},A}(1^k) := \Pr[1 \leftarrow \mathrm{Ver}(pk, \pi, m) \wedge 1 \leftarrow \mathrm{Ver}(pk, \pi, m') \wedge m \neq m'].$$

The probability should be negligible for any PPT adversary A, ensuring that each proof uniquely authenticates a single message.

The PKEMNO NIZK Secure Messaging Protocol is both strongly proof sound and strongly committing under these definitions.

```
1 :  **Setup:** $\mathcal{C} \leftarrow \mathbf{Gen}(1^\lambda)(\mathsf{pk}, \mathsf{sk})$
2 :    $\mathcal{C}$ gives $\mathsf{pk}$ to adversary $\mathcal{A}$
3 :  **Query:**
4 :    $\mathcal{A}$ makes polynomially many queries $m_i$
5 :    $\mathcal{C}$ computes $\pi_i \leftarrow \mathrm{Prove}(sk, m_i)$
6 :    $\mathcal{C}$ sends $\pi_i$ to $\mathcal{A}$
7 :  **Forgery Attempt:**
8 :    $\mathcal{A}$ outputs $\pi$'$m_0, m_1$ s.t. $m_0 \neq m_1$
9 :  **Winning Condition:**
10 :   $\mathcal{A}$ wins if Verify$(pk, \pi$'$m_0) = 1 \wedge$
11 :   Verify$(pk, \pi$'$m_1) = 1$
12 : **Define advantage as:**
13 :   $\mathrm{Adv}^{\mathrm{int}}\mathcal{A}(\lambda) = \Pr[\mathcal{A}$ wins$]$
14 : **The scheme satisfies integrity if**
15 :   $\mathrm{Adv}^{\mathrm{int}}\mathcal{A}(\lambda)$ is negligible
16 :
```

The computation of the probability that an honest verifier accepts a forged proof (proof soundness) is complex and is not covered in this paper. Based on experimental testing, we conservatively estimate the error rate as $p \leq .001$, meaning the probability of accepting a random forgery is extremely low. To date, no random forgeries have ever been accepted. If further analysis shows the mathematically calculated forgery probability is higher than desired, additional proof techniques can be implemented, such as appending successive proof values to lower the chance of forgery. Each additional proof value exponentially reduces the forgery acceptance probability.

Appendix

A Lattice Cryptography

A.1 Introduction

Most explanations of lattice cryptography introduce the concept using multidimensional graph paper as a metaphor, but this often falls short of conveying a comprehensive mental model. We attempt to provide a more useful perspective without relying heavily on formulas. This section aims to elucidate lattice cryptography and to apply this understanding to existing systems and our novel findings. We explore the novel application of Number Theoretic Transforms (NTT) to create functions that are binding and hiding while maintaining a usable linear relationship between inputs. This work can be seen as a lattice trapdoor-based proof of knowledge or a new subcategory of 'somewhat' or 'partially' homomorphic cryptographic systems. By amplifying the dimensionality of the lattice construct internally, we significantly reduce the size of cryptographic keys and variables needed for communication at a given security level.

To grasp the details, it's easier to set aside the concept of dimensions initially and focus on the methodology of applied cryptography and how math represents the vast space of operations. Rather than visualizing the geometry of numbers, discussing the set of linear equations representing the lattice proves more insightful.

A.2 Algebraic Representation

The computational complexity of lattice problems stems from the volume of variables required to construct the system of linear equations describing a lattice. Each variable in the equations adds another dimension to the lattice that needs to be solved for. Applied cryptography often involves efficiently emulated lattice computation, avoiding actual algebra as much as possible while maintaining functionality. Cryptographers leverage the fact that many computations on long polynomials can be condensed into a single instance of a number theoretic transform. The degree of the polynomial (number of coefficients) determines the NTT size and ultimately the lattice's dimensionality. A single NTT transformation, followed by pairwise multiplication and an inverse transform, costs $\mathrm{O}(n \log n)$, compared to $O(n^2)$, a faster path to the same result. This process involves two arrays of polynomial coefficients defined over an algebraic ring, typically represented by a cyclotomic polynomial, as inputs. The forward NTT transform function is applied to these input polynomials, resulting in the "NTT representation" of these inputs.

A.3 Operations in NTT Space

Once inputs are transformed into "representations," an fixed set of modular operations follows, involving pairwise addition and/or multiplication, yielding a new polynomial defined within the modulus. After the arithmetic, applying the inverse NTT transform returns the newly computed polynomial coefficients to the original input domain. In lattice cryptography, variations to this ordering of operations, or operations beyond modular addition and multiplication, are rare. This pattern is standard in the field.

Performing a forward transformation, applying modular arithmetic, and then inversely transforming back to the input domain appears to be universal design pattern. The expected result is that modular addition of two polynomials in NTT space becomes vector addition in the input space, while pairwise modular multiplication in NTT domain performs a more complex formal convolutional operation on the two inputs in the input domain. The key idea is that pairwise multiplication of two polynomials in NTT space results in their convolution in the input space.

NIST describes the NTT as identical to the Discrete Fourier Transform, which is more widely recognized and studied. While the Number Theoretic Transform is often treated as a mathematical trick in cryptography to speed up convolutional calculations, it holds untapped potential for further exploration.

A.4 Reframing the Problem

Problems defined over high-dimensional lattice structures can be reframed as related to multidimensional convolution and deconvolution in signal processing, with added noise to hinder harmonic analysis or heuristic deconvolution attempts. The deconvolution of original inputs, be they signals or numbers in a lattice, is considered a hard problem. It's crucial to note that convolution output results using modular arithmetic are both associative and commutative, allowing for flexible orderings of operations.

A.5 Visualization Section

One can visualize the transformation process as teleporting (NTT forward transform) chains of numbers to another dimension, blending them (modular arithmetic), and then teleporting back (NTT inverse transform). This standard process in lattice cryptography forms the foundation of our methods.

Instead, imagine teleporting chains of numbers to an alternate dimension and accompanying them. In this first new dimension, chains combine into a third chain, and everything shifts to another, new, more complex dimension. After five such teleportations, a "5th-degree convolution" summation is stage by stage un-teleported(NTT inversion) back to the original input universe.

A.6 Complexity

Algebraically, each additional nested transform adds a block of variables to the equations representing the lattice problem. Using polynomials with 256 coefficients, each iteration adds 256 variables to the system of linear equations. After five nested transforms, we require 1280 variables, akin to solving a complex subset sum problem over a 1280-dimensional compound non-uniform lattice. The process is described semi-formally below:

Algorithm 19 Polynomial Convolution via NTT

```
Inputs: $A, B$ as degree $n$ polynomials
Operations: $\mathrm{NTT}_{1...5}$ as NTT transforms using moduli $p_{1...5}$
and $\mathrm{INV\_NTT}_{1...5}$ as corresponding inverse functions
Output: $C$ as a merged degree $n$ polynomial
$A \leftarrow \mathrm{NTT}_1(A)$
$B \leftarrow \mathrm{NTT}_1(B)$
$C \leftarrow (A + B) \mod p_1$
for $i = 2$ to 5 do
    $A \leftarrow \mathrm{NTT}_i(A)$
    $B \leftarrow \mathrm{NTT}_i(B)$
    $C \leftarrow \mathrm{NTT}_i(C)$
    $C \leftarrow (A + B + C) \mod p_i$
end for
$C \leftarrow \mathrm{INV\_NTT}_5(C)$
$C \leftarrow \mathrm{INV\_NTT}_4(C)$
$C \leftarrow \mathrm{INV\_NTT}_3(C)$
$C \leftarrow \mathrm{INV\_NTT}_2(C)$
$C \leftarrow \mathrm{INV\_NTT}_1(C)$
return $C$
```

A.7 Initial Examples

The basic construction we are exploring begins with an NTT transform based on the Fermat prime 257, which then performs modular arithmetic, and then repeats this process four additional times. Each layer transforms the inputs, aggregates them, and carries, by means of NTT transformation, the cumulative results to a final stage NTT using a large-bit prime modulus. This complex process results in a final mixed output polynomial from a geometric space that is challenging to articulate.

Nevertheless, we apply the five corresponding number theoretic inverse functions to this output polynomial, transforming it back to p257 and then to the original input domain. The resulting polynomial has convolution-like properties — it is associative, commutative, and serves as a proof of knowledge of the inputs without revealing any useful information about them beyond the fact that they were known.

A.8 Internal Convolution Operations

A critical aspect of the NTT forward transform is that it inherently includes butterfly addition and multiplication, thereby creating a circulant convolutional function. The ability to invert such a complex convolutional scheme implies the capability to invert standard NTT convolutions, which would challenge the foundations of lattice cryptography and many deconvolution assumptions. Rather than addressing problems like the shortest vector problem (SVP) or shortest integer solution (SIS), our approach resembles a spanning subset sum problem that requires solutions to traverse multiple distinct lattices. This internal amplification of dimensionality is what facilitates reduced communication costs, resulting in smaller keys and ciphertexts while maintaining comparable security to other lattice systems.

A.9 Equivalence Check

Our approach leverages the fact that traditional convolutions between two 'signals' or polynomial vectors are both associative and commutative. This property allows us to use modular convolution for relational comparison as a proof of knowledge. Given that A and B have undergone a form of complex convolution to create C, where deconvolution of C is well-posed and computationally infeasible, we can utilize the associative and commutative nature of modular convolution. If A is our secret polynomial and B is a public challenge polynomial, our public key is the complex convolution of (A+B)=PK', with the full public key being (PK',B).

To prove knowledge of A, which was used to create PK' using B, we convolve a second challenge M as (A,M) = C and publish C as the proof. A verifier can compute the convolution of (PK',M) and (C,B), equating to ((A+B)+M) and ((A+M)+B). Without knowledge of A, forging a complex convolution (A+M) such that ((A+M)+B) == ((A+B)+M) is computationally challenging. This problem reduces to a variant of the subset sum over lattice problem or a well-posed deconvolution/reconstruction problem, both believed to be quantum resilient[4][6].

A.10 Final Remarks

In conclusion, we informally describe the difference between out lattice construction and a traditional one. We describe a OW-CPA (One-Way Chosen Plaintext Attack Resistant) scheme related to the subset sum problem, defined over high-dimensional lattices as the basis for computational complexity. From this foundation, we can apply various transforms to produce an IND-CCPA secure Key Exchange Mechanism (KEM) and an EU-CMA digital signature using the Fiat-Shamir[7] sigma transform. Other unique applications, such as identity and key-policy based cryptography, are also conceivable.

B Python NTT and Inverse NTT

```python
#!/usr/bin/python3
import numpy as np
import operator
import random
np.set_printoptions(suppress=True, precision=6)

def pointwise_multiplication(vec1, vec2, modulus):
    return [(x * y) % modulus for x, y in zip(vec1, vec2)]

def pointwise_addition(vec1, vec2, modulus):
    return [(x + y) % modulus for x, y in zip(vec1, vec2)]

def ntt_inverse(a, MODULUS, ROOT_OF_UNITY, original_n=128):
    NTT_SIZE = 128
    n = len(a)
    if n == 1:
        return a
    inv_n = pow(original_n, MODULUS - 2, MODULUS)
    root_inv = pow(ROOT_OF_UNITY, (MODULUS - 1) - (MODULUS - 1) // n, MODULUS)
    w_inv = 1
    y0 = ntt_inverse(a[::2], MODULUS, ROOT_OF_UNITY, original_n=original_n)
    y1 = ntt_inverse(a[1::2], MODULUS, ROOT_OF_UNITY, original_n=original_n)
    y = [0] * n
    for k in range(n // 2):
        y[k] = (y0[k] + w_inv * y1[k]) % MODULUS
        y[k + n // 2] = (y0[k] - w_inv * y1[k]) % MODULUS
        w_inv = (w_inv * root_inv) % MODULUS
    return [(x * inv_n) % MODULUS for x in y] if n == original_n else y

def ntt(a, MODULUS, ROOT_OF_UNITY, depth=0):
    NTT_SIZE = 128
    n = len(a)
    if n == 1:
       return a
    w = 1
    root = pow(ROOT_OF_UNITY, (MODULUS - 1) // n, MODULUS)
    a0 = ntt(a[::2], MODULUS, ROOT_OF_UNITY, depth+1)
    a1 = ntt(a[1::2], MODULUS, ROOT_OF_UNITY, depth+1)
    y = [0] * n
    for k in range(n // 2):
        y[k] = (a0[k] + w * a1[k]) % MODULUS
        y[k + n // 2] = (a0[k] - w * a1[k]) % MODULUS
        w = w * root % MODULUS
    return y
```

Bibliography

[1] William Barker, William Polk, and Murugiah Souppaya. "Getting ready for post-quantum cryptography: explore challenges associated with adoption and use of post-quantum cryptographic algorithms". In: *The Publications of NIST Cyber Security White Paper (DRAFT), CSRC, NIST, GOV* 26 (2020).

[2] Josh Benaloh and Michael de Mare. "One-Way Accumulators: A Decentralized Alternative to Digital Sinatures (Extended Abstract)". In: *International Conference on the Theory and Application of Cryptographic Techniques*. 1994. URL: https://api.semanticscholar.org/CorpusID:3075498.

[3] Daniel J. Bernstein et al. "The SPHINCS+ Signature Framework". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 2129–2146. ISBN: 9781450367479. DOI: 10.1145/3319535.3363229. URL: https://doi.org/10.1145/3319535.3363229.

[4] Pierre Bonnetain et al. "Improved Classical and Quantum Algorithms for Subset-Sum". In: *IACR Cryptology ePrint Archive* 2020.168 (2020).

[5] Ivan Damgård et al. “Public-Key Encryption with Non-interactive Opening”. In: Jan. 2008, pp. 239–255. ISBN: 978-3-540-79262-8. DOI: 10.1007/978-3-540-79263-5_15.

[6] Alexey Yu Daskin. “Quantum Approach to Subset-Sum and Similar Problems”. In: *arXiv preprint arXiv:1707.08730* (2017).

[7] Jelle Don et al. “Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model”. In: *CoRR* abs/1902.07556 (2019). arXiv: 1902.07556. URL: http://arxiv.org/abs/1902.07556.

[8] Léo Ducas et al. “Dilithium: A high-speed lattice-based digital signature scheme”. In: *CCS 2019*. 2019, pp. 897–918.

[9] David Galindo et al. “Public-Key Encryption with Non-Interactive Opening: New Constructions and Stronger Definitions”. In: vol. 6055. May 2010, pp. 333–350. ISBN: 978-3-642-12677-2. DOI: 10.1007/978-3-642-12678-9_20.

[10] Andreas Huelsing et al. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391. URL: https://www.rfc-editor.org/info/rfc8391.

[11] Zhichuang Liang and Yunlei Zhao. *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*. 2022. arXiv: 2211.13546 [cs.CR].

[12] Vasileios Mavroeidis et al. “The Impact of Quantum Computing on Present Cryptography”. In: *International Journal of Advanced Computer Science and Applications* 9.3 (2018). ISSN: 2158-107X. DOI: 10.14569/ijacsa.2018.090354. URL: http://dx.doi.org/10.14569/IJACSA.2018.090354.

[13] Xiaoyun Wang, Guangwu Xu, and Yang Yu. “Lattice-Based Cryptography: A Survey”. In: *Chinese Annals of Mathematics, Series B* 44.6 (2023), pp. 945–960. DOI: 10.1007/s11401-023-0053-6. URL: https://doi.org/10.1007/s11401-023-0053-6.