Workgroup: None
Internet-Draft: draft-cfrg-schwabe-kyber-04
Published: 2 January 2024
Intended Status: Informational
Expires: 5 July 2024
Authors: P. Schwabe
MPI-SP & Radboud University
B. E. Westerbaan
Cloudflare

Kyber Post-Quantum KEM

Abstract

This memo specifies a preliminary version ("draft00", "v3.02") of Kyber, an IND-CCA2 secure Key Encapsulation Method.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://bwesterb.github.io/draft-schwabe-cfrg-kyber/draft-cfrg-schwabe-kyber.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-cfrg-schwabe-kyber/.

Source for this draft and an issue tracker can be found at https://github.com/bwesterb/draft-schwabe-cfrg-kyber.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 5 July 2024.

Copyright Notice

Table of Contents

1. Introduction
  1.1. Warning on stability and relation to ML-KEM
2. Conventions and Definitions
3. Overview
4. The field GF(q)
  4.1. Size
  4.2. Compression
5. The ring Rq
  5.1. Operations
    5.1.1. Size of polynomials
    5.1.2. Addition and subtraction
    5.1.3. Multiplication
6. Symmetric cryptographic primitives
7. Sampling of polynomials
  7.1. Uniformly
    7.1.1. sampleMatrix
  7.2. From a binomial distribution
    7.2.1. sampleNoise
8. Vector and matrices
  8.1. Operations on vectors
  8.2. Dot product and matrix multiplication
  8.3. Transpose
9. Serialization
  9.1. OctetsToBits
  9.2. Encode and Decode
    9.2.1. Polynomials
    9.2.2. Vectors
10. Inner malleable public-key encryption scheme
  10.1. Parameters
  10.2. Key generation
  10.3. Encryption
  10.4. Decryption
11. Kyber
  11.1. Key generation
  11.2. Encapsulation
  11.3. Decapsulation
12. Parameter sets

13. Machine-readable specification
14. Security Considerations
15. References
  15.1. Normative References
  15.2. Informative References
Appendix A. Acknowledgments
Appendix B. Change Log
  B.1. Since draft-schwabe-cfrg-kyber-03
  B.2. Since draft-schwabe-cfrg-kyber-02
  B.3. Since draft-schwabe-cfrg-kyber-01
  B.4. Since draft-schwabe-cfrg-kyber-00
Authors' Addresses

1. Introduction

Kyber is NIST's pick for a post-quantum key agreement [NISTR3].

Kyber is not a Diffie-Hellman (DH) style non-interactive key agreement, but instead, Kyber is a Key Encapsulation Method (KEM). A KEM is a three-tuple of algorithms (*KeyGen*, *Encapsulate*, *Decapsulate*):

- *KeyGen* takes no inputs and generates a private key and a public key;

- *Encapsulate* takes as input a public key and produces as output a ciphertext and a shared secret;

- *Decapsulate* takes as input a ciphertext and a private key and produces a shared secret.

Like DH, a KEM can be used as an unauthenticated key-agreement protocol, for example in TLS [HYBRID] [XYBERTLS]. However, unlike DH, a KEM-based key agreement is *interactive*, because the party executing Encapsulate can compute its protocol message (the ciphertext) only after having received the input (public key) from the party running *KeyGen* and *Decapsulate*.

A KEM can be transformed into a PKE scheme using HPKE [RFC9180] [XYBERHPKE].

1.1. Warning on stability and relation to ML-KEM

NOTE This draft is not stable and does not (yet) match the final NIST standard ML-KEM (FIPS 203) expected in 2024. It also does not match the draft for ML-KEM published by NIST August 2023. [MLKEM]

Currently it matches Kyber as submitted to round 3 of the NIST PQC process [KYBERV302].

2. Conventions and Definitions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Overview

Kyber is an IND-CCA2 secure KEM. It is constructed by applying a Fujisaki-Okamato style transformation on InnerPKE, which is the underlying IND-CPA secure Public Key Encryption scheme. We cannot use InnerPKE directly, as its ciphertexts are malleable.

```
                F.O. transform
InnerPKE   ---------------------->   Kyber
IND-CPA                              IND-CCA2
```

Kyber is a lattice-based scheme. More precisely, its security is based on the learning-with-errors-and-rounding problem in module lattices (MLWER). The underlying polynomial ring R (defined in Section 5) is chosen such that multiplication is very fast using the number theoretic transform (NTT, see Section 5.1.3.1).

An InnerPKE private key is a vector *s* over R of length k which is *small* in a particular way. Here k is a security parameter akin to the size of a prime modulus. For Kyber512, which targets AES-128's security level, the value of k is 2, for Kyber768 (AES-192 security level) k is 3, and for Kyber1024 (AES-256 security level) k is 4.

The public key consists of two values:

* *A* a k-by-k matrix over R sampled uniformly at random *and*

* *t = A s + e*, where e is a suitably small masking vector.

Distinguishing between such A s + e and a uniformly sampled t is the decision module learning-with-errors (MLWE) problem. If that is hard, then it is also hard to recover the private key from the public key as that would allow you to distinguish between those two.

To save space in the public key, A is recomputed deterministically from a 256bit seed *rho*. Strictly speaking, A is not uniformly random anymore, but it's computationally indistuinguishable from it.

A ciphertext for a message m under this public key is a pair (c_1, c_2) computed roughly as follows:

```
c_1 = Compress(A^T r + e_1, d_u)
c_2 = Compress(t^T r + e_2 + Decompress(m, 1), d_v)
```

where

- *e_1, e_2 and r are small blinds;
- *Compress(-, d) removes some information, leaving d bits per coefficient and Decompress is an "approximate inverse" of Compress;
- *d_u, d_v are scheme parameters; and
- *superscript T denotes transposition, so *A^T* is the transpose of A, see Section 8.3 and *t^T r* is the dot product of t and r, see Section 8.2.

Distinguishing such a ciphertext and uniformly sampled (c_1, c_2) is an example of the full MLWER problem, see Section 4.4 of [KYBERV302].

To decrypt the ciphertext, one computes

```
m = Compress(Decompress(c_2, d_v) - s^T Decompress(c_1, d_u), 1).
```

It it not straight-forward to see that this formula is correct. In fact, there is negligable but non-zero probability that a ciphertext does not decrypt correctly given by the DFP column in Table 4. This failure probability can be computed by a careful automated analysis of the probabilities involved, see kyber_failure.py of [SECEST].

To define all these operations precisely, we first define the field of coefficients for our polynomial ring; what it means to be small; and how to compress. Then we define the polynomial ring R; its operations and in particular the NTT. We continue with the different methods of sampling and (de)serialization. Then, we first define InnerPKE and finally Kyber proper.

4. The field GF(q)

Kyber is defined over GF(q) = Z/qZ, the integers modulo q = 13*2^8+1 = 3329.

4.1. Size

To define the size of a field element, we need a signed modulo. For any odd m, we write

```
a smod m
```

for the unique integer b with -(m-1)/2 < b <= (m-1)/2 and b = a modulo m.

To avoid confusion, for the more familiar modulo we write umod; that is

```
a umod m
```

is the unique integer b with 0 <= b < m and b = a modulo m.

Now we can define the norm of a field element:

```
|| a || = abs(a smod q)
```

Examples:

```
 3325 smod q = -4        ||  3325 || = 4
-3320 smod q =  9        || -3320 || = 9
```

TODO (#23) Should we define smod and umod at all, since we don't use it.Bas

4.2. Compression

In several parts of the algorithm, we will need a method to compress fied elements down into d bits. To do this, we use the following method.

For any positive integer d, integer x and integer 0 <= y < 2^d, we define

```
  Compress(x, d) = Round( (2^d / q) x ) umod 2^d
Decompress(y, d) = Round( (q / 2^d) y )
```

where Round(x) rounds any fraction to the nearest integer going up with ties.

Note that in Section 8.1 we extend Compress and Decompress to polynomials and vectors of polynomials.

These two operations have the following properties:

- *0 <= Compress(x, d) < 2^d
- *0 <= Decompress(y, d) < q
- *Compress(Decompress(y, d), d) = y
- *If Decompress(Compress(x, d), d) = x', then || x' - x || <= Round(q/2^(d+1))

*If x = x' modulo q, then Compress(x, d) = Compress(x', d)

For implementation efficiency, these can be computed as follows.

```
  Compress(x, d) = Div( (x << d) + q/2), q ) & ((1 << d) - 1)
Decompress(y, d) = (q*y + (1 << (d-1))) >> d
```

where Div(x, q) = Floor(x / q). TODO Do we want to include the proof that this is correct? Do we need to define >> and <<?Bas To prevent leaking the secret key, this must be computed in constant time [KYBERSLASH]. On platforms where Div is not constant-time, the following equation is useful, which holds for those x that appear in the previous formula for 0 < d < 12.

```
Div(x, q) = (20642679 * x) >> 36
```

5. The ring Rq

Kyber is defined over a polynomial ring Rq = GF(q)[x]/(x^n+1) where n=256 (and q=3329). Elements of Rq are tuples of 256 integers modulo q. We will call them polynomials or elements interchangeably.

A tuple a = (a_0, ..., a_255) represents the polynomial

```
a_0 + a_1 x + a_2 x^2 + ... + a_255 x^255.
```

With polynomial coefficients, vector and matrix indices, we will start counting at zero.

5.1. Operations

5.1.1. Size of polynomials

For a polynomial a = (a_0, ..., a_255) in R, we write:

```
|| a || = max_i || a_i ||
```

Thus a polynomial is considered large if one of its components is large.

5.1.2. Addition and subtraction

Addition and subtraction of elements is componentwise. Thus

```
(a_0, ..., a_255) + (b_0, ..., b_255) = (a_0 + b_0, ..., a_255 + b_255),
```

and

```
(a_0, ..., a_255) - (b_0, ..., b_255) = (a_0 - b_0, ..., a_255 - b_255),
```

where addition/subtractoin in each component is computed modulo q.

5.1.3. Multiplication

Multiplication is that of polynomials (convolution) with the additional rule that x^256=-1. To wit

```
(a_0, ..., a_255) \* (b_0, ..., b_255)
    = (a_0 * b_0 - a_255 * b_1 - ... - a_1 * b_255,
       a_0 * b_1 + a_1 * b_0 - a_255 * b_2 - ... - a_2 * b_255,

            ...

       a_0 * b_255 + ... + a_255 * b_0)
```

We will not use this schoolbook multiplication to compute the product. Instead we will use the more efficient, number theoretic transform (NTT), see Section 5.1.3.1.

5.1.3.1. Background on the Number Theoretic Transform (NTT)

The modulus q was chosen such that 256 divides into q-1. This means that there are zeta with

```
zeta^128 = -1  modulo  q
```

With such a zeta, we can almost completely split the polynomial x^256+1 used to define R over GF(q):

```
x^256 + 1 = x^256 - zeta^128
          = (x^128 - zeta^64)(x^128 + zeta^64)
          = (x^128 - zeta^64)(x^128 - zeta^192)
          = (x^64 - zeta^32)(x^64 + zeta^32)
                (x^64 - zeta^96)(x^64 + zeta^96)

            ...

          = (x^2 - zeta)(x^2 + zeta)(x^2 - zeta^65)(x^2 + zeta^65)
                   ... (x^2 - zeta^127)(x^2 + zeta^127)
```

Note that the powers of zeta that appear in the second, fourth, ..., and final lines are in binary:

```
0100000 1100000
0010000 1010000 0110000 1110000
0001000 1001000 0101000 1101000 0011000 1011000 0111000 1111000
            ...
0000001 1000001 0100001 1100001 0010001 1010001 0110001 ... 1111111
```

That is: brv(2), brv(3), brv(4), ..., where brv(x) denotes the 7-bit bitreversal of x. The final line is brv(64), brv(65), ..., brv(127).

These polynomials x^2 +- zeta^i are irreducible and coprime, hence by the Chinese Remainder Theorem for commutative rings, we know

```
R = GF(q)[x]/(x^256+1) -> GF(q)[x]/(x^2-zeta) x ... x GF(q)[x]/(x^2+zeta
```

given by a |-> (a mod x^2 - zeta, ..., a mod x^2 + zeta^127) is an isomorphism. This is the Number Theoretic Transform (NTT). Multiplication on the right is much easier: it's almost componentwise, see Section 5.1.3.3.

A propos, the the constant factors that appear in the moduli in order can be computed efficiently as follows (all modulo q):

```
-zeta     = -zeta^brv(64)  = -zeta^{1 + 2 brv(0)}
 zeta     =  zeta^brv(64)  = -zeta^{1 + 2 brv(1)}
-zeta^65  = -zeta^brv(65)  = -zeta^{1 + 2 brv(2)}
 zeta^65  =  zeta^brv(65)  = -zeta^{1 + 2 brv(3)}
-zeta^33  = -zeta^brv(66)  = -zeta^{1 + 2 brv(4)}
 zeta^33  =  zeta^brv(66)  = -zeta^{1 + 2 brv(5)}

             ...

-zeta^127 = -zeta^brv(127) = -zeta^{1 + 2 brv(126)}
 zeta^127 =  zeta^brv(127) = -zeta^{1 + 2 brv(127)}
```

To compute a multiplication in R efficiently, one can first use the NTT, to go to the right "into the NTT domain"; compute the multiplication there and move back with the inverse NTT.

The NTT can be computed efficiently by performing each binary split of the polynomial separately as follows:

```
a |-> ( a mod x^128 - zeta^64, a mod x^128 + zeta^64 ),
  |-> ( a mod  x^64 - zeta^32, a mod  x^64 + zeta^32,
        a mod  x^64 - zeta^96, a mod  x^64 + zeta^96 ),

    et cetera
```

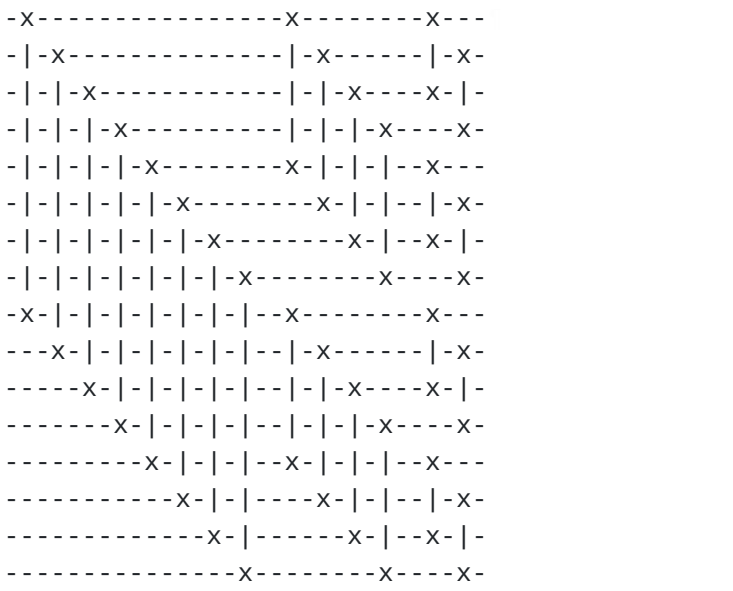
If we concatenate the resulting coefficients, expanding the definitions, for the first step we get:

```
a |-> (   a_0 + zeta^64 a_128,   a_1 + zeta^64 a_129,
         ...
        a_126 + zeta^64 a_254, a_127 + zeta^64 a_255,
          a_0 - zeta^64 a_128,   a_1 - zeta^64 a_129,
         ...
        a_126 - zeta^64 a_254, a_127 - zeta^64 a_255)
```

We can see this as 128 applications of the linear map CT_64, where

```
CT_i: (a, b) |-> (a + zeta^i b, a - zeta^i b)   modulo q
```

for the appropriate i in the following order, pictured in the case of n=16:



For n=16 there are 3 levels with 1, 2 and 4 row groups respectively. For the full n=256, there are 7 levels with 1, 2, 4, 8, 16, 32 and 64 row groups respectively. The appropriate power of zeta in the first level is brv(1)=64. The second level has brv(2) and brv(3) as powers of zeta for the top and bottom row group respectively, and so on.

The CT_i is known as a Cooley-Tukey butterfly. Its inverse is given by the Gentleman-Sande butterfly:

```
GS_i: (a, b) |-> ( (a+b)/2, zeta^-i (a-b)/2 )    modulo q
```

The inverse NTT can be computed by replacing CS_i by GS_i and flipping the diagram horizontally. TODO (#8) This section gives background not necessary for the implementation. Should we keep it?B as

5.1.3.1.1. Optimization notes

The modular divisions by two in the InvNTT can be collected into a single modular division by 128.

zeta^-i can be computed as -zeta^(128-i), which allows one to use the same precomputed table of powers of zeta for both the NTT and InvNTT.

TODO Add hints on lazy Montgomery reduction? Including https://eprint.iacr.org/2020/1377.pdfBas

5.1.3.2. NTT and InvNTT

As primitive 256th root of unity we use zeta=17.

As before, brv(i) denotes the 7-bit bitreversal of i, so brv(1)=64 and brv(91)=109.

The NTT is a linear bijection R -> R given by the matrix:

```
             [ zeta^{ (2 brv(i>>1) + 1) (j>>1) }              if i=j mo
(NTT)_{ij} = [
             [ 0                                              otherwise
```

Recall that we start counting rows and columns at zero. The NTT can be computed more efficiently as described in section Section 5.1.3.1.

The inverse of the NTT is called InvNTT. It is given by the matrix:

```
                    [ zeta^{ 256 - (2 brv(j>>1) + 1) (i>>1) }  if i=j mo
128 (InvNTT)_{ij} = [
                    [ 0                                       otherwise
```

Examples:

```
NTT(1, 1, 0, ..., 0)   = (1, 1, ..., 1, 1)
NTT(0, 1, 2, ..., 255) = (2429, 2845, 425, 1865, ..., 2502, 2134, 2717,
```

5.1.3.3. Multiplication in NTT domain

For elements a, b in R, we write a o b for multiplication in the NTT domain. That is: a * b = InvNTT(NTT(a) o NTT(b)). Concretely:

```
             [ a_i b_i + zeta^{2 brv(i >> 1) + 1} a_{i+1} b_{i+1}   if i
(a o b)_i = [
             [ a_{i-1} b_i + a_i b_{i-1}                            other
```

6. Symmetric cryptographic primitives

Kyber makes use of various symmertic primitives PRF, XOF, KDF, H and G, where

```
XOF(seed) = SHAKE-128(seed)
PRF(seed, counter) = SHAKE-256(seed || counter)
KDF(prekey) = SHAKE-256(msg)[:32]
H(msg) = SHA3-256(msg)
G(msg) = (SHA3-512(msg)[:32], SHA3-512(msg)[32:])
```

Here counter is an octet; seed is 32 octets; prekey is 64 octets; and the length of msg varies.

On the surface, they look different, but they are all based on the same flexible Keccak XOF that uses the f1600 permutation, see [FIPS202]:

```
XOF(seed)      =  Keccak[256](seed || 1111, .)
PRF(seed, ctr) =  Keccak[512](seed || ctr || 1111, .)
KDF(prekey)    =  Keccak[512](prekey || 1111, 256)
H(msg)         =  Keccak[512](msg || 01, 256)
G(msg)         = (Keccak[1024](msg || 01, 512)[:32],
                  Keccak[1024](msg || 01, 512)[32:])

Keccak[c] = Sponge[Keccak-f[1600], pad10*1, 1600-c]
```

The reason five different primitives are used is to ensure domain separation, which is crucial for security, cf. [H2CURVE] §2.2.5. Additionally, a smaller sponge capacity is used for performance where permissable by the security requirements.

7. Sampling of polynomials

7.1. Uniformly

The polynomials in the matrix A are sampled uniformly and deterministically from an octet stream (XOF) using rejection sampling as follows.

Three octets b_0, b_1, b_2 are read from the stream at a time. These are interpreted as two 12-bit unsigned integers d_1, d_2 via

```
d_1 + d_2 2^12 = b_0 + b_1 2^8 + b_2 2^16
```

This creates a stream of 12-bit ds. Of these, the elements >= q are ignored. From the resultant stream, the coefficients of the polynomial are taken in order. In Python:

```
def sampleUniform(stream):
    cs = []
    while True:
        b = stream.read(3)
        d1 = b[0] + 256*(b[1] % 16)
        d2 = (b[1] >> 4) + 16*b[2]
        for d in [d1, d2]:
            if d >= q: continue
            cs.append(d)
            if len(cs) == n: return Poly(cs)
```

Example:

```
sampleUniform(SHAKE-128('')) = (3199, 697, 2212, 2302, ..., 255, 846, 1)
```

7.1.1. sampleMatrix

Now, the *k* by *k* matrix *A* over *R* is derived deterministically from a 32-octet seed *rho* using sampleUniform as follows.

```
sampleMatrix(rho)_{ij} = sampleUniform(XOF(rho || octet(j) || octet(i))
```

That is, to derive the polynomial at the *i*th row and *j*th column, sampleUniform is called with the 34-octet seed created by first appending the octet *j* and then the octet *i* to *rho*. Recall that we start counting rows and columns from zero.

As the NTT is a bijection, it does not matter whether we interpret the polynomials of the sampled matrix in the NTT domain or not. For efficiency, we do interpret the sampled matrix in the NTT domain.

7.2. From a binomial distribution

Noise is sampled from a centered binomial distribution Binomial(2eta, 1/2) - eta deterministically as follows.

An octet array a of length 64*eta is converted to a polynomial CBD(a, eta)

```
CBD(a, eta)_i = b_{2i eta} + b_{2i eta + 1} + ... + b_{2i eta + eta-1}
              - b_{2i eta + eta} - ... - b_{2i eta + 2eta - 1},
```

where b = OctetsToBits(a).

Examples:

```
CBD((0, 1, 2, ..., 127), 2) = (0, 0, 1, 0, 1, 0, ..., 3328, 1, 0, 1)
CBD((0, 1, 2, ..., 191), 3) = (0, 1, 3328, 0, 2, ..., 3328, 3327, 3328,
```

7.2.1. sampleNoise

A *k* component small vector *v* is derived from a seed 32-octet seed *sigma*, an offset *offset* and size *eta* as follows:

```
sampleNoise(sigma, eta, offset)_i = CBD(PRF(sigma, octet(i+offset)), eta
```

Recall that we start counting vector indices at zero.

8. Vector and matrices

8.1. Operations on vectors

Recall that Compress(x, d) maps a field element x into {0, ..., 2^d-1}. In Kyber d is at most 11 and so we can interpret Compress(x, d) as a field element again.

In this way, we can extend Compress(-, d) to polynomials by applying to each coefficient separately and in turn to vectors by applying to each polynomial. That is, for a vector v and polynomial p:

```
Compress(p, d)_i = Compress(p_i, d)
Compress(v, d)_i = Compress(v_i, d)
```

8.2. Dot product and matrix multiplication

We will also use "o", from section Section 5.1.3.3, to denote the dot product and matrix multiplication in the NTT domain. Concretely:

1. For two length k vector v and w, we write

   ```
   v o w = v_0 o w_0 + ... + v_{k-1} o w_{k-1}
   ```

2. For a k by k matrix A and a length k vector v, we have

   ```
   (A o v)_i = A_i o v,
   ```

   where A_i denotes the (i+1)th row of the matrix A as we start counting at zero.

8.3. Transpose

For a matrix A, we denote by A^T the tranposed matrix. To wit:

```
A^T_ij = A_ji.
```

We define Decompress(-, d) for vectors and polynomials in the same way.

9. Serialization

9.1. OctetsToBits

For any list of octets a_0, ..., a_{s-1}, we define OctetsToBits(a), which is a list of bits of length 8s, defined by

```
OctetsToBits(a)_i = ((a_(i>>3)) >> (i umod 8)) umod 2.
```

Example:

```
OctetsToBits(12,45) = (0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0)
```

9.2. Encode and Decode

For an integer 0 < w <= 12, we define Decode(a, w), which converts any list a of w*l/8 octets into a list of length l with values in {0, ..., 2^w-1} as follows.

```
Decode(a, w)_i = b_{wi} + b_{wi+1} 2 + b_{wi+2} 2^2 + ... + b_{wi+w-1} 2
```

where b = OctetsToBits(a).

Encode(-, w) is the unique inverse of Decode(-, w)

9.2.1. Polynomials

A polynomial p is encoded by passing its coefficients to Encode:

```
EncodePoly(p, w) = Encode(p_0, p_1, ..., p_{n-1}, w)
```

DecodePoly(-, w) is the unique inverse of EncodePoly(-, w).

9.2.2. Vectors

A vector v of length k over R is encoded by concatenating the coefficients in the obvious way:

```
EncodeVec(v, w) = Encode((v_0)_0, ..., (v_0)_{n-1},
                         (v_1)_{0}, ..., (v_1)_{n-1},
                                ..., (v_{k-1})_{n-1}, w)
```

DecodeVec(-, w) is the unique inverse of EncodeVec(-, w).

10. Inner malleable public-key encryption scheme

We are ready to define the IND-CPA secure Public-Key Encryption scheme that underlies Kyber. It is unsafe to use this underlying scheme directly as its ciphertexts are malleable. Instead, a Public-Key Encryption scheme can be constructed on top of Kyber by using HPKE [RFC9180] [XYBERHPKE].

10.1. Parameters

We have already been introduced to the following parameters:

q Order of field underlying *R*.

n Length of polynomials in *R*.

zeta Primitive root of unity in GF(q), used for NTT in R.

XOF, H, G, PRF, KDF Various symmetric primitives.

k Main security parameter: the number of rows and columns in the matrix *A*.

Additionally, Kyber takes the following parameters

eta1, eta2

Size of small coefficients used in the private key and noise vectors.

d_u*, *d_v How many bits to retain per coefficient of the *u* and *v* components of the ciphertext.

The values of these parameters are given in Section 12.

10.2. Key generation

InnerKeyGen(seed) takes a 32 octet **seed** and deterministically produces a keypair as follows.

1. Set (rho, sigma) = G(seed).
2. Derive
    1. AHat = sampleMatrix(rho).
    2. s = sampleNoise(sigma, eta1, 0)
    3. e = sampleNoise(sigma, eta1, k)
3. Compute
    1. sHat = NTT(s)
    2. tHat = AHat o sHat + NTT(e)
4. Return
    1. publicKey = EncodeVec(tHat, 12) || rho
    2. privateKey = EncodeVec(sHat, 12)

Note that in essence we're simply computing t = A s + e.

10.3. Encryption

InnerEnc(msg, publicKey, seed) takes a 32-octet seed, and deterministically encrypts the 32-octet msg for the InnerPKE public key publicKey as follows.

1. Split publicKey into
    1. k*(n/8)*12-octet tHatPacked
    2. 32-octet rho
2. Parse tHat = DecodeVec(tHatPacked, 12)

3. Derive

    1. AHat = sampleMatrix(rho)

    2. r = sampleNoise(seed, eta1, 0)

    3. e_1 = sampleNoise(seed, eta2, k)

    4. e_2 = sampleNoise(seed, eta2, 2k)_0

4. Compute

    1. rHat = NTT(r)

    2. u = InvNTT(AHat^T o rHat) + e_1

    3. v = InvNTT(tHat o rHat) + e_2 + Decompress(DecodePoly(msg, 1), 1)

    4. c_1 = EncodeVec(Compress(u, d_u), d_u)

    5. c_2 = EncodePoly(Compress(v, d_v), d_v)

5. Return

    1. cipherText = c_1 || c_2

10.4. Decryption

InnerDec(cipherText, privateKey) takes an InnerPKE private key privateKey and decrypts a cipher text cipherText as follows.

1. Split cipherText into

    1. d_u*k*n/8-octet c_1

    2. d_v*n/8-octet c_2

2. Parse

    1. u = Decompress(DecodeVec(c_1, d_u), d_u)

    2. v = Decompress(DecodePoly(c_2, d_v), d_v)

    3. sHat = DecodeVec(privateKey, 12)

3. Compute

    1. m = v - InvNTT(sHat o NTT(u))

4. Return

    1. plainText = EncodePoly(Compress(m, 1), 1)

11. Kyber

Now we are ready to define Kyber itself.

11.1. Key generation

A Kyber keypair is derived deterministically from a 64-octet seed as follows.

1. Split seed into

    1. A 32-octet cpaSeed

    2. A 32-octet z

2. Compute

    1. (cpaPublicKey, cpaPrivateKey) = InnerKeyGen(cpaSeed)

    2. h = H(cpaPublicKey)

3. Return

    1. publicKey = cpaPublicKey

    2. privateKey = cpaPrivateKey || cpaPublicKey || h || z

11.2. Encapsulation

Kyber encapsulation takes a public key and generates a shared secret and ciphertext for the public key as follows.

1. Sample secret cryptographically-secure random 32-octet seed.

2. Compute

    1. m = H(seed)

    2. (Kbar, cpaSeed) = G(m || H(publicKey))

    3. cpaCipherText = InnerEnc(m, publicKey, cpaSeed)

3. Return

    1. cipherText = cpaCipherText

    2. sharedSecret = KDF(KBar || H(cpaCipherText))

11.3. Decapsulation

Kyber decapsulation takes a private key and a cipher text and returns a shared secret as follows.

1. Split privateKey into
    1. A 12*k*n/8-octet cpaPrivateKey
    2. A 12*k*n/8+32-octet cpaPublicKey
    3. A 32-octet h
    4. A 32-octet z
2. Compute
    1. m2 = InnerDec(cipherText, cpaPrivateKey)
    2. (KBar2, cpaSeed2) = G(m2 || h)
    3. cipherText2 = InnerEnc(m2, cpaPublicKey, cpaSeed2)
    4. K1 = KDF(KBar2 || H(cipherText))
    5. K2 = KDF(z || H(cipherText))
3. In constant-time, set K = K1 if cipherText == cipherText2 else set K = K2.
4. Return
    1. sharedSecret = K

For security, the implementation **MUST NOT** explicitly return or otherwise leak via a side-channel, decapsulation succeeded, viz cipherText == cipherText2.

12. Parameter sets

Name	Value	Description
q	3329	Order of base field
n	256	Degree of polynomials
zeta	17	nth root of unity in base field

Table 1: Common parameters to all versions of Kyber

Primitive	Instantiation
XOF	SHAKE-128

Primitive	Instantiation
H	SHA3-256
G	SHA3-512
PRF(s,b)	SHAKE-256(s || b)
KDF	SHAKE-256

Table 2: Instantiation of symmetric primitives in Kyber

Name	Description
k	Dimension of module
eta1, eta2	Size of "small" coefficients used in the private key and noise vectors.
d_u	How many bits to retain per coefficient of u, the private-key independent part of the ciphertext
d_v	How many bits to retain per coefficient of v, the private-key dependent part of the ciphertext.

Table 3: Description of kyber parameters

Parameter set	k	eta1	eta2	d_u	d_v	sec	DFP
Kyber512	2	3	2	10	4	I	2^-139
Kyber768	3	2	2	10	4	III	2^-164
Kyber1024	4	2	2	11	5	V	2^-174

Table 4: Kyber parameter sets with NIST security level (sec) and decryption failure probability (DFP)

Parameter set	ss	pk	ct	sk
Kyber512	32	800	768	1632
Kyber768	32	1184	1088	2400
Kyber1024	32	1568	1568	3168

Table 5: Kyber parameter sets with sizes of shared secret (ss), public key (pk), cipher text (ct) and private key (sk)

13. Machine-readable specification

```
# WARNING This is a specification of Kyber; not a production ready
# implementation. It is slow and does not run in constant time.

# Requires the CryptoDome for SHAKE. To install, run
#
#   pip install pycryptodome pytest
from Crypto.Hash import SHAKE128, SHAKE256

import io
import hashlib
import functools
import collections

from math import floor

q = 3329
nBits = 8
zeta = 17
eta2 = 2

n = 2**nBits
inv2 = (q+1)//2 # inverse of 2

params = collections.namedtuple('params', ('k', 'du', 'dv', 'eta1'))

params512  = params(k = 2, du = 10, dv = 4, eta1 = 3)
params768  = params(k = 3, du = 10, dv = 4, eta1 = 2)
params1024 = params(k = 4, du = 11, dv = 5, eta1 = 2)

def smod(x):
    r = x % q
    if r > (q-1)//2:
        r -= q
    return r

# Rounds to nearest integer with ties going up
def Round(x):
    return int(floor(x + 0.5))

def Compress(x, d):
    return Round((2**d / q) * x) % (2**d)

def Decompress(y, d):
    assert 0 <= y and y <= 2**d
    return Round((q / 2**d) * y)

def BitsToWords(bs, w):
    assert len(bs) % w == 0
    return [sum(bs[i+j] * 2**j for j in range(w))
            for i in range(0, len(bs), w)]
```

```
def WordsToBits(bs, w):
    return sum([[(b >> i) % 2 for i in range(w)] for b in bs], [])

def Encode(a, w):
    return bytes(BitsToWords(WordsToBits(a, w), 8))

def Decode(a, w):
    return BitsToWords(WordsToBits(a, 8), w)

def brv(x):
    """ Reverses a 7-bit number """
    return int(''.join(reversed(bin(x)[2:].zfill(nBits-1))), 2)

class Poly:
    def __init__(self, cs=None):
        self.cs = (0,)*n if cs is None else tuple(cs)
        assert len(self.cs) == n

    def __add__(self, other):
        return Poly((a+b) % q for a,b in zip(self.cs, other.cs))

    def __neg__(self):
        return Poly(q-a for a in self.cs)
    def __sub__(self, other):
        return self + -other

    def __str__(self):
        return f"Poly({self.cs}"

    def __eq__(self, other):
        return self.cs == other.cs

    def NTT(self):
        cs = list(self.cs)
        layer = n // 2
        zi = 0
        while layer >= 2:
            for offset in range(0, n-layer, 2*layer):
                zi += 1
                z = pow(zeta, brv(zi), q)

                for j in range(offset, offset+layer):
                    t = (z * cs[j + layer]) % q
                    cs[j + layer] = (cs[j] - t) % q
                    cs[j] = (cs[j] + t) % q
            layer //= 2
        return Poly(cs)

    def RefNTT(self):
```

```
        # Slower, but simpler, version of the NTT.
        cs = [0]*n
        for i in range(0, n, 2):
            for j in range(n // 2):
                z = pow(zeta, (2*brv(i//2)+1)*j, q)
                cs[i] = (cs[i] + self.cs[2*j] * z) % q
                cs[i+1] = (cs[i+1] + self.cs[2*j+1] * z) % q
        return Poly(cs)

    def InvNTT(self):
        cs = list(self.cs)
        layer = 2
        zi = n//2
        while layer < n:
            for offset in range(0, n-layer, 2*layer):
                zi -= 1
                z = pow(zeta, brv(zi), q)

                for j in range(offset, offset+layer):
                    t = (cs[j+layer] - cs[j]) % q
                    cs[j] = (inv2*(cs[j] + cs[j+layer])) % q
                    cs[j+layer] = (inv2 * z * t) % q
            layer *= 2
        return Poly(cs)

    def MulNTT(self, other):
        """ Computes self o other, the multiplication of self and other
            in the NTT domain. """
        cs = [None]*n
        for i in range(0, n, 2):
            a1 = self.cs[i]
            a2 = self.cs[i+1]
            b1 = other.cs[i]
            b2 = other.cs[i+1]
            z = pow(zeta, 2*brv(i//2)+1, q)
            cs[i] = (a1 * b1 + z * a2 * b2) % q
            cs[i+1] = (a2 * b1 + a1 * b2) % q
        return Poly(cs)

    def Compress(self, d):
        return Poly(Compress(c, d) for c in self.cs)

    def Decompress(self, d):
        return Poly(Decompress(c, d) for c in self.cs)

    def Encode(self, d):
        return Encode(self.cs, d)

def sampleUniform(stream):
```

```
    cs = []
    while True:
        b = stream.read(3)
        d1 = b[0] + 256*(b[1] % 16)
        d2 = (b[1] >> 4) + 16*b[2]
        assert d1 + 2**12 * d2 == b[0] + 2**8 * b[1] + 2**16*b[2]
        for d in [d1, d2]:
            if d >= q:
                continue
            cs.append(d)
            if len(cs) == n:
                return Poly(cs)

def CBD(a, eta):
    assert len(a) == 64*eta
    b = WordsToBits(a, 8)
    cs = []
    for i in range(n):
        cs.append((sum(b[:eta]) - sum(b[eta:2*eta])) % q)
        b = b[2*eta:]
    return Poly(cs)

def XOF(seed, j, i):
    h = SHAKE128.new()
    h.update(seed + bytes([j, i]))
    return h

def PRF(seed, nonce):
    assert len(seed) == 32
    h = SHAKE256.new()
    h.update(seed + bytes([nonce]))
    return h

def G(seed):
    h = hashlib.sha3_512(seed).digest()
    return h[:32], h[32:]

def H(msg): return hashlib.sha3_256(msg).digest()
def KDF(msg): return hashlib.shake_256(msg).digest(length=32)

class Vec:
    def __init__(self, ps):
        self.ps = tuple(ps)

    def NTT(self):
        return Vec(p.NTT() for p in self.ps)

    def InvNTT(self):
        return Vec(p.InvNTT() for p in self.ps)
```

```
    def DotNTT(self, other):
        """ Computes the dot product <self, other> in NTT domain. """
        return sum((a.MulNTT(b) for a, b in zip(self.ps, other.ps)),
                   Poly())

    def __add__(self, other):
        return Vec(a+b for a,b in zip(self.ps, other.ps))

    def Compress(self, d):
        return Vec(p.Compress(d) for p in self.ps)

    def Decompress(self, d):
        return Vec(p.Decompress(d) for p in self.ps)

    def Encode(self, d):
        return Encode(sum((p.cs for p in self.ps), ()), d)

    def __eq__(self, other):
        return self.ps == other.ps

def EncodeVec(vec, w):
    return Encode(sum([p.cs for p in vec.ps], ()), w)
def DecodeVec(bs, k, w):
    cs = Decode(bs, w)
    return Vec(Poly(cs[n*i:n*(i+1)]) for i in range(k))
def DecodePoly(bs, w):
    return Poly(Decode(bs, w))

class Matrix:
    def __init__(self, cs):
        """ Samples the matrix uniformly from seed rho """
        self.cs = tuple(tuple(row) for row in cs)

    def MulNTT(self, vec):
        """ Computes matrix multiplication A*vec in the NTT domain. """
        return Vec(Vec(row).DotNTT(vec) for row in self.cs)

    def T(self):
        """ Returns transpose of matrix """
        k = len(self.cs)
        return Matrix((self.cs[j][i] for j in range(k))
                      for i in range(k))

def sampleMatrix(rho, k):
    return Matrix([[sampleUniform(XOF(rho, j, i))
            for j in range(k)] for i in range(k)])

def sampleNoise(sigma, eta, offset, k):
    return Vec(CBD(PRF(sigma, i+offset).read(64*eta), eta)
               for i in range(k))
```

```
def constantTimeSelectOnEquality(a, b, ifEq, ifNeq):
    # WARNING! In production code this must be done in a
    # data-independent constant-time manner, which this implementation
    # is not. In fact, many more lines of code in this
    # file are not constant-time.
    return ifEq if a == b else ifNeq

def InnerKeyGen(seed, params):
    assert len(seed) == 32
    rho, sigma = G(seed)
    A = sampleMatrix(rho, params.k)
    s = sampleNoise(sigma, params.eta1, 0, params.k)
    e = sampleNoise(sigma, params.eta1, params.k, params.k)
    sHat = s.NTT()
    eHat = e.NTT()
    tHat = A.MulNTT(sHat) + eHat
    pk = EncodeVec(tHat, 12) + rho
    sk = EncodeVec(sHat, 12)
    return (pk, sk)

def InnerEnc(pk, msg, seed, params):
    assert len(msg) == 32
    tHat = DecodeVec(pk[:-32], params.k, 12)
    rho = pk[-32:]
    A = sampleMatrix(rho, params.k)
    r = sampleNoise(seed, params.eta1, 0, params.k)
    e1 = sampleNoise(seed, eta2, params.k, params.k)
    e2 = sampleNoise(seed, eta2, 2*params.k, 1).ps[0]
    rHat = r.NTT()
    u = A.T().MulNTT(rHat).InvNTT() + e1
    m = Poly(Decode(msg, 1)).Decompress(1)
    v = tHat.DotNTT(rHat).InvNTT() + e2 + m
    c1 = u.Compress(params.du).Encode(params.du)
    c2 = v.Compress(params.dv).Encode(params.dv)
    return c1 + c2

def InnerDec(sk, ct, params):
    split = params.du * params.k * n // 8
    c1, c2 = ct[:split], ct[split:]
    u = DecodeVec(c1, params.k, params.du).Decompress(params.du)
    v = DecodePoly(c2, params.dv).Decompress(params.dv)
    sHat = DecodeVec(sk, params.k, 12)
    return (v - sHat.DotNTT(u.NTT()).InvNTT()).Compress(1).Encode(1)

def KeyGen(seed, params):
    assert len(seed) == 64
    z = seed[32:]
    pk, sk2 = InnerKeyGen(seed[:32], params)
```

```
    h = H(pk)
    return (pk, sk2 + pk + h + z)

def Enc(pk, seed, params):
    assert len(seed) == 32

    m = H(seed)
    Kbar, r = G(m + H(pk))
    ct = InnerEnc(pk, m, r, params)
    K = KDF(Kbar + H(ct))
    return (ct, K)

def Dec(sk, ct, params):
    sk2 = sk[:12 * params.k * n//8]
    pk = sk[12 * params.k * n//8 : 24 * params.k * n//8 + 32]
    h = sk[24 * params.k * n//8 + 32 : 24 * params.k * n//8 + 64]
    z = sk[24 * params.k * n//8 + 64 : 24 * params.k * n//8 + 96]
    m2 = InnerDec(sk, ct, params)
    Kbar2, r2 = G(m2 + h)
    ct2 = InnerEnc(pk, m2, r2, params)
    return constantTimeSelectOnEquality(
        ct2, ct,
        KDF(Kbar2 + H(ct)),  # if ct == ct2
        KDF(z + H(ct)),      # if ct != ct2
    )
```

14. Security Considerations

Kyber512, Kyber768 and Kyber1024 are designed to be post-quantum IND-CCA2 secure KEMs, at the security levels of AES-128, AES-192 and AES-256.

The designers of Kyber recommend Kyber768.

The inner public key encryption **SHOULD NOT** be used directly, as its ciphertexts are malleable. Instead, for public key encryption, HPKE can be used to turn Kyber into IND-CCA2 secure PKE [RFC9180] [XYBERHPKE].

Any implementation **MUST** use implicit rejection as specified in Section 11.3.

15. References

15.1. Normative References

[FIPS202] National Institute of Standards and Technology, "FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", n.d., <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

15.2. Informative References

[H2CURVE] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[HYBRID] Stebila, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", Work in Progress, Internet-Draft, draft-stebila-tls-hybrid-design-03, 12 February 2020,

<https://datatracker.ietf.org/doc/html/draft-stebila-tls-hybrid-design-03>.

[KYBERSLASH] Bernstein, D. J., "KyberSlash: division timings depending on secrets in Kyber software", n.d., <https://kyberslash.cr.yp.to>.

[KYBERV302] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J., Schwabe, P., Seiler, G., and D. Stehle, "CRYSTALS-Kyber, Algorithm Specification And Supporting Documentation (version 3.02)", 2021, <https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf>.

[MLKEM] National Institute of Standards and Technology, "FIPS 203 (Initial Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard", n.d., <https://csrc.nist.gov/pubs/fips/203/ipd>.

[NISTR3] The NIST PQC Team, "PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates", n.d., <https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[SECEST] Ducas, L. and J. Schanck, "CRYSTALS security estimate scripts", n.d., <https://github.com/pq-crystals/security-estimates>.

[XYBERHPKE] Westerbaan, B. and C. A. Wood, "X25519Kyber768Draft00 hybrid post-quantum KEM for HPKE", Work in Progress, Internet-Draft, draft-westerbaan-cfrg-hpke-xyber768d00-02, 4 May 2023, <https://datatracker.ietf.org/doc/html/draft-westerbaan-cfrg-hpke-xyber768d00-02>.

[XYBERTLS] Westerbaan, B. and D. Stebila, "X25519Kyber768Draft00 hybrid post-quantum key agreement", Work in Progress, Internet-Draft, draft-tls-westerbaan-xyber768d00-03, 24 September 2023, <https://datatracker.ietf.org/doc/html/draft-tls-westerbaan-xyber768d00-03>.

Appendix A. Acknowledgments

The authors would like to thank C. Wood, Florence D., I. Liusvaara, J. Crawford, J. Schanck, M. Thomson, and N. Sullivan for their input and assistance.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-schwabe-cfrg-kyber-03

*Add note on KyberSlash.

B.2. Since draft-schwabe-cfrg-kyber-02

*Fix a typo in the machine-readable specification, and use a proper SHAKE implementation. #5

*Add table with sizes.

*Reordered sections.

*Add reference to Kyber in HPKE.

*Miscellaneous editorial changes.

*Remove encapsulation seed as an explicit parameter in the written specification.

*Write security recommendations. #18

*Explain relation with ML-KEM.

B.3. Since draft-schwabe-cfrg-kyber-01

*Fix various typos.

*Move sections around.

*Elaborate domain separation and encoding of nonces in symmetric primitives.

*Add explicit formula for InvNTT.

*Add acknowledgements.

B.4. Since draft-schwabe-cfrg-kyber-00

*Test specification against NIST test vectors.

* Fix two unintentional mismatches between this document and the reference implementation:

  1. KDF uses SHAKE-256 instead of SHAKE-128.

  2. Reverse order of seed. (z comes at the end.)

* Elaborate text in particular introduction, and symmetric key section.

Authors' Addresses

Peter Schwabe
MPI-SP & Radboud University

Bas Westerbaan
Cloudflare

Email: bas@cloudflare.com