FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security   **Subcategory: Cryptography**

Foreword

The Federal Information Processing Standards (FIPS) Publication Series of the National Institute of Standards and Technology is the official series of publications relating to standards and guidelines developed under 15 U.S.C. 278g-3, and issued by the Secretary of Commerce under 40 U.S.C. 11331.

Comments concerning this Federal Information Processing Standard publication are welcomed and should be submitted using the contact information in the "Inquiries and Comments" clause of the announcement section.

Kevin M. Stine, Director
Information Technology Laboratory

Abstract

A key-encapsulation mechanism (KEM) is a set of algorithms that, under certain conditions, can be used by two parties to establish a shared secret key over a public channel. A shared secret key that is securely established using a KEM can then be used with symmetric-key cryptographic algorithms to perform basic tasks in secure communications, such as encryption and authentication. This standard specifies a key-encapsulation mechanism called ML-KEM. The security of ML-KEM is related to the computational difficulty of the Module Learning with Errors problem. At present, ML-KEM is believed to be secure, even against adversaries who possess a quantum computer. This standard specifies three parameter sets for ML-KEM. In order of increasing security strength and decreasing performance, these are ML-KEM-512, ML-KEM-768, and ML-KEM-1024.

Keywords: computer security; cryptography; encryption; Federal Information Processing Standards; key-encapsulation mechanism; lattice-based cryptography; post-quantum; public-key cryptography.

Federal Information Processing Standards Publication 203

Published: August 13, 2024
Effective: August 13, 2024

Announcing the

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Federal Information Processing Standards (FIPS) publications are developed by the National Institute of Standards and Technology (NIST) under 15 U.S.C. 278g-3 and issued by the Secretary of Commerce under 40 U.S.C. 11331.

1. **Name of Standard.** Module-Lattice-Based Key-Encapsulation Mechanism Standard (FIPS 203).

2. **Category of Standard.** Computer Security. **Subcategory.** Cryptography.

3. **Explanation.** A cryptographic key (or simply "key") is represented in a computer as a string of bits. A *shared secret key* is a cryptographic key that is computed jointly by two parties (e.g., Alice and Bob) using a set of algorithms. Under certain conditions, these algorithms ensure that both parties will produce the same key and that this key is secret from adversaries. Such a shared secret key can then be used with symmetric-key cryptographic algorithms (specified in other NIST standards) to perform tasks such as encryption and authentication of digital information.

   This standard specifies a set of algorithms for establishing a shared secret key. While there are many methods for establishing a shared secret key, the particular method described in this standard is a key-encapsulation mechanism (KEM).

   In a KEM, the computation of the shared secret key begins with Alice generating a *decapsulation key* and an *encapsulation key*. Alice keeps the decapsulation key private and makes the encapsulation key available to Bob. Bob then uses Alice's encapsulation key to generate one copy of a shared secret key along with an associated *ciphertext*. Bob then sends the ciphertext to Alice. Finally, Alice uses the ciphertext from Bob along with Alice's private decapsulation key to compute another copy of the shared secret key.

   The security of the particular KEM specified in this standard is related to the computational difficulty of solving certain systems of noisy linear equations, specifically the *Module Learning With Errors* (MLWE) problem. At present, it is believed that this particular method of establishing a shared secret key is secure, even against adversaries who possess a quantum computer. In the future, additional KEMs may be specified and approved in FIPS publications or in NIST Special Publications.

4. **Approving Authority.** Secretary of Commerce.

5. **Maintenance Agency.** Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. **Applicability.** Federal Information Processing Standards apply to information systems used or operated by federal agencies or by a contractor of an agency or other organization on behalf of an agency. They do not apply to national security systems as defined in 44 U.S.C. 3552.

    This standard, or other FIPS or NIST Special Publications that specify alternative mechanisms, shall be used wherever the establishment of a shared secret key (or shared secret from which keying material can be generated) is required for federal applications, including the use of such a key with symmetric-key cryptographic algorithms, in accordance with applicable Office of Management and Budget and agency policies.

    The adoption and use of this standard are available to private and commercial organizations.

7. **Implementations.** A key-encapsulation mechanism may be implemented in software, firmware, hardware, or any combination thereof. For every computational procedure that is specified in this standard, a conforming implementation may replace the given set of steps with any mathematically equivalent set of steps. In other words, different procedures that produce the correct output for every input are permitted.

    NIST will develop a validation program to test implementations for conformance to the algorithms in this standard. Information about validation programs is available at https://csrc.nist.gov/projects/cmvp. Example values will be available at https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

8. **Other Approved Security Functions.** Implementations that comply with this standard **shall** employ cryptographic algorithms that have been **approved** for protecting Federal Government-sensitive information. **Approved** cryptographic algorithms and techniques include those that are either:

    (a) Specified in a Federal Information Processing Standards (FIPS) publication,

    (b) Adopted in a FIPS or NIST recommendation, or

    (c) Specified in the list of **approved** security functions in SP 800-140C.

9. **Export Control.** Certain cryptographic devices and technical data regarding them are subject to federal export controls. Exports of cryptographic modules that implement this standard and technical data regarding them must comply with all federal laws and regulations and be licensed by the Bureau of Industry and Security of the U.S. Department of Commerce. Information about export regulations is available at https://www.bis.doc.gov.

10. **Patents.** NIST has entered into two patent license agreements to facilitate the adoption of NIST's announced selection of the PQC key-encapsulation mechanism CRYSTALS-KYBER. NIST and the licensing parties share a desire, in the public interest, the licensed patents be freely available to be practiced by any implementer of the ML-KEM algorithm as published by NIST. ML-KEM is the name given to the algorithm in this standard derived from CRYSTALS-KYBER. For a summary and extracts from the license, please see https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/selected-algos-2022/nist-pqc-license-summary-and-excerpts.pdf. Implementation of the algorithm specified in the standard may be covered by U.S. and foreign patents of which NIST is not aware.

11. **Implementation Schedule.** This standard becomes effective immediately upon final publication.

12. **Specifications.** Federal Information Processing Standards (FIPS) 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard (affixed).

13. **Qualifications.** In applications, the security guarantees of a KEM only hold under certain conditions (see SP 800-227 [1]). One such condition is the secrecy of several values, including the randomness used by the two parties, the decapsulation key, and the shared secret key itself. Users **shall**, therefore, guard against the disclosure of these values.

    While it is the intent of this standard to specify general requirements for implementing ML-KEM algorithms, conformance to this standard does not ensure that a particular implementation is secure. It is the responsibility of the implementer to ensure that any module that implements a key establishment capability is designed and built in a secure manner.

    Similarly, the use of a product containing an implementation that conforms to this standard does not guarantee the security of the overall system in which the product is used. The responsible authority in each agency or department **shall** ensure that an overall implementation provides an acceptable level of security.

    NIST will continue to follow developments in the analysis of the ML-KEM algorithm. As with its other cryptographic algorithm standards, NIST will formally reevaluate this standard every five years.

    Both this standard and possible threats that reduce the security provided through the use of this standard will undergo review by NIST as appropriate, taking into account newly available analysis and technology. In addition, the awareness of any breakthrough in technology or any mathematical weakness of the algorithm will cause NIST to reevaluate this standard and provide necessary revisions.

14. **Waiver Procedure.** The Federal Information Security Management Act (FISMA) does not allow for waivers to Federal Information Processing Standards (FIPS) that are made mandatory by the Secretary of Commerce.

15. **Where to Obtain Copies of the Standard.** This publication is available by accessing https://csrc.nist.gov/publications. Other computer security publications are available at the same website.

16. **How to Cite This Publication.** NIST has assigned **NIST FIPS 203** as the publication identifier for this FIPS, per the NIST Technical Series Publication Identifier Syntax. NIST recommends that it be cited as follows:

    National Institute of Standards and Technology (2024) Module-Lattice-Based Key-Encapsulation Mechanism Standard. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 203. https://doi.org/10.6028/NIST.FIPS.203

17. **Inquiries and Comments.** Inquiries and comments about this FIPS may be submitted to fips-203-comments@nist.gov.

Federal Information Processing Standards Publication 203

Specification for the
Module-Lattice-Based Key-Encapsulation Mechanism Standard

Table of Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

1.1 Purpose and Scope

This standard specifies the *Module-Lattice-Based Key-Encapsulation Mechanism* (ML-KEM). A key-encapsulation mechanism (KEM) is a set of algorithms that can be used to establish a shared secret key between two parties communicating over a public channel. A KEM is a particular type of key establishment scheme. Other NIST-**approved** key establishment schemes are specified in NIST Special Publication (SP) 800-56A, *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm-Based Cryptography* [2], and SP 800-56B, *Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography* [3].

The key establishment schemes specified in SP 800-56A and SP 800-56B are vulnerable to attacks that use sufficiently-capable quantum computers. ML-KEM is an **approved** alternative that is presently believed to be secure, even against adversaries in possession of a large-scale fault-tolerant quantum computer. ML-KEM is derived from the round-three version of the CRYSTALS-KYBER KEM [4], a submission in the NIST Post-Quantum Cryptography Standardization project. For the differences between ML-KEM and CRYSTALS-KYBER, see Appendix C.

This standard specifies the algorithms and parameter sets of the ML-KEM scheme. It aims to provide sufficient information to implement ML-KEM in a manner that can pass validation (see https://csrc.nist.gov/projects/cryptographic-module-validation-program). For general definitions and properties of KEMs, including requirements for the secure use of KEMs in applications, see SP 800-227 [1].

This standard specifies three parameter sets for ML-KEM that offer different trade-offs in security strength versus performance. All three parameter sets of ML-KEM are **approved** to protect sensitive, non-classified communication systems of the U.S. Federal Government.

1.2 Context

Over the past several years, there has been steady progress toward building quantum computers. If large-scale quantum computers are realized, the security of many commonly used public-key cryptosystems will be at risk. This would include key-establishment schemes and digital signature schemes whose security depends on the difficulty of solving the integer factorization and discrete logarithm problems (both over finite fields and elliptic curves). As a result, in 2016, NIST initiated a public Post-Quantum Cryptography (PQC) Standardization process to select quantum-resistant public-key cryptographic algorithms. A total of 82 candidate algorithms were submitted to NIST for consideration.

After three rounds of evaluation and analysis, NIST selected the first four algorithms for standardization. These algorithms are intended to protect sensitive U.S. Government information well into the foreseeable future, including after the advent of cryptographically-relevant quantum computers. This standard specifies a variant of the selected algorithm CRYSTALS-KYBER, a lattice-based key-encapsulation mechanism (KEM) [4]. Throughout this standard, the KEM specified here will be referred to as ML-KEM, as it is based on the Module Learning With Errors assumption.

2.   Terms, Acronyms, and Notation

2.1   Terms and Definitions

approved	FIPS-approved and/or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST recommendation, 2) adopted in a FIPS or NIST recommendation, or 3) specified in a list of NIST-**approved** security functions.
(KEM) ciphertext	A bit string that is produced by encapsulation and used as an input to decapsulation.
cryptographic module	The set of hardware, software, and/or firmware that implements approved cryptographic functions (including key generation) that are contained within the cryptographic boundary of the module.
decapsulation	The process of applying the Decaps algorithm of a KEM. This algorithm accepts a KEM ciphertext and the decapsulation key as input and produces a shared secret key as output.
decapsulation key	A cryptographic key produced by a KEM during key generation and used during the decapsulation process. The decapsulation key must be kept private and must be destroyed after it is no longer needed. (See Section 3.3.)
decryption key	A cryptographic key that is used with a PKE in order to decrypt ciphertexts into plaintexts. The decryption key must be kept private and must be destroyed after it is no longer needed.
destroy	An action applied to a key or other piece of secret data. After a piece of secret data is destroyed, no information about its value can be recovered.
encapsulation	The process of applying the Encaps algorithm of a KEM. This algorithm accepts the encapsulation key as input, requires private randomness, and produces a shared secret key and an associated ciphertext as output.
encapsulation key	A cryptographic key produced by a KEM during key generation and used during the encapsulation process. The encapsulation key can be made public. (See Section 3.3.)
encryption key	A cryptographic key that is used with a PKE in order to encrypt plaintexts into ciphertexts. The encryption key can be made public.
equivalent process	Two processes are equivalent if the same output is produced when the same values are input to each process (either as input parameters, as values made available during the process, or both).
fresh random value	An output that was produced by a random bit generator and has not been previously used.

hash function	A function on bit strings in which the length of the output is fixed. **Approved** hash functions (such as those specified in FIPS 180 [5] and FIPS 202 [6]) are designed to satisfy the following properties:

1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output.

2. (Collision-resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.

input checking	Examination of a potential input to an algorithm for the purpose of determining whether it conforms to certain requirements.
key	A bit string that is used in conjunction with a cryptographic algorithm, such as the encapsulation and decapsulation keys (of a KEM), the shared secret key (produced by a KEM), and the encryption and decryption keys (of a PKE). (See Section 3.3.)
key-encapsulation mechanism (KEM)	A set of three cryptographic algorithms (KeyGen, Encaps, and Decaps) that can be used by two parties to establish a shared secret key over a public channel.
key establishment	A procedure that results in secret keying material that is shared among different parties.
key pair	A set of two keys with the property that one key can be made public while the other key must be kept private. In this standard, this could refer to either the (encapsulation key, decapsulation key) key pair of a KEM or the (encryption key, decryption key) key pair of a PKE.
little-endian	The property of a byte string having its bytes positioned in order of increasing significance. In particular, the leftmost (first) byte is the least significant, and the rightmost (last) byte is the most significant. The term "little-endian" may also be applied in the same manner to bit strings (e.g., the 8-bit string 11010001 corresponds to the byte $2^0 + 2^1 + 2^3 + 2^7 = 139$).
party	An individual person, organization, device, or process. In this specification, there are two parties (e.g., Party A and Party B, or Alice and Bob) who jointly perform the key establishment process using a KEM.
pseudorandom	A process (or data produced by a process) is said to be pseudorandom when the outcome is deterministic yet also appears random as long as the internal action of the process is hidden from observation. For cryptographic purposes, "effectively random" means "computationally indistinguishable from random within the limits of the intended security strength."
public channel	A communication channel between two parties. Such a channel can be observed and possibly also corrupted by third parties.

public-key encryption scheme (PKE)	A set of three cryptographic algorithms (KeyGen, Encrypt, and Decrypt) that can be used by two parties to send secret data over a public channel. Also known as an asymmetric encryption scheme.
shared secret	A secret value that has been computed during a key-establishment scheme, is known by both participants, and is used as input to a key-derivation method to produce keying material.
shared secret key	A shared secret that can be used directly as a cryptographic key in symmetric-key cryptography. It does not require additional key derivation. The shared secret key must be kept private and must be destroyed when no longer needed.
security category	A number associated with the security strength of a post-quantum cryptographic algorithm, as specified by NIST (see [7]).
security strength	A number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system.
shall	Used to indicate a requirement of this standard.
should	Used to indicate a strong recommendation but not a requirement of this standard. Ignoring the recommendation could lead to undesirable results.

2.2   Acronyms

AES	Advanced Encryption Standard
CBD	Centered Binomial Distribution
FIPS	Federal Information Processing Standard
KEM	Key-Encapsulation Mechanism
LWE	Learning with Errors
MLWE	Module Learning with Errors
NIST	National Institute of Standards and Technology
NISTIR	NIST Interagency or Internal Report
NTT	Number-Theoretic Transform
PKE	Public-Key Encryption
PQC	Post-Quantum Cryptography
PRF	Pseudorandom Function
RBG	Random Bit Generator
SHA	Secure Hash Algorithm

SHAKE   Secure Hash Algorithm KECCAK

SP   Special Publication

XOF   Extendable-Output Function

2.3 Mathematical Symbols

n	Denotes the integer 256 throughout this document.
q	Denotes the prime integer $3329 = 2^8 \cdot 13 + 1$ throughout this document.
ζ	Denotes the integer 17, which is a primitive n-th root of unity modulo q.
$\mathbb{B}$	The set $\{0, 1, \ldots, 255\}$ of unsigned 8-bit integers (bytes).
$\mathbb{Q}$	The set of rational numbers.
$\mathbb{Z}$	The set of integers.
$\mathbb{Z}_m$	The ring of integers modulo m (i.e., the set $\{0, 1, \ldots, m-1\}$ equipped with the operations of addition and multiplication modulo m.)
$\mathbb{Z}_m^n$	The set of n-tuples over $\mathbb{Z}_m$ equipped with $\mathbb{Z}_m$-module structure. As a data type, this is the set of length-n arrays whose entries are in $\mathbb{Z}_m$.
R_q	The ring $\mathbb{Z}_q[X]/(X^n + 1)$ consisting of polynomials of the form $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$, where $f_j \in \mathbb{Z}_q$ for all j. The ring operations are addition and multiplication modulo $X^n + 1$.
T_q	The image of R_q under the number-theoretic transform. Its elements are called "NTT representations" of polynomials in R_q. (See Section 4.3.)
$\mathcal{D}_\eta(R_q)$	A certain distribution of polynomials in R_q with small coefficients, from which noise is sampled. The distribution is parameterized by $\eta \in \{2, 3\}$. (See Section 4.2.2.)
S^*	If S is a set, this denotes the set of finite-length tuples (or arrays) of elements from the set S, including the empty tuple (or empty array).
S^k	If S is a set, this denotes the set of k-tuples (or length-k arrays) of elements from the set S.
f_j	The coefficient of X^j of a polynomial $f = f_0 + f_1 X + \cdots + f_{255} X^{255} \in R_q$.
$\hat{f}$	The element of T_q that is equal to the NTT representation of a polynomial $f \in R_q$. (See Sections 2.4.4 and 4.3.)
$\mathbf{v}^T, \mathbf{A}^T$	The transpose of a row or column vector $\mathbf{v}$. In general, the transpose of a matrix $\mathbf{A}$.

$\circ$	Denotes linear-algebraic composition with coefficients in R_q or T_q (e.g., $\mathbf{A} \circ \mathbf{v}$ denotes the vector resulting from applying matrix $\mathbf{A}$ to vector $\mathbf{v}$). (See Section 2.4.7.)
$\times_{T_q}$	Denotes the operation on coefficient arrays that implements product in the ring T_q. (See Sections 2.4.5 and 4.3.1.)
$A\|B$	The concatenation of two arrays or bit strings A and B.
$B[i]$	The entry at index i in the array B. All arrays have indices that begin at zero.
$B[k:m]$	The subarray $(B[k], B[k+1], ..., B[m-1])$ of the array B.
$|B|$	If B is a number, this denotes the absolute value of B. If B is an array, this denotes its length.
$\lceil x \rceil$	The ceiling of x (i.e., the smallest integer greater than or equal to x).
$\lfloor x \rfloor$	The floor of x (i.e., the largest integer less than or equal to x).
$\lceil x \rfloor$	The rounding of x to the nearest integer. If $x = y + 1/2$ for some $y \in \mathbb{Z}$, then $\lceil x \rfloor = y + 1$.
$:=$	Denotes that the left-hand side is defined to be the expression on the right-hand side.
$r \bmod m$	The unique integer r' in $\{0, 1, ..., m-1\}$ such that m divides $r - r'$.
$\mathsf{BitRev}_7(r)$	Bit reversal of a seven-bit integer r. Specifically, if $r = r_0 + 2r_1 + 4r_2 + \cdots + 64r_6$ with $r_i \in \{0, 1\}$, then $\mathsf{BitRev}_7(r) = r_6 + 2r_5 + 4r_4 + \cdots + 64r_0$.
$s \leftarrow x$	In pseudocode, this notation means that the variable s is assigned the value of the expression x.
$s \xleftarrow{\$} \mathbb{B}^\ell$	In pseudocode, this notation means that the variable s is assigned the value of an array of ℓ random bytes. The bytes must be freshly generated using randomness from an **approved** RBG. (See Section 3.3.)
$\perp$	A symbol indicating failure or the lack of output from an algorithm.

2.4 Interpreting the Pseudocode

This section outlines the conventions of the pseudocode used to describe the algorithms in this standard. All algorithms are understood to have access to two global integer constants: $n = 256$ and $q = 3329$. There are also five global integer variables: k, η_1, η_2, d_u, and d_v. All other variables are local. The five global variables are set to particular values when a parameter set is selected (see Section 8).

When algorithms in this specification invoke other algorithms as subroutines, all arguments (i.e., inputs) are passed by value. In other words, a copy of the inputs is created, and the subroutine is invoked with the copy. There is no "passing by reference."

Pseudocode assignments are performed using the symbol "$\leftarrow$." For example, the statement $z \leftarrow y$ means that the variable z is assigned the value of variable y. Pseudocode comparisons

are performed via the symbol "$==$." For example, the expression $x == w$ is a boolean value that is TRUE if and only if the variables x and w have the same value.

In regular text (i.e., outside of the pseudocode), a different convention is applied. There, the "$=$" symbol is used both for assigning values and for comparisons, in keeping with standard mathematical notation. When emphasis is needed, assignments will be made with "$:=$" instead.

Variables will always have a valid value that is appropriate to their data type, with two exceptions:

1. The outputs of a random bit generator (RBG) have the byte array data type but are also allowed to have the special value NULL. This value indicates that randomness generation failed. This can only occur in ML-KEM.KeyGen and ML-KEM.Encaps.

2. The outputs of ML-KEM.KeyGen and ML-KEM.Encaps have the byte array data type but are also allowed to have the special value ⊥. When ML-KEM.KeyGen or ML-KEM.Encaps return the value ⊥, this indicates that the algorithm failed due to a failure of randomness generation.

2.4.1   Data Types

For variables that represent the input or output of an algorithm, the data type (e.g., bit, byte, array of bits) will be explicitly described at the start of the algorithm. For most local variables in the pseudocode, the data type is easily deduced from context. For all other variables, the data type will be declared in a comment. In a single algorithm, the data type of a variable is determined the first time that the variable is used and will not be changed. Variable names can and will be reused across different algorithms, including with different data types.

In addition to standard atomic data types (e.g., bits, bytes) and data structures (e.g., arrays), integers modulo m (i.e., elements of $\mathbb{Z}_m$) will also be used as an abstract data type. It is implicit that reduction modulo m takes place whenever an assignment is made to a variable in $\mathbb{Z}_m$. For example, for $z \in \mathbb{Z}_m$ and integers x and y, the statement

$$z \leftarrow x + y \qquad\qquad\qquad (2.1)$$

means that z is assigned the value $x + y \bmod m$. The pseudocode is agnostic regarding how an integer modulo m is represented in actual implementations or how modular reduction is computed.

2.4.2   Loop Syntax

The pseudocode will make use of both "while" and "for" loops. The "while" syntax is self-explanatory. In the case of "for" loops, the syntax will be in the style of the programming language C. Two simple examples are given in Algorithm 1. The standard mathematical expression (e.g., $\sum_{i \leftarrow 1}^{n}(i + 3)$) will be used for simple summations instead of a "for" loop.

2.4.3   Arithmetic With Arrays of Integers

This standard makes extensive use of arrays of integers modulo m (i.e., elements of $\mathbb{Z}_m^\ell$). In a typical case, the relevant values are $m = q = 3329$ and $\ell = n = 256$. Arithmetic with arrays in

---

Algorithm 1 ForExample()

Performs two simple "for" loops.

1: **for** $(i \leftarrow 0; i < 10; i\mathbin{++})$
2:     $A[i] \leftarrow i$ $\triangleright\ A$ is an integer array of length 10
3: **end for** $\triangleright\ A$ now has the value $(0,1,2,3,4,5,6,7,8,9)$
4: $j \leftarrow 0$
5: **for** $(k \leftarrow 256; k > 1; k \leftarrow k/2)$
6:     $B[j] \leftarrow k$ $\triangleright\ B$ is an integer array of length 8
7:     $j \leftarrow j+1$
8: **end for** $\triangleright\ B$ now has the value $(256,128,64,32,16,8,4,2)$

$\mathbb{Z}_m^\ell$ will be done as follows. Let $a \in \mathbb{Z}_m$ and $X, Y \in \mathbb{Z}_m^\ell$. The statements

$$Z \leftarrow a \cdot X \tag{2.2}$$

$$W \leftarrow X + Y \tag{2.3}$$

will result in two arrays $Z, W \in \mathbb{Z}_m^\ell$, with the property that $Z[i] = a \cdot X[i]$ and $W[i] = X[i] + Y[i]$ for all i. Multiplication of arrays in $\mathbb{Z}_m^\ell$ will only be meaningful when $m = q$ and $\ell = n = 256$, in which case it corresponds to multiplication in a particular ring. This operation will be described in (2.8).

2.4.4   Representations of Algebraic Objects

An essential operation in ML-KEM is the number-theoretic transform (NTT), which maps a polynomial f in a certain ring R_q to its "NTT representation" $\hat{f}$ in an isomorphic ring T_q. The rings R_q and T_q and the NTT are discussed in detail in Section 4.3. This standard will represent elements of R_q and T_q in pseudocode using arrays of integers modulo q as follows.

An element f of R_q is a polynomial of the form

$$f = f_0 + f_1 X + \cdots + f_{255} X^{255} \in R_q \tag{2.4}$$

and will be represented in pseudocode by the array

$$(f_0, f_1, \ldots, f_{255}) \in \mathbb{Z}_q^{256}, \tag{2.5}$$

whose entries contain the coefficients of f. Overloading notation, the array in (2.5) will also be denoted by f. The i-th entry of the array f will thus contain the i-th coefficient of the polynomial f (i.e., $f[i] = f_i$).

An element (sometimes called "NTT representation") $\hat{g}$ of T_q is a tuple of 128 polynomials, each of degree at most one. Specifically,

$$\hat{g} = \left(\hat{g}_{0,0} + \hat{g}_{0,1} X, \hat{g}_{1,0} + \hat{g}_{1,1} X, \ldots, \hat{g}_{127,0} + \hat{g}_{127,1} X\right) \in T_q. \tag{2.6}$$

Such an algebraic object will be represented in pseudocode by the array

$$(\hat{g}_{0,0}, \hat{g}_{0,1}, \hat{g}_{1,0}, \hat{g}_{1,1}, \ldots, \hat{g}_{127,0}, \hat{g}_{127,1}) \in \mathbb{Z}_q^{256}. \qquad (2.7)$$

Overloading notation, the array in (2.7) will also be denoted by $\hat{g}$. In this case, the mapping between array entries and coefficients is $\hat{g}[2i] = \hat{g}_{i,0}$ and $\hat{g}[2i+1] = \hat{g}_{i,1}$ for $i \in \{0, 1, \ldots, 127\}$.

Converting between a polynomial $f \in R_q$ and its NTT representation $\hat{f} \in T_q$ will be done via the algorithms NTT (Algorithm 9) and NTT^{-1} (Algorithm 10). These algorithms act on arrays of coefficients, as described above, and satisfy $\hat{f} = \text{NTT}(f)$ and $f = \text{NTT}^{-1}(\hat{f})$.

2.4.5 Arithmetic With Polynomials and NTT Representations

The algebraic operations of addition and scalar multiplication in R_q and T_q are done coordinate-wise. For example, if $a \in \mathbb{Z}_q$ and $f \in R_q$, the i-th coefficient of the polynomial $a \cdot f \in R_q$ is equal to $a \cdot f_i \bmod q$. In pseudocode, elements of both R_q and T_q are represented by coefficient arrays (i.e., elements of $\mathbb{Z}_q^{256}$). The algebraic operations of addition and scalar multiplication are thus performed by addition and scalar multiplication of the corresponding coefficient arrays, as in (2.3) and (2.2). For example, the addition of two NTT representations in pseudocode is performed by a statement of the form $\hat{h} \leftarrow \hat{f} + \hat{g}$, where $\hat{h}, \hat{f}, \hat{g} \in \mathbb{Z}_q^{256}$ are coefficient arrays.

The algebraic operations of multiplication in R_q and T_q are treated as follows. For efficiency reasons, multiplication in R_q will not be used. The algebraic meaning of multiplication in T_q is discussed in Section 4.3.1. In pseudocode, it will be performed by the algorithm MultiplyNTTs (Algorithm 11). Specifically, if $\hat{f}, \hat{g} \in \mathbb{Z}_q^{256}$ are a pair of arrays (each representing the NTT of some polynomial), then

$$\hat{h} \leftarrow \hat{f} \times_{T_q} \hat{g} \qquad \text{means} \qquad \hat{h} \leftarrow \text{MultiplyNTTs}(\hat{f}, \hat{g}). \qquad (2.8)$$

The result is an array $\hat{h} \in \mathbb{Z}_q^{256}$.

2.4.6 Matrices and Vectors

In addition to arrays of integers modulo q, the pseudocode will also make use of arrays whose entries are themselves elements of $\mathbb{Z}_q^{256}$. For example, an element $\mathbf{v} \in (\mathbb{Z}_q^{256})^3$ will be a length-three array whose entries $\mathbf{v}[0]$, $\mathbf{v}[1]$, and $\mathbf{v}[2]$ are themselves elements of $\mathbb{Z}_q^{256}$ (i.e., arrays). One can think of each of these entries as representing a polynomial in R_q so that $\mathbf{v}$ itself represents an element of the module R_q^3.

When arrays are used to represent matrices and vectors whose entries are elements of R_q, they will be denoted with bold letters (e.g., $\mathbf{v}$ for vectors and $\mathbf{A}$ for matrices). When arrays are used to represent matrices and vectors whose entries are elements of T_q, they will be denoted with a "hat" (e.g., $\hat{\mathbf{v}}$ and $\hat{\mathbf{A}}$). Unless an explicit transpose operation is performed, it is understood that vectors are column vectors. One can then view vectors as the special case of matrices with only one column.

Converting between matrices over R_q and matrices over T_q will be done coordinate-wise. For

example, if $\mathbf{v} \in (\mathbb{Z}_q^{256})^k$, then the statement

$$\hat{\mathbf{v}} \leftarrow \mathsf{NTT}(\mathbf{v}) \tag{2.9}$$

will result in $\hat{\mathbf{v}} \in (\mathbb{Z}_q^{256})^k$ such that $\hat{\mathbf{v}}[i] = \mathsf{NTT}(\mathbf{v}[i])$ for all i. This involves running NTT a total of k times.

2.4.7 Arithmetic With Matrices and Vectors

The following describes how to perform arithmetic with matrices over R_q and T_q with vectors as a special case.

Addition and scalar multiplication are performed coordinate-wise, so the addition of matrices over R_q and T_q is straightforward. In the case of T_q, scalar multiplication is done via (2.8). For example, if $\hat{f} \in \mathbb{Z}_q^{256}$ and $\hat{\mathbf{u}}, \hat{\mathbf{v}} \in (\mathbb{Z}_q^{256})^k$, then

$$\hat{\mathbf{w}} \leftarrow \hat{f} \cdot \hat{\mathbf{u}} \tag{2.10}$$

$$\hat{\mathbf{z}} \leftarrow \hat{\mathbf{u}} + \hat{\mathbf{v}} \tag{2.11}$$

will result in $\hat{\mathbf{w}}, \hat{\mathbf{z}} \in (\mathbb{Z}_q^{256})^k$ satisfying $\hat{\mathbf{w}}[i] = \hat{f} \times_{T_q} \hat{\mathbf{u}}[i]$ and $\hat{\mathbf{z}}[i] = \hat{\mathbf{u}}[i] + \hat{\mathbf{v}}[i]$ for all i. Here, the multiplication and addition of individual entries are performed using the appropriate arithmetic for coefficient arrays of elements of T_q (i.e., as in (2.3)).

It will also be necessary to multiply matrices with entries in T_q, which is done by using standard matrix multiplication with the base-case multiplication (i.e., multiplication of individual entries) being multiplication in T_q. If $\hat{\mathbf{A}}$ and $\hat{\mathbf{B}}$ are two matrices with entries in T_q, their matrix product will be denoted $\hat{\mathbf{A}} \circ \hat{\mathbf{B}}$. Some example pseudocode statements involving matrix multiplication are given in (2.12), (2.13), and (2.14). In these examples, $\hat{\mathbf{A}}$ is a $k \times k$ matrix, while $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ are vectors of length k. All three of these objects are represented in pseudocode by arrays: a $k \times k$ array for $\hat{\mathbf{A}}$ and length-k arrays for $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$. The entries of $\hat{\mathbf{A}}$, $\hat{\mathbf{u}}$, and $\hat{\mathbf{v}}$ are elements of $\mathbb{Z}_q^{256}$. In (2.12) and (2.13), the pseudocode statement on the left produces a new length-k vector whose entries are specified on the right. In (2.14), the pseudocode statement on the left computes a dot product. The result is in the base ring (i.e., T_q) and is represented by an element $\hat{z}$ of $\mathbb{Z}_q^{256}$.

$$\hat{\mathbf{w}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{u}} \qquad\qquad \hat{\mathbf{w}}[i] = \sum_{j=0}^{k-1} \hat{\mathbf{A}}[i,j] \times_{T_q} \hat{\mathbf{u}}[j] \tag{2.12}$$

$$\hat{\mathbf{y}} \leftarrow \hat{\mathbf{A}}^\top \circ \hat{\mathbf{u}} \qquad\qquad \hat{\mathbf{y}}[i] = \sum_{j=0}^{k-1} \hat{\mathbf{A}}[j,i] \times_{T_q} \hat{\mathbf{u}}[j] \tag{2.13}$$

$$\hat{z} \leftarrow \hat{\mathbf{u}}^\top \circ \hat{\mathbf{v}} \qquad\qquad \hat{z} = \sum_{j=0}^{k-1} \hat{\mathbf{u}}[j] \times_{T_q} \hat{\mathbf{v}}[j] \tag{2.14}$$

The multiplication $\times_{T_q}$ of individual entries above is performed using MultiplyNTTs, as described in (2.8).

2.4.8 Applying Algorithms to Arrays, Examples

In the previous examples, arithmetic over $\mathbb{Z}_m$ was extended to arithmetic with arrays over $\mathbb{Z}_m$ and then further extended to arithmetic with matrices whose entries are themselves arrays over $\mathbb{Z}_m$. Similarly, algorithms defined with a given data type as input will be applied to arrays and matrices over that data type. When the pseudocode invokes such an algorithm on an array or matrix input, it is implied that the algorithm is invoked repeatedly and applied to each entry of the input.

For example, consider the function $\mathsf{Compress}_d : \mathbb{Z}_q \to \mathbb{Z}_{2^d}$ defined in Section 4. It can be invoked on an array input $F \in \mathbb{Z}_q^{256}$ with the statement

$$K \leftarrow \mathsf{Compress}_d(F). \tag{2.15}$$

The result will be an array $K \in \mathbb{Z}_{2^d}^{256}$ such that $K[i] = \mathsf{Compress}_d(F[i])$ for every $i \in \{0, 1, \dots, 255\}$. The computation (2.15) involves running the $\mathsf{Compress}$ algorithm 256 times.

For a second example, consider the algorithm NTT defined in Section 4.3. It takes an array $f \in \mathbb{Z}_q^{256}$ (representing an element of R_q) as input and outputs another array $\hat{f} \in \mathbb{Z}_q^{256}$ (representing an element of T_q). If the NTT algorithm is invoked on a vector $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$ (representing an element of R_q^k) with the pseudocode statement

$$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s}), \tag{2.16}$$

the result is a vector $\hat{\mathbf{s}} \in (\mathbb{Z}_q^{256})^k$ such that $\hat{\mathbf{s}}[i] = \mathsf{NTT}(\mathbf{s}[i])$ for all $i \in \{0, 1, \dots, k-1\}$. The vector $\hat{\mathbf{s}}$ represents an element of T_q^k. The computation (2.16) involves running the NTT algorithm k times.

For a third example, consider line 2 of K-PKE.Encrypt in Section 5.2:

$$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[0 : 384k]). \tag{2.17}$$

$\mathsf{ByteDecode}_{12}$ is defined to receive a byte array of length $32 \cdot 12 = 384$ as input and produce an integer array in $\mathbb{Z}_q^{256}$ as output. The computation (2.17) is run on the first $384k$ bytes of byte array $\mathsf{ek}_{\mathsf{PKE}}$ and results in $\hat{\mathbf{t}} \in (\mathbb{Z}_q^{256})^k$. $\mathsf{ByteDecode}_{12}$ will thus be applied k times, once for each subarray $\mathsf{ek}_{\mathsf{PKE}}[384 \cdot j, 384 \cdot (j+1)]$, and will result in an integer array $\hat{\mathbf{t}}[j] \in \mathbb{Z}_q^{256}$ such that $\hat{\mathbf{t}}[j] = \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[384 \cdot j, 384 \cdot (j+1)])$ for each j from 0 to $k-1$.

3.   Overview of the ML-KEM Scheme

This section gives a high-level overview of the ML-KEM scheme.

3.1   Key-Encapsulation Mechanisms

The following is a high-level overview of key-encapsulation mechanisms (KEMs). For details, see SP 800-227 [1].

A KEM is a cryptographic scheme that, under certain conditions, can be used to establish a shared secret key between two communicating parties. This shared secret key can then be used for symmetric-key cryptography.

A KEM consists of three algorithms and a collection of parameter sets. The three algorithms are:

1. A probabilistic key generation algorithm denoted by KeyGen

2. A probabilistic "encapsulation" algorithm denoted by Encaps

3. A deterministic "decapsulation" algorithm denoted by Decaps

The collection of parameter sets is used to select a trade-off between security and efficiency. Each parameter set in the collection is a list of specific (typically numerical) values, one for each parameter required by the three algorithms.
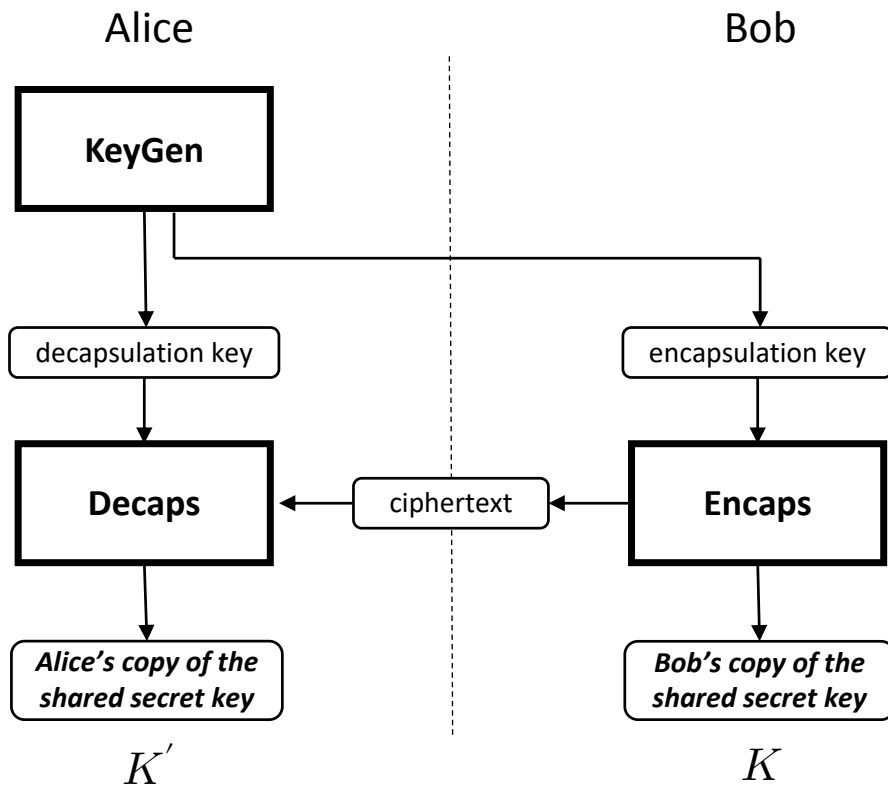

Figure 1. A simple view of key establishment using a KEM

In the typical application, a KEM is used to establish a shared secret key between two parties (here referred to as Alice and Bob) as described in Figure 1. Alice begins by running KeyGen in order to generate a (public) encapsulation key and a (private) decapsulation key. Upon obtaining Alice's encapsulation key, Bob runs the Encaps algorithm, which produces Bob's copy K of the shared secret key along with an associated ciphertext. Bob sends the ciphertext to Alice, and Alice completes the process by running the Decaps algorithm using her decapsulation key and the ciphertext. This final step produces Alice's copy K' of the shared secret key.

After completing this process, Alice and Bob would like to conclude that their outputs satisfy $K' = K$ and that this value is a secure, random, shared secret key. However, these properties only hold if certain important conditions are satisfied, as discussed in SP 800-227 [1].

3.2   The ML-KEM Scheme

ML-KEM is a key-encapsulation mechanism based on CRYSTALS-KYBER [4], a scheme that was initially described in [8]. The following is a brief and informal description of the computational assumption underlying ML-KEM and how the ML-KEM scheme is constructed.

The computational assumption. The security of ML-KEM is based on the presumed hardness of the so-called Module Learning with Errors (MLWE) problem [9], which is a generalization of the Learning With Errors (LWE) problem introduced by Regev in 2005 [10]. The hardness of the MLWE problem is itself based on the presumed hardness of certain computational problems in module lattices [9]. This motivates the name of the scheme ML-KEM.

In the LWE problem, the input is a set of random "noisy" linear equations in some secret variables $x \in \mathbb{Z}_q^n$, and the task is to recover x. The noise in the equations is such that standard algorithms (e.g., Gaussian elimination) are intractable. The LWE problem naturally lends itself to cryptographic applications. For example, if x is interpreted as a secret key, then one can encrypt a one-bit plaintext value by sampling either an approximately correct linear equation (if the plaintext is zero) or a far-from-correct linear equation (if the plaintext is one). Plausibly, only a party in possession of x can distinguish these two cases. Encryption can then be delegated to another party by publishing a large collection of noisy linear equations, which can be combined appropriately by the encrypting party. The result is an asymmetric encryption scheme.

The MLWE problem is similar to the LWE problem. An important difference is that, in MLWE, $\mathbb{Z}_q^n$ is replaced by a certain module R_q^k, which is constructed by taking the k-fold Cartesian product of a certain polynomial ring R_q. In particular, the secret in the MLWE problem is an element $\mathbf{x}$ of the module R_q^k. The ring R_q is discussed in detail in Section 4.3.

The ML-KEM construction. At a high level, the construction of the scheme ML-KEM proceeds in two steps. First, the ideas discussed previously are used to construct a public-key encryption (PKE) scheme from the MLWE problem. Second, this PKE scheme is converted into a key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [11, 12]. Due to certain properties of the FO transform, the resulting KEM provides security in a significantly more general attack model than the PKE scheme. As a result, ML-KEM is believed to satisfy so-called IND-CCA2 security [1, 4, 13, 14].

The specification of the ML-KEM algorithms in this standard will follow the same pattern. Specifically, this standard will first describe a public-key encryption scheme called K-PKE (in Section 5) and then use the algorithms of K-PKE as subroutines when describing the algorithms of ML-KEM (in Sections 6 and 7). The cryptographic transformation from K-PKE to ML-KEM is crucial for achieving IND-CCA2 security. The scheme K-PKE is not IND-CCA2-secure and **shall not** be used as a stand-alone scheme (see Section 3.3).

A notable feature of ML-KEM is the use of the *number-theoretic transform* (NTT). The NTT converts a polynomial $f \in R_q$ to an alternative representation as a vector $\hat{f}$ of linear polynomials. Working with NTT representations enables significantly faster multiplication of polynomials. Other operations (e.g., addition, rounding, and sampling) can be done in either representation.

ML-KEM satisfies the essential KEM property of correctness. This means that in the absence of corruption or interference, the process in Figure 1 will result in $K' = K$ with overwhelming probability. ML-KEM also comes with a proof of asymptotic theoretical security in a certain heuristic model [4]. Each of the parameter sets of ML-KEM comes with an associated security strength that was estimated based on current cryptanalysis (see Section 8 for details).

Parameter sets and algorithms. Recall that a KEM consists of algorithms KeyGen, Encaps, and Decaps, along with a collection of parameter sets. In the case of ML-KEM, the three aforementioned algorithms are:

1. ML-KEM.KeyGen (Algorithm 19)

2. ML-KEM.Encaps (Algorithm 20)

3. ML-KEM.Decaps (Algorithm 21)

These algorithms are described and discussed in detail in Section 7.

ML-KEM comes equipped with three parameter sets:

- ML-KEM-512 (security category 1)

- ML-KEM-768 (security category 3)

- ML-KEM-1024 (security category 5)

These parameter sets are described and discussed in detail in Section 8. The security categories 1-5 are defined in SP 800-57, Part 1 [7]. Each parameter set assigns a particular numerical value to five integer variables: k, η_1, η_2, d_u, and d_v. The values of these variables in each parameter set are given in Table 2 of Section 8. In addition to these five variable parameters, there are also two constants: $n = 256$ and $q = 3329$.

Decapsulation failures. Provided that all inputs are well-formed and randomness generation is successful, the key establishment procedure of ML-KEM will never explicitly fail, meaning that both ML-KEM.Encaps and ML-KEM.Decaps will each output a 256-bit value. Moreover, if no corruption or interference is present, the two 256-bit values produced by ML-KEM.Encaps and ML-KEM.Decaps will be equal with overwhelming probability (i.e., K' will equal K in the process described in Figure 1). The event that $K' \neq K$ under these conditions is called a *decapsulation*

failure. Formally, the decapsulation failure probability is defined to be the probability (conditioned on no RGB failures) that the process

$$1. \ (\mathrm{ek}, \mathrm{dk}) \leftarrow \mathrm{ML\text{-}KEM.KeyGen}() \tag{3.1}$$

$$2. \ (c, K) \leftarrow \mathrm{ML\text{-}KEM.Encaps}(\mathrm{ek}) \tag{3.2}$$

$$3. \ K' \leftarrow \mathrm{ML\text{-}KEM.Decaps}(\mathrm{dk}, c) \tag{3.3}$$

results in $K' \neq K$. The probability is taken over uniformly random seeds d, z (sampled in ML-KEM.KeyGen) and m (sampled in ML-KEM.Encaps) and under the heuristic assumption that hash functions and XOFs behave like uniformly random functions. The decapsulation failure rates for ML-KEM are listed in Table 1. For details, see Theorem 1 in [8] and the scripts in [15].

Table 1. Decapsulation failure rates for ML-KEM

Parameter set	Decapsulation failure rate
ML-KEM-512	$2^{-138.8}$
ML-KEM-768	$2^{-164.8}$
ML-KEM-1024	$2^{-174.8}$

Terminology for keys. A KEM involves three different types of keys: encapsulation keys, decapsulation keys, and shared secret keys. ML-KEM is built on top of the component public-key encryption scheme K-PKE, which has two additional key types: encryption keys and decryption keys. In the literature, encapsulation keys and encryption keys are sometimes referred to as "public keys," while decapsulation keys and decryption keys are sometimes referred to as "private keys." In order to reduce confusion, this standard will not use the terms "public key" or "private key." Instead, keys will be referred to only using the more specific terms, i.e., one of "encapsulation key", "decapsulation key", "encryption key", "decryption key", and "shared secret key".

3.3　Requirements for ML-KEM Implementations

This section describes several requirements for cryptographic modules that implement ML-KEM. Implementation requirements specific to particular algorithms will be described in later sections. Additional requirements, including requirements for using ML-KEM in specific applications, are given in SP 800-227 [1]. While conforming implementations must adhere to all of these requirements, adherence does not guarantee that the result will be secure (see Point 13 in the announcement).

K-PKE is only a component. The public-key encryption scheme K-PKE described in Section 5 **shall not** be used as a stand-alone cryptographic scheme. Instead, the algorithms that comprise K-PKE may only be used as subroutines in the algorithms of ML-KEM. In particular, the algorithms K-PKE.KeyGen (Algorithm 13), K-PKE.Encrypt (Algorithm 14), and K-PKE.Decrypt (Algorithm 15) are **not approved** for use as a public-key encryption scheme.

Controlled access to internal functions. The key-encapsulation mechanism ML-KEM makes use of internal, "derandomized" functions ML-KEM.KeyGen_internal, ML-KEM.Encaps_internal, and ML-KEM.Decaps_internal, specified in Section 6. The interfaces for these functions **should not** be made available to applications other than for testing purposes. In particular, the sampling of random values required for key generation (as specified in ML-KEM.KeyGen) and encapsulation (as specified in ML-KEM.Encaps) **shall** be performed by the cryptographic module.

Equivalent implementations. For every algorithm that is specified in this standard, a conforming implementation may replace the given set of steps with any mathematically equivalent set of steps. In other words, the specified algorithm may be replaced with a different procedure that produces the correct output for every input (where "input" includes the specified input as well as all parameter values and all randomness).

Approved usage of the shared secret key. If randomness generation is successful, the values K and K' returned by ML-KEM.Encaps and ML-KEM.Decaps, respectively, are always 256-bit values. Under appropriate conditions (see Sections 3.1 and 3.2, and SP 800-227 [1]), these values match (i.e., $K' = K$) and can be used directly as a shared secret key for symmetric cryptography. If further key derivation is needed, the final symmetric keys **shall** be derived from this 256-bit shared secret key in an **approved** manner, as specified in SP 800-108 and SP 800-56C [16, 17].

As discussed in Section 3.2, ML-KEM is an IND-CCA2-secure KEM. However, a combined KEM that includes ML-KEM as a component might not meet IND-CCA2 security. Implementers should assess the security of any procedure in which the key derivation methods of SP 800-56C are applied to ML-KEM in combination with another key establishment procedure. More guidance regarding combined KEMs is given in SP 800-227 [1].

Randomness generation. Two algorithms in this standard require the generation of randomness as an internal step: ML-KEM.KeyGen and ML-KEM.Encaps. In pseudocode, this randomness generation is denoted by a statement of the form $m \xleftarrow{\$} \mathbb{B}^{32}$. A fresh string of random bytes must be generated for every such invocation. These random bytes **shall** be generated using an **approved** RBG, as prescribed in SP 800-90A, SP 800-90B, and SP 800-90C [18, 19, 20]. Moreover, this RBG **shall** have a security strength of at least 128 bits for ML-KEM-512, at least 192 bits for ML-KEM-768, and at least 256 bits for ML-KEM-1024.

Input checking. The algorithms ML-KEM.Encaps and ML-KEM.Decaps require input checking. Implementers **shall** ensure that ML-KEM.Encaps and ML-KEM.Decaps are only executed on inputs that have been checked, as described in Section 7.

Destruction of intermediate values. Data used in intermediate computation steps of KEM algorithms could be used by an adversary to compromise security. Therefore, implementers **shall** ensure that intermediate data is destroyed as soon as it is no longer needed. In particular, for ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps, only the designated output can be retained in memory after the algorithm terminates. All other data **shall** be destroyed prior to

16

the algorithm terminating.

There are two exceptions to this rule:

1. The seed (d, z) generated in steps 1 and 2 of ML-KEM.KeyGen can be stored for later expansion using ML-KEM.KeyGen_internal. As this seed can be used to compute the decapsulation key, it is sensitive data and **shall** be treated with the same safeguards as a decapsulation key (see SP 800-227 [1]).

2. The matrix $\hat{\mathbf{A}}$ generated in steps 3-7 of K-PKE.KeyGen (as a subroutine of ML-KEM.KeyGen) can be stored so that it need not be recomputed in later operations (e.g., ML-KEM.Decaps). The same matrix $\hat{\mathbf{A}}$ is also generated in steps 4-8 of K-PKE.Encrypt (as a subroutine of ML-KEM.Encaps or ML-KEM.Decaps); it can also then be stored. In either case, the matrix $\hat{\mathbf{A}}$ is data that is easily computed from the public encapsulation key and thus does not require any special protections.

No floating-point arithmetic. Implementations of ML-KEM **shall not** use floating-point arithmetic, as rounding errors in floating-point operations may lead to incorrect results in some cases. In all pseudocode in this standard in which division is performed (e.g., x/y) and y may not divide x, either $\lfloor x/y \rfloor$, $\lceil x/y \rceil$, or $\lceil x/y \rfloor$ is used. All of these can be computed without floating-point arithmetic, as ordinary integer division x/y computes $\lfloor x/y \rfloor$, and $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for non-negative integers x and positive integers y.

4.   Auxiliary Algorithms

4.1   Cryptographic Functions

The algorithms specified in this publication require the use of several cryptographic functions. Each function **shall** be instantiated by means of an **approved** hash function or an **approved** eXtendable-Output Function (XOF), as prescribed below. The relevant hash functions and XOFs are described in detail in FIPS 202 [6]. They will be used as follows.

SHA3-256 and SHA3-512 are hash functions with one variable-length input and one fixed-length output. In this standard, invocations of these functions on an input M will be denoted by SHA3-256(M) and SHA3-512(M), respectively. The inputs and outputs for both SHA3-256 and SHA3-512 are always byte arrays.

SHAKE128 and SHAKE256 are XOFs with one variable-length input and one variable-length output. This standard will adhere to the following conventions [6]:

- The inputs and outputs for both SHAKE128 and SHAKE256 are always **byte arrays**.

- When invoking SHAKE128 or SHAKE256, desired output length is always specified **in bits**.

For example, the expression

$$r := \mathsf{SHAKE128}(M, 8 \cdot 64) \tag{4.1}$$

implies that M is an array of bytes and that r is an array of 64 bytes.

The aforementioned functions play several different roles in the algorithms specified in this standard and will only be invoked using the wrapper functions defined below. Importantly, these wrappers will avoid any potential "byte array" versus "bit-length" confusion by only working with bytes and byte array lengths.

Pseudorandom function (PRF). The function PRF takes a parameter $\eta \in \{2, 3\}$, one 32-byte input, and one 1-byte input. It produces one $(64 \cdot \eta)$-byte output. It will be denoted by

$$\mathsf{PRF} : \{2, 3\} \times \mathbb{B}^{32} \times \mathbb{B} \to \mathbb{B}^{64\eta}, \tag{4.2}$$

and it **shall** be instantiated as

$$\mathsf{PRF}_\eta(s, b) := \mathsf{SHAKE256}(s \| b, 8 \cdot 64 \cdot \eta), \tag{4.3}$$

where $\eta \in \{2, 3\}$, $s \in \mathbb{B}^{32}$, and $b \in \mathbb{B}$. Note that η is only used to specify the desired output length and not to perform domain separation.

Hash functions. The specification will also make use of three hash functions H, J and G, which are defined as follows.

The functions H and J each take one variable-length input and produce one 32-byte output. They will be denoted by $\mathsf{H} : \mathbb{B}^* \to \mathbb{B}^{32}$ and $\mathsf{J} : \mathbb{B}^* \to \mathbb{B}^{32}$, respectively, and **shall** be instantiated as

$$\mathsf{H}(s) := \mathsf{SHA3\text{-}256}(s) \qquad \text{and} \qquad \mathsf{J}(s) := \mathsf{SHAKE256}(s, 8 \cdot 32) \tag{4.4}$$

where $s \in \mathbb{B}^*$.

The function G takes one variable-length input and produces two 32-byte outputs. It will be denoted by $\mathsf{G} : \mathbb{B}^* \to \mathbb{B}^{32} \times \mathbb{B}^{32}$. The two outputs of G will be denoted by $(a, b) \leftarrow \mathsf{G}(c)$, where $a, b \in \mathbb{B}^{32}$, $c \in \mathbb{B}^*$, and $\mathsf{G}(c) = a \| b$. The function G **shall** be instantiated as

$$\mathsf{G}(c) := \mathsf{SHA3\text{-}512}(c). \tag{4.5}$$

eXtendable-Output Function (XOF). This standard uses a XOF wrapper defined in terms of the incremental API for SHAKE128 in SP 800-185 [21]. This SHAKE128 API consists of three functions:

- $\mathsf{ctx} \leftarrow \mathsf{SHAKE128.Init}()$
  Initializes a XOF "context" ctx.

- $\mathsf{ctx} \leftarrow \mathsf{SHAKE128.Absorb}(\mathsf{ctx}, \mathsf{str})$
  Injects data to be used in the "absorbing" phase of SHAKE128 and updates the context accordingly.

- $(\mathsf{ctx}, B) \leftarrow \mathsf{SHAKE128.Squeeze}(\mathsf{ctx}, 8 \cdot z)$
  Extracts z output bytes produced during the "squeezing" phase of SHAKE128 and updates the context accordingly.

While the above functions are constructed using the Keccak-f permutation rather than the XOF SHAKE128 directly, they are defined so that a single SHAKE128 call of the form

$$\mathsf{output} \leftarrow \mathsf{SHAKE128}(\mathsf{str}_1 \| \dots \| \mathsf{str}_m, 8b_1 + \dots + 8b_\ell) \tag{4.6}$$

is equivalent to performing Algorithm 2. This equivalence holds whether or not $|\mathsf{str}_i|$ and b_j are multiples of the SHAKE128 block length.

Algorithm 2 $\mathsf{SHAKE128example}(\mathsf{str}_1, \dots, \mathsf{str}_m, b_1, \dots, b_\ell)$

Performs a sequence of absorbing operations followed by a sequence of squeezing operations.

Input: byte arrays $\mathsf{str}_1, \dots, \mathsf{str}_m$.
Input: positive integers $b_1, \dots, b_\ell$.
Output: a byte array of length $\sum_{j=1}^{\ell} b_j$.

1: $\mathsf{ctx} \leftarrow \mathsf{SHAKE128.Init}()$ ▷ initialize context
2: **for** $(i \leftarrow 1; i \leq m; i\text{++})$
3:     $\mathsf{ctx} \leftarrow \mathsf{SHAKE128.Absorb}(\mathsf{ctx}, \mathsf{str}_i)$ ▷ absorb byte array str_i
4: **end for**
5: **for** $(j \leftarrow 1; j \leq \ell; j\text{++})$
6:     $(\mathsf{ctx}, \mathsf{out}_j) \leftarrow \mathsf{SHAKE128.Squeeze}(\mathsf{ctx}, 8 \cdot b_j)$ ▷ squeeze b_j-many bytes
7: **end for**
8: $\mathsf{output} \leftarrow \mathsf{out}_1 \| \dots \| \mathsf{out}_\ell$ ▷ return the concatenation of all the results

---

In this standard, the incremental API for SHAKE128 will only be invoked through a wrapper XOF,

which is defined as follows.

1. $\mathsf{XOF}.\mathsf{Init}() = \mathsf{SHAKE128}.\mathsf{Init}()$.

2. $\mathsf{XOF}.\mathsf{Absorb}(\mathsf{ctx}, \mathsf{str}) = \mathsf{SHAKE128}.\mathsf{Absorb}(\mathsf{ctx}, \mathsf{str})$.

3. $\mathsf{XOF}.\mathsf{Squeeze}(\mathsf{ctx}, \ell) = \mathsf{SHAKE128}.\mathsf{Squeeze}(\mathsf{ctx}, 8 \cdot \ell)$.

Note that $\mathsf{XOF}.\mathsf{Squeeze}$ requires the input length to be specified in bytes. This is consistent with the convention that all wrapper functions treat inputs and outputs as byte arrays and measure the lengths of all such arrays in terms of bytes.

4.2   General Algorithms

This section specifies a number of algorithms that will be used as subroutines in ML-KEM.

4.2.1   Conversion and Compression Algorithms

This section specifies several algorithms for converting between bit arrays, byte arrays, and arrays of integers modulo m. It also specifies a certain operation for compressing integers modulo q, and the corresponding decompression operation.

Algorithm 3 BitsToBytes(b)

Converts a bit array (of a length that is a multiple of eight) into an array of bytes.

Input: bit array $b \in \{0,1\}^{8 \cdot \ell}$.
Output: byte array $B \in \mathbb{B}^{\ell}$.

  1: $B \leftarrow (0, \dots, 0)$
  2: **for** $(i \leftarrow 0; i < 8\ell; i\text{++})$
  3:     $B[\lfloor i/8 \rfloor] \leftarrow B[\lfloor i/8 \rfloor] + b[i] \cdot 2^{i \bmod 8}$
  4: **end for**
  5: **return** B

Algorithm 4 BytesToBits(B)

Performs the inverse of BitsToBytes, converting a byte array into a bit array.

Input: byte array $B \in \mathbb{B}^{\ell}$.
Output: bit array $b \in \{0,1\}^{8 \cdot \ell}$.

  1: $C \leftarrow B$ $\triangleright$ copy B into array $C \in \mathbb{B}^{\ell}$
  2: **for** $(i \leftarrow 0; i < \ell; i\text{++})$
  3:     **for** $(j \leftarrow 0; j < 8; j\text{++})$
  4:         $b[8i + j] \leftarrow C[i] \bmod 2$
  5:         $C[i] \leftarrow \lfloor C[i]/2 \rfloor$
  6:     **end for**
  7: **end for**
  8: **return** b

Converting between bits and bytes. The algorithms BitsToBytes (Algorithm 3) and BytesToBits (Algorithm 4) convert between bit arrays and byte arrays. The inputs to BitsToBytes and the outputs of BytesToBits are bit arrays, with each segment of eight bits representing a byte in little-endian order. As an example, the 8-bit string 11010001 corresponds to the byte $2^0 + 2^1 + 2^3 + 2^7 = 139$.

Compression and decompression. Recall that $q = 3329$, and that the bit length of q is 12. For $d < 12$, define

$$\text{Compress}_d : \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \tag{4.7}$$
$$x \longmapsto \lceil (2^d/q) \cdot x \rfloor \bmod 2^d \,.$$
$$\text{Decompress}_d : \mathbb{Z}_{2^d} \longrightarrow \mathbb{Z}_q \tag{4.8}$$
$$y \longmapsto \lceil (q/2^d) \cdot y \rfloor \,.$$

The input and output types of these functions are integers modulo m (see Section 2.4.1). Division and rounding in the computation of these functions are performed in the set of rational numbers. Floating-point computations **shall not** be used.

The Compress and Decompress algorithms satisfy two important properties. First, decompression followed by compression preserves the input. That is, $\text{Compress}_d(\text{Decompress}_d(y)) = y$ for all $y \in \mathbb{Z}_{2^d}$ and all $d < 12$. Second, if d is large (i.e., close to 12), compression followed by decompression does not significantly alter the value.

Encoding and decoding. The algorithms ByteEncode (Algorithm 5) and ByteDecode (Algorithm 6) will be used for conversion between integers modulo m and bytes. The algorithm ByteEncode converts an array of $n = 256$ integers modulo m into a corresponding array of bytes. ByteDecode performs the inverse operation, converting an array of bytes into an array of integers modulo m. Specifying the modulus m is done as described below.

For the following description, it is convenient to view ByteDecode and ByteEncode as converting between integers and bits. The conversion between bits and bytes is straightforward and done using BitsToBytes and BytesToBits. The valid range of values for the parameter d is $1 \leq d \leq 12$. Bit arrays are divided into d-bit segments. The operations are performed in two different ways, depending on the value of d:

- For d satisfying $1 \leq d \leq 11$, the conversion is one-to-one. ByteDecode_d converts each d-bit segment of its input into one integer modulo 2^d, while ByteEncode_d performs the inverse operation.

- For $d = 12$, ByteDecode_{12} produces integers modulo q as output, while ByteEncode_{12} receives integers modulo q as input. Specifically, ByteDecode_{12} converts each 12-bit segment of its input into an integer modulo $2^{12} = 4096$ and then reduces the result modulo q. This is no longer a one-to-one operation. Indeed, some 12-bit segments could correspond to an integer greater than $q - 1 = 3328$ but less than 4096. However, this cannot occur for arrays produced by ByteEncode_{12}. These aspects of ByteDecode_{12} and

ByteEncode$_{12}$ will be important when considering checking of the ML-KEM encapsulation key in Section 7.

Algorithm 5 ByteEncode$_d(F)$

Encodes an array of d-bit integers into a byte array for $1 \leq d \leq 12$.

Input: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$, and $m = q$ if $d = 12$.
Output: byte array $B \in \mathbb{B}^{32d}$.

1: **for** $(i \leftarrow 0; i < 256; i$++$)$
2:      $a \leftarrow F[i]$ $\triangleright \, a \in \mathbb{Z}_m$
3:      **for** $(j \leftarrow 0; j < d; j$++$)$
4:          $b[i \cdot d + j] \leftarrow a \bmod 2$ $\triangleright \, b \in \{0,1\}^{256 \cdot d}$
5:          $a \leftarrow (a - b[i \cdot d + j])/2$ $\triangleright$ note $a - b[i \cdot d + j]$ is always even
6:      **end for**
7: **end for**
8: $B \leftarrow$ BitsToBytes(b)
9: **return** B

Algorithm 6 ByteDecode$_d(B)$

Decodes a byte array into an array of d-bit integers for $1 \leq d \leq 12$.

Input: byte array $B \in \mathbb{B}^{32d}$.
Output: integer array $F \in \mathbb{Z}_m^{256}$, where $m = 2^d$ if $d < 12$ and $m = q$ if $d = 12$.

1: $b \leftarrow$ BytesToBits(B)
2: **for** $(i \leftarrow 0; i < 256; i$++$)$
3:      $F[i] \leftarrow \sum_{j \leftarrow 0}^{d-1} b[i \cdot d + j] \cdot 2^j \bmod m$
4: **end for**
5: **return** F

4.2.2 Sampling Algorithms

The algorithms of ML-KEM require two sampling subroutines that are specified in Algorithms 7 and 8. Both of these algorithms can be used to convert a stream of uniformly random bytes into a sample from some desired distribution. In this standard, these algorithms will be invoked with a stream of pseudorandom bytes as the input. It follows that the output will then be a sample from a distribution that is computationally indistinguishable from the desired distribution.

Uniform sampling of NTT representations. The algorithm SampleNTT (Algorithm 7) converts a seed together with two indexing bytes into a polynomial in the NTT domain. If the seed is uniformly random, the resulting polynomial will be drawn from a distribution that is computationally indistinguishable from the uniform distribution on T_q. The output of SampleNTT is an array in $\mathbb{Z}_q^{256}$ that contains the coefficients of the sampled element of T_q (see Section 2.4.4). See Appendix B for a note on (optionally) safely bounding the algorithm's while-loop iterations.

Algorithm 7 SampleNTT(B)

Takes a 32-byte seed and two indices as input and outputs a pseudorandom element of T_q.

Input: byte array $B \in \mathbb{B}^{34}$.   ▷ a 32-byte seed along with two indices
Output: array $\hat{a} \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the NTT of a polynomial

1: ctx ← XOF.Init()
2: ctx ← XOF.Absorb(ctx, B)   ▷ input the given byte array into XOF
3: $j \leftarrow 0$
4: **while** $j < 256$ **do**
5:     (ctx, C) ← XOF.Squeeze(ctx, 3)   ▷ get a fresh 3-byte array C from XOF
6:     $d_1 \leftarrow C[0] + 256 \cdot (C[1] \bmod 16)$ ▷ $0 \leq d_1 < 2^{12}$
7:     $d_2 \leftarrow \lfloor C[1]/16 \rfloor + 16 \cdot C[2]$ ▷ $0 \leq d_2 < 2^{12}$
8:     **if** $d_1 < q$ **then**
9:         $\hat{a}[j] \leftarrow d_1$ ▷ $\hat{a} \in \mathbb{Z}_q^{256}$
10:        $j \leftarrow j+1$
11:     **end if**
12:     **if** $d_2 < q$ and $j < 256$ **then**
13:        $\hat{a}[j] \leftarrow d_2$
14:        $j \leftarrow j+1$
15:     **end if**
16: **end while**
17: **return** $\hat{a}$

Sampling from the centered binomial distribution. ML-KEM makes use of a special distribution $\mathcal{D}_\eta(R_q)$ of polynomials in R_q with small coefficients. Such polynomials are sometimes referred to as "errors" or "noise." The distribution is parameterized by an integer $\eta \in \{2, 3\}$. To sample a polynomial from $\mathcal{D}_\eta(R_q)$, each of its coefficients is independently sampled from a certain centered binomial distribution (CBD) on $\mathbb{Z}_q$. The algorithm SamplePolyCBD (Algorithm 8) samples the coefficient array of a polynomial $f \in R_q$ according to the distribution $\mathcal{D}_\eta(R_q)$, provided that

---

Algorithm 8 SamplePolyCBD$_\eta(B)$

Takes a seed as input and outputs a pseudorandom sample from the distribution $\mathcal{D}_\eta(R_q)$.

Input: byte array $B \in \mathbb{B}^{64\eta}$.
Output: array $f \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the sampled polynomial

1: $b \leftarrow$ BytesToBits(B)
2: **for** ($i \leftarrow 0$; $i < 256$; i++)
3:     $x \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + j]$ ▷ $0 \leq x \leq \eta$
4:     $y \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + \eta + j]$ ▷ $0 \leq y \leq \eta$
5:     $f[i] \leftarrow x - y \bmod q$ ▷ $0 \leq f[i] \leq \eta$ or $q-\eta \leq f[i] \leq q-1$
6: **end for**
7: **return** f

---

its input is a stream of uniformly random bytes.

4.3   The Number-Theoretic Transform

The Number-Theoretic Transform (NTT) can be viewed as a specialized, exact version of the discrete Fourier transform. In the case of ML-KEM, the NTT is used to improve the efficiency of multiplication in the ring R_q. Recall that R_q is the ring $\mathbb{Z}_q[X]/(X^n + 1)$ of polynomials of the form $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$ (where $f_j \in \mathbb{Z}_q$ for all j), with the ring operations defined by arithmetic modulo $X^n + 1$.

The ring R_q is isomorphic to another ring T_q, which is a direct sum of 128 quadratic extensions of $\mathbb{Z}_q$. The NTT is a computationally efficient isomorphism between these two rings. When a polynomial $f \in R_q$ is input, the NTT outputs an element $\hat{f} := \mathsf{NTT}(f)$ of the ring T_q, where $\hat{f}$ is called the "NTT representation" of f. The isomorphism property implies that

$$f \times_{R_q} g = \mathsf{NTT}^{-1}(\hat{f} \times_{T_q} \hat{g}), \tag{4.9}$$

where $\times_{R_q}$ and $\times_{T_q}$ denote multiplication in R_q and T_q, respectively. Moreover, since T_q is a product of 128 rings that each consist of polynomials of degree at most one, the operation $\times_{T_q}$ is much more efficient than the operation $\times_{R_q}$. For these reasons, the NTT is considered to be an integral part of ML-KEM and not merely an optimization.

As the rings R_q and T_q have a vector space structure over $\mathbb{Z}_q$, the most natural abstract data type to represent elements from either of these rings is $\mathbb{Z}_q^n$. For this reason, the choice of data structure for the inputs and outputs of NTT and NTT^{-1} are length-n arrays of integers modulo q. These arrays are understood to represent elements of T_q or R_q, respectively (see Section 2.4.4). Algorithms 9 and 10 describe an efficient means of computing NTT and NTT^{-1} in place. However, to clarify the distinction between the algebraic objects before and after the conversion, the algorithms are written with explicit inputs and outputs. This is consistent with this standard's convention that all inputs are passed by copy.

The mathematical structure of the NTT. In ML-KEM, q is the prime $3329 = 2^8 \cdot 13 + 1$, and $n = 256$. There are 128 primitive 256-th roots of unity and no primitive 512-th roots of unity in $\mathbb{Z}_q$. Note that $\zeta = 17 \in \mathbb{Z}_q$ is a primitive 256-th root of unity modulo q. Thus, $\zeta^{128} \equiv -1$.

Define $\mathsf{BitRev}_7(i)$ to be the integer represented by bit-reversing the unsigned 7-bit value that corresponds to the input integer $i \in \{0, \dots, 127\}$.

The polynomial $X^{256} + 1$ factors into 128 polynomials of degree 2 modulo q as follows:

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1} \right). \tag{4.10}$$

Therefore, $R_q := \mathbb{Z}_q[X]/(X^{256} + 1)$ is isomorphic to a direct sum of 128 quadratic extension

fields of $\mathbb{Z}_q$, denoted T_q. Specifically, this ring has the structure

$$T_q := \bigoplus_{i=0}^{127} \mathbb{Z}_q[X]/\left(X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1}\right). \tag{4.11}$$

Thus, the NTT representation $\hat{f} \in T_q$ of a polynomial $f \in R_q$ is the vector that consists of the corresponding residues of degree at most one:

$$\hat{f} := \left(f \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(0)+1}), \ldots, f \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(127)+1})\right). \tag{4.12}$$

As discussed in Section 2.4.4, the algorithms in this standard represent $\hat{f}$ as an array of 256 integers modulo q. Specifically,

$$f \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1}) = \hat{f}[2i] + \hat{f}[2i+1]X, \tag{4.13}$$

for i from 0 to 127.

The ML-KEM NTT algorithms. An algorithm for the ML-KEM NTT is described in Algorithm 9. An algorithm for the inverse operation (NTT^{-1}) is described in Algorithm 10. These two algorithms will be used to transform elements of R_q to elements of T_q (using NTT) and vice versa (using NTT^{-1}). In addition, as discussed in Section 2.4.8, these algorithms represent the coordinate-wise transformation of structures over those rings. Specifically, they map matrices/vectors with entries in R_q to matrices/vectors with entries in T_q (using NTT) and vice versa (using NTT^{-1}).

The values $\zeta^{\mathsf{BitRev}_7(i)} \bmod q$ for $i = 1, \ldots, 127$ used in line 5 of Algorithm 9 and line 5 of Algorithm 10 may be precomputed into an array. This array is given in Appendix A.

Algorithm 9 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the input polynomial
Output: array $\hat{f} \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the NTT of the input polynomial

1: $\hat{f} \leftarrow f$ ▷ will compute in place on a copy of input array
2: $i \leftarrow 1$
3: **for** ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
4:        **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5:              $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(i)} \bmod q$
6:              $i \leftarrow i + 1$
7:              **for** ($j \leftarrow start$; $j < start + len$; j++)
8:                    $t \leftarrow zeta \cdot \hat{f}[j + len]$ ▷ steps 8-10 done modulo q
9:                    $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
10:                  $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11:            **end for**
12:      **end for**
13: **end for**
14: **return** $\hat{f}$

Algorithm 10 NTT$^{-1}(\hat{f})$

Computes the polynomial $f \in R_q$ that corresponds to the given NTT representation $\hat{f} \in T_q$.

Input: array $\hat{f} \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of input NTT representation
Output: array $f \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the inverse NTT of the input

1: $f \leftarrow \hat{f}$ ▷ will compute in place on a copy of input array
2: $i \leftarrow 127$
3: **for** ($len \leftarrow 2$; $len \leq 128$; $len \leftarrow 2 \cdot len$)
4:        **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5:              $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(i)} \bmod q$
6:              $i \leftarrow i - 1$
7:              **for** ($j \leftarrow start$; $j < start + len$; j++)
8:                    $t \leftarrow f[j]$
9:                    $f[j] \leftarrow t + f[j + len]$ ▷ steps 9-10 done modulo q
10:                  $f[j + len] \leftarrow zeta \cdot (f[j + len] - t)$
11:            **end for**
12:      **end for**
13: **end for**
14: $f \leftarrow f \cdot 3303 \bmod q$ ▷ multiply every entry by $3303 \equiv 128^{-1} \bmod q$
15: **return** f

4.3.1   Multiplication in the NTT Domain

The addition and scalar multiplication of elements of T_q can be done using the corresponding coordinate-wise arithmetic operations on the coefficient arrays (see Section 2.4.5). This section describes how to do the remaining ring operation (i.e., multiplication in T_q).

Recall from (4.12) that $\hat{f} \in T_q$ is a vector of 128 degree-one residues modulo quadratic polynomials. Algebraically, multiplication in the ring T_q consists of independent multiplication in each of the 128 coordinates with respect to the quadratic modulus of that coordinate. Specifically, the i-th coordinate in T_q of the product $\hat{h} = \hat{f} \times_{T_q} \hat{g}$ is determined by the calculation

$$\hat{h}[2i] + \hat{h}[2i+1]X = (\hat{f}[2i] + \hat{f}[2i+1]X)(\hat{g}[2i] + \hat{g}[2i+1]X) \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1}).$$
(4.14)

Algorithm 11 MultiplyNTTs$(\hat{f}, \hat{g})$

Computes the product (in the ring T_q) of two NTT representations.

Input: Two arrays $\hat{f} \in \mathbb{Z}_q^{256}$ and $\hat{g} \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of two NTT representations
Output: An array $\hat{h} \in \mathbb{Z}_q^{256}$.   ▷ the coefficients of the product of the inputs

 1: **for** $(i \leftarrow 0; i < 128; i\text{++})$
 2:     $(\hat{h}[2i], \hat{h}[2i+1]) \leftarrow$ BaseCaseMultiply$(\hat{f}[2i], \hat{f}[2i+1], \hat{g}[2i], \hat{g}[2i+1], \zeta^{2\mathsf{BitRev}_7(i)+1})$
 3: **end for**
 4: **return** $\hat{h}$

Thus, one can compute the product of two elements of T_q using the algorithm MultiplyNTTs (Algorithm 11), which uses BaseCaseMultiply (Algorithm 12) as a subroutine. The values $\zeta^{2\mathsf{BitRev}_7(i)+1}$ used in Algorithm 11 may be precomputed and stored in an array (see Appendix A). MultiplyNTTs also enables linear-algebraic arithmetic with matrices and vectors whose entries are in T_q (see Section 2.4.7).

Algorithm 12 BaseCaseMultiply$(a_0, a_1, b_0, b_1, \gamma)$

Computes the product of two degree-one polynomials with respect to a quadratic modulus.

Input: $a_0, a_1, b_0, b_1 \in \mathbb{Z}_q$.   ▷ the coefficients of $a_0 + a_1 X$ and $b_0 + b_1 X$
Input: $\gamma \in \mathbb{Z}_q$.   ▷ the modulus is $X^2 - \gamma$
Output: $c_0, c_1 \in \mathbb{Z}_q$.   ▷ the coefficients of the product of the two polynomials

 1: $c_0 \leftarrow a_0 \cdot b_0 + a_1 \cdot b_1 \cdot \gamma$ ▷ steps 1-2 done modulo q
 2: $c_1 \leftarrow a_0 \cdot b_1 + a_1 \cdot b_0$
 3: **return** (c_0, c_1)

---

5. The K-PKE Component Scheme

This section describes the component scheme K-PKE. As discussed in Section 3.3, K-PKE is **not approved** for use in a stand-alone fashion. It serves only as a collection of subroutines for use in the algorithms of the **approved** scheme ML-KEM, as described in Section 7.

K-PKE consists of three algorithms:

1. Key generation (K-PKE.KeyGen)

2. Encryption (K-PKE.Encrypt)

3. Decryption (K-PKE.Decrypt)

When K-PKE is instantiated as part of ML-KEM, K-PKE inherits the parameter set selected for ML-KEM. Each parameter set specifies numerical values for each parameter. While n is always 256 and q is always 3329, the values of the remaining parameters k, η_1, η_2, d_u, and d_v vary among the three parameter sets. Parameters and parameter sets are described in Section 8.

The algorithms in this section do not perform any input checking because they are only invoked as subroutines of the main ML-KEM algorithms. The algorithms of ML-KEM themselves do perform input checking as needed.

Each of the algorithms of K-PKE is accompanied by a brief, informal description in text. For simplicity, this description is written in terms of vectors and matrices whose entries are elements of R_q. In the actual algorithm, most of the computations occur in the NTT domain in order to improve the efficiency of multiplication. The relevant vectors and matrices will then have entries in T_q. Linear-algebraic arithmetic with such vectors and matrices (e.g., line 18 of K-PKE.KeyGen) is performed as described in Sections 2.4.7 and 4.3.1. The encryption and decryption keys of K-PKE are also stored in the NTT form.

5.1 K-PKE Key Generation

The key generation algorithm K-PKE.KeyGen of K-PKE (Algorithm 13) receives a seed as input and outputs an encryption key ek_{PKE} and a decryption key dk_{PKE}. As is typically the case for public-key encryption, the encryption key can be made public, while the decryption key and the randomness must remain private. Indeed, the encryption key of K-PKE will serve as the encapsulation key of ML-KEM (see ML-KEM.KeyGen below) and can thus be made public. Meanwhile, the decryption key and seed of K-PKE.KeyGen must remain private as they can be used to perform decapsulation in ML-KEM.

The matrix $\hat{\mathbf{A}}$ generated in steps 3-7 of K-PKE.KeyGen can be stored, as specified in Section 3.3. This allows later operations to use $\hat{\mathbf{A}}$ directly rather than re-expanding it from the public seed ρ.

Informal description. The decryption key of K-PKE.KeyGen is a length-k vector $\mathbf{s}$ of elements of R_q (i.e., $\mathbf{s} \in R_q^k$). Roughly speaking, $\mathbf{s}$ is a set of secret variables, while the encryption key is a collection of "noisy" linear equations $(\mathbf{A}, \mathbf{As} + \mathbf{e})$ in the secret variables $\mathbf{s}$. The rows of the matrix $\mathbf{A}$ form the equation coefficients. This matrix is generated pseudorandomly using XOF with only a seed stored in the encryption key. The secret $\mathbf{s}$ and the "noise" $\mathbf{e}$ are sampled from

---

Algorithm 13 K-PKE.KeyGen(d)

Uses randomness to generate an encryption key and a corresponding decryption key.

Input: randomness $d \in \mathbb{B}^{32}$.
Output: encryption key $ek_{PKE} \in \mathbb{B}^{384k+32}$.
Output: decryption key $dk_{PKE} \in \mathbb{B}^{384k}$.

1: $(\rho, \sigma) \leftarrow G(d\|k)$　　　　　　　▷ expand 32+1 bytes to two pseudorandom 32-byte seeds[1]
2: $N \leftarrow 0$
3: **for** $(i \leftarrow 0; i < k; i\text{++})$　　　　　　　　　　　　　　　▷ generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
4: 　　**for** $(j \leftarrow 0; j < k; j\text{++})$
5: 　　　　$\hat{\mathbf{A}}[i,j] \leftarrow \text{SampleNTT}(\rho\|j\|i)$　　　　▷ j and i are bytes 33 and 34 of the input
6: 　　**end for**
7: **end for**
8: **for** $(i \leftarrow 0; i < k; i\text{++})$　　　　　　　　　　　　　　　　▷ generate $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$
9: 　　$\mathbf{s}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$　　　▷ $\mathbf{s}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
10: 　　$N \leftarrow N+1$
11: **end for**
12: **for** $(i \leftarrow 0; i < k; i\text{++})$　　　　　　　　　　　　　　　　▷ generate $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$
13: 　　$\mathbf{e}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$　　　▷ $\mathbf{e}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
14: 　　$N \leftarrow N+1$
15: **end for**
16: $\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$　　　　　　　　　　　▷ run NTT k times (once for each coordinate of $\mathbf{s}$)
17: $\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$　　　　　　　　　　　　　　　　　　　▷ run NTT k times
18: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$　　　　　　　　　　　　▷ noisy linear system in NTT domain
19: $ek_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{t}})\|\rho$　　　　▷ run ByteEncode_{12} k times, then append $\hat{\mathbf{A}}$-seed
20: $dk_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{s}})$　　　　　　　　▷ run ByteEncode_{12} k times
21: **return** (ek_{PKE}, dk_{PKE})

---

centered binomial distributions using randomness expanded from another seed σ via PRF. Once $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{e}$ are generated, the computation of the final part $\mathbf{t} = \mathbf{As} + \mathbf{e}$ of the encryption key takes place. The results are appropriately encoded into byte arrays and output.

In K-PKE.KeyGen, the choice of parameter set affects the length of the secret $\mathbf{s}$ (via the parameter k) and, as a consequence, the sizes of the noise vector $\mathbf{e}$ and the pseudorandom matrix $\mathbf{A}$. The choice of parameter set also affects the noise distribution (via the parameter η_1) used to sample the entries of $\mathbf{s}$ and $\mathbf{e}$.

5.2　K-PKE Encryption

The encryption algorithm K-PKE.Encrypt of K-PKE (Algorithm 14) takes an encryption key ek_{PKE}, a 32-byte plaintext m, and randomness r as input and produces a single output: a ciphertext c.

---

[1]Byte 33 of the input to G is the module dimension $k \in \{2, 3, 4\} \subset \mathbb{B}$. This is included to establish domain separation between the three parameter sets. For implementations that use the seed in place of the private key, this ensures that the expansion will produce an unrelated key if the seed is mistakenly expanded using a parameter set other than the originally intended one.

The matrix $\hat{\mathbf{A}}$ generated in steps 4-8 of K-PKE.Encrypt can be stored, as specified in Section 3.3. This allows later operations to use $\hat{\mathbf{A}}$ directly rather than re-expanding it from the public seed ρ.

Algorithm 14 K-PKE.Encrypt$(\mathrm{ek}_{\mathsf{PKE}}, m, r)$

Uses the encryption key to encrypt a plaintext message using the randomness r.

Input: encryption key $\mathrm{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$.
Input: message $m \in \mathbb{B}^{32}$.
Input: randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

1: $N \leftarrow 0$
2: $\hat{\mathbf{t}} \leftarrow \mathrm{ByteDecode}_{12}(\mathrm{ek}_{\mathsf{PKE}}[0 : 384k])$ ▷ run $\mathrm{ByteDecode}_{12}$ k times to decode $\hat{\mathbf{t}} \in (\mathbb{Z}_q^{256})^k$
3: $\rho \leftarrow \mathrm{ek}_{\mathsf{PKE}}[384k : 384k + 32]$ ▷ extract 32-byte seed from $\mathrm{ek}_{\mathsf{PKE}}$
4: **for** $(i \leftarrow 0; i < k; i\text{++})$ ▷ re-generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$ sampled in Alg. 13
5:     **for** $(j \leftarrow 0; j < k; j\text{++})$
6:         $\hat{\mathbf{A}}[i, j] \leftarrow \mathrm{SampleNTT}(\rho \| j \| i)$ ▷ j and i are bytes 33 and 34 of the input
7:     **end for**
8: **end for**
9: **for** $(i \leftarrow 0; i < k; i\text{++})$ ▷ generate $\mathbf{y} \in (\mathbb{Z}_q^{256})^k$
10:     $\mathbf{y}[i] \leftarrow \mathrm{SamplePolyCBD}_{\eta_1}(\mathrm{PRF}_{\eta_1}(r, N))$ ▷ $\mathbf{y}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
11:     $N \leftarrow N + 1$
12: **end for**
13: **for** $(i \leftarrow 0; i < k; i\text{++})$ ▷ generate $\mathbf{e_1} \in (\mathbb{Z}_q^{256})^k$
14:     $\mathbf{e_1}[i] \leftarrow \mathrm{SamplePolyCBD}_{\eta_2}(\mathrm{PRF}_{\eta_2}(r, N))$ ▷ $\mathbf{e_1}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
15:     $N \leftarrow N + 1$
16: **end for**
17: $e_2 \leftarrow \mathrm{SamplePolyCBD}_{\eta_2}(\mathrm{PRF}_{\eta_2}(r, N))$ ▷ sample $e_2 \in \mathbb{Z}_q^{256}$ from CBD
18: $\hat{\mathbf{y}} \leftarrow \mathrm{NTT}(\mathbf{y})$ ▷ run NTT k times
19: $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}) + \mathbf{e_1}$ ▷ run NTT^{-1} k times
20: $\mu \leftarrow \mathrm{Decompress}_1(\mathrm{ByteDecode}_1(m))$
21: $v \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}) + e_2 + \mu$ ▷ encode plaintext m into polynomial v
22: $c_1 \leftarrow \mathrm{ByteEncode}_{d_u}(\mathrm{Compress}_{d_u}(\mathbf{u}))$ ▷ run $\mathrm{ByteEncode}_{d_u}$ and $\mathrm{Compress}_{d_u}$ k times
23: $c_2 \leftarrow \mathrm{ByteEncode}_{d_v}(\mathrm{Compress}_{d_v}(v))$
24: **return** $c \leftarrow (c_1 \| c_2)$

Informal description. The algorithm K-PKE.Encrypt begins by extracting the vector $\mathbf{t}$ and the seed from the encryption key. The seed is then expanded to re-generate the matrix $\mathbf{A}$ in the same manner as was done in K-PKE.KeyGen. If $\mathbf{t}$ and $\mathbf{A}$ are derived correctly from an encryption key produced by K-PKE.KeyGen, then they are equal to their corresponding values in K-PKE.KeyGen.

Recall from the description of key generation that the pair $(\mathbf{A}, \mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e})$ can be thought of as a system of noisy linear equations in the secret variables $\mathbf{s}$. One can generate an additional noisy linear equation in the same secret variables — without knowing $\mathbf{s}$ — by picking a random linear

combination of the noisy equations in the system $(\mathbf{A}, \mathbf{t})$. One can then encode information in the "constant term" (i.e., the entry that is a linear combination of entries of $\mathbf{t}$) of such a combined equation. This information can then be deciphered by a party in possession of $\mathbf{s}$. For example, one could encode a single bit by deciding whether or not to significantly alter the constant term, thus making either a nearly correct equation that corresponds to the decrypted bit value of 0 or a far-from-correct equation that corresponds to the decrypted bit value of 1. In the case of K-PKE, the constant term is a polynomial with 256 coefficients, so one can encode more information: one bit in each coefficient.

To this end, K-PKE.Encrypt proceeds by generating a vector $\mathbf{y} \in R_q^k$ and noise terms $\mathbf{e_1} \in R_q^k$ and $e_2 \in R_q$, all of which are sampled from the centered binomial distribution using pseudorandomness expanded via PRF from the input randomness r. One then computes the "new noisy equation," which is (up to some details) $(\mathbf{A}^\top \mathbf{y} + \mathbf{e_1}, \mathbf{t}^\top \mathbf{y} + e_2)$. An appropriate encoding μ of the input message m is then added to the latter term in the pair. Finally, the resulting pair $(\mathbf{u}, v)$ is compressed, serialized into a byte array, and output as the ciphertext.

5.3   K-PKE Decryption

The decryption algorithm K-PKE.Decrypt of K-PKE (Algorithm 15) takes a decryption key $\mathsf{dk_{PKE}}$ and a ciphertext c as input, requires no randomness, and outputs a plaintext m.

Informal description. The algorithm K-PKE.Decrypt begins by recovering a pair $(\mathbf{u}', v')$ from the ciphertext c (see the description of K-PKE.Encrypt). Here, one can think of $\mathbf{u}'$ as the coefficients of the equation and v' as the constant term. The decryption key $\mathsf{dk_{PKE}}$ contains the vector of secret variables $\mathbf{s}$. The decryption algorithm can thus use the decryption key to compute the true constant term $v = \mathbf{s}^\top \mathbf{u}'$ and calculate $v' - v$. The decryption algorithm ends by decoding the plaintext message m from $v' - v$ and outputting m.

Algorithm 15 K-PKE.Decrypt($\mathsf{dk_{PKE}}, c$)

Uses the decryption key to decrypt a ciphertext.

Input: decryption key $\mathsf{dk_{PKE}} \in \mathbb{B}^{384k}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: message $m \in \mathbb{B}^{32}$.

1: $c_1 \leftarrow c[0 : 32 d_u k]$
2: $c_2 \leftarrow c[32 d_u k : 32(d_u k + d_v)]$
3: $\mathbf{u}' \leftarrow \mathsf{Decompress}_{d_u}(\mathsf{ByteDecode}_{d_u}(c_1))$ ▷ run $\mathsf{Decompress}_{d_u}$ and $\mathsf{ByteDecode}_{d_u}$ k times
4: $v' \leftarrow \mathsf{Decompress}_{d_v}(\mathsf{ByteDecode}_{d_v}(c_2))$
5: $\hat{\mathbf{s}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{dk_{PKE}})$ ▷ run $\mathsf{ByteDecode}_{12}$ k times
6: $w \leftarrow v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^\top \circ \mathsf{NTT}(\mathbf{u}'))$ ▷ run NTT k times; run NTT^{-1} once
7: $m \leftarrow \mathsf{ByteEncode}_1(\mathsf{Compress}_1(w))$ ▷ decode plaintext m from polynomial v
8: **return** m

---

6.   Main Internal Algorithms

This section specifies three algorithms: ML-KEM.KeyGen_internal, ML-KEM.Encaps_internal, and ML-KEM.Decaps_internal. These three algorithms are all *deterministic*, meaning that their output is completely determined by their input. No randomness is sampled inside of these algorithms. These three algorithms will be used to construct ML-KEM in Section 7. The algorithms in this section make use of the parameters n, q, k, d_u, and d_v. The subroutines they invoke additionally make use of the parameters η_1 and η_2. While n is always 256 and q is always 3329, the remaining parameters vary among the possible parameter sets (see Section 8).

The interfaces specified in this section will be used to test ML-KEM implementations through the Cryptographic Algorithm Validation Program (CAVP). The key generation function in this section may also be used to re-expand a key from a seed (see Section 3.3), including when obtaining assurance of private key possession via regeneration. As prescribed in Section 3.3, the interfaces specified in this section **should not** be made available to applications other than for testing purposes, and the random seeds (as specified in ML-KEM.KeyGen and ML-KEM.Encaps in Section 7) **shall** be generated by the cryptographic module.

6.1   Internal Key Generation

The algorithm ML-KEM.KeyGen_internal (Algorithm 16) accepts two random seeds as input, and produces an encapsulation key and a decapsulation key.

Informal description. The core subroutine of ML-KEM.KeyGen_internal is the key generation algorithm of K-PKE (Algorithm 13). The encapsulation key is simply the encryption key of K-PKE. The decapsulation key consists of the decryption key of K-PKE, the encapsulation key, a hash of the encapsulation key, and a random 32-byte value. This random value will be used in the "implicit rejection" mechanism of the internal decapsulation algorithm (Algorithm 18).

Algorithm 16 ML-KEM.KeyGen_internal(d, z)

Uses randomness to generate an encapsulation key and a corresponding decapsulation key.

Input: randomness $d \in \mathbb{B}^{32}$.
Input: randomness $z \in \mathbb{B}^{32}$.
Output: encapsulation key ek $\in \mathbb{B}^{384k+32}$.
Output: decapsulation key dk $\in \mathbb{B}^{768k+96}$.

  1: $(\text{ek}_{\text{PKE}}, \text{dk}_{\text{PKE}}) \leftarrow$ K-PKE.KeyGen(d) ▷ run key generation for K-PKE
  2: $\text{ek} \leftarrow \text{ek}_{\text{PKE}}$ ▷ KEM encaps key is just the PKE encryption key
  3: $\text{dk} \leftarrow (\text{dk}_{\text{PKE}} \| \text{ek} \| H(\text{ek}) \| z)$ ▷ KEM decaps key includes PKE decryption key
  4: **return** (ek, dk)

6.2   Internal Encapsulation

The algorithm ML-KEM.Encaps_internal (Algorithm 17) accepts an encapsulation key and a random byte array as input and outputs a ciphertext and a shared key.

Algorithm 17 ML-KEM.Encaps_internal(ek, m)

Uses the encapsulation key and randomness to generate a key and an associated ciphertext.

Input: encapsulation key ek $\in \mathbb{B}^{384k+32}$.
Input: randomness $m \in \mathbb{B}^{32}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

1: $(K, r) \leftarrow$ G($m \| $H(ek)) $\triangleright$ derive shared secret key K and randomness r
2: $c \leftarrow$ K-PKE.Encrypt(ek, m, r) $\triangleright$ encrypt m using K-PKE with randomness r
3: **return** (K, c)

Informal description. The core subroutine of ML-KEM.Encaps_internal is the encryption algorithm of K-PKE, which is used to encrypt a random value m into a ciphertext c. A copy of the shared secret key K and the randomness used during encryption are derived from m and the encapsulation key ek via hashing. Specifically, H is applied to ek, and the result is concatenated with m and then hashed using G. Finally, the algorithm outputs the shared secret key K and the ciphertext c.

6.3   Internal Decapsulation

The algorithm ML-KEM.Decaps_internal (Algorithm 18) accepts a decapsulation key and a ciphertext as input, does not use any randomness, and outputs a shared secret key.

Informal description. The algorithm ML-KEM.Decaps_internal begins by parsing out the components of the decapsulation key dk of ML-KEM. These components are an (encryption key, decryption key) pair for K-PKE, a hash value h, and a random value z. The decryption key of K-PKE is then used to decrypt the input ciphertext c to get a plaintext m'. The decapsulation algorithm then re-encrypts m' and computes a candidate shared secret key K' in the same manner as should have been done in encapsulation. Specifically, K' and the encryption randomness r' are computed by hashing m' and the encryption key of K-PKE, and a ciphertext c' is generated by encrypting m' using randomness r'. Finally, decapsulation checks whether the resulting ciphertext c' matches the provided ciphertext c. If it does not, the algorithm performs an "implicit rejection": the value of K' is changed to a hash of c together with the random value z stored in the ML-KEM secret key (see the discussion of decapsulation failures in Section 3.2). In either case, decapsulation outputs the resulting shared secret key K'.

The "implicit reject" flag computed in step 9 (by comparing c and c') is a secret piece of intermediate data. As specified in the requirements in Section 3.3, this flag **shall** be destroyed prior to ML-KEM.Decaps_internal terminating. In particular, returning the value of the flag as an output in any form is not permitted.

Algorithm 18 ML-KEM.Decaps_internal(dk, c)

Uses the decapsulation key to produce a shared secret key from a ciphertext.

Input: decapsulation key $\mathrm{dk} \in \mathbb{B}^{768k+96}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.

1: $\mathrm{dk}_{\mathsf{PKE}} \leftarrow \mathrm{dk}[0:384k]$ $\triangleright$ extract (from KEM decaps key) the PKE decryption key
2: $\mathrm{ek}_{\mathsf{PKE}} \leftarrow \mathrm{dk}[384k:768k+32]$ $\triangleright$ extract PKE encryption key
3: $h \leftarrow \mathrm{dk}[768k+32:768k+64]$ $\triangleright$ extract hash of PKE encryption key
4: $z \leftarrow \mathrm{dk}[768k+64:768k+96]$ $\triangleright$ extract implicit rejection value
5: $m' \leftarrow$ K-PKE.Decrypt$(\mathrm{dk}_{\mathsf{PKE}}, c)$ $\triangleright$ decrypt ciphertext
6: $(K', r') \leftarrow \mathsf{G}(m' \| h)$
7: $\bar{K} \leftarrow \mathsf{J}(z \| c)$
8: $c' \leftarrow$ K-PKE.Encrypt$(\mathrm{ek}_{\mathsf{PKE}}, m', r')$ $\triangleright$ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10:      $K' \leftarrow \bar{K}$ $\triangleright$ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

34

7. The ML-KEM Key-Encapsulation Mechanism

This section describes the three main algorithms of the ML-KEM scheme:

1. Key generation (ML-KEM.KeyGen)

2. Encapsulation (ML-KEM.Encaps)

3. Decapsulation (ML-KEM.Decaps)

To instantiate ML-KEM, one must select a parameter set. Each parameter set is associated with a particular trade-off between security and performance. The three possible parameter sets are called ML-KEM-512, ML-KEM-768, and ML-KEM-1024 and are described in detail in Table 2 of Section 8. Each parameter set assigns specific numerical values to the individual parameters n, q, k, η_1, η_2, d_u, and d_v. While n is always 256 and q is always 3329, the remaining parameters vary among the three parameter sets. Implementers **shall** ensure that the three algorithms of ML-KEM listed above are only invoked with a valid parameter set, and that this parameter set is selected appropriately for the desired application. Moreover, implementers **shall** ensure that the parameter set used in any particular invocation of ML-KEM.Encaps or ML-KEM.Decaps matches the parameter set associated to the provided inputs.

7.1 ML-KEM Key Generation

The key generation algorithm ML-KEM.KeyGen for ML-KEM (Algorithm 19) accepts no input, generates randomness internally, and produces an encapsulation key and a decapsulation key. While the encapsulation key can be made public, the decapsulation key **shall** remain private.

The seed (d, z) generated in steps 1 and 2 of ML-KEM.KeyGen can be stored for later expansion using ML-KEM.KeyGen_internal (see Section 3.3). As the seed can be used to compute the decapsulation key, it is sensitive data and **shall** be treated with the same safeguards as a decapsulation key (see SP 800-227 [1]).

Algorithm 19 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.

Output: encapsulation key $ek \in \mathbb{B}^{384k+32}$.
Output: decapsulation key $dk \in \mathbb{B}^{768k+96}$.

1: $d \xleftarrow{\$} \mathbb{B}^{32}$ ▷ d is 32 random bytes (see Section 3.3)
2: $z \xleftarrow{\$} \mathbb{B}^{32}$ ▷ z is 32 random bytes (see Section 3.3)
3: **if** $d ==$ NULL **or** $z ==$ NULL **then**
4:     **return** $\perp$ ▷ return an error indication if random bit generation failed
5: **end if**
6: $(ek, dk) \leftarrow$ ML-KEM.KeyGen_internal(d, z) ▷ run internal key generation algorithm
7: **return** (ek, dk)

Secure key establishment depends on the use of key pairs that have been properly generated via ML-KEM.KeyGen. If the owner of a KEM key pair did not generate the key pair but instead received it from a trusted third party or other source, the owner may optionally perform certain

checks on the key pair. While these checks can detect certain corruptions, they do not guarantee that the key pair was properly generated.

Key pair check. To check a candidate key pair[1] $(\overline{ek}, \overline{dk})$, perform the following checks:

1. *(Seed consistency)* If a seed (d, z) is available, run ML-KEM.KeyGen_internal(d, z), and verify that the output is equal to $(\overline{ek}, \overline{dk})$.

2. *(Encapsulation key check)* Check $\overline{ek}$ as specified in Section 7.2.

3. *(Decapsulation key check)* Check $\overline{dk}$ as specified in Section 7.3.

4. *(Pair-wise consistency)* Perform the following steps:

    i. Generate an array of 32 random bytes by performing $m \xleftarrow{\$} \mathbb{B}^{32}$.

    ii. Perform $(K, c) \leftarrow$ ML-KEM.Encaps_internal$(\overline{ek}, m)$.

    iii. Perform $K' \leftarrow$ ML-KEM.Decaps_internal$(\overline{dk}, c)$.

    iv. Reject unless $K == K'$.

It is important to note that this checking process does *not* guarantee that the key pair is a properly produced output of ML-KEM.KeyGen.

7.2   ML-KEM Encapsulation

The encapsulation algorithm ML-KEM.Encaps of ML-KEM (Algorithm 20) accepts an encapsulation key as input, generates randomness internally, and outputs a ciphertext and a shared key. This algorithm requires input checking, as specified below.

Encapsulation key check. To check a candidate encapsulation key $\overline{ek}$, perform the following:

1. *(Type check)* If $\overline{ek}$ is not an array of bytes of length $384k + 32$ for the value of k specified by the relevant parameter set, then input checking failed.

2. *(Modulus check)* Perform the computation

$$\text{test} \leftarrow \text{ByteEncode}_{12}(\text{ByteDecode}_{12}(\overline{ek}[0 : 384k])) \tag{7.1}$$

(see Section 4.2.1). If test $\neq \overline{ek}[0 : 384k]$, then input checking failed. This check ensures that the integers encoded in the public key are in the valid range $[0, q-1]$.

If both checks pass, then ML-KEM.Encaps can be run with input ek $:= \overline{ek}$. It is important to note that this checking process does *not* guarantee that ek is a properly produced output of ML-KEM.KeyGen.

ML-KEM.Encaps **shall not** be run with an encapsulation key that has not been checked as above. However, checking of the encapsulation key need not be performed by the encapsulating party,

---

[1]In discussions of input checking, the "low overline" in the notation indicates that the input has not yet been checked (e.g., $\overline{ek}$ has not yet been checked, while ek has passed the check).

nor with every execution of ML-KEM.Encaps. Instead, assurance that these checks have been performed can be acquired through other means (see SP 800-227 [1]).

Algorithm 20 ML-KEM.Encaps(ek)

Uses the encapsulation key to generate a shared secret key and an associated ciphertext.

Checked input: encapsulation key ek $\in \mathbb{B}^{384k+32}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

1: $m \xleftarrow{\$} \mathbb{B}^{32}$ $\triangleright$ m is 32 random bytes (see Section 3.3)
2: **if** $m ==$ NULL **then**
3:     **return** $\perp$ $\triangleright$ return an error indication if random bit generation failed
4: **end if**
5: $(K, c) \leftarrow$ ML-KEM.Encaps_internal(ek, m) $\triangleright$ run internal encapsulation algorithm
6: **return** (K, c)

7.3   ML-KEM Decapsulation

The decapsulation algorithm ML-KEM.Decaps of ML-KEM (Algorithm 21) accepts a decapsulation key and an ML-KEM ciphertext as input, does not use any randomness, and outputs a shared secret. This algorithm requires input checking, as specified below.

Decapsulation input check. To check a candidate decapsulation key $\overline{dk}$ and ciphertext $\overline{c}$, perform the following checks:

1. *(Ciphertext type check)* If $\overline{c}$ is not a byte array of length $32(d_u k + d_v)$ for the values of d_u, d_v, and k specified by the relevant parameter set, then input checking has failed.

2. *(Decapsulation key type check)* If $\overline{dk}$ is not a byte array of length $768k + 96$ for the value of k specified by the relevant parameter set, then input checking has failed.

3. *(Hash check)* Perform the computation

$$\text{test} \leftarrow \mathsf{H}(\overline{dk}[384k : 768k + 32])). \tag{7.2}$$

   If test $\neq \overline{dk}[768k + 32 : 768k + 64]$, then input checking has failed.

If all of the above checks pass, then ML-KEM.Decaps can be run with inputs dk $:= \overline{dk}$ and $c := \overline{c}$. It is important to note that this checking process does *not* guarantee that dk is a properly produced output of ML-KEM.KeyGen, nor that c is a properly produced output of ML-KEM.Encaps.

ML-KEM.Decaps **shall not** be run with a decapsulation key or a ciphertext unless both have been checked. However, checking of the decapsulation key need not be performed by the decapsulating party, nor with every execution of ML-KEM.Decaps. Instead, assurance that this check has been performed can be acquired through other means (see SP 800-227 [1]). Ciphertext checking **shall** be performed with every execution of ML-KEM.Decaps.

Algorithm 21 ML-KEM.Decaps(dk, c)

Uses the decapsulation key to produce a shared secret key from a ciphertext.

Checked input: decapsulation key $\text{dk} \in \mathbb{B}^{768k+96}$.
Checked input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.

1: $K' \leftarrow$ ML-KEM.Decaps_internal(dk, c) $\triangleright$ run internal decapsulation algorithm
2: **return** K'

---

8.   Parameter Sets

ML-KEM is equipped with three parameter sets, each of the which comprises five individual parameters: k, η_1, η_2, d_u, and d_v. There are also two constants: $n = 256$ and $q = 3329$. The following is a brief and informal description of the roles played by the variable parameters in the algorithms of K-PKE and ML-KEM. See Section 5 for details.

- The parameter k determines the dimensions of the matrix $\hat{\mathbf{A}}$ that appears in K-PKE.KeyGen and K-PKE.Encrypt. It also determines the dimensions of vectors $\mathbf{s}$ and $\mathbf{e}$ in K-PKE.KeyGen and the dimensions of vectors $\mathbf{y}$ and $\mathbf{e}_1$ in K-PKE.Encrypt.

- The parameter η_1 is required to specify the distribution for generating the vectors $\mathbf{s}$ and $\mathbf{e}$ in K-PKE.KeyGen and the vector $\mathbf{y}$ in K-PKE.Encrypt.

- The parameter η_2 is required to specify the distribution for generating the vectors $\mathbf{e}_1$ and e_2 in K-PKE.Encrypt.

- The parameters d_u and d_v serve as parameters and inputs for the functions Compress, Decompress, ByteEncode, and ByteDecode in K-PKE.Encrypt and K-PKE.Decrypt.

This standard approves the parameter sets given in Table 2. Each parameter set is associated with a required security strength for randomness generation (see Section 3.3). The sizes of the ML-KEM keys and ciphertexts for each parameter set are summarized in Table 3.

Table 2. Approved parameter sets for ML-KEM

	n	q	k	η_1	η_2	d_u	d_v	required RBG strength (bits)
ML-KEM-512	256	3329	2	3	2	10	4	128
ML-KEM-768	256	3329	3	2	2	10	4	192
ML-KEM-1024	256	3329	4	2	2	11	5	256

Table 3. Sizes (in bytes) of keys and ciphertexts of ML-KEM

	encapsulation key	decapsulation key	ciphertext	shared secret key
ML-KEM-512	800	1632	768	32
ML-KEM-768	1184	2400	1088	32
ML-KEM-1024	1568	3168	1568	32

A parameter set name can also be said to denote a (parameter-free) KEM. Specifically, ML-KEM-x can be used to denote the parameter-free KEM that results from instantiating the scheme ML-KEM with the parameter set ML-KEM-x.

The three parameter sets included in Table 2 were designed to meet certain security strength categories defined by NIST in its original Call for Proposals [4, 22]. These security strength categories are explained further in SP 800-57, Part 1 [7].

Using this approach, security strength is not described by a single number, such as "128 bits of security." Instead, each ML-KEM parameter set is claimed to be at least as secure as a generic

block cipher with a prescribed key size or a generic hash function with a prescribed output length. More precisely, it is claimed that the computational resources needed to break ML-KEM are greater than or equal to the computational resources needed to break the block cipher or hash function when those computational resources are estimated using any realistic model of computation. Different models of computation can be more or less realistic and, accordingly, lead to more or less accurate estimates of security strength. Some commonly studied models are discussed in [23].

Concretely, ML-KEM-512 is claimed to be in security category 1, ML-KEM-768 is claimed to be in security category 3, and ML-KEM-1024 is claimed to be in security category 5. For additional discussion of the security strength of MLWE-based cryptosystems, see [4].

Selecting an appropriate parameter set. When initially establishing cryptographic protections for data, the strongest possible parameter set **should** be used. This has a number of advantages, including reducing the likelihood of costly transitions to higher-security parameter sets in the future. At the same time, it should be noted that some parameter sets might have adverse performance effects for the relevant application (e.g., the algorithm may be unacceptably slow, or objects such as keys or ciphertexts may be unacceptably large).

NIST recommends using ML-KEM-768 as the default parameter set, as it provides a large security margin at a reasonable performance cost. In cases where this is impractical or even higher security is required, other parameter sets may be used.

References

[1] National Institute of Standards and Technology (2024) Recommendations for key-encapsulation mechanisms, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-227. [Forthcoming; will be available at https://csrc.nist.gov/publications].

[2] Barker EB, Chen L, Roginsky AL, Vassilev A, Davis R (2018) Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-56A Revision 3. https://doi.org/10.6028/NIST.SP.800-56Ar3.

[3] Barker EB, Chen L, Roginsky AL, Vassilev A, Davis R, Simon S (2019) Recommendation for pair-wise key-establishment using integer factorization cryptography (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-56B Revision 2. https://doi.org/10.6028/NIST.SP.800-56Br2.

[4] Avanzi R, Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanck JM, Schwabe P, Seiler G, Stehlé D (2020) CRYSTALS-Kyber algorithm specifications and supporting documentation, Third-round submission to the NIST's post-quantum cryptography standardization process. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[5] National Institute of Standards and Technology (2015) Secure hash standard (SHS), (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 180-4. https://doi.org/10.6028/NIST.FIPS.180-4.

[6] National Institute of Standards and Technology (2015) SHA-3 standard: Permutation-based hash and extendable-output functions, (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. https://doi.org/10.6028/NIST.FIPS.202.

[7] Barker EB (2020) Recommendation for key management: Part 1 - General, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 1, Rev. 5 [or as amended]. https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[8] Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanck JM, Schwabe P, Seiler G, Stehlé D (2018) CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM. *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp 353–367. https://doi.org/10.1109/EuroSP.2018.00032.

[9] Langlois A, Stehlé D (2015) Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* 75(3):565–599. https://doi.org/10.1007/s10623-014-9938-4.

[10] Regev O (2005) On lattices, learning with errors, random linear codes, and cryptography. *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing* STOC '05 (Association for Computing Machinery, New York, NY, USA), pp 84—93. https://doi.org/10.1145/1060590.1060603.

[11] Fujisaki E, Okamoto T (2013) Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology* 26:80–101. https://doi.org/10.1007/s00145-011-9114-1.

[12] Hofheinz D, Hövelmanns K, Kiltz E (2017) A modular analysis of the Fujisaki-Okamoto transformation. *Theory of Cryptography*, eds Kalai Y, Reyzin L (Springer International Publishing, Cham), pp 341–371. https://doi.org/10.1007/978-3-319-70500-2_12.

[13] Katz J, Lindell Y (2020) *Introduction to Modern Cryptography* (Chapman & Hall/CRC), 3rd Ed.

[14] Almeida JB, Olmos SA, Barbosa M, Barthe G, Dupressoir F, Grégoire B, Laporte V, Léchenet JC, Low C, Oliveira T, Pacheco H, Quaresma M, Schwabe P, Strub PY (2024) Formally verifying Kyber episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt, Cryptology ePrint Archive, Paper 2024/843. Available at https://eprint.iacr.org/2024/843.

[15] Ducas L, Schanck J (2021) Security estimation scripts for Kyber and Dilithium, Github repository. Available at https://github.com/pq-crystals/security-estimates.

[16] Chen L (2022) Recommendation for key derivation using pseudorandom functions, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-108r1-upd1, Includes updates as of February 2, 2024. https://doi.org/10.6028/NIST.SP.800-108r1-upd1.

[17] Barker EB, Chen L, Davis R (2020) Recommendation for key-derivation methods in key-establishment schemes (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-56C Revision 2. https://doi.org/10.6028/NIST.SP.800-56Cr2.

[18] Barker EB, Kelsey JM (2015) Recommendation for random number generation using deterministic random bit generators, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90A, Rev. 1. https://doi.org/10.6028/NIST.SP.800-90Ar1.

[19] Sönmez Turan M, Barker EB, Kelsey JM, McKay KA, Baish ML, Boyle M (2018) Recommendation for the entropy sources used for random bit generation, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B. https://doi.org/10.6028/NIST.SP.800-90B.

[20] Barker E, Kelsey J, McKay K, Roginsky A, Turan MS (2024) Recommendation for random bit generator (RBG) constructions, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90C 4pd. https://doi.org/10.6028/NIST.SP.800-90C.4pd.

[21] Kelsey J, Chang S, Perlner R (2016) SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-185 [or as amended]. https://doi.org/10.6028/NIST.SP.800-185.

[22] National Institute of Standards and Technology (2016) Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[23] Alagic G, Apon D, Cooper D, Dang Q, Dang T, Kelsey J, Lichtinger J, Liu YK, Miller C, Moody D, Peralta R, Perlner R, Robinson A, Smith-Tone D (2022) Status report on the third round of the NIST post-quantum cryptography standardization process (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 8413. https://doi.org/10.6028/NIST.IR.8413-upd1.

[24] CRYSTALS-Kyber submission team (2023) "Discussion about Kyber's tweaked FO transform", PQC-Forum Post. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/WFRDl8DqYQ4.

[25] CRYSTALS-Kyber submission team (2023) "Kyber decisions, part 2: FO transform", PQC-Forum Post. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/C0D3W1KoINY/m/99kIvydoAwAJ.

Appendix A — Precomputed Values for the NTT

The following 128 numbers are the values of $\zeta^{\mathrm{BitRev}_7(i)}$ mod q for $i \in \{0, \dots, 127\}$. These numbers are used in Algorithms 9 and 10.

$$
\begin{array}{rrrrrrrr}
\{\,1 & 1729 & 2580 & 3289 & 2642 & 630 & 1897 & 848 \\[6pt]
1062 & 1919 & 193 & 797 & 2786 & 3260 & 569 & 1746 \\[6pt]
296 & 2447 & 1339 & 1476 & 3046 & 56 & 2240 & 1333 \\[6pt]
1426 & 2094 & 535 & 2882 & 2393 & 2879 & 1974 & 821 \\[6pt]
289 & 331 & 3253 & 1756 & 1197 & 2304 & 2277 & 2055 \\[6pt]
650 & 1977 & 2513 & 632 & 2865 & 33 & 1320 & 1915 \\[6pt]
2319 & 1435 & 807 & 452 & 1438 & 2868 & 1534 & 2402 \\[6pt]
2647 & 2617 & 1481 & 648 & 2474 & 3110 & 1227 & 910 \\[6pt]
17 & 2761 & 583 & 2649 & 1637 & 723 & 2288 & 1100 \\[6pt]
1409 & 2662 & 3281 & 233 & 756 & 2156 & 3015 & 3050 \\[6pt]
1703 & 1651 & 2789 & 1789 & 1847 & 952 & 1461 & 2687 \\[6pt]
939 & 2308 & 2437 & 2388 & 733 & 2337 & 268 & 641 \\[6pt]
1584 & 2298 & 2037 & 3220 & 375 & 2549 & 2090 & 1645 \\[6pt]
1063 & 319 & 2773 & 757 & 2099 & 561 & 2466 & 2594 \\[6pt]
2804 & 1092 & 403 & 1026 & 1143 & 2150 & 2775 & 886 \\[6pt]
1722 & 1212 & 1874 & 1029 & 2110 & 2935 & 885 & 2154\,\}
\end{array}
$$

When implementing Algorithm 11, the values $\zeta^{2\mathrm{BitRev}_7(i)+1} \bmod q$ need to be computed. The following array contains these values for $i \in \{0, \dots, 127\}$:

{ 17   -17   2761   -2761   583   -583   2649   -2649

1637   -1637   723   -723   2288   -2288   1100   -1100

1409   -1409   2662   -2662   3281   -3281   233   -233

756   -756   2156   -2156   3015   -3015   3050   -3050

1703   -1703   1651   -1651   2789   -2789   1789   -1789

1847   -1847   952   -952   1461   -1461   2687   -2687

939   -939   2308   -2308   2437   -2437   2388   -2388

733   -733   2337   -2337   268   -268   641   -641

1584   -1584   2298   -2298   2037   -2037   3220   -3220

375   -375   2549   -2549   2090   -2090   1645   -1645

1063   -1063   319   -319   2773   -2773   757   -757

2099   -2099   561   -561   2466   -2466   2594   -2594

2804   -2804   1092   -1092   403   -403   1026   -1026

1143   -1143   2150   -2150   2775   -2775   886   -886

1722   -1722   1212   -1212   1874   -1874   1029   -1029

2110   -2110   2935   -2935   885   -885   2154   -2154 }

Appendix B — SampleNTT Loop Bounds

In SampleNTT (Algorithm 7), the algorithm repeatedly generates byte arrays from the XOF to create an element of T_q. If a generated byte array value is out of bounds for a coefficient of T_q, the algorithm tries again until all 256 coefficients are created. On average, this while loop will resolve within a reasonable number of iterations. However, there may be cases in which the generated byte arrays are consistently out of bounds and the algorithm may run for a higher number of iterations.

Implementations **should not** bound this loop, if at all possible. An incorrect limit will cause interoperability errors, and the chances for SampleNTT to iterate longer become exponentially rare. If an implementation does bound the number of iterations of SampleNTT, it **shall not** use a limit lower than those presented in Table 4. The calculated probability of SampleNTT exceeding the limit is included and calculated under standard assumptions about the output distributions of XOFs and hash functions.

Table 4. While-loop limits and probabilities of occurrence for SampleNTT

Number of iterations	Probability of reaching limit
280	2^{-261}

If a limit is used and the number of iterations exceeds the limit, then the algorithm **shall** destroy all intermediate results. If a return value or exception is produced, it **shall** be the same value for any execution in which the maximum number of iterations is exceeded.

Appendix C — Differences From the CRYSTALS-Kyber Submission

This appendix lists the differences between CRYSTALS-KYBER (as described in [4]) and the ML-KEM scheme (specified in this document) that result in differing input-output behavior of the main algorithms (i.e., KeyGen, Encaps, Decaps). Since a conforming implementation need only match the input-output behavior of these three algorithms (see "Implementations" and Section 3.3 below), the list does not include any of the numerous differences in how the main algorithms actually produce outputs from inputs (e.g., via different computational steps or different subroutines), nor any differences in presentation between this standard and [4].

C.1   Differences Between CRYSTALS-Kyber and FIPS 203 Initial Public Draft

- In the third-round specification [4], the shared secret key was treated as a variable-length value whose length depended on how it would be used in the relevant application. In this specification, the length of the shared secret key is fixed to 256 bits. It can be used directly in applications as a symmetric key, or symmetric keys can be derived from it, as specified in Section 3.3.

- The ML-KEM.Encaps and ML-KEM.Decaps algorithms in this specification use a different variant of the Fujisaki-Okamoto transform (see [24, 25]) than the third-round specification [4]. Specifically, ML-KEM.Encaps no longer includes a hash of the ciphertext in the derivation of the shared secret, and ML-KEM.Decaps has been adjusted to match this change.

- In the third-round specification [4], the initial randomness m in the ML-KEM.Encaps algorithm was first hashed before being used. Specifically, between lines 1 and 2 in Algorithm 20, there was an additional step that performed the operation $m \leftarrow H(m)$. The purpose of this step was to safeguard against the use of flawed randomness generation processes. As this standard requires the use of NIST-approved randomness generation, this step is unnecessary and is not performed in ML-KEM.

- This specification includes explicit input checking steps that were not part of the third-round specification [4]. For example, ML-KEM.Encaps requires that the byte array containing the encapsulation key correctly decodes to an array of integers modulo q without any modular reductions.

C.2   Changes From FIPS 203 Initial Public Draft

The differences between CRYSTALS-KYBER and ML-KEM as described in Appendix C were included in the initial public draft (ipd) of FIPS 203, which was posted on August 24, 2023. Based on comments submitted on the draft ML-KEM, domain separation was added to K-PKE.KeyGen to prevent the misuse of keys generated to target one security level from being used for a different security level when saving a key as a seed.

Additionally, FIPS 203 ipd had inadvertently swapped the indices of matrix $\hat{\mathbf{A}}$ in K-PKE.KeyGen and K-PKE.Encrypt. This was changed back in the final version of ML-KEM to match CRYSTALS-KYBER.