Withdrawn Draft

Warning Notice

The attached draft document has been withdrawn and is provided solely for historical purposes. It has been followed by the document identified below.

Withdrawal Date	August 13, 2024
Original Release Date	August 24, 2023

The attached draft document is followed by:

Status	Final
Series/Number	NIST FIPS 203
Title	Module-Lattice-Based Key-Encapsulation Mechanism Standard
Publication Date	August 2024
DOI	https://doi.org/10.6028/NIST.FIPS.203
CSRC URL	https://csrc.nist.gov/pubs/fips/203/final
Additional Information	https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY U.S. DEPARTMENT OF COMMERCE

FIPS 203 (Draft)

Federal Information Processing Standards Publication

Module-Lattice-based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
https://doi.org/10.6028/NIST.FIPS.203.ipd

Published August 24, 2023

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

Foreword

The Federal Information Processing Standards Publication (FIPS) 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.

James A. St Pierre, Acting Director
Information Technology Laboratory

Abstract

A key-encapsulation mechanism (or 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 so-called 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 parameter sets are ML-KEM-512, ML-KEM-768, and ML-KEM-1024.

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

Federal Information Processing Standards Publication 203

Published: August 24, 2023

Announcing the

Module-Lattice-based Key-Encapsulation Mechanism Standard

Federal Information Processing Standards Publications (FIPS) are issued 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 (ML-KEM) (FIPS PUB 203).

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

3. **Explanation.** This standard specifies a set of algorithms for applications that require a secret cryptographic key that is shared by two parties who can only communicate over a public channel. A cryptographic key (or simply "key") is represented in a computer as a string of bits. A *shared secret key* is computed jointly by two parties (e.g., Party A and Party B) using a set of rules and parameters. Under certain conditions, these rules and parameters ensure that both parties will produce the same key and that this shared 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.

   While there are many methods for establishing a shared secret key, the particular method described in this specification is a key-encapsulation mechanism (KEM). In a KEM, the computation of the shared secret key begins with Party A generating a *decapsulation key* and an *encapsulation key*. Party A keeps the decapsulation key private and makes the encapsulation key available to Party B. Party B then uses Party A's encapsulation key to generate one copy of a shared secret key along with an associated *ciphertext*. Party B then sends the ciphertext to Party A over the same channel. Finally, Party A uses the ciphertext from Party B along with Party A's private decapsulation key to compute another copy of the shared secret key.

   The security of the particular KEM specified here is related to the computational difficulty of solving certain systems of noisy linear equations, specifically the so-called *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 must be implemented wherever the establishment of a shared secret key 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. Federal agencies may also use alternative methods that NIST has indicated are appropriate for this purpose.

   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. A conforming implementation may replace the given sequence of steps in the top-level algorithms of ML-KEM (i.e., ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps) with any equivalent process. In other words, different procedures that produce the correct output for every input are permitted. In particular, conforming implementations are not required to use the same subroutines (of the aforementioned main algorithms) as are used in this specification.

   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 for cryptographic algorithms are 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 for FIPS 140-3.

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 public-key encryption PQC algorithm 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 NIST 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 ipd** 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 (2023) Module-Lattice-based Key-Encapsulation Mechanism Standard. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 203 ipd. https://doi.org/10.6028/NIST.FIPS.203.ipd

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

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

   (i) under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or

   (ii) without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed 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*, or ML-KEM. A key-encapsulation mechanism (or 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. Current NIST-**approved** key establishment schemes are specified in NIST SP-800-56A, *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm-Based Cryptography* [2], and NIST SP-800-56B, *Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography* [3].

It is well-known that the key establishment schemes specified in NIST SP-800-56A and NIST SP-800-56B are vulnerable to attacks using sufficiently capable quantum computers. ML-KEM is an **approved** alternative that is presently believed to be secure even against adversaries in possession of a 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 Section 1.3.

This standard specifies the algorithms and parameter sets of the ML-KEM scheme. It aims to provide sufficient information for implementing 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 NIST SP 800-227 [1].

This standard specifies three parameter sets for ML-KEM. These parameter sets 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 that are based on integer factorization and discrete logarithms (both over finite fields and elliptic curves). As a result, in 2016, the National Institute of Standards and Technology (NIST) initiated a public process to select quantum-resistant public-key cryptographic algorithms for standardization. A total of 82 candidate algorithms were submitted to NIST for consideration for standardization.

After three rounds of evaluation and analysis, NIST selected the first four algorithms to standardize as a result of the Post-Quantum Cryptography (PQC) Standardization process. These algorithms are intended to protect sensitive U.S. Government information well into the foreseeable future, including after the advent of 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 so-called Module Learning With Errors assumption.

1.3 Differences From the CRYSTALS-KYBER Submission

Below is a list of all scheme differences between CRYSTALS-KYBER (as described in [4]) and the ML-KEM scheme specified in this document. The list consists only of those differences that result in differing input-output behavior of the main algorithms (i.e., KeyGen, Encaps, Decaps) of CRYSTALS-KYBER and ML-KEM. Recall that a conforming implementation need only match the input-output behavior of these three algorithms (see "Implementations" above, and Section 3.3 below). Consequently, the list below 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). The list below also does not include any differences in presentation between this standard and [4].

- In the third-round specification [4], the shared secret key was treated as a variable-length value whose length depends on how this key would be used in the relevant application. In this specification, the length of the shared secret key is fixed to 256 bits. In this specification, this key can be used directly in applications as a symmetric key; alternatively, symmetric keys can be derived from this key, 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 [5, 6]) 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 16, 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 validation 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.

2. Glossary of Terms, Acronyms, and Mathematical Symbols

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.
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.
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 private randomness and the encapsulation key as input 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.
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).
hash function	A function on bit strings in which the length of the output is fixed. **Approved** hash functions relevant to this standard are specified in FIPS 202 [7].
KEM ciphertext	A bit string that is produced by encapsulation and used as an input to decapsulation.
key	A bit string that is used in conjunction with a cryptographic algorithm. Examples applicable to this standard include: 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).

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 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.
party	An individual (person), organization, device, or process. In this specification, there are two parties (Party A and Party B, or Alice and Bob), and they 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 key	The final result of a KEM key establishment process. It is a cryptographic key that can be used for symmetric-key cryptography. It must be kept private, and it 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 Appendix A, Table 4).
security strength	A number associated with the amount of work 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

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.
$\mathsf{BitRev}_7(r)$	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 $\mathsf{BitRev}_7(r) = r_6 + 2r_5 + 4r_4 + \cdots + 64r_0$.
$\hat{f}$	The element of T_q that is equal to the NTT representation of a polynomial $f \in R_q$ (see Section 4.3).
$\mathbb{Q}$	The set of rational numbers.
$\mathbb{Z}_m$	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.
$\mathbb{Z}$	The set of integers.
$\mathbf{v}^T$, $\mathbf{A}^T$	The transpose of a row or column $\mathbf{v}$; also, the transpose of a matrix $\mathbf{A}$.
f_j	The coefficient of X^j of a polynomial $f = f_0 + f_1X + \cdots + f_{255}X^{255} \in R_q$.
$r \bmod m$	The unique integer r' in $\{0,1,\ldots,m-1\}$ such that m divides $r - r'$.

$r \bmod^{\pm} m$	For m even (respectively, odd), this denotes the unique integer r' such that $-m/2 < r' \leq m/2$ (respectively, $-(m-1)/2 \leq r' \leq (m-1)/2$) and m divides $r - r'$.
$\lvert B \rvert$	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.
$\lceil x \rfloor$	The rounding of x to the nearest integer; if $x = y + 1/2$ for some $y \in \mathbb{Z}$, then $\lceil x \rfloor = y + 1$.
$\lfloor x \rfloor$	The floor of x, i.e., the largest integer less than or equal to x.
$\mathbb{B}$	The set $\{0, 1, \ldots, 255\}$ of unsigned 8-bit integers (bytes).
$A \| B$	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 $(B[k], B[k+1], \ldots, B[m-1])$ of the array B.
n	Denotes the integer 256 throughout this document.
q	Denotes the prime integer $3329 = 2^8 \cdot 13 + 1$ throughout this document.
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, equipped with addition and multiplication modulo $X^n + 1$.
$s \leftarrow x$	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 generated using randomness from an **approved** RBG (see Section 3.3).
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).

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 7).

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

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 any integers x, y, the statement

$$z \leftarrow x + y \tag{2.1}$$

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.

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.

Algorithm 1 ForExample

Performs two simple "for" loops.

```
for (i ← 0; i < 10; i++)
    A[i] ← i                          ▷ A is an integer array of length 10
end for                               ▷ A now has the value (0,1,2,3,4,5,6,7,8,9)
j ← 0
for (k ← 256; k > 1; k ← k/2)
    B[j] ← k                          ▷ B is an integer array of length 8
    j ← j + 1
end for                               ▷ B now has the value (256,128,64,32,16,8,4,2)
```

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 $m = q$ and $\ell = n = 256$. Arithmetic with arrays in $\mathbb{Z}_m^\ell$ will be done as follows. Let $a \in \mathbb{Z}_m$ and $X, Y \in \mathbb{Z}_m^\ell$. The statements

$$Z \leftarrow a \cdot X$$
$$W \leftarrow X + Y$$

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 (2.2) below.

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 a different 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 elements of T_q in pseudocode using arrays of integers modulo q, as follows.

An element f of R_q is a polynomial of the form

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

and will be represented in pseudocode by the array

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

whose entries contain the coefficients of f. Abusing notation somewhat, this array 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,

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

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

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

Abusing notation somewhat, this array 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 8) and NTT^{-1} (Algorithm 9). These algorithms act on arrays of coefficients, as described above, and satisfy $\hat{f} = \mathsf{NTT}(f)$ and $f = \mathsf{NTT}^{-1}(\hat{f})$.

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 $\mathbb{Z}_q^{256}$), as described above. The algebraic operations of addition and scalar multiplication are thus performed by addition and scalar multiplication of the corresponding coefficient arrays. 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 10). Specifically, if $\hat{f}, \hat{g} \in \mathbb{Z}_q^{256}$ are a pair of arrays (each representing the NTT of some polynomial), then

$$\hat{h} \leftarrow \hat{f} \times_{T_q} \hat{g} \qquad \text{means} \qquad \hat{h} \leftarrow \text{MultiplyNTTs}(\hat{f}, \hat{g})\,. \tag{2.2}$$

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

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. Specifically, if $\mathbf{A} \in (\mathbb{Z}_q^{256})^{k \times \ell}$, then the statement

$$\hat{\mathbf{A}} \leftarrow \text{NTT}(\mathbf{A})$$

will result in $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times \ell}$ such that $\hat{\mathbf{A}}[i,j] = \text{NTT}(\mathbf{A}[i,j])$ for all i,j. This involves running NTT a total of $k \cdot \ell$ times. Note that the case of vectors corresponds to $\ell = 1$.

Arithmetic with matrices and vectors. The following describes how to perform arithmetic with matrices while continuing to view vectors as a special case of matrices.

Addition and scalar multiplication is performed coordinate-wise. Addition of matrices over R_q and T_q is then straightforward. In the case of T_q, scalar multiplication is done via (2.2). For example, if $\hat{f} \in \mathbb{Z}_q^{256}$ and $\hat{\mathbf{u}}, \hat{\mathbf{v}} \in (\mathbb{Z}_q^{256})^k$, then

$$\begin{aligned} \hat{\mathbf{w}} &\leftarrow \hat{f} \cdot \hat{\mathbf{u}} \\ \hat{\mathbf{z}} &\leftarrow \hat{\mathbf{u}} + \hat{\mathbf{v}} \end{aligned}$$

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. Note that the multiplication and addition of individual entries here is performed using the appropriate arithmetic for coefficient arrays of elements of T_q.

It will also be necessary to multiply matrices with entries in T_q. This is done 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 below. 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 $\mathbb{Z}_q^{256}$. The first two pseudocode statements below produce a new length-k vector whose entries are specified on the right-hand side. The third pseudocode statement computes a dot product; the result is therefore in the base ring (i.e., T_q), and is represented by an element $\hat{z}$ of $\mathbb{Z}_q^{256}$.

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

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

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

The multiplication $\times_{T_q}$ of individual entries above is performed using MultiplyNTTs, as described in (2.2) above.

Applying algorithms to arrays. The conventions of coordinate-wise arithmetic described above will also be extended to algorithms that act on (and/or produce) an atomic data type. When the pseudocode invokes such an algorithm on an array input, it is implied that the algorithm is invoked repeatedly for each entry of the array. For example, the function $\mathsf{Compress}_d : \mathbb{Z}_q \to \mathbb{Z}_{2^d}$ defined in Section 4 can be invoked on an array input $F \in \mathbb{Z}_q^{256}$ with the statement

$$K \leftarrow \mathsf{Compress}_d(F)\,. \tag{2.3}$$

The result will be that $K \in \mathbb{Z}_{2^d}^{256}$ and $K[i] = \mathsf{Compress}_d(F[i])$ for every i. This computation involves running the Compress algorithm 256 times.

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 brief and informal overview of key-encapsulation mechanisms (or KEMs). For more details, see NIST SP 800-227 [1].

A key-encapsulation mechanism (or KEM) is a set of algorithms that can be used, under certain conditions, 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:

- a key generation algorithm denoted by KeyGen;
- an "encapsulation" algorithm denoted by Encaps;
- a "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 numerical values, one for each parameter required by the above algorithms.

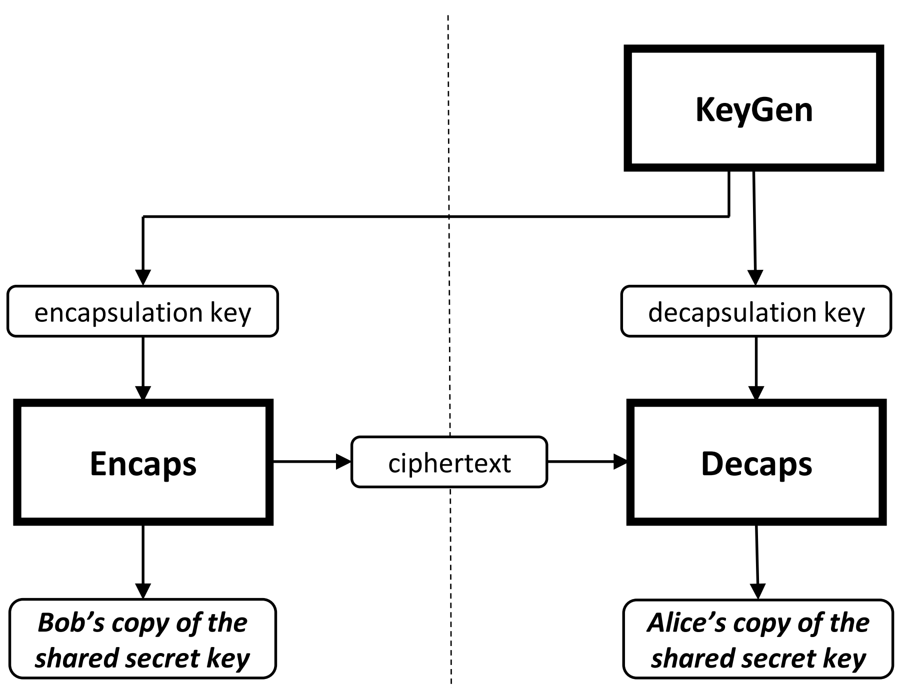

Figure 1. A simple view of key establishment using a KEM

A KEM can be used to establish a shared secret key between two parties (see Figure 1) referred to here as Alice and Bob. 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; this produces Bob's copy K_B 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 step produces Alice's copy K_A of the shared secret key.

After completing the process above, Alice and Bob would like to conclude that their individual outputs satisfy $K_A = K_B$ and that this value is a secure, random, shared secret key. However, these properties only hold under certain important assumptions, as discussed in NIST 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 difficulty of solving the so-called Module Learning with Errors (MLWE) problem [9], 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 lends itself naturally to cryptographic applications. For example, if x is interpreted as a secret key, then one can encrypt a one-bit value by sampling either an approximately correct linear equation (if the bit value is zero) or a far-from-correct linear equation (if the bit value is one). Plausibly, only a party in possession of x can then 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.

At a high level, the MLWE problem poses the same task as LWE but with $\mathbb{Z}_q^n$ replaced with the module R_q^k constructed by taking the k-fold Cartesian product of a certain polynomial ring R_q for some integer $k > 1$. In particular, the secret is now an element $\mathbf{x}$ of the module R_q^k.

The ML-KEM construction. At a high level, the ML-KEM construction proceeds in two steps. First, the idea mentioned above is used to construct a public-key encryption scheme from the MLWE problem. Second, this public-key encryption scheme is converted into a key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [11, 12]. In addition to producing a KEM, the FO transform is also intended to provide security in a significantly more general adversarial attack model. As a result, ML-KEM is believed to satisfy so-called IND-CCA security [1, 4, 13].

The specification of the ML-KEM algorithms in this standard will follow the above pattern. Specifically, this standard will first describe a public-key encryption scheme called K-PKE and then use the algorithms of K-PKE as subroutines when describing the algorithms of ML-KEM. The cryptographic transformation from K-PKE to ML-KEM is crucial for achieving full security.

The scheme K-PKE is not sufficiently 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. Although NTT representations enable fast multiplication, other operations such as rounding and sampling must be applied to standard polynomial representations.

ML-KEM satisfies the key properties of KEM correctness, and a proof of asymptotic theoretical security (in a certain heuristic model) is known [4]. Each of the parameter sets of ML-KEM comes with an associated security strength, which was estimated based on current cryptanalysis (see Section 7 for details).

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

- ML-KEM.KeyGen (Algorithm 15);
- ML-KEM.Encaps (Algorithm 16);
- ML-KEM.Decaps (Algorithm 17).

These algorithms are described and discussed in detail in Section 6.

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 7; the security categories 1-5 are defined in Appendix A. 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 7. In addition to these five variable parameters, there are also two constants: $n = 256$ and $q = 3329$.

Decapsulation failures. Provided all inputs are well-formed, the key establishment procedure of ML-KEM will never explicitly fail. Specifically, the ML-KEM.Encaps and ML-KEM.Decaps algorithms will always output a value with the same data type as a shared secret key, and will never output an error or failure symbol. However, it is possible (though extremely unlikely) that the process will fail in the sense that Alice (via ML-KEM.Decaps) and Bob (via ML-KEM.Encaps) will produce different outputs, even though both of them are behaving honestly and no adversarial interference is present. In this case, Alice and Bob clearly did not succeed in producing a shared secret key. This event is called a decapsulation failure. The decapsulation failure probability is defined to be the probability that the process

1. $(\mathsf{ek}, \mathsf{dk}) \leftarrow$ ML-KEM.KeyGen()

2. $(c, K) \leftarrow$ ML-KEM.Encaps(ek)

3. $K' \leftarrow$ ML-KEM.Decaps(c, dk)

results in $K \neq K'$ (i.e., the encapsulated key is different from the decapsulated key). Estimates for the decapsulation failure probability (or rate) for each of the ML-KEM parameter sets are given in Table 1 (see [4]).

Table 1. Decapsulation failure rates for ML-KEM

Parameter set	Decapsulation failure rate
ML-KEM-512	2^{-139}
ML-KEM-768	2^{-164}
ML-KEM-1024	2^{-174}

A note on 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, and K-PKE 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 can sometimes be referred to as "private keys." In order to reduce confusion, this standard will not use the terms "public key" and "private key." Instead, keys will be referred to using the more specific terms above (i.e., encapsulation key, decapsulation key, encryption key, decryption key, or shared secret key).

3.3 Requirements for ML-KEM Implementations

This section describes several requirements for implementing the algorithms of ML-KEM. Requirements for using ML-KEM in specific applications are given in NIST SP 800-227 [1].

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 12), K-PKE.Encrypt (Algorithm 13), and K-PKE.Decrypt (Algorithm 14) are **not approved** for use as a public-key encryption scheme.

Equivalent implementations. Each of the three top-level algorithms (i.e., ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps) defines a particular mathematical operation, mapping any given input to a corresponding output. For example, the operation defined by the algorithm ML-KEM.Encaps takes one byte array as input and produces two byte arrays as output.

In this standard, the three operations defined by ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps are described using particular sequences of computational steps. A conforming implementation can replace each of these sequences with a different sequence of steps, provided that the resulting operation is an equivalent process to the one specified in this standard.

For example, a conforming implementation of the encapsulation operation must have the property that, for any parameter set and any input byte array ek, the distribution of output byte arrays is identical to the distribution ML-KEM.Encaps(ek) as specified in this standard.

Approved usage of the shared secret key. The output of the encapsulation and decapsulation algorithms of ML-KEM is always a 256-bit value. Under appropriate conditions (see above; see also NIST SP 800-227 [1]), this output is a shared secret key K. This shared secret key K can be used directly as a key for symmetric cryptography. When key derivation is needed, the final symmetric key(s) **shall** be derived from this 256-bit shared secret key K in an **approved** manner, as specified in NIST SP 800-108 [14].

Randomness generation. Three algorithms in this standard require the generation of randomness: K-PKE.KeyGen, ML-KEM.KeyGen, and ML-KEM.Encaps. In pseudocode, the step in which this randomness is generated is denoted by a pseudocode 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 NIST SP 800-90A, NIST SP 800-90B, and NIST SP 800-90C [15, 16, 17]. Moreover, the RBG used **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 validation. The algorithms ML-KEM.Encaps and ML-KEM.Decaps require input validation. Implementers **shall** ensure that ML-KEM.Encaps and ML-KEM.Decaps are only executed on validated inputs, as described in Section 6. As discussed above, implementers can choose to implement the top-level algorithms (i.e., ML-KEM.Encaps, ML-KEM.Decaps, or ML-KEM.KeyGen) using any equivalent process; the validation of inputs is considered part of this process. A conforming implementation **shall** be equivalent to first validating the input, and then running the appropriate algorithm.

Destruction of intermediate values. Data used internally by KEM algorithms in intermediate computation steps could be used by an adversary to compromise security. Implementers **shall**, therefore, ensure that such intermediate data is destroyed as soon as it is no longer needed.

No floating-point arithmetic. Implementations of ML-KEM **should not** use floating-point arithmetic. All division and rounding steps in the algorithms of ML-KEM can be performed within the set of rational numbers.

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 [7]. They will be used as follows.

SHA3-256 and SHA3-512 are hash functions with variable-length input and 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.

SHAKE128 and SHAKE256 are XOFs with variable-length input and variable-length output. Invocations of these functions on an input M will be denoted in two different ways, depending on whether the desired output length ℓ (in bytes) is known at invocation time. If ℓ is known at invocation time, the invocation will be denoted by SHAKE128(M,ℓ) or SHAKE256(M,ℓ). For SHAKE128, the output length will sometimes not be known at invocation time; in those cases, the invocation will be denoted by SHAKE128(M) and the hashing routine will behave like a byte stream that provides pseudorandom bytes (by performing additional "squeezing" rounds [7]) until no more bytes are needed.

The above functions will play several different roles in the algorithms specified in this standard. It will be convenient to assign a specific notation to each of these roles, as follows.

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 PRF : $\{2,3\} \times \mathbb{B}^{32} \times \mathbb{B} \to \mathbb{B}^{64\eta}$, and it **shall** be instantiated as

$$\mathsf{PRF}_\eta(s,b) := \mathrm{SHAKE256}(s\|b, 64\cdot\eta)\,, \tag{4.1}$$

where $\eta \in \{2,3\}$, $s \in \mathbb{B}^{32}$, and $b \in \mathbb{B}$. Here, η is only used to specify the desired output length and not to perform domain separation. Note that the output length parameter for SHAKE256 is specified in bytes.

eXtendable-output function (XOF). The function XOF takes one 32-byte input and two 1-byte inputs. It produces a variable-length output. This function will be denoted by XOF : $\mathbb{B}^{32} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}^*$, and it **shall** be instantiated as

$$\mathsf{XOF}(\rho,i,j) := \mathrm{SHAKE128}(\rho\|i\|j)\,, \tag{4.2}$$

where $\rho \in \mathbb{B}^{32}$, $i \in \mathbb{B}$, and $j \in \mathbb{B}$. The function XOF will only be invoked to provide a stream of pseudorandom bytes for the sampling algorithm SampleNTT (Algorithm 6). As SampleNTT performs rejection sampling, the total number of needed bytes will not be known at the time that XOF is invoked.

Three hash functions. The specification will also make use of three hash function instantiations H, J, and G, as follows.

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

$$H(s) := \text{SHA3-256}(s) \qquad \text{and} \qquad J(s) := \text{SHAKE256}(s, 32) \tag{4.3}$$

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

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

$$G(c) := \text{SHA3-512}(c)\,. \tag{4.4}$$

4.2 General Algorithms

This section specifies a number of algorithms that will be used as subroutines in the main ML-KEM algorithms. For a discussion of how to interpret the pseudocode of these algorithms, see Section 2.4.

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 compression operation for integers modulo q, as well as the corresponding decompression operation.

Converting between bits and bytes. Algorithms 2 and 3 convert between bit arrays and byte arrays. The inputs to BitsToBytes and the outputs of BytesToBits are bit arrays, with each segment of 8 bits representing a byte in little-endian order.

Algorithm 2 BitsToBytes(b)

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

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

$B \leftarrow (0, \ldots, 0)$
for ($i \leftarrow 0$; $i < 8\ell$; i++)
  $B[\lfloor i/8 \rfloor] \leftarrow B[\lfloor i/8 \rfloor] + b[i] \cdot 2^{i \bmod 8}$
end for
return B

Algorithm 3 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 $b \in \{0,1\}^{8 \cdot \ell}$.

for ($i \leftarrow 0$; $i < \ell$; i++)
  for ($j \leftarrow 0$; $j < 8$; j++)
    $b[8i+j] \leftarrow B[i] \bmod 2$
    $B[i] \leftarrow \lfloor B[i]/2 \rfloor$
  end for
end for
return b

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

$$\begin{aligned} \mathsf{Compress}_d : \mathbb{Z}_q &\longrightarrow \mathbb{Z}_{2^d} \\ x &\longmapsto \lceil (2^d/q) \cdot x \rfloor . \end{aligned} \tag{4.5}$$

$$\begin{aligned} \mathsf{Decompress}_d : \mathbb{Z}_{2^d} &\longrightarrow \mathbb{Z}_q \\ y &\longmapsto \lceil (q/2^d) \cdot y \rfloor . \end{aligned} \tag{4.6}$$

Note that the input and output types of these functions are integers modulo m (see discussion of types in Section 2.4). Division and rounding in the computation of the above functions are performed in the set of rational numbers. Floating-point computations **should not** be used.

Informally, Compress discards low-order bits of the input, and Decompress adds low-order bits set to zero. These algorithms satisfy two important properties. First, decompression followed by compression preserves the input, that is, $\mathsf{Compress}_d(\mathsf{Decompress}_d(y)) = y$ for all $y \in \mathbb{Z}_q$ and all $d < 12$. Second, if d is large (i.e., close to 12) — meaning that the number of discarded bits is small — compression followed by decompression does not significantly alter the value. Specifically,

$$[\mathsf{Decompress}_d(\mathsf{Compress}_d(x)) - x] \bmod^{\pm} q \ \leq \lceil q/2^{d+1} \rfloor \tag{4.7}$$

for all $x \in \mathbb{Z}_q$ and all $d < 12$.

Encoding and decoding. The algorithms ByteEncode (Algorithm 4) and ByteDecode (Algorithm 5) will be used for serialization and deserialization of arrays of integers modulo m. All serialized arrays will be of length $n = 256$. $\mathsf{ByteEncode}_d$ serializes an array of d-bit integers into an array of $32 \cdot d$ bytes. $\mathsf{ByteDecode}_d$ performs the corresponding deserialization operation, converting an array of $32 \cdot d$ bytes into an array of d-bit integers.

For the following discussion, 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 $1 \le d \le 12$. Bit arrays are divided into d-bit segments. In the case where $1 \le d \le 11$, $\mathsf{ByteDecode}_d$ converts each d-bit segment of the input into one integer modulo 2^d, and $\mathsf{ByteEncode}_d$ performs the inverse operation. In this case, the conversion is one-to-one.

The case $d = 12$ is treated differently. In this case, $\mathsf{ByteEncode}_{12}$ receives integers modulo q as input, and $\mathsf{ByteDecode}_{12}$ produces integers modulo q as output. $\mathsf{ByteDecode}_{12}$ converts each 12-bit segment of the 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 $q = 3329$ but less than 4096; however, this cannot occur for arrays produced by $\mathsf{ByteEncode}_{12}$. These aspects of $\mathsf{ByteDecode}_{12}$ and $\mathsf{ByteEncode}_{12}$ will be important when considering validation of the ML-KEM encapsulation key in Section 6.

Algorithm 4 $\mathsf{ByteEncode}_d(F)$

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

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 $B \in \mathbb{B}^{32d}$.

for $(i \leftarrow 0;\ i < 256;\ i{+}{+})$
  $a \leftarrow F[i]$ ▷ $a \in \mathbb{Z}_{2^d}$
  for $(j \leftarrow 0;\ j < d;\ j{+}{+})$
    $b[i \cdot d + j] \leftarrow a \bmod 2$ ▷ $b \in \{0,1\}^{256 \cdot d}$
    $a \leftarrow (a - b[i \cdot d + j])/2$ ▷ note $a - b[i \cdot d + j]$ is always even.
  end for
end for
$B \leftarrow \mathsf{BitsToBytes}(b)$
return B

Algorithm 5 $\mathsf{ByteDecode}_d(B)$

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

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 $d = 12$.

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

4.2.2 Sampling Algorithms

The algorithms of ML-KEM require two sampling subroutines that are specified in Algorithms 6 and 7. 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 6) converts a stream of bytes into a polynomial in the NTT domain. If the input stream consists of uniformly random bytes, then the result will be drawn uniformly at random from T_q. The output is an array in $\mathbb{Z}_q^{256}$ that contains the coefficients of the sampled element of T_q (see Section 2.4).

Algorithm 6 SampleNTT(B)

If the input is a stream of uniformly random bytes, the output is a uniformly random element of T_q.

Input: byte stream $B \in \mathbb{B}^*$.
Output: array $\hat{a} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of a polynomial

$i \leftarrow 0$
$j \leftarrow 0$
while $j < 256$ **do**
    $d_1 \leftarrow B[i] + 256 \cdot (B[i+1] \bmod 16)$
    $d_2 \leftarrow \lfloor B[i+1]/16 \rfloor + 16 \cdot B[i+2]$
    if $d_1 < q$ **then**
        $\hat{a}[j] \leftarrow d_1$ ▷ $\hat{a} \in \mathbb{Z}_q^{256}$
        $j \leftarrow j+1$
    end if
    if $d_2 < q$ and $j < 256$ **then**
        $\hat{a}[j] \leftarrow d_2$
        $j \leftarrow j+1$
    end if
    $i \leftarrow i+3$
end while
return $\hat{a}$

Algorithm 7 SamplePolyCBD$_\eta(B)$

If the input is a stream of uniformly random bytes, outputs a 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

$b \leftarrow$ BytesToBits(B)
for $(i \leftarrow 0;\ i < 256;\ i\text{++})$
    $x \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + j]$
    $y \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + \eta + j]$
    $f[i] \leftarrow x - y \bmod q$ ▷ $f \in \mathbb{Z}_q^{256}$
end for
return f

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 will sometimes be 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 7) samples the coefficient array of a polynomial $f \in R_q$ according to the distribution $\mathcal{D}_\eta(R_q)$, provided that its input is a stream of uniformly random bytes.

4.3 The Number-Theoretic Transform

The number-theoretic transform (or 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 $\mathbb{Z}_q[X]/(X^n+1)$ consisting of polynomials of the form $f = f_0 + f_1X + \cdots + f_{255}X^{255}$ where $f_j \in \mathbb{Z}_q$ for all j, equipped with arithmetic modulo X^n+1.

The ring R_q is naturally isomorphic to another ring, denoted 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. On input a polynomial $f \in R_q$, 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

$$f \times_{R_q} g = \mathsf{NTT}^{-1}(\hat{f} \times_{T_q} \hat{g}), \tag{4.8}$$

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, each consisting of degree-one polynomials, 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). Both NTT and NTT^{-1} can be computed in-place. In fact, Algorithms 8 and 9 demonstrate an efficient means of computing NTT and NTT^{-1} in-place. However, for clarity in understanding the distinction of the algebraic objects before and after the conversion, the algorithms are written with explicit inputs and outputs.

The mathematical structure of a simple NTT. Recall that, 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:

$$X^{256}+1 = \prod_{k=0}^{127}\left(X^2 - \zeta^{2\mathsf{BitRev}_7(k)+1}\right). \tag{4.9}$$

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

$$T_q := \bigoplus_{k=0}^{127} \mathbb{Z}_q[X]/\left(X^2 - \zeta^{2\mathsf{BitRev}_7(k)+1}\right). \tag{4.10}$$

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

$$\hat{f} := \left(f \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(0)+1}), \ldots, f \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(127)+1})\right). \tag{4.11}$$

In the algorithms below, $\hat{f}$ is stored as an array of 256 integers modulo q. Specifically,

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

for i from 0 to 127.

Algorithm 8 $\mathsf{NTT}(f)$

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: f̂ ← f                                   ▷ will compute NTT in-place on a copy of input array
2: k ← 1
3: for (len ← 128; len ≥ 2; len ← len/2)
4:     for (start ← 0; start < 256; start ← start + 2·len)
5:         zeta ← ζ^{BitRev7(k)} mod q
6:         k ← k + 1
7:         for (j ← start; j < start + len; j++)
8:             t ← zeta · f̂[j + len]        ▷ steps 8-10 done modulo q
9:             f̂[j + len] ← f̂[j] − t
10:            f̂[j] ← f̂[j] + t
11:        end for
12:    end for
13: end for
14: return f̂
```

The ML-KEM NTT algorithms. An algorithm for the NTT is described in Algorithm 8. An algorithm for the Inverse-NTT is described in Algorithm 9. These two algorithms are overloaded in this standard. First, they represent the transformation used to map elements of R_q to elements of T_q (using NTT) and vice versa (using NTT^{-1}). Second, they 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}).

Algorithm 9 $\mathsf{NTT}^{-1}(\hat{f})$

Computes the polynomial $f \in R_q$ corresponding 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: $k \leftarrow 127$
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:         $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(k)} \bmod q$
6:         $k \leftarrow k - 1$
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

As discussed in Section 2.4, addition and scalar multiplication of elements of T_q is straightforward: it can be done using the corresponding coordinate-wise arithmetic operations on the coefficient arrays. This section describes how to do the remaining ring operation (i.e., multiplication in T_q).

Recall from (4.11) 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

$$\hat{h}[2i] + \hat{h}[2i+1]X = (\hat{f}[2i] + \hat{f}[2i+1]X)(\hat{g}[2i] + \hat{g}[2i+1]X) \bmod (X^2 - \zeta^{2\mathsf{BitRev}_7(i)+1}). \tag{4.12}$$

Thus, one can compute the product of two elements of T_q using the algorithm MultiplyNTTs (Algorithm 10). Note that MultiplyNTTs uses BaseCaseMultiply (Algorithm 11) as a subroutine. As discussed in Section 2.4, MultiplyNTTs enables one to perform linear-algebraic arithmetic operations with matrices and vectors with entries in T_q.

Algorithm 10 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

```
for (i ← 0; i < 128; i++)
    (ĥ[2i], ĥ[2i+1]) ← BaseCaseMultiply(f̂[2i], f̂[2i+1], ĝ[2i], ĝ[2i+1], ζ^(2BitRev7(i)+1))
end for
return ĥ
```

$$(\hat{h}[2i], \hat{h}[2i+1]) \leftarrow \mathsf{BaseCaseMultiply}(\hat{f}[2i], \hat{f}[2i+1], \hat{g}[2i], \hat{g}[2i+1], \zeta^{2\mathsf{BitRev}_7(i)+1})$$

Algorithm 11 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 $a_0 + a_1X$ and $b_0 + b_1X$
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 6.

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. The individual parameters and the parameter sets are described in Section 7.

The algorithms in this section do not perform any input validation. This is because they are only invoked as subroutines of the main ML-KEM algorithms. The algorithms of ML-KEM do perform input validation as needed; they also ensure that all inputs to K-PKE algorithms (invoked as subroutines) will be valid.

Each of the algorithms of K-PKE below 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 (see, e.g., line 19 of K-PKE.KeyGen) is performed as described in Sections 2.4 and 4.3.1. The encryption and decryption key 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 12) takes no input, requires randomness, and outputs an encryption key ek_{PKE} and a decryption key dk_{PKE}. From the typical point of view of public-key encryption, the encryption key can be made public, while the decryption key and the randomness must remain private. This will be the case in the context of this standard as well. 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 randomness of K-PKE.KeyGen must remain private as they can be used to perform decapsulation in ML-KEM.

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 $(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e})$ 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 the seed stored in the encryption key. The secret $\mathbf{s}$ and the "noise" $\mathbf{e}$ are sampled from the

Algorithm 12 K-PKE.KeyGen()

Generates an encryption key and a corresponding decryption key.

Output: encryption key $\mathsf{ek}_{\mathrm{PKE}} \in \mathbb{B}^{384k+32}$.
Output: decryption key $\mathsf{dk}_{\mathrm{PKE}} \in \mathbb{B}^{384k}$.

$d \xleftarrow{\$} \mathbb{B}^{32}$ ▷ d is 32 random bytes (see Section 3.3)
$(\rho, \sigma) \leftarrow G(d)$ ▷ expand to two pseudorandom 32-byte seeds
$N \leftarrow 0$
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
  for $(j \leftarrow 0;\ j < k;\ j{+}{+})$
    $\hat{\mathbf{A}}[i, j] \leftarrow \mathsf{SampleNTT}(\mathsf{XOF}(\rho, i, j))$ ▷ each entry of $\hat{\mathbf{A}}$ uniform in NTT domain
  end for
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{s}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$ ▷ $\mathbf{s}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$ ▷ $\mathbf{e}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$ ▷ NTT is run k times (once for each coordinate of $\mathbf{s}$)
$\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$ ▷ NTT is run k times
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ noisy linear system in NTT domain
$\mathsf{ek}_{\mathrm{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{t}}) \| \rho$ ▷ $\mathsf{ByteEncode}_{12}$ is run k times; include seed for $\hat{\mathbf{A}}$
$\mathsf{dk}_{\mathrm{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\mathbf{s}})$ ▷ $\mathsf{ByteEncode}_{12}$ is run k times
return $(\mathsf{ek}_{\mathrm{PKE}}, \mathsf{dk}_{\mathrm{PKE}})$

centered binomial distribution using randomness expanded from a seed via PRF. Once $\mathbf{A}$ and $\mathbf{s}$ and $\mathbf{e}$ are generated, the computation of the final part $\mathbf{t} = \mathbf{As} + \mathbf{e}$ of the encryption key takes place.

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 13) takes an encryption key $\mathsf{ek}_{\mathrm{PKE}}$ and a plaintext m as input, requires randomness r, and outputs a ciphertext c. While many algorithms specified in this document require randomness, only the description of K-PKE.Encrypt interprets this randomness as part of the input. This is because ML-KEM will need to invoke K-PKE.Encrypt with a specific choice of randomness (see Algorithm 16 for details).

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.

Algorithm 13 K-PKE.Encrypt$(\mathsf{ek}_{\mathsf{PKE}}, m, r)$

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

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

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[0 : 384k])$
$\rho \leftarrow \mathsf{ek}_{\mathsf{PKE}}[384k : 384k + 32]$ ▷ extract 32-byte seed from $\mathsf{ek}_{\mathsf{PKE}}$
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ re-generate matrix $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
  for $(j \leftarrow 0;\ j < k;\ j{+}{+})$
    $\hat{\mathbf{A}}[i, j] \leftarrow \mathsf{SampleNTT}(\mathsf{XOF}(\rho, i, j))$
  end for
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{r} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{r}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$ ▷ $\mathbf{r}[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
for $(i \leftarrow 0;\ i < k;\ i{+}{+})$ ▷ generate $\mathbf{e_1} \in (\mathbb{Z}_q^{256})^k$
  $\mathbf{e}_1[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ $\mathbf{e}_1[i] \in \mathbb{Z}_q^{256}$ sampled from CBD
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$ ▷ sample $e_2 \in \mathbb{Z}_q^{256}$ from CBD
$\hat{\mathbf{r}} \leftarrow \mathsf{NTT}(\mathbf{r})$ ▷ NTT is run k times
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^{\intercal} \circ \hat{\mathbf{r}}) + \mathbf{e_1}$ ▷ NTT^{-1} is run k times
$\mu \leftarrow \mathsf{Decompress}_1(\mathsf{ByteDecode}_1(m)))$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^{\intercal} \circ \hat{\mathbf{r}}) + e_2 + \mu$ ▷ encode plaintext m into polynomial v.
$c_1 \leftarrow \mathsf{ByteEncode}_{d_u}(\mathsf{Compress}_{d_u}(\mathbf{u}))$ ▷ $\mathsf{ByteEncode}_{d_u}$ is run k times
$c_2 \leftarrow \mathsf{ByteEncode}_{d_v}(\mathsf{Compress}_{d_v}(v))$
return $c \leftarrow (c_1 \| c_2)$

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 which 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 (corresponding to the decrypted bit value of 0) or a far-from-correct equation (corresponding 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 $\mathbf{r} \in R_q^k$ 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) computed by $(\mathbf{u}, v) \leftarrow (\mathbf{A}^\intercal \mathbf{r} + \mathbf{e}_1, \mathbf{t}^\intercal \mathbf{r} + e_2)$. An appropriate encoding μ of the input message m is then added to the term $\mathbf{t}^\intercal \mathbf{r} + e_2$. Finally, the 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 14) takes a decryption key $\mathsf{dk}_{\mathrm{PKE}}$ and a ciphertext c as input, requires no randomness, and outputs a plaintext m.

Informal description. The algorithm K-PKE.Decrypt begins by computing the "noisy equation" $(\mathbf{u}, v)$ underlying the ciphertext c, as discussed in 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. Recall that the decryption key $\mathsf{dk}_{\mathrm{PKE}}$ contains the vector of secret variables $\mathbf{s}$. The decryption algorithm can thus use the decryption key to compute the true constant term $v' = \mathbf{s}^\intercal \mathbf{u}$ and then calculate $v - v'$. The decryption algorithm ends by decoding the plaintext message m from $v - v'$ and outputting m.

Algorithm 14 K-PKE.Decrypt($\mathsf{dk}_{\mathrm{PKE}}, c$)

Uses the decryption key to decrypt a ciphertext.

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

```
c_1 ← c[0 : 32 d_u k]
c_2 ← c[32 d_u k : 32(d_u k + d_v)]
u ← Decompress_{d_u}(ByteDecode_{d_u}(c_1))              ▷ ByteDecode_{d_u} invoked k times
v ← Decompress_{d_v}(ByteDecode_{d_v}(c_2))
ŝ ← ByteDecode_12(dk_PKE)
w ← v − NTT^{−1}(ŝ^T ∘ NTT(u))                           ▷ NTT^{−1} and NTT invoked k times
m ← ByteEncode_1(Compress_1(w))                          ▷ decode plaintext m from polynomial v
return m
```

6. The ML-KEM Key-Encapsulation Mechanism

The ML-KEM scheme consists of three algorithms:

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 of which 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 7. 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. In addition, the algorithms ML-KEM.Encaps and ML-KEM.Decaps require validation of inputs, as discussed below.

6.1 ML-KEM Key Generation

The key generation algorithm ML-KEM.KeyGen for ML-KEM (Algorithm 15) accepts no input, requires randomness, and produces an encapsulation key and a decapsulation key. While the encapsulation key can be made public, the decapsulation key must remain private.

Informal description. The core subroutine of ML-KEM.KeyGen is the key generation algorithm of K-PKE (Algorithm 12). The ML-KEM encapsulation key is simply the encryption key of K-PKE. The ML-KEM decapsulation key is comprised of the decryption key of K-PKE, the encapsulation key, a hash of the encapsulation key, and a pseudorandom 32-byte value. This random value will be used in the "implicit rejection" mechanism of the decapsulation algorithm (Algorithm 17).

Algorithm 15 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.

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

```
z ←$ B^32                                    ▷ z is 32 random bytes (see Section 3.3)
(ek_PKE, dk_PKE) ← K-PKE.KeyGen()            ▷ run key generation for K-PKE
ek ← ek_PKE                                  ▷ KEM encaps key is just the PKE encryption key
dk ← (dk_PKE || ek || H(ek) || z)            ▷ KEM decaps key includes PKE decryption key
return (ek, dk)
```

6.2 ML-KEM Encapsulation

The encapsulation algorithm ML-KEM.Encaps of ML-KEM (Algorithm 16) accepts an encapsulation key as input, requires randomness, and outputs a ciphertext and a shared key.

Input validation. To validate a given input[1] $\widetilde{\mathsf{ek}}$ to ML-KEM.Encaps, perform the following checks.

1. *(Type check.)* If $\widetilde{\mathsf{ek}}$ is not an array of bytes of length $384k + 32$ for the value of k specified by the relevant parameter set, the input is invalid.
2. *(Modulus check.)* Perform the computation $\mathsf{ek} \leftarrow \mathsf{ByteEncode}_{12}(\mathsf{ByteDecode}_{12}(\widetilde{\mathsf{ek}}))$. If $\mathsf{ek} \neq \widetilde{\mathsf{ek}}$, the input is invalid. (See Section 4.2.1.)

If either of the above checks declare the input to be invalid, then ML-KEM.Encaps **shall not** be performed with input $\widetilde{\mathsf{ek}}$. Instead, application-appropriate steps **shall** be taken to abort. If both of the above checks pass (i.e., none of them declare the input to be invalid), then the input is considered valid and ML-KEM.Encaps can be performed with input $\mathsf{ek} = \widetilde{\mathsf{ek}}$.

It is important to note that the above input validation process does not ensure that $\widetilde{\mathsf{ek}}$ is an actual output of ML-KEM.KeyGen. In fact, the ability to ensure that (without using the decapsulation key) would violate the security assumption.

Recall that, as discussed in Section 3.3, implementations are only required to correctly reproduce the input-output behavior of the top-level algorithms. In the case of ML-KEM.Encaps, this means that an implementation can perform any process that is equivalent to executing checks 1 and 2 above and then running Algorithm 16. (For example, the second check could be performed during the execution of $\mathsf{ByteDecode}_{12}$ in line 2 of K-PKE.Encrypt.)

Algorithm 16 ML-KEM.Encaps(ek)

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

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

$m \xleftarrow{\$} \mathbb{B}^{32}$ ▷ m is 32 random bytes (see Section 3.3)
$(K, r) \leftarrow G(m \| H(\mathsf{ek}))$ ▷ derive shared secret key K and randomness r
$c \leftarrow \mathsf{K\text{-}PKE.Encrypt}(\mathsf{ek}, m, r)$ ▷ encrypt m using K-PKE with randomness r
return (K, c)

Informal description. The core subroutine of ML-KEM.Encaps 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 K and the randomness used during encryption are derived from m and the encapsulation

[1] In discussions of input validation, the tilde in the notation indicates that the input might not be properly formed, e.g., $\widetilde{\mathsf{ek}}$ for a candidate encapsulation key input, as opposed to ek for a valid input.

key ek via hashing. Specifically, H is applied to ek, and the result is concatenated with m and then hashed using G. The algorithm completes by outputting the ciphertext c and the shared secret K.

6.3 ML-KEM Decapsulation

The decapsulation algorithm ML-KEM.Decaps of ML-KEM (Algorithm 16) accepts a decapsulation key and a ML-KEM ciphertext as input, does not use any randomness, and outputs a shared secret.

Input validation. To validate a given pair of inputs $\widetilde{c}$ (candidate ciphertext) and $\widetilde{\mathsf{dk}}$ (candidate decapsulation key) to ML-KEM.Decaps, perform the following checks.

1. *(Ciphertext type check.)* If $\widetilde{c}$ 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, the input is invalid.

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

If either of the above checks declares the input to be invalid, then ML-KEM.Decaps **shall not** be performed with input $(\widetilde{c}, \widetilde{\mathsf{dk}})$. Instead, application-appropriate steps **shall** be taken to abort. If both of the checks pass (i.e., neither one declares the input to be invalid), then the input is considered valid and ML-KEM.Decaps can be performed with input $(c, \mathsf{dk}) = (\widetilde{c}, \widetilde{\mathsf{dk}})$.

For some applications, further validation of the decapsulation key $\widetilde{\mathsf{dk}}$ may be appropriate. For instance, in cases where $\widetilde{\mathsf{dk}}$ was generated by a third party, users may want to ensure that the four components of $\widetilde{\mathsf{dk}}$ have the correct relationship with each other, as in line 4 of ML-KEM.KeyGen. In all cases, implementers **shall** validate the inputs to ML-KEM.Decaps in a manner that is appropriate for their application.

Informal description. The algorithm ML-KEM.Decaps 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 on decapsulation failures in Section 3.2). In either case, decapsulation outputs the resulting shared secret key K'.

Algorithm 17 ML-KEM.Decaps(c, dk)

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

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

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

7. Parameter Sets

ML-KEM is equipped with three parameter sets. Each of the three parameter sets is comprised of 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 hence in ML-KEM). See Section 5 for details.

- The parameter k determines the dimensions of the vectors $\mathbf{s}$ and $\mathbf{e}$ in K-PKE.KeyGen, as well as the dimensions of the matrix $\hat{\mathbf{A}}$ and the vectors $\mathbf{r}$, $\mathbf{e}_1$, and $\mathbf{e}_2$ in K-PKE.Encrypt.
- The parameter η_1 is required for specifying the distribution for generating the vectors $\mathbf{s}$ and $\mathbf{e}$ in K-PKE.KeyGen and the vector $\mathbf{r}$ in K-PKE.Encrypt.
- The parameter η_2 is required for specifying 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.

	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 2. Approved parameter sets for 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

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

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, 18]. These security strength categories are explained further in Appendix A.

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 these 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 [19].

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).

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 where even higher security is required, other parameter sets may be used.

References

[1] NIST. Special Publication 800-227: Recommendations for key-encapsulation mechanisms, 2024.

[2] Elaine B. Barker, Lily Chen, Allen L. Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. Technical Report Special Publication 800-56A Revision 3, U.S. Department of Commerce, Washington, D.C., April 2018.

[3] Elaine B. Barker, Lily Chen, Allen L. Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. Recommendation for pair-wise key-establishment using integer factorization cryptography. Technical Report Special Publication 800-56B Revision 2, U.S. Department of Commerce, Washington, D.C., March 2019.

[4] Robert Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation. Third-round submission to the NIST's post-quantum cryptography standardization process, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[5] CRYSTALS-Kyber submission team. "Discussion about Kyber's tweaked FO transform". Forum post on pqc-forum, available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/WFRDl8DqYQ4, 2023.

[6] CRYSTALS-Kyber submission team. "Kyber decisions, part 2: FO transform". Forum post on pqc-forum, available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/C0D3W1KoINY/m/99kIvydoAwAJ, 2023.

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

[8] Joppe Bos, Léo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, 2018.

[9] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[10] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

[11] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26:80–101, 2013.

[12] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing.

[13] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, third edition, 2020.

[14] Lily Chen. Recommendation for key derivation using pseudorandom functions. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-108 Rev. 1, August 2022. https://doi.org/10.6028/NIST.SP.800-108r1.

[15] Elaine B. Barker and John M. Kelsey. 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, June 2015. https://doi.org/10.6028/NIST.SP.800-90Ar1.

[16] Meltem Sönmez Turan, Elaine B. Barker, John M. Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. Recommendation for the entropy sources used for random bit generation. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B, January 2018. https://doi.org/10.6028/NIST.SP.800-90B.

[17] Elaine B. Barker, John M. Kelsey, Kerry McKay, Allen Roginsky, and Meltem Sönmez Turan. Recommendation for random bit generator (RBG) constructions. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90C (Third Public Draft), September 2022. https://csrc.nist.gov/publications/detail/sp/800-90c/draft.

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

[19] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. Technical Report NIST Interagency or Internal Report (IR) 8413, National Institute of Standards and Technology, Gaithersburg, MD, July 2022.

[20] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. Implementing Grover oracles for quantum key search on AES and LowMC. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 280–310, Cham, 2020. Springer International Publishing.

[21] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

Appendix A — Security Strength Categories

NIST understands that there are significant uncertainties in estimating the security strengths of post-quantum cryptosystems. These uncertainties come from two sources: first, the possibility that new quantum algorithms will be discovered, leading to new cryptanalytic attacks; and second, our limited ability to predict the performance characteristics of future quantum computers, such as their cost, speed, and memory size.

In order to address these uncertainties, NIST proposed the following approach in its original Call for Proposals [18]. Instead of defining the strength of an algorithm using precise estimates of the number of "bits of security," NIST defined a collection of broad security strength categories. Each category is defined by a comparatively easy-to-analyze reference primitive, whose security will serve as a floor for a wide variety of metrics that NIST deems potentially relevant to practical security. A given cryptosystem may be instantiated using different parameter sets in order to fit into different categories. The goals of this classification are:

- To facilitate meaningful performance comparisons between various post-quantum algorithms by ensuring – insofar as possible – that the parameter sets being compared provide comparable security
- To allow NIST to make prudent future decisions regarding when to transition to longer keys
- To help submitters make consistent and sensible choices regarding what symmetric primitives to use in padding mechanisms or other components of their schemes that require symmetric cryptography
- To better understand the security/performance trade-offs involved in a given design approach

In accordance with the second and third goals above, NIST based its classification on the range of security strengths offered by the existing NIST standards in symmetric cryptography, which NIST expects to offer significant resistance to quantum cryptanalysis. In particular, NIST defined a separate category for each of the following security requirements (listed in order of increasing strength):

1. Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g., AES-128) .
2. Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 256-bit hash function (e.g., SHA-256/ SHA3-256).
3. Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 192-bit key (e.g., AES-192).
4. Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for collision search on a 384-bit hash function (e.g., SHA-384/ SHA3-384).
5. Any attack that breaks the relevant security definition must require computational resources

comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g., AES-256).

Table 4. NIST Security Strength Categories

Security Category	Corresponding Attack Type	Example
1	Key search on block cipher with 128-bit key	AES-128
2	Collision search on 256-bit hash function	SHA3-256
3	Key search on block cipher with 192-bit key	AES-192
4	Collision search on 384-bit hash function	SHA3-384
5	Key search on block cipher with 256-bit key	AES-256

Here, computational resources may be measured using a variety of different metrics (e.g., number of classical elementary operations, quantum circuit size). In order for a cryptosystem to satisfy one of the above security requirements, any attack must require computational resources comparable to or greater than the stated threshold, with respect to all metrics that NIST deems to be potentially relevant to practical security.

NIST intends to consider a variety of possible metrics, reflecting different predictions about the future development of quantum and classical computing technology, and the cost of different computing resources (such as the cost of accessing extremely large amounts of memory).[2] NIST will also consider input from the cryptographic community regarding this question.

In an example metric provided to submitters, NIST suggested an approach where quantum attacks are restricted to a fixed running time or circuit depth. Call this parameter MAXDEPTH. This restriction is motivated by the difficulty of running extremely long serial computations. Plausible values for MAXDEPTH range from 2^{40} logical gates (the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year) through 2^{64} logical gates (the approximate number of gates that current classical computing architectures can perform serially in a decade), to no more than 2^{96} logical gates (the approximate number of gates that atomic scale qubits with speed of light propagation times could perform in a millennium). The most basic version of this cost metric ignores costs associated with physically moving bits or qubits so they are physically close enough to perform gate operations. This simplification may result in an underestimate of the cost of implementing memory-intensive computations on real hardware.

The complexity of quantum attacks can then be measured in terms of circuit size. These numbers can be compared to the resources required to break AES and SHA-3. During the post-quantum standardization process, NIST gave the following estimates for the classical and quantum gate counts[3] for the optimal key recovery and collision attacks on AES and SHA-3, respectively, where

[2] See the discussion in [19, Appendix B].

[3] Quantum circuit sizes are based on the work in [20].

circuit depth is limited to MAXDEPTH][4].

Table 5. Estimates for classical and quantum gate counts for the optimal key recovery and collision attacks on AES and SHA-3

AES-128	2^{157}/MAXDEPTH quantum gates or 2^{143} classical gates
SHA3-256	2^{146} classical gates
AES-192	2^{221}/MAXDEPTH quantum gates or 2^{207} classical gates
SHA3-384	2^{210} classical gates
AES-256	2^{285}/MAXDEPTH quantum gates or 2^{272} classical gates
SHA3-512	2^{274} classical gates

It is worth noting that the security categories based on these reference primitives provide substantially more quantum security than a naïve analysis might suggest. For example, categories 1, 3, and 5 are defined in terms of block ciphers, which can be broken using Grover's algorithm [21], with a quadratic quantum speedup. However, Grover's algorithm requires a long-running serial computation, which is difficult to implement in practice. In a realistic attack, one has to run many smaller instances of the algorithm in parallel, which makes the quantum speedup less dramatic.

Finally, for attacks that use a combination of classical and quantum computation, one may use a cost metric that rates logical quantum gates as being several orders of magnitude more expensive than classical gates. Presently envisioned quantum computing architectures typically indicate that the cost per quantum gate could be billions or trillions of times the cost per classical gate. However, especially when considering algorithms claiming a high security strength (e.g., equivalent to AES-256 or SHA-384), it is likely prudent to consider the possibility that this disparity will narrow significantly or even be eliminated.

[4]NIST believes the above estimates are accurate for the majority of values of MAXDEPTH that are relevant to its security analysis, but the above estimates may understate the security of SHA for very small values of MAXDEPTH and may understate the quantum security of AES for very large values of MAXDEPTH.