# FIPS 204

**Federal Information Processing Standards Publication**

# Module-Lattice-Based Digital Signature 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.204

Published August 13, 2024


**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 Series of the National Institute of Standards and Technology is the official series of publications relating to standards and guidelines developed under 15 U.S.C. 278g-3, and issued by the Secretary of Commerce under 40 U.S.C. 11331.

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

Kevin M. Stine, Director
Information Technology Laboratory

## Abstract

Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of the signatory. In addition, the recipient of signed data can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation since the signatory cannot easily repudiate the signature at a later time. This standard specifies ML-DSA, a set of algorithms that can be used to generate and verify digital signatures. ML-DSA is believed to be secure, even against adversaries in possession of a large-scale quantum computer.


**Keywords:** cryptography; digital signatures; Federal Information Processing Standards; lattice; post-quantum; public-key cryptography.

**Federal Information Processing Standards Publication 204**

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

# Announcing the Module-Lattice-Based Digital Signature Standard

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

1. **Name of Standard.** Module-Lattice-Based Digital Signature Standard (FIPS 204).

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

3. **Explanation.** This standard specifies ML-DSA, a lattice-based digital signature algorithm for applications that require a digital signature rather than a written signature. Additional digital signature schemes are specified and approved in other NIST Special Publications and FIPS publications (e.g., FIPS 186-5 [1]). A digital signature is represented in a computer as a string of bits and computed using a set of rules and parameters that allow the identity of the signatory and the integrity of the data to be verified. Digital signatures may be generated on both stored and transmitted data.

   Signature generation uses a private key to generate a digital signature. Signature verification uses a public key that corresponds to but is not the same as the private key. Each signatory possesses a key-pair composed of a private key and a corresponding public key. Public keys may be known by the public, but private keys must be kept secret. Anyone can verify the signature by employing the signatory's public key. Only the user who possesses the private key can generate a signature that can be verified by the corresponding public key.

   The digital signature is provided to the intended verifier along with the signed data. The verifying entity verifies the signature by using the claimed signatory's public key. Similar procedures may be used to generate and verify signatures for both stored and transmitted data.

   This standard specifies several parameter sets for ML-DSA that are **approved** for use. Additional parameter sets may be specified and approved in future 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.** This standard is applicable to all federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502 (2) of Title 44, United States Code. Either this standard, FIPS 205, FIPS 186-5, or NIST Special Publication 800-208 **shall** be used in designing and implementing public-key-based signature systems that federal departments and agencies operate or that are operated for them under contract. In the future, additional digital signature schemes may be specified and approved in FIPS or NIST Special Publications.

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

7. **Applications.** A digital signature algorithm allows an entity to authenticate the integrity of signed data and the identity of the signatory. The recipient of a signed message can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation since the signatory cannot easily repudiate the signature at a later time. A digital signature algorithm is intended for use in electronic mail, electronic funds transfer, electronic data interchange, software distribution, data storage, and other applications that require data integrity assurance and data origin authentication.

8. **Implementations.** A digital signature algorithm may be implemented in software, firmware, hardware, or any combination thereof. NIST will develop a validation program to test implementations for conformance to the algorithm in this standard. For every computational procedure that is specified in this standard, a conforming implementation may replace the given set of steps with any mathematically equivalent set of steps. In other words, different procedures that produce the correct output for every input are permitted.

   Information about validation programs is available at https://csrc.nist.gov/projects/cmvp. Examples for digital signature algorithms are available at https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

   Agencies are advised that digital signature key pairs **shall not** be used for other purposes.

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

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

   b. Adopted in a FIPS or NIST recommendation, or

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

10. **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 these federal 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.

11. **Patents.** The algorithm in this standard may be covered by U.S. or foreign patents.

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

13. **Specifications.** Federal Information Processing Standards (FIPS) 204, Module-Lattice-Based Digital Signature Standard (affixed).

14. **Qualifications.** The security of a digital signature system depends on maintaining the secrecy of the signatory's private keys. Signatories **shall**, therefore, guard against the disclosure of their private keys. While it is the intent of this standard to specify general security requirements for generating digital signatures, conformance to this standard does not ensure that a particular implementation is secure. It is the responsibility of an implementer to ensure that any module that implements a digital signature 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.

Since a standard of this nature must be flexible enough to adapt to advancements and innovations in science and technology, this standard will be reviewed every five years in order to assess its adequacy.

15. **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.

16. **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.

17. **How to Cite This Publication.** NIST has assigned **NIST FIPS 204** as the publication identifier for this FIPS, per the NIST Technical Series Publication Identifier Syntax. NIST recommends that it be cited as follows:

    National Institute of Standards and Technology (2024) Module-Lattice-Based Digital Signature Standard. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 204. https://doi.org/10.6028/NIST.FIPS.204

18. **Inquiries and Comments.** Inquiries and comments about this FIPS may be submitted to fips-204-comments@nist.gov.

# Federal Information Processing Standards Publication 204

## Specification for the
# Module-Lattice-Based Digital Signature Standard

## Table of Contents

## List of Tables



## List of Algorithms

# 1. Introduction

## 1.1 Purpose and Scope

This standard defines a digital signature scheme, which includes a method for digital signature generation that can be used for the protection of binary data (commonly called a "message") and a method for the verification and validation of those digital signatures. NIST Special Publication (SP) 800-175B [2], *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*, includes a general discussion of digital signatures.

This standard specifies the mathematical steps that need to be performed for key generation, signature generation, and signature verification. In order for digital signatures to be valid, additional assurances are required, such as assurance of identity and of private key possession. SP 800-89, *Recommendation for Obtaining Assurances for Digital Signature Applications* [3], specifies the required assurances and the methods for obtaining them.

The digital signature scheme approved in this standard is the Module-Lattice-Based Digital Signature Algorithm (ML-DSA), which is based on the Module Learning With Errors problem [4]. ML-DSA is believed to be secure, even against adversaries in possession of a large-scale fault-tolerant quantum computer. In particular, ML-DSA is believed to be strongly unforgeable, which implies that the scheme can be used to detect unauthorized modifications to data and to authenticate the identity of the signatory (one bound to the possession of the private-key). In addition, a signature generated by this scheme can be used as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. The latter property is known as non-repudiation since the signatory cannot easily repudiate the signature at a later time.

This standard gives algorithms for ML-DSA key generation, signature generation, and signature verification (Section 5), and for supporting algorithms used by them (Section 7). ML-DSA is standardized with three possible parameter sets, each of which corresponds to a different security strength. Section 4 describes the global parameters used by these algorithms and enumerates the parameter sets for ML-DSA that are approved by this standard. ML-DSA can be used in place of other digital signature schemes specified in NIST FIPS and Special Publications (e.g., FIPS 186-5, *Digital Signature Standard (DSS)* [1]).

## 1.2 Context

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

After three rounds of evaluation and analysis, NIST selected the first four algorithms for standardization. ML-DSA is derived from one of the selected schemes, CRYSTALS-DILITHIUM [5, 6], and is intended to protect sensitive U.S. Government information well into the foreseeable future, including after the advent of cryptographically relevant quantum computers. For the differences between ML-DSA and CRYSTALS-DILITHIUM, see Appendix D.

# 2. Glossary of Terms, Acronyms, and 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. |
| assurance of possession | Confidence that an entity possesses a private key and any associated keying material. |
| asymmetric cryptography | Cryptography that uses two separate keys to exchange data — one to encrypt or digitally sign the data and one to decrypt the data or verify the digital signature. Also known as *public-key cryptography*. |
| bit string | An ordered sequence of zeros and ones. |
| byte | An integer from the set {0, 1, 2, …, 255}. |
| byte string | An ordered sequence of bytes. |
| certificate | A set of data that uniquely identifies a public key that has a corresponding private key and an owner that is authorized to use the key pair. The certificate contains the owner's public key and possibly other information and is digitally signed by a certification authority (i.e., a trusted party), thereby binding the public key to the owner. |
| certification authority (CA) | The entity in a public-key infrastructure (PKI) that is responsible for issuing certificates and exacting compliance with a PKI policy. |
| claimed signatory | From the verifier's perspective, the claimed signatory is the entity that purportedly generated a digital signature. |
| destroy | An action applied to a key or a piece of secret data. After a key or a piece of secret data is destroyed, no information about its value can be recovered. |
| digital signature | The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism to verify origin authenticity and data integrity and to enforce signatory non-repudiation. |
| entity | An individual person, organization, device, or process. Used interchangeably with *party*. |
| 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). |
| eXtendable-Output Function (XOF) | A function on bit strings in which the output can be extended to any desired length. **Approved** XOFs (e.g., those specified in FIPS 202 [7]) are designed to satisfy the following properties as long as the specified output length is sufficiently long to prevent trivial attacks: |

1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output.
2. (Collision-resistant) It is computationally infeasible to find any two distinct inputs that map to the same output.

| | |
|---|---|
| fresh random value | A previously unused output of a random bit generator. |
| hash function | A function on bit strings in which the length of the output is fixed. **Approved** hash functions (such as those specified in FIPS 180 [8] and FIPS 202 [7]) are designed to satisfy the following properties:<br>1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output.<br>2. (Collision-resistant) It is computationally infeasible to find any two distinct inputs that map to the same output. |
| hash value | See *message digest*. |
| key | A parameter used in conjunction with a cryptographic algorithm that determines its operation. Examples of cryptographic algorithms applicable to this standard include:<br>1. The computation of a digital signature from data<br>2. The verification of a digital signature |
| key pair | A public key and its corresponding private key. |
| little-endian | The property of a byte string having its bytes positioned in order of increasing significance. In particular, the leftmost (first) byte is the least significant, and the rightmost (last) byte is the most significant. The term "little-endian" may also be applied in the same manner to bit strings (e.g., the 8-bit string 11010001 corresponds to the byte $2^0 + 2^1 + 2^3 + 2^7 = 139$). |
| message | The data that is signed. Also known as *signed data* during the signature verification and validation process. |
| message digest | The result of applying a hash function to a message. Also known as a *hash value*. |
| non-repudiation | A service that is used to provide assurance of the integrity and origin of data in such a way that the integrity and origin can be verified and validated by a third party as having originated from a specific entity in possession of the private key (i.e., the signatory). |
| owner | A key pair owner is the entity authorized to use the private key of a key pair. |
| party | An individual person, organization, device, or process. Used interchangeably with *entity*. |
| public-key infrastructure (PKI) | A framework that is established to issue, maintain, and revoke public key certificates. |

| | |
|---|---|
| private key | A cryptographic key that is used with an asymmetric (public-key) cryptographic algorithm. The private key is uniquely associated with the owner and is not made public. The private key is used to compute a digital signature that may be verified using the corresponding public key. |
| pseudorandom | A process or data produced by a process is said to be pseudorandom when the outcome is deterministic yet also effectively 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 key | A cryptographic key that is used with an asymmetric (public-key) cryptographic algorithm and is associated with a private key. The public key is associated with an owner and may be made public. In the case of digital signatures, the public key is used to verify a digital signature that was generated using the corresponding private key. |
| security category | A number associated with the security strength of a post-quantum cryptographic algorithm, as specified by NIST (see [9, Sect. 5.6]). |
| security strength | A number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system. |
| seed | A bit string used as input to a pseudorandom process. |
| **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. |
| signatory | The entity that generates a digital signature on data using a private key. |
| signature generation | The process of using a digital signature algorithm and a private key to generate a digital signature on data. |
| signature validation | The mathematical verification of the digital signature along with obtaining the appropriate assurances (e.g., public-key validity, private-key possession, etc.). |
| signature verification | The process of using a digital signature algorithm and a public key to verify a digital signature on data. |
| signed data | The data or message upon which a digital signature has been computed. Also see *message*. |
| trusted third party (TTP) | An entity other than the key pair owner and the verifier that is trusted by the owner, the verifier, or both. Sometimes shortened to "trusted party." |
| verifier | The entity that verifies the authenticity of a digital signature using the public key of the signatory. |

## 2.2 Acronyms

| | |
|---|---|
| AES | Advanced Encryption Standard |
| API | Application Programming Interface |
| DER | Distinguished Encoding Rules |
| FIPS | Federal Information Processing Standard |
| ML-DSA | Module-Lattice-Based Digital Signature Algorithm |
| MLWE | Module Learning With Errors |
| NIST | National Institute of Standards and Technology |
| NIST IR | NIST Interagency or Internal Report |
| NTT | Number Theoretic Transform |
| OID | Object Identifier |
| PQC | Post-Quantum Cryptography |
| RBG | Random Bit Generator |
| SHA | Secure Hash Algorithm |
| SHAKE | Secure Hash Algorithm KECCAK |
| SP | Special Publication |
| SUF-CMA | Strongly existentially UnForgeable under Chosen Message Attack |
| XOF | eXtendable-Output Function |

## 2.3 Mathematical Symbols

The following symbols and mathematical expressions are used in this standard.

| | |
|---|---|
| $q$ | The prime number $q = 2^{23} - 2^{13} + 1 = 8380417$. |
| $\mathbb{B}$ | The set $\{0, 1, \ldots, 255\}$ of integers represented by a byte. |
| $\mathbb{N}$ | The set of natural numbers $\{1, 2, 3, \ldots\}$. |
| $\mathbb{Z}$ | The ring of integers. |
| $\mathbb{Z}_m$ | The ring of integers modulo $m$ whose set of elements is $\{0, 1, \ldots, m-1\}$. |
| $\mathbb{Z}_m^n$ | The set of $n$-tuples over $\mathbb{Z}_m$ equipped with the $\mathbb{Z}$-module structure. |
| $R$ | The ring of single-variable polynomials over $\mathbb{Z}$ modulo $X^{256} + 1$, also denoted by $\mathbb{Z}[X]/(X^{256} + 1)$. |
| $R_m$ | The ring of single-variable polynomials over $\mathbb{Z}_m$ modulo $X^{256} + 1$, also denoted by $\mathbb{Z}_m[X]/(X^{256} + 1)$. |

| | |
|---|---|
| $B_\tau$ | The set of all polynomials $p = \sum_{i=0}^{255} p_i X^i$ in $R_q$ that are such that exactly $\tau$ of the coefficients of $p_i$ are from the set $\{-1, 1\}$, and all other coefficients are zero. (See Section 7.3.) |
| $\Pi$ | Used to denote a direct product of two or more rings, where addition and multiplication are performed componentwise. |
| $T_q$ | The ring $\Pi_{j=0}^{255} \mathbb{Z}_q$. |
| $A \times B$ | Cartesian product of two sets $A, B$. |
| $[a, b]$ | For two integers $a \leq b$, $[a, b]$ denotes the set of integers $\{a, a+1, \ldots, b\}$. |
| bitlen $a$ | The bit length of a positive integer $a$. The bit length of $a$ is the number of digits that would appear in a base-2 representation of $a$, where the most significant digit in the representation is assumed to be a 1 (e.g., bitlen $32 = 6$ and bitlen $31 = 5$).[1] |
| $\text{BitRev}_8(r)$ | Bit reversal of an 8-bit integer $r$. If $r = r_0 + 2r_1 + 4r_2 + \ldots + 128r_7$ with $r_i \in \{0, 1\}$, then $\text{BitRev}_8(r) = r_7 + 2r_6 + 4r_5 + \ldots + 128r_0$. |
| 0x | Prefix to an integer written in hexadecimal representation. |
| $\log_2 x$ | The base 2 logarithm of $x$. For example, $\log_2(16) = 4$. |
| mod | If $\alpha$ is a positive integer and $m \in \mathbb{Z}$ or $m \in \mathbb{Z}_\alpha$, then $m$ mod $\alpha$ denotes the unique element $m' \in \mathbb{Z}$ in the range $0 \leq m' < \alpha$ such that $m$ and $m'$ are congruent modulo $\alpha$. |
| $\text{mod}^{\pm}$ | If $\alpha$ is a positive integer and $m \in \mathbb{Z}$ or $m \in \mathbb{Z}_\alpha$, then $m$ $\text{mod}^{\pm}\alpha$ denotes the unique element $m' \in \mathbb{Z}$ in the range $-\lceil \alpha/2 \rceil < m' \leq \lfloor \alpha/2 \rfloor$ such that $m$ and $m'$ are congruent modulo $\alpha$. |
| $\lfloor x \rfloor$ | The largest integer less than or equal to the real number $x$, called the floor of $x$. For example, $\lfloor 2.1 \rfloor = 2$, and $\lfloor 4 \rfloor = 4$. |
| $\lceil x \rceil$ | The least integer greater than or equal to the real number $x$, called the ceiling of $x$. For example, $\lceil 5 \rceil = 5$ and $\lceil 5.3 \rceil = 6$. |
| $\|\cdot\|_\infty$ | The infinity norm. For an element $w \in \mathbb{Z}$, $\|w\|_\infty = \|w\|$, the absolute value of $w$. For an element $w \in \mathbb{Z}_q$, $\|w\|_\infty = \left\|w \text{ mod}^{\pm} q\right\|$. For an element $w$ of $R$ or $R_q$, $\|w\|_\infty = \max_{0 \leq i < 256} \|w_i\|_\infty$. For a length-$m$ vector $\mathbf{w}$ with entries from $R$ or $R_q$, $\|\mathbf{w}\|_\infty = \max_{0 \leq i < m} \|w[i]\|_\infty$. |
| $a!$ | The factorial quantity $1 \cdot 2 \cdot 3 \cdot \ldots \cdot a$. The value $0!$ is defined as 1. |
| $\binom{a}{b}$ | For $a \geq b$, the quantity $a!/(b!(a-b)!)$. |
| $s \leftarrow x$ | In pseudocode, this notation means that the variable $s$ is assigned the value of the expression $x$. |
| $s \leftarrow \mathbb{B}^\ell$ | In pseudocode, this notation means that the variable $s$ is assigned the value of an array of $\ell$ random bytes. The bytes must be generated using randomness from an **approved** RBG. See Section 3.6.1. |

[1] In this specification, bitlen $a$ is only ever called with a small finite number of values for $a$, so it may be helpful to precompute bitlen $a$ for these values and hard code the results, rather than computing them on the fly.

| | |
|---|---|
| $x \in S \leftarrow y$ | Type casting. The variable $x$ is assigned a value in a set $S$ that is constructed from the value of an expression $y$ in a possibly different set $T$. The set $T$ and the mapping from $T$ to $S$ are not explicitly specified, but they should be obvious from the context in which this statement appears. |
| $[[a < b]]$ | A Boolean predicate. A comparison operator inside double square brackets $[[a < b]]$ denotes that the expression should be evaluated as a Boolean. Booleans can also be interpreted as elements of $\mathbb{Z}_2$ with 1 denoting true and 0 denoting false. |
| $\langle\langle f(x) \rangle\rangle$ | A temporary variable that stores the output of a computation $f(x)$ so that it can be used many times without needing to be recomputed. This is equivalent to defining a temporary variable $y \leftarrow f(x)$. Naming the variable $\langle\langle f(x) \rangle\rangle$ makes the pseudocode less cluttered. |
| $a \|\| b$ | Concatenation of two bit or byte strings $a$ and $b$. |
| $a \circ b$ | Multiplication of $a$ and $b$ in the ring $T_q$. |
| $a \cdot b$ or $ab$ | Multiplication in any of the rings $\mathbb{Z}, \mathbb{Z}_m, R$, or $R_m$. |
| $a + b$ | Addition of $a$ and $b$. |
| $a/b$ | Division of integers. When this notation is used, $a$ and $b$ are always integers. If $b$ cannot be assumed to divide $a$, then either $\lfloor a/b \rfloor$ or $\lceil a/b \rceil$ is used. |
| $\perp$ | Blank symbol that indicates failure or the lack of an output from an algorithm. |

## 2.4 Notation

### 2.4.1 Rings

Elements of the rings $\mathbb{Z}$, $\mathbb{Z}_q$, $\mathbb{Z}_2$, $R$, and $R_q$ are denoted by italicized lowercase letters (e.g., $w$). Elements of the ring $T_q$ are length-$256$ arrays of elements of $\mathbb{Z}_q$, and they are denoted by italicized letters with a hat symbol (e.g., $\hat{w}$). The addition and multiplication of elements of $T_q$ are performed entry-wise. Thus, the $i$th entry of the product of two elements $\hat{u}$ and $\hat{v}$ of $T_q$ is $\hat{u}[i] \cdot \hat{v}[i] \in \mathbb{Z}_q$. The multiplication operation in $T_q$ is denoted by the symbol $\circ$ (see Section 2.3).

When a product $a \cdot b$ or a sum $a + b$ is written and either $a$ or $b$ is a congruence class modulo $m$ (i.e., if either $a$ or $b$ is an element of $\mathbb{Z}_m$ or $R_m$), then the product or sum is also understood to be a congruence class modulo $m$ (i.e., an element of $\mathbb{Z}_m$ or $R_m$). Likewise, an element of $R$ or $\mathbb{Z}$ may be "typecast" to an element of $R_m$ or $\mathbb{Z}_m$, respectively, and may be used as the input of a function specified to act on an element of $R_m$ or $\mathbb{Z}_m$, respectively. In both cases, the element itself or its coefficients are mapped from $\mathbb{Z}$ to $\mathbb{Z}_m$ by taking the unique congruence class modulo $m$ that contains the integer.

The coefficients of an element $w$ of $R$ or $R_m$ are denoted by $w_i$ so that $w = w_0 + w_1X + ... + w_{255}X^{255}$. If $w$ is in $R$ (respectively, $R_m$) and $t$ is in $\mathbb{Z}$ (respectively, $\mathbb{Z}_d$), then $w(t)$ denotes the polynomial $w = w_0 + w_1X + ... + w_{255}X^{255}$ evaluated at $X = t$.

### 2.4.2 Vectors and Matrices

Vectors with elements in $R$ or $R_m$ are denoted by bold lowercase letters (e.g., $\mathbf{v}$). Matrices with elements in $R$ or $R_m$ are denoted by bold uppercase letters (e.g., $\mathbf{A}$).

If $S$ is a ring and $\mathbf{v}$ is a length-$L$ vector over $S$, then the entries in the vector $\mathbf{v}$ are expressed as

$$v[0], v[1], \ldots, v[L-1].$$

The entries of a $K \times L$ matrix $\mathbf{A}$ over $S$ are denoted as $\mathbf{A}[i, j]$, where $0 \leq i < K$ and $0 \leq j < L$.

The set of all length-$L$ vectors over $S$ is denoted by $S^L$. The set of all $K \times L$ matrices over $S$ is denoted by $S^{K \times L}$. A length-$L$ vector can also be treated as an $L \times 1$ matrix.

## 2.5 NTT Representation

The Number Theoretic Transform (NTT) is a specific isomorphism between the rings $R_q$ and $T_q$. Let $\zeta = 1753 \in \mathbb{Z}_q$, which is a $512^{\text{th}}$ root of unity. If $w \in R_q$, then

$$\text{NTT}(w) = (w(\zeta_0), w(\zeta_1), \ldots, w(\zeta_{255})) \in T_q, \tag{2.1}$$

where $\zeta_i = w(\zeta^{2\text{BitRev}_8(i)+1}) \bmod q$. See Section 7.5 for an implementation discussion for NTT and $\text{NTT}^{-1}$.

The motivation for using NTT is that multiplication is considerably faster in the ring $T_q$. Since NTT is an isomorphism, for any $a, b \in R_q$,

$$\text{NTT}(ab) = \text{NTT}(a) \circ \text{NTT}(b). \tag{2.2}$$

If $\mathbf{A}$ is a matrix with entries from $R_q$, then $\text{NTT}(\mathbf{A})$ denotes the matrix computed via the entry-wise application of NTT to $\mathbf{A}$. The symbol $\circ$ is also used to denote the matrix multiplication of matrices with entries in $T_q$. Thus, $\text{NTT}(\mathbf{AB}) = \text{NTT}(\mathbf{A}) \circ \text{NTT}(\mathbf{B})$. Explicit algorithms for linear algebra over $T_q$ are given in Section 7.6.

# 3. Overview of the ML-DSA Signature Scheme

ML-DSA is a digital signature scheme based on CRYSTALS-DILITHIUM [6]. It consists of three main algorithms: ML-DSA.KeyGen (Algorithm 1), ML-DSA.Sign (Algorithm 2), and ML-DSA.Verify (Algorithm 3). The ML-DSA scheme uses the Fiat-Shamir With Aborts construction [10, 11] and bears the most resemblance to the schemes proposed in [12, 13].

This document also defines a closely related but domain-separated signature scheme, HashML-DSA, which differs from ML-DSA in that it includes an additional pre-hashing step before signing. It consists of three main algorithms: ML-DSA.KeyGen (Algorithm 1), which is the same key generation algorithm used for ML-DSA; HashML-DSA.Sign (Algorithm 4); and HashML-DSA.Verify (Algorithm 5).

## 3.1 Security Properties

ML-DSA is designed to be strongly existentially unforgeable under chosen message attack (SUF-CMA). That is, it is expected that even if an adversary can get the honest party to sign arbitrary messages, the adversary cannot create any additional valid signatures based on the signer's public key, including on messages for which the signer has already provided a signature.

Beyond unforgeability, ML-DSA is designed to satisfy additional security properties described in [14].

## 3.2 Computational Assumptions

Security for lattice-based digital signature schemes is typically related to the Learning With Errors (LWE) problem and the short integer solution (SIS) problem. The LWE problem [15] is to recover a vector $\mathbf{s} \in \mathbb{Z}_q^n$ given a set of random "noisy" linear equations[2] satisfied by $\mathbf{s}$. The SIS problem is to find a non-zero solution $\mathbf{t} \in \mathbb{Z}_q^n$ for a given linear system over $\mathbb{Z}_q$ of the form $\mathbf{At} = \mathbf{0}$ such that $\|\mathbf{t}\|_\infty$ is small. For appropriate choices of parameters, these problems are intractable for the best known techniques, including Gaussian elimination.

When the module $\mathbb{Z}_q^n$ in LWE and SIS is replaced by a module over a ring larger than $\mathbb{Z}_q$ (e.g., $R_q$), the resulting problems are called Module Learning With Errors (MLWE) [4] and Module Short Integer Solution (MSIS). The security of ML-DSA is based on the MLWE problem over $R_q$ and a nonstandard variant of MSIS called SelfTargetMSIS [16].

## 3.3 ML-DSA Construction

ML-DSA is a Schnorr-like signature with several optimizations. The Schnorr signature scheme applies the Fiat-Shamir heuristic to an interactive protocol between a verifier who knows $g$ (the generator of a group in which discrete logs are believed to be difficult) and the value $y = g^x$ and a prover who knows $g$ and $x$. The interactive protocol, where the prover demonstrates knowledge of $x$ to the verifier, consists of three steps:

1. Commitment: The prover generates a random positive integer $r$ that is less than the order of $g$ and commits to its value by sending $g^r$ to the verifier.
2. Challenge: The verifier sends a random positive integer $c$ that is less than the order of $g$ to the prover.

[2] Specifically, the LWE problem is to solve a system of equations of the form $\mathbf{As} + \mathbf{e} = b$, where $\mathbf{A}$ and $b$ are given and $\mathbf{e}$ is not given but known to be small.

3. Response: The prover returns $s = r - cx$ reduced modulo the order of $g$, and the verifier checks whether $g^s \cdot y^c = g^r$.

This protocol is made noninteractive and turned into a signature scheme by replacing the verifier's random choice of $c$ in step 2 with a deterministic process that pseudorandomly derives $c$ from a hash of the commitment $g^r$ concatenated with the message to be signed. For this signature scheme, $y$ is the public key, and $x$ is the private key.

The basic idea of ML-DSA and similar lattice signature schemes is to build a signature scheme from an analogous interactive protocol, where a prover who knows matrices $\mathbf{A} \in \mathbb{Z}_q^{K \times L}$, $\mathbf{S}_1 \in \mathbb{Z}_q^{L \times n}$, and $\mathbf{S}_2 \in \mathbb{Z}_q^{K \times n}$ with small coefficients (for $\mathbf{S}_1$ and $\mathbf{S}_2$) demonstrates knowledge of these matrices to a verifier who knows $\mathbf{A}$ and $\mathbf{T} \in \mathbb{Z}_q^{K \times n} = \mathbf{A}\mathbf{S}_1 + \mathbf{S}_2$. Such an interactive protocol would proceed as follows:

1. Commitment: The prover generates $\mathbf{y} \in \mathbb{Z}_q^L$ with small coefficients and commits to its value by sending $\mathbf{w}_{\text{Approx}} = \mathbf{A}\mathbf{y} + \mathbf{y}_2$ to the verifier, where $\mathbf{y}_2 \in \mathbb{Z}_q^K$ is a vector with small coefficients.
2. Challenge: The verifier sends a vector $\mathbf{c} \in \mathbb{Z}_q^n$ with small coefficients to the prover.
3. Response: The prover returns $\mathbf{z} = \mathbf{y} + \mathbf{S}_1\mathbf{c}$, and the verifier checks that $\mathbf{z}$ has small coefficients and that $\mathbf{A}\mathbf{z} - \mathbf{T}\mathbf{c} \approx \mathbf{w}_{\text{Approx}}$.

As written, the above protocol has a security flaw: the response $\mathbf{z}$ will be biased in a direction related to the private value $\mathbf{S}_1$. Likewise $\mathbf{r} = \mathbf{w}_{\text{Approx}} - \mathbf{A}\mathbf{z} + \mathbf{T}\mathbf{c} = \mathbf{y}_2 + \mathbf{S}_2\mathbf{c}$ is biased in a direction related to the private value $\mathbf{S}_2$. However, this flaw can be corrected when converting the interactive protocol into a signature scheme. As with Schnorr signatures, the signer derives the challenge by a pseudorandom process from a hash of the commitment concatenated with the message. To correct the bias, the signer applies rejection sampling to $\mathbf{z}$; if coefficients of $\mathbf{z}$ fall outside of a specified range, the signing process is aborted, and the signer starts over from a new value of $\mathbf{y}$. Likewise, similar rejection sampling must also be applied to $\mathbf{r}$. These checks are analogous to those done at Line 23 of Algorithm 7. In the simplified Fiat-Shamir With Aborts signature, the public key is $(\mathbf{A}, \mathbf{T})$, and the private key is $(\mathbf{S}_1, \mathbf{S}_2)$.

In the ML-DSA standard, a number of tweaks and modifications are added to this basic framework for security or efficiency reasons:

- To reduce key and signature size and to use fast NTT-based polynomial multiplication, ML-DSA uses module-structured matrices. Relative to the basic scheme described above, it replaces dimension-$n \times n$ blocks of matrices and dimension-$n$ blocks of vectors with polynomials in the ring $R_q$. Thus, instead of $\mathbf{A} \in \mathbb{Z}_q^{K \times L}$, $\mathbf{T} \in \mathbb{Z}_q^{K \times n}$, $\mathbf{S}_1 \in \mathbb{Z}_q^{L \times n}$, $\mathbf{S_2} \in \mathbb{Z}_q^{K \times n}$, $\mathbf{y} \in \mathbb{Z}_q^L$, $\mathbf{c} \in \mathbb{Z}_q^n$, ML-DSA has $\mathbf{A} \in R_q^{k \times \ell}$, $\mathbf{t} \in R_q^k$, $\mathbf{s}_1 \in R_q^\ell$, $\mathbf{s}_2 \in R_q^k$, $\mathbf{y} \in R_q^\ell$, $c \in R_q$, where $\ell = L/n$ and $k = K/n$.
- To further reduce the size of the public key, the matrix $\mathbf{A}$ is pseudorandomly derived from a 256-bit public seed $\rho$, which is included in the ML-DSA public key in place of $\mathbf{A}$.
- For a still further reduction in public key size, the ML-DSA public key substitutes a compressed value $\mathbf{t}_1$ for $\mathbf{t}$, which drops the $d$ low-order bits of each coefficient.
- To obtain beyond unforgeability (BUFF) properties noted in [14], ML-DSA does not directly sign the message $M$ but rather signs a message representative $\mu$ that is obtained by hashing the concatenation of a hash of the public key and $M$.

- To reduce signature size, rather than including the commitment $\mathbf{w}_{\text{Approx}} = \mathbf{A}\mathbf{y} + \mathbf{y}_2$ in the signature, the ML-DSA signature uses a rounded version of $\mathbf{w} = \mathbf{A}\mathbf{y}$ as a commitment $\mathbf{w}_1$ and includes only the hash $\tilde{c}$ of $\mathbf{w}_1 || \mu$.
- To ensure that $\mathbf{w}_1$ can be reconstructed by the verifier from $\mathbf{z}$ and the compressed value $\mathbf{t}_1$, the signature must also include a *hint* $\mathbf{h} \in R_2^k$ computed by the signer using the signer's private key.
- Additionally, to ensure correctness, a second stage of rejection sampling must be included (Line 28 of Algorithm 7)

In this document, the abbreviations ML-DSA-44, ML-DSA-65, and ML-DSA-87 are used to refer to ML-DSA with the parameter choices given in Table 1. In these abbreviations, the numerical suffix refers to the dimension of the matrix $\mathbf{A}$. For example, in ML-DSA-65, the matrix $\mathbf{A}$ is a $6 \times 5$ matrix over $R_q$.

## 3.4 Hedged and Deterministic Signing

For ML-DSA to be secure, the signer's commitment value $\mathbf{y}$ must not be used to sign more than one message, and it must not be easily guessed by an attacker. This requires randomness. In principle, the randomness leading to $\mathbf{y}$ can be produced either with the use of fresh randomness at signing time or pseudorandomly from the message being signed and a precomputed random value included in the signer's private key.

By default, this standard specifies the signing algorithm to use both types of randomness. This is referred to as the "hedged" variant of the signing procedure. The use of fresh randomness during signing helps mitigate side-channel attacks, while the use of precomputed randomness protects against the possibility that there may be flaws in the random number generator used by the signer at signing time.

This document also permits a fully deterministic variant of the signing procedure in case the signer has no access to a fresh source of randomness at signing time. However, the lack of randomness in the deterministic variant makes the risk of side-channel attacks (particularly fault attacks) more difficult to mitigate. Therefore, this variant **should not** be used on platforms where side-channel attacks are a concern and where they cannot be otherwise mitigated.

Only implementing the hedged variant (i.e., without the deterministic variant) is sufficient to guarantee interoperability. The same verification algorithm will work to verify signatures produced by either variant, so implementing the deterministic variant in addition to the hedged variant does not enhance interoperability. The hedged and deterministic signing procedure differ only at line 5 of Algorithm 2 and line 5 of Algorithm 4.

## 3.5 Use of Digital Signatures

Secure key management is an essential requirement for the use of digital signatures. This is context-dependent and involves more than the key generation, signing, and signature verification algorithms in this document. Guidance for key management is provided in the SP 800-57 series [9, 17, 18].

Digital signatures are most useful when bound to an identity. Binding a public key to an identity requires proof of possession of the private key. In the PKI context, issuing certificates involves assurances of identity and proof of possession. When a public-key certificate is not available, users of digital signatures should determine whether a public key needs to be bound to an identity. Methods for obtaining assurances of identity and proof of possession are provided in [3].

## 3.6 Additional Requirements

This section describes several required assurances when implementing ML-DSA. These are in addition to the considerations in Section 3.5.

### 3.6.1 Randomness Generation

Algorithm 1, implementing key generation for ML-DSA, uses an RBG to generate the 256-bit random seed $\xi$. The seed $\xi$ **shall** be a fresh (i.e., not previously used) random value generated using an **approved** RBG, as prescribed in SP 800-90A, SP 800-90B, and SP 800-90C [19, 20, 21]. Moreover, the RBG used **shall** have a security strength of at least 192 bits for ML-DSA-65 and 256 bits for ML-DSA-87. For ML-DSA-44, the RBG **should** have a security strength of at least 192 bits and **shall** have a security strength of at least 128 bits. If an **approved** RBG with at least 128 bits of security but less than 192 bits of security is used, then the claimed security strength of ML-DSA-44 is reduced from category 2 to category 1.

Additionally, the value $rnd$ is generated using an RBG in the default "hedged" variants of Algorithms 2 and 4 for ML-DSA and HashML-DSA signing, respectively. While this value **should** ideally be generated by an **approved** RBG, other methods for generating fresh random values may be used. The primary purpose of $rnd$ is to facilitate countermeasures to side-channel attacks and fault attacks on deterministic signatures, such as [22, 23, 24].[3] For this purpose, even a weak RBG may be preferable to the fully deterministic variants of Algorithms 2 and 4.

### 3.6.2 Public-Key and Signature Length Checks

Algorithm 3, implementing verification for ML-DSA, and Algorithm 5, implementing verification for HashML-DSA, specify the length of the signature $\sigma$ and the public key $pk$ in terms of the parameters described in Table 1. If an implementation of ML-DSA can accept inputs for $\sigma$ or $pk$ of any other length, it **shall** return false whenever the lengths of either of these inputs differ from their lengths specified in this standard. Failing to check the length of $pk$ or $\sigma$ may interfere with the security properties that ML-DSA is designed to have, like strong unforgeability.

### 3.6.3 Intermediate Values

The data used internally by the key generation and signing algorithms in intermediate computation steps could be used by an adversary to gain information about the private key and thereby compromise security. The data used internally by verification algorithms is similarly sensitive for some applications, including the verification of signatures that are used as bearer tokens (i.e., authentication secrets) or the verification of signatures on plaintext messages that are intended to be confidential. Intermediate values of the verification algorithm may reveal information about its inputs (i.e., the message, signature, and public key), and in some applications, security or privacy requires one or more of these inputs to be confidential. Therefore, implementations of ML-DSA **shall** ensure that any potentially sensitive intermediate data is destroyed as soon as it is no longer needed.

Two particular cases in which implementations may refrain from destroying intermediate data are:

1. The seed $\xi$ generated in step 1 of ML-DSA.KeyGen can be stored for the purpose of later expansion

[3] In addition, when signing is deterministic, there is leakage through timing side channels of information about the message but not the private key). If the signer does not want to reveal the message being signed, hedged signatures should be used (see Section 3.2 in [6]).

using ML-DSA.KeyGen_internal. As the seed can be used to compute the private key, it is sensitive data and **shall** be treated with the same safeguards as a private key.

2. The matrix $\hat{\mathbf{A}}$ generated in step 3 of ML-DSA.KeyGen_internal can be stored so that it need not be recomputed in later operations. Likewise, the matrix $\hat{\mathbf{A}}$ generated in step 5 of the verification algorithm ML-DSA.Verify_internal can also be stored. In either case, the matrix $\hat{\mathbf{A}}$ is data that is easily computed from the public key and does not require any special protections.

In certain situations (e.g., deterministic signing and the verification of confidential messages and signatures), additional care must be taken to protect implementations against side-channel attacks or fault attacks. A cryptographic device may leak critical information through side channels, which allows internal data or keying material to be extracted without breaking the cryptographic primitives.

### 3.6.4 No Floating-Point Arithmetic

Implementations of ML-DSA **shall not** use floating-point arithmetic, as rounding errors in floating point operations may lead to incorrect results in some cases. Either $\lfloor x/y \rfloor$ or $\lceil x/y \rceil$ is used in all pseudocode in this standard in which division is performed (e.g., $x/y$), and $y$ may not divide $x$. Both of these can be computed without floating-point arithmetic, as ordinary integer division $x/y$ computes $\lfloor x/y \rfloor$, and $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for non-negative integers $x$ and positive integers $y$. If $y$ is a power of two, it may be more efficient to use bit shift operations than integer division.

## 3.7 Use of Symmetric Cryptography

This standard makes use of the functions SHAKE256 and SHAKE128, as defined in FIPS 202 [7]. While FIPS 202 specifies these functions as inputting and outputting bit strings, most implementations treat inputs and outputs as byte strings.

This standard will always call these functions with an output length of a multiple of eight bits and treat the output of these functions as a byte string, which will be the same byte string that would result from taking the bit string expected from a literal reading of FIPS 202 and processing it with BitsToBytes. However, to allow the signing of messages that are not a whole number of bytes, this document will overload SHAKE256 so that its input may be a bit string but will usually be a byte string. The following equivalence will hold for any byte string str and integer $\ell \geq 1$:

$$\text{SHAKE256}(\text{str}, 8\ell) = \text{SHAKE256}(\text{BytesToBits}(\text{str}), 8\ell).$$

In addition to using a mostly byte-oriented variant of the API defined in FIPS 202 for SHAKE256 and SHAKE128, this standard sometimes makes use of the incremental API defined in SP 800-185 [25]. This API consists of three functions for each variant of SHAKE. These functions can be used to absorb a sequence of arbitrary-length strings and squeeze a sequence of arbitrary-length strings. These functions perform buffering to handle any incomplete data blocks while absorbing or squeezing. For example, for SHAKE256:

- ctx $\leftarrow$ SHAKE256.Init()
  Initializes a hash function context.
- ctx $\leftarrow$ SHAKE256.Absorb(ctx, str)
  Injects data to be used in the absorbing phase of SHAKE256 and updates context ctx.

- $(\text{ctx}, \text{out}) \leftarrow \text{SHAKE256.Squeeze}(\text{ctx}, 8\ell)$
  Extracts $\ell$ output bytes produced during the squeezing phase of SHAKE256 and updates context ctx.

While the above functions are specified in terms of the Keccak-$f$ permutation rather than the eXtendable-Output Function (XOF), SHAKE256, they are defined so that any series of commands of the following form:

1. $\text{ctx} \leftarrow \text{SHAKE256.Init}()$
2. **For** $i = 1$ to $m$: $\text{ctx} \leftarrow \text{SHAKE256.Absorb}(\text{ctx}, \text{str}_i)$
3. **For** $j = 1$ to $k$: $(\text{ctx}, \text{out}_j) \leftarrow \text{SHAKE256.Squeeze}(\text{ctx}, 8b_j)$
4. $\text{output} \leftarrow \text{out}_1 || \ ... \ || \text{out}_k$

will yield the same output as a single SHAKE256 call:

$$\text{output} \leftarrow \text{SHAKE256}(\text{str}_1 || \ ... \ || \text{str}_m, 8b_1 + ... + 8b_k).$$

This equivalence holds whether or not $|\text{str}_i|$ and $b_j$ are multiples of the SHAKE256 block length.

Since all outputs of SHAKE128 and SHAKE256 in this document give a whole number of bytes, the wrapper functions H and G are defined as follows:

1. $\text{H}(\text{str}, \ell) = \text{SHAKE256}(\text{str}, 8\ell)$
2. $\text{G}(\text{str}, \ell) = \text{SHAKE128}(\text{str}, 8\ell)$
3. $\text{H.Init}() = \text{SHAKE256.Init}()$
4. $\text{G.Init}() = \text{SHAKE128.Init}()$
5. $\text{H.Absorb}(\text{ctx}, \text{str}) = \text{SHAKE256.Absorb}(\text{ctx}, \text{str})$
6. $\text{G.Absorb}(\text{ctx}, \text{str}) = \text{SHAKE128.Absorb}(\text{ctx}, \text{str})$
7. $\text{H.Squeeze}(\text{ctx}, \ell) = \text{SHAKE256.Squeeze}(\text{ctx}, 8\ell)$
8. $\text{G.Squeeze}(\text{ctx}, \ell) = \text{SHAKE128.Squeeze}(\text{ctx}, 8\ell)$

In addition to SHAKE128 and SHAKE256, HashML-DSA.Sign and HashML-DSA.Verify may call other **approved** hash functions for pre-hashing. The pseudocode in this standard also treats these functions as returning a byte string as output while supporting either a bit string or a byte string as input. Here, it should be noted that the hash functions defined in [8] use different rules (i.e., big-endian ordering) to relate bits, bytes, and words.

# 4. Parameter Sets

**Table 1. ML-DSA parameter sets**

| Parameters (see Sections 6.1 and 6.2 of this document) | Values assigned by each parameter set | | |
|---|---|---|---|
| | ML-DSA-44 | ML-DSA-65 | ML-DSA-87 |
| $q$ - modulus [see §6.1] | 8380417 | 8380417 | 8380417 |
| $\zeta$ - a $512$th root of unity in $\mathbb{Z}_q$ [see §7.5] | 1753 | 1753 | 1753 |
| $d$ - # of dropped bits from $\mathbf{t}$ [see §6.1] | 13 | 13 | 13 |
| $\tau$ - # of $\pm 1$'s in polynomial $c$ [see §6.2] | 39 | 49 | 60 |
| $\lambda$ - collision strength of $\tilde{c}$ [see §6.2] | 128 | 192 | 256 |
| $\gamma_1$ - coefficient range of $\mathbf{y}$ [see §6.2] | $2^{17}$ | $2^{19}$ | $2^{19}$ |
| $\gamma_2$ - low-order rounding range [see §6.2] | $(q-1)/88$ | $(q-1)/32$ | $(q-1)/32$ |
| $(k, \ell)$ - dimensions of $\mathbf{A}$ [see §6.1] | (4,4) | (6,5) | (8,7) |
| $\eta$ - private key range [see §6.1] | 2 | 4 | 2 |
| $\beta = \tau \cdot \eta$ [see §6.2] | 78 | 196 | 120 |
| $\omega$ - max # of 1's in the hint $\mathbf{h}$ [see §6.2] | 80 | 55 | 75 |
| Challenge entropy $\log_2 \binom{256}{\tau} + \tau$ [see §6.2] | 192 | 225 | 257 |
| Repetitions (see explanation below) | 4.25 | 5.1 | 3.85 |
| Claimed security strength | Category 2 | Category 3 | Category 5 |

Three ML-DSA parameter sets are included in Table 1. Each parameter set assigns values for all of the parameters used in the ML-DSA algorithms for key generation, signing, and verification. For informational purposes, some parameters used in the analysis of these algorithms are also included in the table. In particular, "repetitions" refers to the expected number of repetitions of the main loop in the signing algorithm from eq. 5 in [5]. The names of the parameter sets are of the form "ML-DSA-$k\ell$," where $(k, \ell)$ are the dimensions of the matrix $\mathbf{A}$.

These parameter sets were designed to meet certain security strength categories defined by NIST in its original Call for Proposals [26]. These security strength categories are explained further in SP 800-57, Part 1 [9].

Using this approach, security strength is not described by a single number, such as "128 bits of security." Instead, each ML-DSA 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-DSA 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 [27].

Concretely, the parameter set ML-DSA-44 is claimed to be in security strength category 2, ML-DSA-65 is claimed to be in category 3, and ML-DSA-87 is claimed to be in category 5 [6]. For additional discussion of the security strength of MLWE-based cryptosystems, see [28].

The sizes of keys and signatures that correspond to each parameter set are given in Table 2. Certain optimizations are possible when storing ML-DSA public and private keys. If additional space is available, one can precompute and store $\hat{\mathbf{A}}$ to speed up signing and verifying. Alternatively, if one wants to reduce

the space needed for the private key, one can store only the 32-byte seed $\xi$, which is sufficient to generate the other parts of the private key. For additional details, see Section 3.1 in [6].

**Table 2. Sizes (in bytes) of keys and signatures of ML-DSA**

| | Private Key | Public Key | Signature Size |
|---|---|---|---|
| ML-DSA-44 | 2560 | 1312 | 2420 |
| ML-DSA-65 | 4032 | 1952 | 3309 |
| ML-DSA-87 | 4896 | 2592 | 4627 |

# 5. External Functions

The signing, verifying, and key generation functions can be split into "external" and "internal" components to simplify APIs and Cryptographic Algorithm Validation Program (CAVP) testing. The external components generate randomness and perform various checks before calling their internal counterparts. The internal components are deterministic and can assume that the external components did not encounter error conditions.

The distinction between external and internal functions also simplifies the presentation of algorithms for signing and verification by grouping the operations that are shared between ML-DSA.Sign and HashML-DSA.Sign in ML-DSA.Sign_internal and grouping the operations that are shared between ML-DSA.Verify and HashML-DSA.Verify in ML-DSA.Verify_internal.

## 5.1 ML-DSA Key Generation

The key generation algorithm ML-DSA.KeyGen takes no input and outputs a public key and a private key, which are both encoded as byte strings.

The algorithm uses an **approved** RBG to generate a 256-bit (32-byte) random seed $\xi$ that is given as input to ML-DSA.KeyGen_internal (Algorithm 6), which produces the public and private keys.

**Algorithm 1** ML-DSA.KeyGen()

*Generates a public-private key pair.*

**Output**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $\xi \leftarrow \mathbb{B}^{32}$ ▷ choose random seed
2: **if** $\xi =$ NULL **then**
3: **return** $\perp$ ▷ return an error indication if random bit generation failed
4: **end if**
5: **return** ML-DSA.KeyGen_internal ($\xi$)

## 5.2 ML-DSA Signing

The signing algorithm ML-DSA.Sign (Algorithm 2) takes a private key, a message, and a context string as input[4]. It outputs a signature that is encoded as a byte string.

For the default "hedged" version of ML-DSA signing, the algorithm (at line 5) uses an **approved** RBG to generate a 256-bit (32-byte) random seed $rnd$. If the deterministic variant is desired, then $rnd$ is set to the fixed zero string $\{0\}^{32}$. The value $rnd$, the private key, and the encoded message are input to ML-DSA.Sign_internal (Algorithm 7), which produces the signature.

[4]By default, the context is the empty string, though applications may specify the use of a non-empty context string.

**Algorithm 2** ML-DSA.Sign$(sk, M, ctx)$

*Generates an ML-DSA signature.*

**Input**: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen }(2\eta)+dk)}$, message $M \in \{0,1\}^*$,
context string $ctx$ (a byte string of 255 or fewer bytes).
**Output**: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.

1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ return an error indication if the context string is too long
3: **end if**
4:
5: $rnd \leftarrow \mathbb{B}^{32}$ ▷ for the optional deterministic variant, substitute $rnd \leftarrow \{0\}^{32}$
6: **if** $rnd =$ NULL **then**
7: **return** $\perp$ ▷ return an error indication if random bit generation failed
8: **end if**
9:
10: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1) \,\|\, $IntegerToBytes$(|ctx|,1) \,\|\, ctx) \,\|\, M$
11: $\sigma \leftarrow$ ML-DSA.Sign_internal$(sk, M', rnd)$
12: **return** $\sigma$

## 5.3 ML-DSA Verifying

The verification algorithm ML-DSA.Verify (Algorithm 3) takes a public key, a message, a signature, and a context string as input. The public key, signature, and context string are all encoded as byte strings, while the message is a bit string. ML-DSA.Verify outputs a Boolean value that is true if the signature is valid with respect to the message and the public key and false if the signature is invalid. The verification is accomplished by calling ML-DSA.Verify_internal (Algorithm 8) with the public key, the encoded message, and the signature.

**Algorithm 3** ML-DSA.Verify$(pk, M, \sigma, ctx)$

*Verifies a signature $\sigma$ for a message $M$.*

**Input**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$, message $M \in \{0,1\}^*$,
signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$,
context string $ctx$ (a byte string of 255 or fewer bytes).
**Output**: Boolean.

1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ return an error indication if the context string is too long
3: **end if**
4:
5: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1) \,\|\, $IntegerToBytes$(|ctx|,1) \,\|\, ctx) \,\|\, M$
6: **return** ML-DSA.Verify_internal$(pk, M', \sigma)$

## 5.4 Pre-Hash ML-DSA

For some cryptographic modules that generate ML-DSA signatures, hashing the message in step 6 of ML-DSA.Sign_internal may result in unacceptable performance if the message $M$ is large. For example,

the platform may require hardware support for hashing to achieve acceptable performance but lack hardware support for SHAKE256 specifically. For some use cases, this may be addressed by signing a digest of the message along with some domain separation information rather than signing the message directly. This version of ML-DSA is known as "pre-hash" ML-DSA or HashML-DSA . In general, the "pure" ML-DSA version is preferred.

While key generation for HashML-DSA is the same as for ML-DSA, it is not the same for the signing algorithm HashML-DSA.Sign or the verification algorithm HashML-DSA.Verify. Like ML-DSA, the signing algorithm of HashML-DSA takes the content to be signed, the private key, and a context as input, as well as a hash function or XOF that is to be used to pre-hash the content to be signed. The context string has a maximum length of 255 bytes. By default, the context is the empty string, though applications may specify the use of a non-empty context string.

The identifier for a signature (e.g., the object identifier [OID]) **should** indicate whether the signature is a ML-DSA signature or a pre-hash HashML-DSA signature. In the case of pre-hash signatures, the identifier **should** also indicate the hash function or XOF used to compute the pre-hash. [5] While a single key pair may be used for both ML-DSA and HashML-DSA signatures, it is recommended that each key pair only be used for one version or the other. If a non-empty context string is to be used, this should either be indicated by the signature's identifier or by the application with which the signature is being used.

If the default "hedged" variant of is used, the 32-byte random value $rnd$ **shall** be generated by the cryptographic module that generates the signature (i.e., that runs ML-DSA.Sign_internal). However, all other steps of signing may be performed outside of the cryptographic module that generates the signature. In the case of pre-hashing, the hash or XOF of the content to be signed must be computed within a FIPS 140-validated cryptographic module, but it may be a different cryptographic module than the one that generates the signature.

If the content to be signed is large, hashing of the content is often performed at the application level. For example, in the Cryptographic Message Syntax [29], a digest of the content may be computed, and that digest is signed along with other attributes. If the content is not hashed at the application level, the pre-hash version of ML-DSA signing may be used.

In order to maintain the same level of security strength when the content is hashed at the application level or using HashML-DSA , the digest that is signed needs to be generated using an **approved** hash function or XOF (e.g., from FIPS 180 [8] or FIPS 202 [7]) that provides at least $\lambda$ bits of classical security strength against both collision and second preimage attacks [7, Table 4].[6] The verification of a signature that is created in this way will require the verify function to generate a digest from the message in the same way to be used as input for the verification function.

### 5.4.1 HashML-DSA Signing and Verifying

In the HashML-DSA version, the message input to ML-DSA.Sign_internal is the result of applying either a hash function or a XOF to the content to be signed. The output of the hash function or XOF is prepended by a one-byte domain separator, one byte that indicates the length of the context string, the context string, and the distinguished encoding rules (DER) encoding of the hash function or XOF's OID. The domain separator has a value of one for "pre-hash" signing. The DER encoding of the OID includes the tag and length.

[5] In the case of a XOF this would also include the length of the output from the XOF.

[6] Obtaining at least $\lambda$ bits of classical security strength against collision attacks requires that the digest to be signed be at least $2\lambda$ bits in length.

Algorithm 4 shows the DER encodings of the OIDs for SHA-256, SHA-512, and SHAKE128. However, it may be used with other hash functions or XOFs.

**Algorithm 4** HashML-DSA.Sign$(sk, M, ctx, \text{PH})$

*Generate a "pre-hash" ML-DSA signature.*

**Input**: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen }(2\eta)+dk)}$, message $M \in \{0,1\}^*$, context string $ctx$ (a byte string of 255 or fewer bytes), pre-hash function PH.
**Output**: ML-DSA signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.

**if** $|ctx| > 255$ **then**
  **return** $\perp$ ▷ return an error indication if the context string is too long
**end if**

$rnd \leftarrow \mathbb{B}^{32}$ ▷ for the optional deterministic variant, substitute $rnd \leftarrow \{0\}^{32}$
**if** $rnd =$ NULL **then**
  **return** $\perp$ ▷ return an error indication if random bit generation failed
**end if**

**switch** PH **do**
  **case** SHA-256:
    OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01 ▷ 2.16.840.1.101.3.4.2.1
    $\text{PH}_M \leftarrow \text{SHA256}(M)$
  **case** SHA-512:
    OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03 ▷ 2.16.840.1.101.3.4.2.3
    $\text{PH}_M \leftarrow \text{SHA512}(M)$
  **case** SHAKE128:
    OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0B ▷ 2.16.840.1.101.3.4.2.11
    $\text{PH}_M \leftarrow \text{SHAKE128}(M, 256)$
  **case** ...
    ...
**end switch**
$M' \leftarrow \text{BytesToBits}(\text{IntegerToBytes}(1,1) \,\|\, \text{IntegerToBytes}(|ctx|,1) \,\|\, ctx \,\|\, \text{OID} \,\|\, \text{PH}_M)$
$\sigma \leftarrow \text{ML-DSA.Sign\_internal}(sk, M', rnd)$
**return** $\sigma$

Algorithm 5 presents the signature verification for HashML-DSA . This function constructs $M'$ in the same way as Algorithm 4 and passes the resulting $M'$ to Algorithm ML-DSA.Verify_internal for verification. As with the pre-hash signature generation, $M'$ may be constructed outside of the cryptographic module that performs ML-DSA.Verify_internal. However, in the case of HashML-DSA , the hash or XOF of the content must be computed within a FIPS 140-validated cryptographic module, which may be a different cryptographic module than the one that performs ML-DSA.Verify_internal.

As noted in Section 5.4, the identifier associated with the signature should indicate whether ML-DSA or

the pre-hash version HashML-DSA of signature verification should be used, as well as the hash function or XOF to be used to compute the pre-hash. A non-empty context string should be used in verification if one is specified either in the signature's identifier or by the application with which the signature is being used.

**Algorithm 5** HashML-DSA.Verify$(pk, M, \sigma, ctx, \text{PH})$

*Verifies a pre-hash HashML-DSA signature.*

**Input**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$, message $M \in \{0,1\}^*$,
signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$,
context string $ctx$ (a byte string of 255 or fewer bytes), pre-hash function PH.
**Output**: Boolean.

1: **if** $|ctx| > 255$ **then**
2: **return** false
3: **end if**
4:
5: **switch** PH **do**
6: **case** SHA-256:
7: OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01 ▷ 2.16.840.1.101.3.4.2.1
8: $\text{PH}_M \leftarrow \text{SHA256}(M)$
9: **case** SHA-512:
10: OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03 ▷ 2.16.840.1.101.3.4.2.3
11: $\text{PH}_M \leftarrow \text{SHA512}(M)$
12: **case** SHAKE128:
13: OID $\leftarrow$ 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0B ▷ 2.16.840.1.101.3.4.2.11
14: $\text{PH}_M \leftarrow \text{SHAKE128}(M, 256)$
15: **case** ...
16: ...
17: **end switch**
18: $M' \leftarrow \text{BytesToBits}(\text{IntegerToBytes}(1,1) \,\|\, \text{IntegerToBytes}(|ctx|,1) \,\|\, ctx \,\|\, \text{OID} \,\|\, \text{PH}_M)$
19: **return** ML-DSA.Verify_internal$(pk, M', \sigma)$

# 6. Internal Functions

This section describes the functions for ML-DSA key generation, signature generation, and signature verification. Where randomness is required, the random values are provided as inputs to the functions. The interfaces specified in this section will be used when testing of ML-DSA implementations is performed through the CAVP.

Other than for testing purposes, the interfaces for key generation and signature generation specified in this section **should not** be made available to applications, as any random values required for key generation and signature generation **shall** be generated by the cryptographic module. Section 5 provides guidance on the interfaces to be made available to applications.[7]

## 6.1 ML-DSA Key Generation (Internal)

The internal key generation algorithm ML-DSA.KeyGen_internal takes a 32-byte random seed as input and outputs a public key and a private key that are both encoded as byte strings.

The seed $\xi$ is expanded as needed using an XOF (i.e., a byte-variant of SHAKE256) denoted by H to produce other random values.[8] In particular:

- A $32$-byte public random seed $\rho$. Using this seed, a polynomial matrix $\mathbf{A} \in R_q^{k \times \ell}$ is pseudorandomly sampled[9] from $R_q^{k \times \ell}$.
- A $64$-byte private random seed $\rho'$. Using this seed, the polynomial vectors $\mathbf{s}_1 \in R_q^{\ell}$ and $\mathbf{s}_2 \in R_q^{k}$ are pseudorandomly sampled from the subset of polynomial vectors whose coordinate polynomials have short coefficients (i.e., in the range $[-\eta, \eta]$).
- A $32$-byte private random seed $K$ for use during signing.

The core cryptographic operation computes the public value

$$\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2.$$

The vector $\mathbf{t}$ together with the matrix $\mathbf{A}$ may be considered an expanded form of the public key. The vector $\mathbf{t}$ is compressed in the actual public key by dropping the $d$ least significant bits from each coefficient, thus producing the polynomial vector $\mathbf{t}_1$. This compression is an optimization for performance, not security. The low-order bits of $\mathbf{t}$ can be reconstructed from a small number of signatures and, therefore, need not be regarded as secret.

The ML-DSA public key $pk$ is a byte encoding of the public random seed $\rho$ and the compressed polynomial vector $\mathbf{t}_1$.

The ML-DSA private key $sk$ is a byte encoding of the public random seed $\rho$, a private random seed $K$ for use during signing, a $64$-byte hash $tr$ of the public key for use during signing, the secret polynomial

[7] In some cases, it is permissible to modify the format of the private key in these interfaces (see Sections 4 and 3.6.3.)

[8] Single-byte encodings of the parameters $k$ and $\ell$ are included in the XOF input for domain separation. For implementations that use the seed in place of the private key, this ensures that the expansion will produce an unrelated key if the seed is mistakenly expanded using a parameter set other than the one originally intended.

[9] More precisely, since only the NTT form of $\mathbf{A}$, $\hat{\mathbf{A}} \in T_q^{k \times \ell} = \text{NTT}(\mathbf{A})$ is needed in subsequent calculations, the code actually computes $\hat{\mathbf{A}}$ as a pseudorandom sample over $T_q^{k \times \ell}$, and the sampling of $\mathbf{A} = \text{NTT}^{-1}(\hat{\mathbf{A}})$ is only implicit (i.e., it could be computed but is not).

vectors $\mathbf{s}_1$ and $\mathbf{s}_2$, and a polynomial vector $\mathbf{t}_0$ encoding the $d$ least significant bits of each coefficient of the uncompressed public-key polynomial $\mathbf{t}$.

**Algorithm 6** ML-DSA.KeyGen_internal($\xi$)

*Generates a public-private key pair from a seed.*

**Input**: Seed $\xi \in \mathbb{B}^{32}$

**Output**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \mathsf{H}(\xi||\mathsf{IntegerToBytes}(k,1)||\mathsf{IntegerToBytes}(\ell,1),128)$
2: ▷ expand seed
3: $\hat{\mathbf{A}} \leftarrow \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \mathsf{ExpandS}(\rho')$
5: $\mathbf{t} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \mathsf{Power2Round}(\mathbf{t})$ ▷ compress $\mathbf{t}$
7: ▷ PowerTwoRound is applied componentwise (see explanatory text in Section 7.4)
8: $pk \leftarrow \mathsf{pkEncode}(\rho, \mathbf{t}_1)$
9: $tr \leftarrow \mathsf{H}(pk, 64)$
10: $sk \leftarrow \mathsf{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ $K$ and $tr$ are for use in signing
11: **return** $(pk, sk)$

## 6.2 ML-DSA Signing (Internal)

ML-DSA.Sign_internal (Algorithm 7) outputs a signature encoded as a byte string. It takes a private key $sk$ encoded as a byte string, a formatted message $M'$ encoded as a bit string, and a 32-byte string $rnd$ as input. There are two ways that a signing algorithm can use ML-DSA.Sign_internal: "hedged" and "deterministic." The default "hedged" variants of ML-DSA.Sign and HashML-DSA.Sign use a fresh random value for $rnd$, while the optional deterministic variants use the constant byte string $\{0\}^{32}$ (see Section 3).

In both variants, the signer first extracts the following from the private key: the public random seed $\rho$, the 32-byte private random seed $K$, the 64-byte hash of the public key $tr$, the secret polynomial vectors $\mathbf{s}_1$ and $\mathbf{s}_2$, and the polynomial vector $\mathbf{t}_0$ encoding the $d$ least significant bits of each coefficient of the uncompressed public-key polynomial $\mathbf{t}$. $\rho$ is then expanded to the same matrix $\mathbf{A}$ as in key generation.

Before the message $M$ is signed, it is concatenated with the public-key hash $tr$ and hashed down to a 64-byte message representative $\mu$ using H.

The signer produces an additional 64-byte seed $\rho''$ for private randomness during each signing operation. $\rho''$ is computed as $\rho'' \leftarrow \mathsf{H}(K||rnd||\mu, 64)$. In the default hedged variant, $rnd$ is the output of an RBG, while in the deterministic variant, $rnd$ is a 32-byte string that consists entirely of zeros. This is the only difference between the deterministic and hedged variant of ML-DSA.Sign.

The main part of the signing algorithm consists of a rejection sampling loop in which each iteration of the loop either produces a valid signature or an invalid signature whose release would leak information about the private key. The loop is repeated until a valid signature is produced, which can then be encoded as a byte string and output.[10] The rejection sampling loop follows the Fiat-Shamir With Aborts paradigm [10]

[10] Implementations may limit the number of iterations in this loop to not exceed a finite maximum value. If this

and (aside from the rejection step) is similar in structure to Schnorr signatures [30] (e.g., EdDSA [31]). The signer first produces a "commitment" $\mathbf{w}_1$ and then pseudorandomly derives a "challenge" $c$ from $\mathbf{w}_1$ and the message representative $\mu$. Finally, the signer computes a response $\mathbf{z}$.

In more detail, the main computations involved in the rejection sampling loop are as follows:

- Using the ExpandMask function (Algorithm 34), the seed $\rho''$, and a counter $\kappa$, a polynomial vector $\mathbf{y} \in R_q^\ell$ is pseudorandomly sampled from the subset of polynomial vectors whose coefficients are moderately short (i.e., in the range $[-\gamma_1 + 1, \gamma_1]$).
- From $\mathbf{y}$, the signer computes the commitment $\mathbf{w}_1$ by computing $\mathbf{w} = \mathbf{A}\mathbf{y}$ and then rounding to a nearby multiple of $2\gamma_2$ using HighBits (Algorithm 37).
- $\mathbf{w}_1$ and $\mu$ are concatenated and hashed to produce the commitment hash $\tilde{c}$. This uses the function w1Encode (Algorithm 28). The byte string $\tilde{c}$ is used to pseudorandomly sample a polynomial $c \in R_q$ that has coefficients in $\{-1, 0, 1\}$ and Hamming weight $\tau$. The sampling is done with the function SampleInBall (Algorithm 29).[11]
- The signer computes the response $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ and performs various validity checks. If any of the checks fails, the signer will continue the rejection sampling loop.
- If the checks pass, the signer can compute a hint polynomial $\mathbf{h}$, which will allow the verifier to reconstruct $\mathbf{w}_1$ using the compressed public key along with the other components of the signature. This uses the function MakeHint (Algorithm 39). The signer will then output the final signature, which is a byte encoding of the commitment hash $\tilde{c}$, the response $\mathbf{z}$, and the hint $\mathbf{h}$.

In addition, there is an alternative way of implementing the validity checks on $\mathbf{z}$ and the computation of $\mathbf{h}$, which is described in Section 5.1 of [6]. This method may also be used in implementations of ML-DSA.

In Algorithm 7, variables are sometimes used to store products to avoid recomputing them later in the signing algorithm. These precomputed products are denoted in the pseudocode by a pair of double angle brackets enclosing the variables being multiplied (e.g., $\langle\langle c\mathbf{s}_1 \rangle\rangle$).

option is used and the maximum number of iterations is exceeded without producing a valid signature, the signing algorithm **shall** return a constant that represents an error and no other output, destroying the results of the unsuccessful signing attempts. See Appendix C.

[11] The length of $\tilde{c}$ is determined by the desired security with respect to the "message-bound signatures" property described in [14]. Here, a length of $\lambda/4$ bytes or equivalently $2\lambda$ bits is required for $\lambda$ bits of classical security.

**Algorithm 7** ML-DSA.Sign_internal$(sk, M', rnd)$

*Deterministic algorithm to generate a signature for a formatted message $M'$.*

**Input**: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen }(2\eta)+dk)}$, formatted message $M' \in \{0,1\}^*$, and per message randomness or dummy variable $rnd \in \mathbb{B}^{32}$.
**Output**: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.

$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow$ skDecode$(sk)$
$\hat{\mathbf{s}}_1 \leftarrow$ NTT$(\mathbf{s}_1)$
$\hat{\mathbf{s}}_2 \leftarrow$ NTT$(\mathbf{s}_2)$
$\hat{\mathbf{t}}_0 \leftarrow$ NTT$(\mathbf{t}_0)$
$\hat{\mathbf{A}} \leftarrow$ ExpandA$(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$\mu \leftarrow$ H(BytesToBits$(tr)||M', 64)$ ▷ message representative that may optionally be computed in a different cryptographic module
$\rho'' \leftarrow$ H$(K||rnd||\mu, 64)$ ▷ compute private random seed
$\kappa \leftarrow 0$ ▷ initialize counter $\kappa$
$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
**while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ rejection sampling loop
    $\mathbf{y} \in R_q^\ell \leftarrow$ ExpandMask$(\rho'', \kappa)$
    $\mathbf{w} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{y}))$
    $\mathbf{w}_1 \leftarrow$ HighBits$(\mathbf{w})$ ▷ signer's commitment
        ▷ HighBits is applied componentwise (see explanatory text in Section 7.4)
    $\tilde{c} \leftarrow$ H$(\mu||\text{w1Encode}(\mathbf{w}_1), \lambda/4)$ ▷ commitment hash
    $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ verifier's challenge
    $\hat{c} \leftarrow$ NTT$(c)$
    $\langle\langle c\mathbf{s}_1 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$
    $\langle\langle c\mathbf{s}_2 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2)$
    $\mathbf{z} \leftarrow \mathbf{y} + \langle\langle c\mathbf{s}_1 \rangle\rangle$ ▷ signer's response
    $\mathbf{r}_0 \leftarrow$ LowBits$(\mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle)$
        ▷ LowBits is applied componentwise (see explanatory text in Section 7.4)
    **if** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ **or** $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ validity checks
    **else**
        $\langle\langle c\mathbf{t}_0 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$
        $\mathbf{h} \leftarrow$ MakeHint$(-\langle\langle c\mathbf{t}_0 \rangle\rangle, \mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle + \langle\langle c\mathbf{t}_0 \rangle\rangle)$ ▷ Signer's hint
            ▷ MakeHint is applied componentwise (see explanatory text in Section 7.4)
        **if** $||\langle\langle c\mathbf{t}_0 \rangle\rangle||_\infty \geq \gamma_2$ **or** the number of 1's in $\mathbf{h}$ is greater than $\omega$, **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
        **end if**
    **end if**
    $\kappa \leftarrow \kappa + \ell$ ▷ increment counter
**end while**
$\sigma \leftarrow$ sigEncode$(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h})$
**return** $\sigma$

## 6.3 ML-DSA Verifying (Internal)

The algorithm ML-DSA.Verify_internal (Algorithm 8) takes a public key $pk$ encoded as a byte string, a message $M$ encoded as a bit string, and a signature $\sigma$ encoded as a byte string as input. No randomness is

required for ML-DSA.Verify_internal. It produces a Boolean value (i.e., a value that is true if the signature is valid with respect to the message and public key and false if the signature is invalid) as output. Algorithm 8 specifies the lengths of the signature $\sigma$ and the public key $pk$ in terms of the parameters described in Table 1. If an implementation of ML-DSA.Verify_internal can accept inputs for $\sigma$ or $pk$ of any other length, it **shall** return false whenever the length of either of these inputs differs from its specified length.

The verifier first extracts the public random seed $\rho$ and the compressed polynomial vector $\mathbf{t}_1$ from the public key $pk$ and then extracts the signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$ from the signature $\sigma$. The verifier may find that the hint was not properly byte-encoded, denoted by the symbol "$\perp$," in which case the verification algorithm will immediately return false to indicate that the signature is invalid.

Assuming that the signature is successfully extracted from its byte encoding, the verifier pseudorandomly derives $\mathbf{A}$ from $\rho$, as is done in key generation and signing, and creates a message representative $\mu$ by hashing the concatenation of $tr$ (i.e., the hash of the public key $pk$) and the message $M$. The verifier then attempts to reconstruct the signer's commitment (i.e., the polynomial vector $\mathbf{w}_1$) from the public key $pk$ and the signature $\sigma$. In ML-DSA.Sign_internal, $\mathbf{w}_1$ is computed by rounding $\mathbf{w} = \mathbf{Ay}$. In ML-DSA.Verify_internal, the reconstructed value of $\mathbf{w}_1$ is called $\mathbf{w}'_1$ since it may have been computed in a different way if the signature is invalid. This $\mathbf{w}'_1$ is computed through the following process:

- Derive the challenge polynomial $c$ from the signer's commitment hash $\tilde{c}$, just as similarly is done in ML-DSA.Sign_internal.
- Use the signer's response $\mathbf{z}$ to compute

  $$\mathbf{w}'_{\text{Approx}} = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d.$$

  Assuming the signature was computed correctly, as in ML-DSA.Sign_internal, it follows that

  $$\mathbf{w} = \mathbf{Ay} = \mathbf{Az} - c\mathbf{t} + c\mathbf{s}_2 \approx \mathbf{w}'_{\text{Approx}} = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$$

  because $c$ and $\mathbf{s}_2$ have small coefficients, and $\mathbf{t}_1 \cdot 2^d \approx \mathbf{t}$ .
- Use the signer's hint $\mathbf{h}$ to obtain $\mathbf{w}'_1$ from $\mathbf{w}'_{\text{Approx}}$.

Finally, the verifier checks that the signer's response $\mathbf{z}$ and the signer's hint $\mathbf{h}$ are valid and that the reconstructed $\mathbf{w}'_1$ is consistent with the signer's commitment hash $\tilde{c}$. More precisely, the verifier checks that all of the coefficients of $\mathbf{z}$ are sufficiently small (i.e., in the range $(-(\gamma_1 - \beta), \gamma_1 - \beta)$), $\mathbf{h}$ contains no more than $\omega$ nonzero coefficients, and $\tilde{c}$ matches the hash $\tilde{c}'$ of the message representative $\mu$ concatenated with $\mathbf{w}'_1$ (represented as a byte string). If all of these checks succeed, then ML-DSA.Verify_internal returns true. Otherwise, it returns false.

**Algorithm 8** ML-DSA.Verify_internal$(pk, M', \sigma)$

*Internal function to verify a signature $\sigma$ for a formatted message $M'$.*

**Input**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$ and message $M' \in \{0,1\}^*$.
**Input**: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.
**Output**: Boolean

1: $(\rho, \mathbf{t}_1) \leftarrow$ pkDecode$(pk)$
2: $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow$ sigDecode$(\sigma)$ ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
3: **if** $\mathbf{h} = \perp$ **then return** false ▷ hint was not properly encoded
4: **end if**
5: $\hat{\mathbf{A}} \leftarrow$ ExpandA$(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
6: $tr \leftarrow$ H$(pk, 64)$
7: $\mu \leftarrow$ (H(BytesToBits$(tr)||M', 64)$) ▷ message representative that may optionally be computed in a different cryptographic module
8: $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
9: $\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
10: $\mathbf{w}'_1 \leftarrow$ UseHint$(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ reconstruction of signer's commitment
11: ▷ UseHint is applied componentwise (see explanatory text in Section 7.4)
12: $\tilde{c}' \leftarrow$ H$(\mu||\text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ ▷ hash it; this should match $\tilde{c}$
13: **return** $[[\ ||\mathbf{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$

# 7. Auxiliary Functions

This section provides pseudocode for subroutines utilized by ML-DSA, including functions for data-type conversions, arithmetic, and sampling.

## 7.1 Conversion Between Data Types

While the primary data type in ML-DSA is a byte string, other data types are used as well. The goal in this section is to construct procedures for translating between the various algebraic objects defined in Section 2.3. Algorithms 9–13 are intermediate procedures for converting between bit strings, byte strings, and integers.

**Algorithm 9** IntegerToBits$(x, \alpha)$

*Computes a base-2 representation of $x$ mod $2^{\alpha}$ using little-endian order.*

**Input**: A nonnegative integer $x$ and a positive integer $\alpha$.
**Output**: A bit string $y$ of length $\alpha$.

$x' \leftarrow x$
**for** $i$ **from** $0$ **to** $\alpha - 1$ **do**
    $y[i] \leftarrow x' \bmod 2$
    $x' \leftarrow \lfloor x'/2 \rfloor$
**end for**
**return** $y$

**Algorithm 10** BitsToInteger$(y, \alpha)$

*Computes the integer value expressed by a bit string using little-endian order.*

**Input**: A positive integer $\alpha$ and a bit string $y$ of length $\alpha$.
**Output**: A nonnegative integer $x$.

$x \leftarrow 0$
**for** $i$ **from** $1$ **to** $\alpha$ **do**
    $x \leftarrow 2x + y[\alpha - i]$
**end for**
**return** $x$

**Algorithm 11** IntegerToBytes$(x, \alpha)$

*Computes a base-256 representation of $x$ mod $256^{\alpha}$ using little-endian order.*

**Input**: A nonnegative integer $x$ and a positive integer $\alpha$.
**Output**: A byte string $y$ of length $\alpha$.

$x' \leftarrow x$
**for** $i$ **from** $0$ **to** $\alpha - 1$ **do**
    $y[i] \leftarrow x' \bmod 256$
    $x' \leftarrow \lfloor x'/256 \rfloor$
**end for**
**return** $y$

**Algorithm 12** BitsToBytes$(y)$

*Converts a bit string into a byte string using little-endian order.*

**Input**: A bit string $y$ of length $\alpha$.
**Output**: A byte string $z$ of length $\lceil \alpha/8 \rceil$.

$z \in \mathbb{B}^{\lceil \alpha/8 \rceil} \leftarrow 0^{\lceil \alpha/8 \rceil}$
**for** $i$ **from** $0$ **to** $\alpha - 1$ **do**
    $z[\lfloor i/8 \rfloor] \leftarrow z[\lfloor i/8 \rfloor] + y[i] \cdot 2^{i \bmod 8}$
**end for**
**return** $z$

**Algorithm 13** BytesToBits$(z)$

*Converts a byte string into a bit string using little-endian order.*

**Input**: A byte string $z$ of length $\alpha$.
**Output**: A bit string $y$ of length $8\alpha$.

$z' \leftarrow z$
**for** $i$ **from** $0$ **to** $\alpha - 1$ **do**
    **for** $j$ **from** $0$ **to** $7$ **do** ▷ convert the byte $z[i]$ into $8$ bits
        $y[8i + j] \leftarrow z'[i] \bmod 2$
        $z'[i] \leftarrow \lfloor z'[i]/2 \rfloor$
    **end for**
**end for**
**return** $y$

Algorithms 14 and 15 translate byte strings into coefficients of polynomials in $R$. CoeffFromThreeBytes uses a 3-byte string to either generate an element of $\{0, 1, \ldots, q-1\}$ or return the blank symbol $\perp$. CoeffFromHalfByte uses an element of $\{0, 1, \ldots, 15\}$ to either generate an element of $\{-\eta, -\eta+1, \ldots, \eta\}$ or return $\perp$. These two procedures will be used in the uniform sampling algorithms RejNTTPoly and RejBoundedPoly, which are discussed in Section 7.3.

**Algorithm 14** CoeffFromThreeBytes$(b_0, b_1, b_2)$

*Generates an element of* $\{0, 1, 2, \ldots, q-1\} \cup \{\perp\}$.

**Input**: Bytes $b_0, b_1, b_2$.
**Output**: An integer modulo $q$ or $\perp$.

$b_2' \leftarrow b_2$
**if** $b_2' > 127$ **then**
    $b_2' \leftarrow b_2' - 128$ ▷ set the top bit of $b_2'$ to zero
**end if**
$z \leftarrow 2^{16} \cdot b_2' + 2^8 \cdot b_1 + b_0$ ▷ $0 \le z \le 2^{23} - 1$
**if** $z < q$ **then return** $z$ ▷ rejection sampling
**else return** $\perp$
**end if**

**Algorithm 15** CoeffFromHalfByte$(b)$

*Let $\eta \in \{2, 4\}$. Generates an element of $\{-\eta, -\eta + 1, \ldots, \eta\} \cup \{\perp\}$.*

**Input**: Integer $b \in \{0, 1, \ldots, 15\}$.
**Output**: An integer between $-\eta$ and $\eta$, or $\perp$.

1: **if** $\eta = 2$ **and** $b < 15$ **then return** $2 - (b \bmod 5)$ ▷ rejection sampling from $\{-2, \ldots, 2\}$
2: **else**
3: **if** $\eta = 4$ **and** $b < 9$ **then return** $4 - b$ ▷ rejection sampling from $\{-4, \ldots, 4\}$
4: **else return** $\perp$
5: **end if**
6: **end if**

Algorithms 16–19 efficiently translate an element $w \in R$ into a byte string and vice versa under the assumption that the coefficients of $w$ are in a restricted range. SimpleBitPack assumes that $w_i \in [0, b]$ for some positive integer $b$ and packs $w$ into a byte string of length $32 \cdot$ bitlen $b$. BitPack allows for the more general restriction $w_i \in [-a, b]$. The BitPack algorithm works by merely subtracting $w$ from the polynomial $\sum_{i=0}^{255} bX^i$.

**Algorithm 16** SimpleBitPack$(w, b)$

*Encodes a polynomial $w$ into a byte string.*

**Input**: $b \in \mathbb{N}$ and $w \in R$ such that the coefficients of $w$ are all in $[0, b]$.
**Output**: A byte string of length $32 \cdot$ bitlen $b$.

1: $z \leftarrow ()$ ▷ set $z$ to the empty bit string
2: **for** $i$ **from** $0$ **to** $255$ **do**
3: $z \leftarrow z||$IntegerToBits$(w_i,$ bitlen $b)$
4: **end for**
5: **return** BitsToBytes$(z)$

**Algorithm 17** BitPack$(w, a, b)$

*Encodes a polynomial $w$ into a byte string.*

**Input**: $a, b \in \mathbb{N}$ and $w \in R$ such that the coefficients of $w$ are all in $[-a, b]$.
**Output**: A byte string of length $32 \cdot$ bitlen $(a + b)$.

1: $z \leftarrow ()$ ▷ set $z$ to the empty bit string
2: **for** $i$ **from** $0$ **to** $255$ **do**
3: $z \leftarrow z||$IntegerToBits$(b - w_i,$ bitlen $(a + b))$
4: **end for**
5: **return** BitsToBytes$(z)$

SimpleBitUnpack and BitUnpack are used to decode the byte strings produced by the above functions. For some choices of $a$ and $b$, there exist malformed byte strings that will cause SimpleBitUnpack and BitUnpack to output polynomials whose coefficients are not in the ranges $[0, b]$ and $[-a, b]$, respectively. This can be a concern when running SimpleBitUnpack and BitUnpack on inputs that may come from an untrusted source.

**Algorithm 18** SimpleBitUnpack$(v, b)$

*Reverses the procedure* SimpleBitPack*.*

**Input**: $b \in \mathbb{N}$ and a byte string $v$ of length $32 \cdot$ bitlen $b$.
**Output**: A polynomial $w \in R$ with coefficients in $[0, 2^c - 1]$, where $c =$ bitlen $b$.
When $b + 1$ is a power of 2, the coefficients are in $[0, b]$.

1: $c \leftarrow$ bitlen $b$
2: $z \leftarrow$ BytesToBits$(v)$
3: **for** $i$ **from** $0$ **to** $255$ **do**
4: $\quad w_i \leftarrow$ BitsToInteger$((z[ic], z[ic+1], \ldots z[ic+c-1]), c)$
5: **end for**
6: **return** $w$

**Algorithm 19** BitUnpack$(v, a, b)$

*Reverses the procedure* BitPack*.*

**Input**: $a, b \in \mathbb{N}$ and a byte string $v$ of length $32 \cdot$ bitlen $(a + b)$.
**Output**: A polynomial $w \in R$ with coefficients in $[b - 2^c + 1, b]$, where $c =$ bitlen $(a + b)$.
When $a + b + 1$ is a power of 2, the coefficients are in $[-a, b]$.

1: $c \leftarrow$ bitlen $(a + b)$
2: $z \leftarrow$ BytesToBits$(v)$
3: **for** $i$ **from** $0$ **to** $255$ **do**
4: $\quad w_i \leftarrow b -$ BitsToInteger$((z[ic], z[ic+1], \ldots z[ic+c-1]), c)$
5: **end for**
6: **return** $w$

Algorithms 20 and 21 carry out byte-string-to-polynomial conversions for polynomials with sparse binary coefficients. In particular, the signing and verification algorithms (Sections 6.2 and 6.3) make use of a "hint," which is a vector of polynomials $\mathbf{h} \in R_2^k$ such that the total number of coefficients in $\mathbf{h}[0], \mathbf{h}[1], \ldots, \mathbf{h}[k-1]$ that are equal to $1$ is no more than $\omega$. This constraint enables encoding and decoding procedures that are more efficient (although more complex) than BitPack and BitUnpack.

HintBitPack $(\mathbf{h})$ outputs a byte string $y$ of length $\omega + k$. The last $k$ bytes of $y$ contain information about how many nonzero coefficients are present in each of the polynomials $\mathbf{h}[0], \mathbf{h}[1], \ldots, \mathbf{h}[k-1]$, and the first $\omega$ bytes of $y$ contain information about exactly where those nonzero terms occur. HintBitUnpack reverses the procedure performed by HintBitPack and recovers the vector $\mathbf{h}$.

**Algorithm 20** HintBitPack$(\mathbf{h})$

*Encodes a polynomial vector* $\mathbf{h}$ *with binary coefficients into a byte string.*

**Input**: A polynomial vector $\mathbf{h} \in R_2^k$ such that the polynomials $\mathbf{h}[0], \mathbf{h}[1]$,...,$\mathbf{h}[k-1]$ have collectively at most $\omega$ nonzero coefficients.
**Output**: A byte string $y$ of length $\omega + k$ that encodes $\mathbf{h}$ as described above.

$y \in \mathbb{B}^{\omega+k} \leftarrow 0^{\omega+k}$
Index $\leftarrow 0$ ▷ Index for writing the first $\omega$ bytes of $y$
**for** $i$ **from** $0$ **to** $k-1$ **do** ▷ look at $\mathbf{h}[i]$
  **for** $j$ **from** $0$ **to** $255$ **do**
    **if** $\mathbf{h}[i]_j \neq 0$ **then**
      $y$[Index] $\leftarrow j$ ▷ store the locations of the nonzero coefficients in $\mathbf{h}[i]$
      Index $\leftarrow$ Index $+ 1$
    **end if**
  **end for**
  $y[\omega + i] \leftarrow$ Index ▷ after processing $\mathbf{h}[i]$, store the value of Index
**end for**
**return** $y$

**Algorithm 21** HintBitUnpack$(y)$

*Reverses the procedure* HintBitPack.

**Input**: A byte string $y$ of length $\omega + k$ that encodes $\mathbf{h}$ as described above.
**Output**: A polynomial vector $\mathbf{h} \in R_2^k$ or $\perp$.

$\mathbf{h} \in R_2^k \leftarrow 0^k$
Index $\leftarrow 0$ ▷ Index for reading the first $\omega$ bytes of $y$
**for** $i$ **from** $0$ **to** $k-1$ **do** ▷ reconstruct $\mathbf{h}[i]$
  **if** $y[\omega + i] <$ Index or $y[\omega + i] > \omega$ **then return** $\perp$ ▷ malformed input
  **end if**
  First $\leftarrow$ Index
  **while** Index $< y[\omega + i]$ **do** ▷ $y[\omega + i]$ says how far one can advance Index
    **if** Index $>$ First **then**
      **if** $y$[Index $- 1] \geq y$[Index] **then return** $\perp$ ▷ malformed input
      **end if**
    **end if**
    $\mathbf{h}[i]_{y[\text{Index}]} \leftarrow 1$ ▷ $y$[Index] says which coefficient in $\mathbf{h}[i]$ should be 1
    Index $\leftarrow$ Index $+ 1$
  **end while**
**end for**
**for** $i$ **from** Index **to** $\omega - 1$ **do** ▷ read any leftover bytes in the first $\omega$ bytes of $y$
  **if** $y[i] \neq 0$ **then return** $\perp$ ▷ malformed input
  **end if**
**end for**
**return** $\mathbf{h}$

## 7.2 Encodings of ML-DSA Keys and Signatures

Algorithms 22–27 translate keys and signatures for ML-DSA into byte strings. These procedures take certain sequences of algebraic objects, encode them consecutively into byte strings, and perform the respective decoding procedures.

First, pkEncode and pkDecode translate ML-DSA public keys into byte strings and vice versa. When verifying a signature, pkDecode might be run on an input that comes from an untrusted source. Thus, care is required when using SimpleBitUnpack. As used here, SimpleBitUnpack always returns values in the correct range.

**Algorithm 22** pkEncode$(\rho, \mathbf{t}_1)$

*Encodes a public key for ML-DSA into a byte string.*

**Input**:$\rho \in \mathbb{B}^{32}, \mathbf{t}_1 \in R^k$ with coefficients in [0, $2^{\text{bitlen }(q-1)-d} - 1$].
**Output**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$.

1: $pk \leftarrow \rho$
2: **for** $i$ **from** $0$ **to** $k-1$ **do**
3: $\quad pk \leftarrow pk \,||\, \text{SimpleBitPack}\,(\mathbf{t}_1[i], 2^{\text{bitlen }(q-1)-d} - 1)$
4: **end for**
5: **return** $pk$

**Algorithm 23** pkDecode$(pk)$

*Reverses the procedure* pkEncode.

**Input**: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$.
**Output**: $\rho \in \mathbb{B}^{32}, \mathbf{t}_1 \in R^k$ with coefficients in [0, $2^{\text{bitlen }(q-1)-d} - 1$].

1: $(\rho, z_0, \ldots, z_{k-1}) \in \mathbb{B}^{32} \times \left(\mathbb{B}^{32(\text{bitlen }(q-1)-d)}\right)^k \leftarrow pk$
2: **for** $i$ **from** $0$ **to** $k-1$ **do**
3: $\quad \mathbf{t}_1[i] \leftarrow \text{SimpleBitUnpack}(z_i, 2^{\text{bitlen }(q-1)-d} - 1)$ $\quad \triangleright$ This is always in the correct range
4: **end for**
5: **return** $(\rho, \mathbf{t}_1)$

Next, skEncode and skDecode translate ML-DSA secret keys into byte strings and vice versa. Note that there exist malformed inputs that can cause skDecode to return values that are not in the correct range. Hence, skDecode should only be run on inputs that come from trusted sources.

**Algorithm 24** skEncode$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

*Encodes a secret key for ML-DSA into a byte string.*

**Input**: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $tr \in \mathbb{B}^{64}$, $\mathbf{s}_1 \in R^{\ell}$ with coefficients in $[-\eta, \eta]$, $\mathbf{s}_2 \in R^{k}$ with coefficients in $[-\eta, \eta]$, $\mathbf{t}_0 \in R^{k}$ with coefficients in $[-2^{d-1}+1, 2^{d-1}]$.
**Output**: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((k+\ell)\cdot \text{bitlen }(2\eta)+dk)}$.

$sk \leftarrow \rho||K||tr$
**for** $i$ **from** $0$ **to** $\ell - 1$ **do**
    $sk \leftarrow sk \,||\, \text{BitPack}\,(\mathbf{s}_1[i], \eta, \eta)$
**end for**
**for** $i$ **from** $0$ **to** $k - 1$ **do**
    $sk \leftarrow sk \,||\, \text{BitPack}\,(\mathbf{s}_2[i], \eta, \eta)$
**end for**
**for** $i$ **from** $0$ **to** $k - 1$ **do**
    $sk \leftarrow sk \,||\, \text{BitPack}\,(\mathbf{t}_0[i], 2^{d-1}-1, 2^{d-1})$
**end for**
**return** $sk$

**Algorithm 25** skDecode$(sk)$

*Reverses the procedure* skEncode.

**Input**: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen }(2\eta)+dk)}$.
**Output**: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $tr \in \mathbb{B}^{64}$,
$\mathbf{s}_1 \in R^{\ell}$, $\mathbf{s}_2 \in R^{k}$, $\mathbf{t}_0 \in R^{k}$ with coefficients in $[-2^{d-1}+1, 2^{d-1}]$.

$(\rho, K, tr, y_0, \ldots, y_{\ell-1}, z_0, \ldots, z_{k-1}, w_0, \ldots, w_{k-1}) \in \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{64} \times \left(\mathbb{B}^{32\cdot \text{bitlen }(2\eta)}\right)^{\ell} \times \left(\mathbb{B}^{32\cdot \text{bitlen }(2\eta)}\right)^{k} \times \left(\mathbb{B}^{32d}\right)^{k} \leftarrow sk$
**for** $i$ **from** $0$ **to** $\ell - 1$ **do**
    $\mathbf{s}_1[i] \leftarrow \text{BitUnpack}(y_i, \eta, \eta)$ ▷ this may lie outside $[-\eta, \eta]$ if input is malformed
**end for**
**for** $i$ **from** $0$ **to** $k - 1$ **do**
    $\mathbf{s}_2[i] \leftarrow \text{BitUnpack}(z_i, \eta, \eta)$ ▷ this may lie outside $[-\eta, \eta]$ if input is malformed
**end for**
**for** $i$ **from** $0$ **to** $k - 1$ **do**
    $\mathbf{t}_0[i] \leftarrow \text{BitUnpack}(w_i, 2^{d-1}-1, 2^{d-1})$ ▷ this is always in the correct range
**end for**
**return** $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Next, sigEncode and sigDecode translate ML-DSA signatures into byte strings and vice versa. When verifying a signature, sigDecode might take input that comes from an untrusted source. Thus, care is required when using BitUnpack. As used here, BitUnpack always returns values in the correct range.

**Algorithm 26** sigEncode$(\tilde{c}, \mathbf{z}, \mathbf{h})$

*Encodes a signature into a byte string.*

**Input**: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ with coefficients in $[-\gamma_1 + 1, \gamma_1]$, $\mathbf{h} \in R_2^k$.
**Output**: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.

$\sigma \leftarrow \tilde{c}$
**for** $i$ **from** $0$ **to** $\ell - 1$ **do**
    $\sigma \leftarrow \sigma \,||\, \text{BitPack}\,(\mathbf{z}[i], \gamma_1 - 1, \gamma_1)$
**end for**
$\sigma \leftarrow \sigma \,||\, \text{HintBitPack}\,(\mathbf{h})$
**return** $\sigma$

**Algorithm 27** sigDecode$(\sigma)$

*Reverses the procedure* sigEncode.

**Input**: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.
**Output**: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ with coefficients in $[-\gamma_1 + 1, \gamma_1]$, $\mathbf{h} \in R_2^k$, or $\perp$.

$(\tilde{c}, x_0, \ldots, x_{\ell-1}, y) \in \mathbb{B}^{\lambda/4} \times \left(\mathbb{B}^{32\cdot(1+\text{bitlen }(\gamma_1-1))}\right)^{\ell} \times \mathbb{B}^{\omega+k} \leftarrow \sigma$
**for** $i$ **from** $0$ **to** $\ell - 1$ **do**
    $\mathbf{z}[i] \leftarrow \text{BitUnpack}(x_i, \gamma_1 - 1, \gamma_1)$ ▷ this is in the correct range, as $\gamma_1$ is a power of 2
**end for**
$\mathbf{h} \leftarrow \text{HintBitUnpack}(y)$
**return** $(\tilde{c}, \mathbf{z}, \mathbf{h})$

w1Encode is a specific subroutine used in ML-DSA.Sign. The procedure w1Encode encodes a polynomial vector $\mathbf{w}_1$ into a string of bytes so that it can be processed by the function H.

**Algorithm 28** w1Encode$(\mathbf{w}_1)$

*Encodes a polynomial vector $\mathbf{w}_1$ into a byte string.*

**Input**: $\mathbf{w}_1 \in R^k$ whose polynomial coordinates have coefficients in $[0, (q-1)/(2\gamma_2) - 1]$.
**Output**: A byte string representation $\tilde{\mathbf{w}}_1 \in \mathbb{B}^{32k\cdot\text{bitlen }((q-1)/(2\gamma_2)-1)}$.

$\tilde{\mathbf{w}}_1 \leftarrow ()$
**for** $i$ **from** $0$ **to** $k - 1$ **do**
    $\tilde{\mathbf{w}}_1 \leftarrow \tilde{\mathbf{w}}_1 \,||\, \text{SimpleBitPack}\,(\mathbf{w}_1[i], (q-1)/(2\gamma_2) - 1)$
**end for**
**return** $\tilde{\mathbf{w}}_1$

## 7.3 Pseudorandom Sampling

This section specifies various algorithms for generating algebraic objects pseudorandomly from a seed $\rho$, where $\rho$ is a byte string whose length varies depending on the algorithm. The first procedure to be defined is SampleInBall. As in Section 2.3, $B_\tau$ denotes the set of all polynomials $c \in R$ such that

- Each coefficient of $c$ is either $-1$, $0$, or $1$, and
- Exactly $\tau$ of the coefficients of $c$ are nonzero.

SampleInBall pseudorandomly generates an element of $B_\tau$ using the XOF of a seed $\rho$. The procedure is based on the Fisher-Yates shuffle. H is applied to $\rho$, and the first $8$ bytes of the output are used to choose the signs of the nonzero entries of $c$.[12] Subsequent bytes are used to choose the positions of those nonzero entries.

**Algorithm 29** SampleInBall($\rho$)

*Samples a polynomial $c \in R$ with coefficients from $\{-1, 0, 1\}$ and Hamming weight $\tau \leq 64$.*

**Input**: A seed $\rho \in \mathbb{B}^{\lambda/4}$
**Output**: A polynomial $c$ in $R$.

$c \leftarrow 0$
ctx $\leftarrow$ H.Init()
ctx $\leftarrow$ H.Absorb(ctx, $\rho$)
(ctx, $s$) $\leftarrow$ H.Squeeze(ctx, $8$)
$h \leftarrow$ BytesToBits($s$) ▷ $h$ is a bit string of length $64$
**for** $i$ **from** $256 - \tau$ **to** $255$ **do**
    (ctx, $j$) $\leftarrow$ H.Squeeze(ctx, $1$)
    **while** $j > i$ **do** ▷ rejection sampling in $\{0, \ldots, i\}$
        (ctx, $j$) $\leftarrow$ H.Squeeze(ctx, $1$)
    **end while** ▷ $j$ is a pseudorandom byte that is $\leq i$
    $c_i \leftarrow c_j$
    $c_j \leftarrow (-1)^{h[i+\tau-256]}$
**end for**
**return** $c$

Algorithms 30–34 are the pseudorandom procedures RejNTTPoly, RejBoundedPoly, ExpandA, ExpandS, and ExpandMask. Each generates elements of $R$ or $T_q$ under different input and output conditions. RejNTTPoly and ExpandA make use of the more efficient XOF G, whereas the other three procedures use the XOF H.

The procedure ExpandMask (Algorithm 34) generates a polynomial vector $\mathbf{y}$ in $R^k$ that disguises the secret key in the ML-DSA.Sign_internal procedure (Algorithm 7). In addition to the seed $\rho$, ExpandMask also accepts an integer input $\mu$ that is incorporated into the pseudorandom procedure that generates $\mathbf{s}$.

[12] The parameter $\tau$ is always less than or equal to $64$, and thus $8$ bytes are sufficient to choose the signs for all $\tau$ nonzero entries of $\mathbf{c}$.

**Algorithm 30** RejNTTPoly($\rho$)

*Samples a polynomial* $\in T_q$.

**Input**: A seed $\rho \in \mathbb{B}^{34}$.
**Output**: An element $\hat{a} \in T_q$.

$j \leftarrow 0$
ctx $\leftarrow$ G.Init()
ctx $\leftarrow$ G.Absorb(ctx, $\rho$)
**while** $j < 256$ **do**
    (ctx, $s$) $\leftarrow$ G.Squeeze(ctx, 3)
    $\hat{a}[j] \leftarrow$ CoeffFromThreeBytes($s[0], s[1], s[2]$)
    **if** $\hat{a}[j] \neq \perp$ **then**
        $j \leftarrow j + 1$
    **end if**
**end while**
**return** $\hat{a}$

**Algorithm 31** RejBoundedPoly($\rho$)

*Samples an element* $a \in R$ *with coefficients in* $[-\eta, \eta]$ *computed via rejection sampling from* $\rho$.

**Input**: A seed $\rho \in \mathbb{B}^{66}$.
**Output**: A polynomial $a \in R$.

$j \leftarrow 0$
ctx $\leftarrow$ H.Init()
ctx $\leftarrow$ H.Absorb(ctx, $\rho$)
**while** $j < 256$ **do**
    $z \leftarrow$ H.Squeeze(ctx, 1)
    $z_0 \leftarrow$ CoeffFromHalfByte($z \bmod 16$)
    $z_1 \leftarrow$ CoeffFromHalfByte($\lfloor z/16 \rfloor$)
    **if** $z_0 \neq \perp$ **then**
        $a_j \leftarrow z_0$
        $j \leftarrow j + 1$
    **end if**
    **if** $z_1 \neq \perp$ **and** $j < 256$ **then**
        $a_j \leftarrow z_1$
        $j \leftarrow j + 1$
    **end if**
**end while**
**return** $a$

**Algorithm 32** ExpandA($\rho$)

*Samples a $k \times \ell$ matrix $\hat{\mathbf{A}}$ of elements of $T_q$.*

**Input**: A seed $\rho \in \mathbb{B}^{32}$.
**Output**: Matrix $\hat{\mathbf{A}} \in (T_q)^{k \times \ell}$.

1: **for** $r$ **from** $0$ **to** $k-1$ **do**
2: **for** $s$ **from** $0$ **to** $\ell-1$ **do**
3: $\rho' \leftarrow \rho || \text{IntegerToBytes}(s, 1) || \text{IntegerToBytes}(r, 1)$
4: $\hat{\mathbf{A}}[r, s] \leftarrow \text{RejNTTPoly}(\rho')$ ▷ seed $\rho'$ depends on $s$ and $r$
5: **end for**
6: **end for**
7: **return** $\hat{\mathbf{A}}$

**Algorithm 33** ExpandS($\rho$)

*Samples vectors $\mathbf{s}_1 \in R^\ell$ and $\mathbf{s}_2 \in R^k$, each with polynomial coordinates whose coefficients are in the interval $[-\eta, \eta]$.*

**Input**: A seed $\rho \in \mathbb{B}^{64}$.
**Output**: Vectors $\mathbf{s}_1, \mathbf{s}_2$ of polynomials in $R$.

1: **for** $r$ **from** $0$ **to** $\ell-1$ **do**
2: $\mathbf{s_1}[r] \leftarrow \text{RejBoundedPoly}(\rho || \text{IntegerToBytes}(r, 2))$ ▷ seed depends on $r$
3: **end for**
4: **for** $r$ **from** $0$ **to** $k-1$ **do**
5: $\mathbf{s_2}[r] \leftarrow \text{RejBoundedPoly}(\rho || \text{IntegerToBytes}(r+\ell, 2))$ ▷ seed depends on $r+\ell$
6: **end for**
7: **return** $(\mathbf{s_1}, \mathbf{s_2})$

**Algorithm 34** ExpandMask($\rho, \mu$)

*Samples a vector $\mathbf{y} \in R^\ell$ such that each polynomial $\mathbf{y}[r]$ has coefficients between $-\gamma_1 + 1$ and $\gamma_1$.*

**Input**: A seed $\rho \in \mathbb{B}^{64}$ and a nonnegative integer $\mu$.
**Output**: Vector $\mathbf{y} \in R^\ell$.

1: $c \leftarrow 1 + \text{bitlen}\ (\gamma_1 - 1)$ ▷ $\gamma_1$ is always a power of 2
2: **for** $r$ **from** $0$ **to** $\ell-1$ **do**
3: $\rho' \leftarrow \rho || \text{IntegerToBytes}(\mu + r, 2)$
4: $v \leftarrow \text{H}(\rho', 32c)$ ▷ seed depends on $\mu + r$
5: $\mathbf{y}[r] \leftarrow \text{BitUnpack}(v, \gamma_1 - 1, \gamma_1)$
6: **end for**
7: **return** $\mathbf{y}$

## 7.4 High-Order and Low-Order Bits and Hints

This specification uses the auxiliary functions Power2Round, Decompose, HighBits, LowBits, MakeHint, and UseHint and explicitly defines these functions, where $r \in \mathbb{Z}_q$, $r_1, r_0 \in \mathbb{Z}$, and $h$ is a Boolean (or equivalently an element of $\mathbb{Z}_2$). However, this specification also uses these functions where $\mathbf{r}, \mathbf{z} \in R_q^k$, $\mathbf{r}_1, \mathbf{r}_0 \in R^k$, and $\mathbf{h} \in R_2^k$. In this case, the functions are applied coefficientwise to the polynomials in the vectors. In particular:

- For $\mathbf{r} \in R_q^k$, define $(\mathbf{r}_1, \mathbf{r}_0) \in (R^k)^2 = \text{Power2Round}(\mathbf{r})$ so that:

$$((\mathbf{r}_1[i])_j, (\mathbf{r}_0[i])_j) = \text{Power2Round}((\mathbf{r}[i])_j).$$

- For $\mathbf{r} \in R_q^k$, define $(\mathbf{r}_1, \mathbf{r}_0) \in (R^k)^2 = \text{Decompose}(\mathbf{r})$ so that:

$$((\mathbf{r}_1[i])_j, (\mathbf{r}_0[i])_j) = \text{Decompose}((\mathbf{r}[i])_j).$$

- For $\mathbf{r} \in R_q^k$, define $\mathbf{r}_1$ = HighBits $(\mathbf{r})$ so that:

$$(\mathbf{r}_1[i])_j = \text{HighBits}((\mathbf{r}[i])_j).$$

- For $\mathbf{r} \in R_q^k$, define $\mathbf{r}_0$ = LowBits$(\mathbf{r})$ so that:

$$(\mathbf{r}_0[i])_j = \text{LowBits}((\mathbf{r}[i])_j).$$

- For $\mathbf{z}, \mathbf{r} \in R_q^k$, define $\mathbf{h} \in R_2^k = \text{MakeHint}(\mathbf{z}, \mathbf{r})$ so that:

$$(\mathbf{h}[i])_j = \text{MakeHint}((\mathbf{z}[i])_j, (\mathbf{r}[i])_j).$$

- For $\mathbf{h} \in R_2^k$ and $\mathbf{r} \in R_q^k$, define $\mathbf{r}_1 \in R^k = \text{UseHint}(\mathbf{h}, \mathbf{r})$ so that:

$$\mathbf{r}_1[i]_j = \text{UseHint}((\mathbf{h}[i])_j, (\mathbf{r}[i])_j).$$

These algorithms are used to support the key compression optimization of ML-DSA. They involve dropping the $d$ low-order bits of each coefficient of the polynomial vector $\mathbf{t}$ from the public key using the function Power2Round. However, in order to make this optimization work, additional information called a "hint" needs to be provided in the signature to allow the verifier to reconstruct enough of the information in the dropped public-key bits to verify the signature. Hints are created during signing and used during verification by the functions MakeHint and UseHint, respectively. In the verification of a valid signature, the hint allows the verifier to recover $\mathbf{w}_1 \in R^k$, which represents $\mathbf{w} \in R_q^k$ rounded to a nearby multiple of $\alpha = 2\gamma_2$. The signer directly obtains $\mathbf{w}_1$ using the function HighBits, and the part rounded off (i.e., $\mathbf{r}_0$) is obtained by LowBits. $\mathbf{r}_0$ is used by the signer in the rejection sampling procedure.

Power2Round decomposes an input $r \in \mathbb{Z}_q$ into integers that represent the high- and low-order bits of $r$ mod $q$ in the straightforward bitwise way, $r \bmod q = r_1 \cdot 2^d + r_0$, where $r_0 = (r \bmod q) \bmod^{\pm} 2^d$ and $r_1 = (r \bmod q - r_0)/2^d$.

However, for the purpose of computations related to hints, this method of decomposing $r$ has the undesirable property that when $r$ is close to $q - 1$ or 0, a small rounding error in $r$ can cause $r_1$ to change by more than 1, even accounting for wrap-around. In contrast to other unequal pairs of values of $r_1 \cdot 2^d$ and $r_1' \cdot 2^d$, the distance (mod $q$) between $\lfloor q/2^d \rfloor \cdot 2^d$ and 0 may be very small.

To avoid this problem, this specification defines Decompose, which is similar to Power2Round except:

- $r$ is generally decomposed as $r \bmod q = r_1 \cdot \alpha + r_0$, where $\alpha = 2\gamma_2$ is a divisor of $q - 1$.
- If the straightforward rounding procedure would return $(r_1 = (q-1)/\alpha, r_0 \in [-(\alpha/2)+1, \alpha/2])$, Decompose instead returns $(r_1 = 0, r_0 - 1)$.

The functions HighBits and LowBits — which only return $r_1$ and $r_0$, respectively — and MakeHint and UseHint use Decompose. For additional discussion of the mathematical properties of these functions that are relevant to the correctness and security of ML-DSA, see Section 2.4 in [6].

**Algorithm 35** Power2Round$(r)$

*Decomposes $r$ into $(r_1, r_0)$ such that $r \equiv r_1 2^d + r_0 \bmod q$.*

**Input**: $r \in \mathbb{Z}_q$.
**Output**: Integers $(r_1, r_0)$.

1: $r^+ \leftarrow r \bmod q$
2: $r_0 \leftarrow r^+ \bmod^{\pm} 2^d$
3: **return** $((r^+ - r_0)/2^d, r_0)$

**Algorithm 36** Decompose$(r)$

*Decomposes $r$ into $(r_1, r_0)$ such that $r \equiv r_1(2\gamma_2) + r_0 \bmod q$.*

**Input**: $r \in \mathbb{Z}_q$.
**Output**: Integers $(r_1, r_0)$.

1: $r^+ \leftarrow r \bmod q$
2: $r_0 \leftarrow r^+ \bmod^{\pm} (2\gamma_2)$
3: **if** $r^+ - r_0 = q - 1$ **then**
4: $\quad r_1 \leftarrow 0$
5: $\quad r_0 \leftarrow r_0 - 1$
6: **else** $r_1 \leftarrow (r^+ - r_0)/(2\gamma_2)$
7: **end if**
8: **return** $(r_1, r_0)$

**Algorithm 37** HighBits$(r)$

*Returns $r_1$ from the output of Decompose $(r)$.*

**Input**: $r \in \mathbb{Z}_q$.
**Output**: Integer $r_1$.

1: $(r_1, r_0) \leftarrow$ Decompose$(r)$
2: **return** $r_1$

**Algorithm 38** LowBits$(r)$

*Returns $r_0$ from the output of* Decompose $(r)$.

**Input**: $r \in \mathbb{Z}_q$.
**Output**: Integer $r_0$.

1: $(r_1, r_0) \leftarrow$ Decompose$(r)$
2: **return** $r_0$

**Algorithm 39** MakeHint$(z, r)$

*Computes hint bit indicating whether adding $z$ to $r$ alters the high bits of $r$.*

**Input**: $z, r \in \mathbb{Z}_q$.
**Output**: Boolean.

1: $r_1 \leftarrow$ HighBits$(r)$
2: $v_1 \leftarrow$ HighBits$(r + z)$
3: **return** $[[r_1 \neq v_1]]$

**Algorithm 40** UseHint$(h, r)$

*Returns the high bits of $r$ adjusted according to hint $h$.*

**Input**: Boolean $h$, $r \in \mathbb{Z}_q$.
**Output**: $r_1 \in \mathbb{Z}$ with $0 \leq r_1 \leq \frac{q-1}{2\gamma_2}$.

1: $m \leftarrow (q-1)/(2\gamma_2)$
2: $(r_1, r_0) \leftarrow$ Decompose$(r)$
3: **if** $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod m$
4: **if** $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod m$
5: **return** $r_1$

## 7.5 NTT and NTT$^{-1}$

The following algorithms implement the NTT and its inverse (NTT$^{-1}$), which is important for efficiency. There are other optimizations that are not included in this standard. In particular, mod $q$ and mod$^{\pm}q$ are expensive operations whose use can be minimized by using Montgomery Multiplication (see Appendix A).

An element of $R_q$ is a polynomial in $\mathbb{Z}_q[X]/(X^{256}+1)$, and an element of $T_q$ is a tuple in $\Pi_{j=0}^{255}\mathbb{Z}_q$. The NTT algorithm takes a polynomial $w \in R_q$ as input and returns $\hat{w} \in T_q$. NTT$^{-1}$ takes $\hat{w} \in T_q$ as input and returns $w$ such that $\hat{w} = \text{NTT}(w)$.

This document always distinguishes between elements of $R_q$ and elements of $T_q$. However, the natural data structure for both of these sets is as an integer array of size $256$. This would allow the NTT and NTT$^{-1}$ algorithms to perform computation in place on an integer array passed by reference. That optimization is not included in this document.

In Section 2.5, $\zeta = 1753 \in \mathbb{Z}_q$, which is a $512$th root of unity modulo $q$. On input $w \in R_q$, the algorithm outputs

$$\text{NTT}(w) = (w(\zeta_0), w(\zeta_1), \ldots, w(\zeta_{255})) \in T_q, \tag{7.1}$$

where $\zeta_i = w(\zeta^{2\text{BitRev}_8(i)+1}) \bmod q$.

The values $\zeta^{\text{BitRev}_8(k)} \bmod q$ for $k = 1, \ldots, 255$ used in line 10 of Algorithms 41 and 42 are pre-computed into an array zetas$[1..255]$. The table of zetas is given in Appendix B. If Montgomery Multiplication is used (see Appendix A), then the zetas array would typically be stored in Montgomery form.

NTT and NTT$^{-1}$ use BitRev$_8$, which reverses the order of bits in an 8-bit integer.

**Algorithm 41** NTT$(w)$

*Computes the NTT.*

**Input**: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

**Output**: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$.

**for** $j$ **from** $0$ **to** $255$ **do**
  $\hat{w}[j] \leftarrow w_j$
**end for**
$m \leftarrow 0$
$len \leftarrow 128$
**while** $len \geq 1$ **do**
  $start \leftarrow 0$
  **while** $start < 256$ **do**
    $m \leftarrow m + 1$
    $z \leftarrow \text{zetas}[m]$ ▷ $z \leftarrow \zeta^{\text{BitRev}_8(m)} \bmod q$
    **for** $j$ **from** $start$ **to** $start + len - 1$ **do**
      $t \leftarrow (z \cdot \hat{w}[j + len]) \bmod q$
      $\hat{w}[j + len] \leftarrow (\hat{w}[j] - t) \bmod q$
      $\hat{w}[j] \leftarrow (\hat{w}[j] + t) \bmod q$
    **end for**
    $start \leftarrow start + 2 \cdot len$
  **end while**
  $len \leftarrow \lfloor len/2 \rfloor$
**end while**
**return** $\hat{w}$

**Algorithm 42** $\text{NTT}^{-1}(\hat{w})$

*Computes the inverse of the NTT.*

**Input**: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$.

**Output**: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

**for** $j$ **from** $0$ **to** $255$ **do**
  $w_j \leftarrow \hat{w}[j]$
**end for**
$m \leftarrow 256$
$len \leftarrow 1$
**while** $len < 256$ **do**
  $start \leftarrow 0$
  **while** $start < 256$ **do**
    $m \leftarrow m - 1$
    $z \leftarrow -zetas[m]$ ▷ $z \leftarrow -\zeta^{\text{BitRev}_8(m)} \bmod q$
    **for** $j$ **from** $start$ **to** $start + len - 1$ **do**
      $t \leftarrow w_j$
      $w_j \leftarrow (t + w_{j+len}) \bmod q$
      $w_{j+len} \leftarrow (t - w_{j+len}) \bmod q$
      $w_{j+len} \leftarrow (z \cdot w_{j+len}) \bmod q$
    **end for**
    $start \leftarrow start + 2 \cdot len$
  **end while**
  $len \leftarrow 2 \cdot len$
**end while**
$f \leftarrow 8347681$ ▷ $f = 256^{-1} \bmod q$
**for** $j$ **from** $0$ **to** $255$ **do**
  $w_j \leftarrow (f \cdot w_j) \bmod q$
**end for**
**return** $w$

**Algorithm 43** $\text{BitRev}_8(m)$

*Transforms a byte by reversing the order of bits in its 8-bit binary expansion.*

**Input**: A byte $m \in [0, 255]$.

**Output**: A byte $r \in [0, 255]$.

$b \leftarrow \text{IntegerToBits}(m, 8)$
$b_{\text{rev}} \in \{0,1\}^8 \leftarrow (0, \ldots, 0)$
**for** $i$ **from** 0 **to** 7 **do**
  $b_{\text{rev}}[i] \leftarrow b[7 - i]$
**end for**
$r \leftarrow \text{BitsToInteger}(b_{\text{rev}}, 8)$
**return** r

## 7.6 Arithmetic Under NTT

The NTT converts elements of the ring $R_q$ (where addition and multiplication are denoted by $+$ and $\cdot$, respectively) into elements of the ring $T_q$ (where addition and multiplication are denoted by $+$ and $\circ$, respectively). This section gives explicit algorithms for linear algebra over the ring $T_q$.

The ring $T_q$ is defined to be the direct product ring $\Pi_{i=0}^{255}\mathbb{Z}_q$. Thus, an element $\hat{a} \in T_q$ is an array of length $256$, and its elements are denoted by $\hat{a}[0], \hat{a}[1], \ldots, \hat{a}[255] \in \mathbb{Z}_q$.

**Algorithm 44** AddNTT$(\hat{a}, \hat{b})$

*Computes the sum $\hat{a} + \hat{b}$ of two elements $\hat{a}, \hat{b} \in T_q$.*

**Input**: $\hat{a}, \hat{b} \in T_q$.
**Output**: $\hat{c} \in T_q$.

1: **for** $i$ **from** $0$ **to** $255$ **do**
2: $\quad \hat{c}[i] \leftarrow \hat{a}[i] + \hat{b}[i]$
3: **end for**
4: **return** $\hat{c}$

**Algorithm 45** MultiplyNTT$(\hat{a}, \hat{b})$

*Computes the product $\hat{a} \circ \hat{b}$ of two elements $\hat{a}, \hat{b} \in T_q$.*

**Input**: $\hat{a}, \hat{b} \in T_q$.
**Output**: $\hat{c} \in T_q$.

1: **for** $i$ **from** $0$ **to** $255$ **do**
2: $\quad \hat{c}[i] \leftarrow \hat{a}[i] \cdot \hat{b}[i]$
3: **end for**
4: **return** $\hat{c}$

**Algorithm 46** AddVectorNTT$(\hat{\mathbf{v}}, \hat{\mathbf{w}})$

*Computes the sum $\hat{\mathbf{v}} + \hat{\mathbf{w}}$ of two vectors $\hat{\mathbf{v}}, \hat{\mathbf{w}}$ over $T_q$.*

**Input**: $\ell \in \mathbb{N}, \hat{\mathbf{v}} \in T_q^{\ell}, \hat{\mathbf{w}} \in T_q^{\ell}$.
**Output**: $\hat{\mathbf{u}} \in T_q^{\ell}$.

1: **for** $i$ **from** $0$ **to** $\ell - 1$ **do**
2: $\quad \hat{\mathbf{u}}[i] \leftarrow$ AddNTT$(\hat{\mathbf{v}}[i], \hat{\mathbf{w}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{u}}$

**Algorithm 47** ScalarVectorNTT$(\hat{c}, \hat{\mathbf{v}})$

*Computes the product $\hat{c} \circ \hat{\mathbf{v}}$ of a scalar $\hat{c}$ and a vector $\hat{\mathbf{v}}$ over $T_q$.*

**Input**: $\hat{c} \in T_q$, $\ell \in \mathbb{N}$, $\hat{\mathbf{v}} \in T_q^{\ell}$.
**Output**: $\hat{\mathbf{w}} \in T_q^{\ell}$.

1: **for** $i$ **from** $0$ **to** $\ell - 1$ **do**
2: $\quad \hat{\mathbf{w}}[i] \leftarrow$ MultiplyNTT$(\hat{c}, \hat{\mathbf{v}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{w}}$

**Algorithm 48** MatrixVectorNTT$(\hat{\mathbf{M}}, \hat{\mathbf{v}})$

*Computes the product $\hat{\mathbf{M}} \circ \hat{\mathbf{v}}$ of a matrix $\hat{\mathbf{M}}$ and a vector $\hat{\mathbf{v}}$ over $T_q$.*

**Input**: $k, \ell \in \mathbb{N}$, $\hat{\mathbf{M}} \in T_q^{k \times \ell}$, $\hat{\mathbf{v}} \in T_q^{\ell}$.
**Output**: $\hat{\mathbf{w}} \in T_q^{k}$.

1: $\hat{\mathbf{w}} \leftarrow 0^k$
2: **for** $i$ **from** $0$ **to** $k - 1$ **do**
3: $\quad$ **for** $j$ **from** $0$ **to** $\ell - 1$ **do**
4: $\quad\quad \hat{\mathbf{w}}[i] \leftarrow$ AddNTT$(\hat{\mathbf{w}}[i],$ MultiplyNTT$(\hat{\mathbf{M}}[i, j], \hat{\mathbf{v}}[j]))$
5: $\quad$ **end for**
6: **end for**
7: **return** $\hat{\mathbf{w}}$

## References


[1] National Institute of Standards and Technology (2023) Digital signature standard (DSS), (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 186-5. https://doi.org/10.6028/NIST.FIPS.186-5.

[2] Barker E (2020) Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-175B, Rev. 1 [or as amended]. https://doi.org/10.6028/NIST.SP.800-175Br1.

[3] Barker E (2006) Recommendation for obtaining assurances for digital signature applications, National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication (SP) 800-89 [or as amended]. https://doi.org/10.6028/NIST.SP.800-89.

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

[5] Bai S, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schwabe P, Seiler G, Stehlé D (2020) CRYSTALS-Dilithium: Algorithm specifications and supporting documentation, Submission to the NIST's post-quantum cryptography standardization process. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[6] Bai S, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schwabe P, Seiler G, Stehlé D (2021) CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (Version 3.1). Available at https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

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

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

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

[10] Lyubashevsky V (2009) Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. *Advances in Cryptology – ASIACRYPT 2009*, ed Matsui M (Springer Berlin Heidelberg, Berlin, Heidelberg), pp 598–616. https://doi.org/10.1007/978-3-642-10366-7_35.

[11] Lyubashevsky V (2012) Lattice signatures without trapdoors. *EUROCRYPT* (Springer), *Lecture Notes in Computer Science*, Vol. 7237, pp 738–755. https://doi.org/10.1007/978-3-642-29011-4_43.

[12] Güneysu T, Lyubashevsky V, Pöppelmann T (2012) Practical lattice-based cryptography: A signature scheme for embedded systems. *CHES* (Springer), Vol. 7428, pp 530–547. https://doi.org/10.1007/978-3-642-33027-8_31.

[13] Bai S, Galbraith SD (2014) An improved compression technique for signatures based on learning with errors. *Topics in Cryptology – CT-RSA 2014*, ed Benaloh J (Springer International Publishing, Cham), pp 28–47. https://doi.org/10.1007/978-3-319-04852-9_2.

[14] Cremers C, Düzlü S, Fiedler R, Janson C, Fischlin M (2021) BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. *2021 IEEE Symposium on Security and Privacy (SP)* (IEEE Computer Society, Los Alamitos, CA, USA), pp 1696–1714. https://doi.org/10.1109/SP40001.2021.00093.

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

[16] Kiltz E, Lyubashevsky V, Schaffner C (2018) A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. *Advances in Cryptology – EUROCRYPT 2018*, eds Nielsen JB, Rijmen V (Springer International Publishing, Cham), pp 552–586. https://doi.org/10.1007/978-3-319-78372-7_18.

[17] Barker E, Barker W (2019) Recommendation for key management: Part 2 - best practices for key management organizations, National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication (SP) 800-57 Part 2, Rev. 1. https://doi.org/10.6028/NIST.SP.800-57pt2r1.

[18] Barker E, Dang Q (2019) Recommendation for key management: Part 3 - application-specific key management guidance, National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication (SP) 800-57 Part 3, Rev. 1. http://doi.org/10.6028/NIST.SP.800-57pt3r1.

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

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

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

[22] Bruinderink LG, Pessl P (2018) Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (3):21–43. https://doi.org/10.13154/tches.v2018.i3.21-43.

[23] Poddebniak D, Somorovsky J, Schinzel S, Lochter M, Rösler P (2018) Attacking deterministic signature schemes using fault attacks. *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (IEEE), pp 338–352. https://doi.org/10.1109/EuroSP.2018.00031.

[24] Samwel N, Batina L, Bertoni G, Daemen J, Susella R (2018) Breaking ed25519 in wolfssl. *Topics in Cryptology–CT-RSA 2018: The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings* (Springer), pp 1–20. https://doi.org/10.1007/978-3-319-76953-0_1.

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

[26] National Institute of Standards and Technology (2016) Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Available at https://csrc.nist.go

v/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

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

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

[29] Housley R (2009) Cryptographic Message Syntax (CMS), Internet Engineering Task Force (IETF) request for comments (RFC) 5652, https://doi.org/10.17487/RFC5652.

[30] Schnorr C (1990) Efficient identification and signatures for smart cards. *Advances in Cryptology — CRYPTO' 89 Proceedings*, ed Brassard G (Springer New York, New York, NY), pp 239–252. https://doi.org/10.1007/0-387-34805-0_22.

[31] Josefsson S, Liusvaara I (2017) Edwards-Curve Digital Signature Algorithm (EdDSA), RFC 8032. https://doi.org/10.17487/RFC8032.

[32] Lyubashevsky V (2021) Round 3 Official Comment: CRYSTALS-DILITHIUM. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/BjfjRMIdnhM/m/W7kkVOFDBAAJ.

[33] Hamburg M (2024) Dilithium hint unpacking. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/TQo-qFbBO1A/m/YcYKjMblAAAJ.

[34] Mattsson (on behalf of Sönke Jendral) JP (2024) Dilithium hint unpacking. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/TQo-qFbBO1A/m/sLjseYlSAwAJ.

[35] Lee S (2024) Updates for FIPS 203. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Rb0nFvfFTEQ/m/lw-k7tVdBQAJ.

## Appendix A — Montgomery Multiplication

This document uses modular multiplications of the form $a \cdot b$ modulo $q$. This is an expensive operation that is often sped up in practice through the use of Montgomery Multiplication.

If $a$ is an integer modulo $q$, then its *Montgomery form with multiplier* $2^{32}$ is $r \equiv a \cdot 2^{32} \bmod q$. [13]

Suppose that two integers $u$ and $v$ modulo $q$ are in Montgomery form. Their product modulo $q$ is $c = u \cdot v \cdot 2^{-32}$, which is also in Montgomery form. If the integer product of $u$ and $v$ does not overflow a 64-bit signed integer, then one can compute $c$ by first performing the integer multiplication $u \cdot v$ and then "reducing" the product by multiplying by $2^{-32}$ modulo $q$. This last operation can be done efficiently as follows.

The MontgomeryReduce function takes an integer $a$ with absolute value at most $2^{31}q$ as input. It returns an integer $r$ such that $r = a \cdot 2^{-32} \bmod q$. The output is in Montgomery form with multiplier $2^{32} \bmod q$. An implementation would typically input a 64-bit input and return a 32-bit output. The "modulo $2^{32}$" operation simply extracts the 32 least significant bits of a 64-bit value. The value $(a - t \cdot q)$ on line 3 is an integer divisible by $2^{32}$. Therefore, the division consists of simply taking the most significant 32 bits of a 64-bit value. Extracting the four low- or high-order bytes is often done using typecasting.

**Algorithm 49** MontgomeryReduce$(a)$

*Computes* $a \cdot 2^{-32} \bmod q$.

**Input**: Integer $a$ with $-2^{31}q \leq a \leq 2^{31}q$.
**Output**: $r \equiv a \cdot 2^{-32} \bmod q$.

1: QINV $\leftarrow 58728449$ ▷ the inverse of $q$ modulo $2^{32}$
2: $t \leftarrow ((a \bmod 2^{32}) \cdot \text{QINV}) \bmod 2^{32}$
3: $r \leftarrow (a - t \cdot q)/2^{32}$
4: return $r$

With this algorithm, the modular product of $a$ and $b$ is $c = \text{MontgomeryReduce}(a \cdot b)$, where $a, b,$ and $c$ are in Montgomery form. The return value of the algorithm is not necessarily less than $q$ in absolute value, but it is less than $2q$ in absolute value. This is not a concern in practice since the objective of Montgomery Multiplication is to efficiently work with modular values that fit in a 32-bit register. If necessary, the result can be normalized to an integer in $(-q, q)$ using a comparison and an integer addition.

Converting an integer modulo $q$ to Montgomery form by multiplying by $2^{32}$ modulo $q$ is an expensive operation. When a sequence of modular operations is to be performed, the operands are converted once to Montgomery form. The operations are then performed, and the factor $2^{32}$ is extracted from the final result.

[13] This section does not distinguish between different versions of the "mod" operator. There are three such versions of "$x = a$ modulo $q$": i) $x \in [0, q-1]$; ii) $x \in [-\lceil q/2 \rceil, \lfloor q/2 \rfloor]$ ; iii) $x \in [-q+1, q-1]$. The last version corresponds to the "%" operator in most programming languages.

## Appendix B — Zetas Array

The values $\zeta^{\text{BitRev}_8(k)} \bmod q$ for $k = 1, \ldots, 255$ used in the NTT Algorithms 41 and 42 may be pre-computed and stored in an array zetas$[1..255]$. This table of zetas is given below.

zetas[0..255] = {

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0, | 4808194, | 3765607, | 3761513, | 5178923, | 5496691, | 5234739, | 5178987, |
| 7778734, | 3542485, | 2682288, | 2129892, | 3764867, | 7375178, | 557458, | 7159240, |
| 5010068, | 4317364, | 2663378, | 6705802, | 4855975, | 7946292, | 676590, | 7044481, |
| 5152541, | 1714295, | 2453983, | 1460718, | 7737789, | 4795319, | 2815639, | 2283733, |
| 3602218, | 3182878, | 2740543, | 4793971, | 5269599, | 2101410, | 3704823, | 1159875, |
| 394148, | 928749, | 1095468, | 4874037, | 2071829, | 4361428, | 3241972, | 2156050, |
| 3415069, | 1759347, | 7562881, | 4805951, | 3756790, | 6444618, | 6663429, | 4430364, |
| 5483103, | 3192354, | 556856, | 3870317, | 2917338, | 1853806, | 3345963, | 1858416, |
| 3073009, | 1277625, | 5744944, | 3852015, | 4183372, | 5157610, | 5258977, | 8106357, |
| 2508980, | 2028118, | 1937570, | 4564692, | 2811291, | 5396636, | 7270901, | 4158088, |
| 1528066, | 482649, | 1148858, | 5418153, | 7814814, | 169688, | 2462444, | 5046034, |
| 4213992, | 4892034, | 1987814, | 5183169, | 1736313, | 235407, | 5130263, | 3258457, |
| 5801164, | 1787943, | 5989328, | 6125690, | 3482206, | 4197502, | 7080401, | 6018354, |
| 7062739, | 2461387, | 3035980, | 621164, | 3901472, | 7153756, | 2925816, | 3374250, |
| 1356448, | 5604662, | 2683270, | 5601629, | 4912752, | 2312838, | 7727142, | 7921254, |
| 348812, | 8052569, | 1011223, | 6026202, | 4561790, | 6458164, | 6143691, | 1744507, |
| 1753, | 6444997, | 5720892, | 6924527, | 2660408, | 6600190, | 8321269, | 2772600, |
| 1182243, | 87208, | 636927, | 4415111, | 4423672, | 6084020, | 5095502, | 4663471, |
| 8352605, | 822541, | 1009365, | 5926272, | 6400920, | 1596822, | 4423473, | 4620952, |
| 6695264, | 4969849, | 2678278, | 4611469, | 4829411, | 635956, | 8129971, | 5925040, |
| 4234153, | 6607829, | 2192938, | 6653329, | 2387513, | 4768667, | 8111961, | 5199961, |
| 3747250, | 2296099, | 1239911, | 4541938, | 3195676, | 2642980, | 1254190, | 8368000, |
| 2998219, | 141835, | 8291116, | 2513018, | 7025525, | 613238, | 7070156, | 6161950, |
| 7921677, | 6458423, | 4040196, | 4908348, | 2039144, | 6500539, | 7561656, | 6201452, |
| 6757063, | 2105286, | 6006015, | 6346610, | 586241, | 7200804, | 527981, | 5637006, |
| 6903432, | 1994046, | 2491325, | 6987258, | 507927, | 7192532, | 7655613, | 6545891, |
| 5346675, | 8041997, | 2647994, | 3009748, | 5767564, | 4148469, | 749577, | 4357667, |
| 3980599, | 2569011, | 6764887, | 1723229, | 1665318, | 2028038, | 1163598, | 5011144, |
| 3994671, | 8368538, | 7009900, | 3020393, | 3363542, | 214880, | 545376, | 7609976, |
| 3105558, | 7277073, | 508145, | 7826699, | 860144, | 3430436, | 140244, | 6866265, |
| 6195333, | 3123762, | 2358373, | 6187330, | 5365997, | 6663603, | 2926054, | 7987710, |
| 8077412, | 3531229, | 4405932, | 4606686, | 1900052, | 7598542, | 1054478, | 7648983 } |

## Appendix C — Loop Bounds

There are four algorithms in this standard with loops that iterate an indeterminate number of times, though the expected number of iterations is a small constant in each case. Three of the four algorithms involve sampling from the output of an XOF, where the amount of output required from the XOF is proportional to the number of iterations that are performed.

Implementations **should not** bound the number of iterations in these loops or the amount of output that is extracted from the XOFs when executing these functions.[14] If an implementation bounds the number of iterations or the number of bytes that may be extracted from the XOF, it **shall not** use a limit lower than those presented in Table 3. The limits yield a probability of approximately $2^{-256}$ (or less) of being reached in a correct implementation of this standard. The probability is calculated under standard assumptions about the output distributions of XOFs and hash functions.

**Table 3. While loop and XOF output limits for a $2^{-256}$ or less probability of failure**

| Algorithm | Minimum allowable limit (Loop iterations) | Minimum allowable limit (XOF output bytes) |
|---|---|---|
| ML-DSA.Sign_internal | 814 | N/A |
| RejBoundedPoly | 481 | 481 |
| RejNTTPoly | 298 | 894 |
| SampleInBall | 121 | 221 |

Implementations may limit the number of iterations of a while loop or the number of bytes drawn from the XOF to not exceed the maximum values in Table 3. If this option is used and the maximum number of iterations or XOF output bytes is exceeded, the algorithm shall destroy all intermediate results. If a return value or exception is produced, it shall be the same for any execution in which the maximum number of iterations or output bytes is exceeded.

There is essentially no performance penalty for using a larger than necessary limit, as the limit will only be reached on a faulty execution of the loop. Because of this, limits were chosen that lower-bound the probability of reaching them to $2^{-256}$.

### ML-DSA.Sign_internal

Table 1 contains the expected repetitions in the rejection sampling loop of ML-DSA.Sign_internal. These are 4.25, 5.1, and 3.85 for Categories 2, 3, and 5, respectively. Therefore, the probability that the number of repetitions exceeds $n$ is less than or equal to $\left(\frac{5.1-1}{5.1}\right)^n$ for all categories. Solving $\left(\frac{5.1-1}{5.1}\right)^n \leq 2^{-256}$ yields $n = 814$.

### RejBoundedPoly

Let $X$ be the number of coefficients generated in $n$ iterations of the while loop of RejBoundedPoly[15]. Then $X$ is $Binomial(2n, \theta)$, where $\theta$ is either $\frac{9}{16}$ or $\frac{15}{16}$, depending on the parameter $\eta$. For $\theta = \frac{9}{16}$, the probability that fewer than $256$ coefficients are generated in 481 iterations of the main loop in RejBoundedPoly is less than $2^{-256}$. Each iteration consumes one byte of output from H.

[14] RejBoundedPoly, RejNTTPoly, and SampleInBall use the incremental APIs described in Section 3.7 in order to extract the amount of output needed from the XOFs, given that the amount needed is not known in advance.

[15] Note that 0, 1, or 2 coefficients are generated in each iteration.

## RejNTTPoly

The number of valid coefficients generated in $n$ calls to G.Squeeze in RejNTTPoly is $Binomial(n, 2^{-23}q)$. It follows that after $298$ calls, the probability of failure is less than $2^{-256}$. Each iteration consumes three bytes of output from G.

## SampleInBall

Step 9 in SampleInBall is executed every time a pseudorandom byte is greater than a value $i$ in the range $[256 - \tau, 255]$. The parameter $\tau$ is 39, 49, and 60 for categories 2, 3, and 5, respectively. Therefore, the probability that this step is executed more than $n$ times in a single iteration of the for loop is less than or equal to $(\frac{59}{256})^n \leq (\frac{\tau}{256})^n$. Solving $(\frac{59}{256})^n \leq 2^{-256}$ yields a bound of $n = 121$ for the while loop on step 8 of SampleInBall. Each iteration consumes one byte of output from H.

Each call to SampleInBall extracts eight bytes from H and then performs $\tau$ iterations of the for loop, each of which extracts an indeterminate amount of data from H. The probability that more than $n$ bytes of output will be required from H during an execution of SampleInBall for a given value of $\tau$ is

$$P(n, \tau) = \begin{cases} 1 & \text{if } n \leq 8 \\ 0 & \text{if } \tau = 1 \text{ and } n > 8 \\ \left(\frac{257-\tau}{256}\right) P(n-1, \tau-1) + \left(\frac{\tau-1}{256}\right) P(n-1, \tau) & \text{if } \tau > 1 \text{ and } n > 8 \end{cases}.$$

$P(n, 60)$ is less than $2^{-256}$ when $n$ is 221 or greater. Implementations may limit the number of bytes extracted from H to $n \geq 221$. Such implementations must stop the execution of SampleInBall, return a constant that represents an error and no other output, and destroy all intermediate results after $n$ bytes of output have been consumed.

# Appendix D — Differences from the CRYSTALS-DILITHIUM Submission

ML-DSA is derived from Version 3.1 of CRYSTALS-DILITHIUM [6]. Version 3.1 differs slightly from the most recent version that appears on the NIST website (i.e., Version 3 CRYSTALS-DILITHIUM [5]). Appendices D.1, D.2, and D.3 document the differences between Versions 3 and 3.1, the differences between Version 3.1 and the initial public draft of the ML-DSA, and the differences between the initial public draft and the ML-DSA standard as published in this document, respectively.

## D.1 Differences Between Version 3.1 and the Round 3 Version of CRYSTALS-DILITHIUM

The lengths of the variables $\rho'$ (private random seed) and $\mu$ (message representative) in the signing algorithm were increased from 384 to 512 bits. The increase in the length of $\mu$ corrects a security flaw that appeared in the third-round submission, where a collision attack against SHAKE256 with a 384-bit output would make it so that parameters targeting NIST security strength category 5 could only meet category 4 [32].

Additionally, the length of the variable $tr$ (the hash of the public key) was reduced from 384 to 256 bits. In key generation, the variable $\varsigma$ was relabeled as $\rho'$ and increased in size from 256 bits to 512 bits.

## D.2 Differences Between Version 3.1 of CRYSTALS-DILITHIUM and FIPS 204 Initial Public Draft

In order to ensure the properties noted in [14], ML-DSA increases the length of $tr$ to 512 bits and increases the length of $\tilde{c}$ to 384 and 512 bits for the parameter sets ML-DSA-65 and ML-DSA-87, respectively. In draft ML-DSA, only the first 256 bits of $\tilde{c}$ are used in the generation of $c$.

In Version 3.1 of the CRYSTALS-DILITHIUM submission, the default version of the signing algorithm is deterministic with $\rho'$ being generated pseudorandomly from the signer's private key and the message, and an optional version of the signing algorithm has $\rho'$ sampled instead as a 512-bit random string. In ML-DSA, $\rho'$ is generated by a "hedged" procedure in which $\rho'$ is pseudorandomly derived from the signer's private key, the message, and a 256-bit string $rnd$, which **should** be generated by an Approved RBG by default. The ML-DSA standard also allows for an optional deterministic version in which $rnd$ is a 256-bit constant string.

The draft ML-DSA standard also included pseudocode that unintentionally omitted a check for malformed input while unpacking the hint [33]. Failure to perform this check results in a signature scheme that is not strongly existentially unforgeable [34].

## D.3 Changes From FIPS 204 Initial Public Draft

In the final version of the ML-DSA standard, the omitted malformed input check was restored to the hint unpacking algorithm (Algorithm 21). Additionally, in the final version of ML-DSA, all of the bits of $\tilde{c}$ are used in the generation of $c$ (Algorithm 29), and ExpandMask (Algorithm 34) is modified to take output bits from the beginning of the output of H.

Based on comments that were submitted on the draft version, more details were provided for the pre-hash version HashML-DSA in Section 5.4. These modifications include domain separation for the cases in which the message is signed directly and cases in which a digest of the message is signed. The changes were made by explicitly defining external functions for both versions of the signing and verification functions

that call an internal function corresponding to the signing or verification functions from the draft FIPS. Domain separation is included in the input to the internal function (see Algorithms 2, 3, 4, 5, 7, and 8). To simplify APIs and for testing purposes, this document also introduced a similar external/internal split for key generation (see Algorithms 1 and 6), but this is a purely editorial change, as the external key generation algorithm is functionally equivalent to the key-generation algorithm from the draft FIPS.

Finally, to offer misuse resistance against the possibility that keys for different parameter sets might be expanded from the same seed [35], domain separation was added to line 1 of Algorithm 6.