



Check for updates

1 FIPS 204 (Draft)

2 Federal Information Processing Standards Publication

3 4 Module-Lattice-Based Digital 5 Signature Standard

6 **Category: Computer Security**

Subcategory: Cryptography

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

10 This publication is available free of charge from:

11 <https://doi.org/10.6028/NIST.FIPS.204.ipd>

12 Published August 24, 2023



13
14 **U.S. Department of Commerce**

15 *Gina M. Raimondo, Secretary*

16 **National Institute of Standards and Technology**

17 *Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology*

18

Foreword

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

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

25

James A. St. Pierre, Acting Director
Information Technology Laboratory

26

Abstract

27 Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity
28 of the signatory. In addition, the recipient of signed data can use a digital signature as evidence in
29 demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is
30 known as non-repudiation since the signatory cannot easily repudiate the signature at a later time.

31 This standard specifies ML-DSA, a set of algorithms that can be used to generate and verify digital
32 signatures. ML-DSA is believed to be secure even against adversaries in possession of a large-scale
33 quantum computer.

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

Federal Information Processing Standards Publication 204

Published: August 24, 2023

Announcing the Module-Lattice-Based Digital Signature Standard

Federal Information Processing Standards Publications (FIPS PUBS) 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 Standard. **Subcategory:** Cryptography.
- 3. Explanation.** This standard specifies a lattice-based digital signature algorithm, ML-DSA, 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 perform the generation of 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 or Federal Information Processing Standard (FIPS) 205 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 publications or in NIST Special Publications.

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

- 73 7. **Applications.** A digital signature algorithm allows an entity to authenticate the integrity of signed
74 data and the identity of the signatory. The recipient of a signed message can use a digital signature
75 as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed
76 signatory. This is known as non-repudiation since the signatory cannot easily repudiate the signature
77 at a later time. A digital signature algorithm is intended for use in electronic mail, electronic funds
78 transfer, electronic data interchange, software distribution, data storage, and other applications that
79 require data integrity assurance and data origin authentication.
- 80 8. **Implementations.** A digital signature algorithm may be implemented in software, firmware, hardware,
81 or any combination thereof. NIST will develop a validation program to test implementations for
82 conformance to the algorithm in this standard. For every computational procedure that is specified in
83 this standard, a conforming implementation may replace the given set of steps with any mathematically
84 equivalent set of steps. In other words, different procedures that produce the correct output for every
85 input are permitted. Information about validation programs is available at [https://csrc.nist.gov/projects](https://csrc.nist.gov/projects/cmvp)
86 [/cmvp](https://csrc.nist.gov/projects/cmvp). Examples for digital signature algorithms are available at [https://csrc.nist.gov/projects/cryptog](https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values)
87 [raphic-standards-and-guidelines/example-values](https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values).
- 88 Agencies are advised that digital signature key pairs **shall not** be used for other purposes.
- 89 9. **Other Approved Security Functions.** Digital signature implementations that comply with this
90 standard **shall** employ cryptographic algorithms that have been approved for protecting Federal
91 Government-sensitive information. **Approved** cryptographic algorithms and techniques include those
92 that are either:
- 93 a. Specified in a Federal Information Processing Standards (FIPS) publication,
 - 94 b. Adopted in a FIPS or NIST recommendation, or
 - 95 c. Specified in the list of **approved** security functions for FIPS 140-3.
- 96 10. **Export Control.** Certain cryptographic devices and technical data regarding them are subject to federal
97 export controls. Exports of cryptographic modules that implement this standard and technical data
98 regarding them must comply with these federal regulations and be licensed by the Bureau of Industry
99 and Security of the U.S. Department of Commerce. Information about export regulations is available at
100 <https://www.bis.doc.gov>.
- 101 11. **Patents.** The algorithm in this standard may be covered by U.S. or foreign patents.
- 102 12. **Implementation Schedule.** This standard becomes effective immediately upon final publication.
- 103 13. **Specifications.** Federal Information Processing Standards (FIPS) 204, Module-Lattice-Based Digital
104 Signature Standard (affixed).
- 105 14. **Qualifications.** The security of a digital signature system is dependent on maintaining the secrecy of
106 the signatory's private keys. Signatories **shall**, therefore, guard against the disclosure of their private
107 keys. While it is the intent of this standard to specify general security requirements for generating
108 digital signatures, conformance to this standard does not ensure that a particular implementation is
109 secure. It is the responsibility of an implementer to ensure that any module that implements a digital
110 signature capability is designed and built in a secure manner.
- 111 Similarly, the use of a product containing an implementation that conforms to this standard does not
112 guarantee the security of the overall system in which the product is used. The responsible authority in
113 each agency or department **shall** ensure that an overall implementation provides an acceptable level of
114 security.

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

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

120 **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.

122 **17. How to Cite this Publication.** NIST has assigned **NIST FIPS 204 ipd** as the publication identifier for
123 this FIPS, per the [NIST Technical Series Publication Identifier Syntax](#). NIST recommends that it be
124 cited as follows:

125 National Institute of Standards and Technology (2023) Module-Lattice-Based Digital
126 Signature Standard. (Department of Commerce, Washington, D.C.), Federal Information
127 Processing Standards Publication (FIPS) NIST FIPS 204 ipd. <https://doi.org/10.6028/NIST.FIPS.204.ipd>

129 **18. Inquiries and Comments.** Inquiries and comments about this FIPS may be submitted to [fips-204-](mailto:fips-204-comments@nist.gov)
130 [comments@nist.gov](mailto:fips-204-comments@nist.gov).

131 **Call for Patent Claims**

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

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

- 140 a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not
141 currently intend holding any essential patent claim(s); or
- 142 b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring
143 to utilize the license for the purpose of complying with the guidance or requirements in this ITL
144 draft publication either:
- 145 (i) under reasonable terms and conditions that are demonstrably free of any unfair discrimination;
146 or
- 147 (ii) without compensation and under reasonable terms and conditions that are demonstrably free
148 of any unfair discrimination.

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

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

156 Such statements should be addressed to: fips-204-comments@nist.gov

157 **Federal Information Processing Standards Publication 204**

158 **Specification for the**
 159 **Module-Lattice-Based Digital Signature Standard**

160 **Table of Contents**

161	1 Introduction	1
162	1.1 Purpose and Scope	1
163	1.2 Context	1
164	1.3 Differences Between the ML-DSA Standard and CRYSTALS-DILITHIUM	2
165	1.3.1 Differences Between Version 3.1 and the Round 3 Version of CRYSTALS-	
166	DILITHIUM	2
167	1.3.2 Differences Between the ML-DSA Standard and Version 3.1 of CRYSTALS-	
168	DILITHIUM	2
169	2 Glossary of Terms, Acronyms, and Symbols	3
170	2.1 Terms and Definitions	3
171	2.2 Acronyms	5
172	2.3 Mathematical Symbols	6
173	2.4 Notation	7
174	2.5 NTT Representation	8
175	3 Overview of the ML-DSA Signature Scheme	9
176	3.1 Security Properties	9
177	3.2 Computational Assumptions	9
178	3.3 The ML-DSA Construction	9
179	3.4 Use of Digital Signatures	11
180	3.5 Additional Requirements	11
181	3.5.1 Randomness Generation	11
182	3.5.2 Public-Key Validity and Signature Length Checks	11
183	3.5.3 Intermediate Values	11
184	4 Parameter Sets	13
185	5 Key Generation	14
186	6 Signing	15
187	7 Verification	18

188	7.1 Prehash ML-DSA	19
189	8 Auxiliary Functions	20
190	8.1 Conversion Between Data Types	20
191	8.2 Encodings of ML-DSA Keys and Signatures	24
192	8.3 Hashing and Pseudorandom Sampling	29
193	8.4 High Order / Low Order Bits and Hints	33
194	8.5 NTT and NTT^{-1}	36
195	References	38
196	Appendix A — Security Strength Categories	41
197	Appendix B — Montgomery Reduction	44

List of Tables

198

199	Table 1	ML-DSA Parameter sets	13
200	Table 2	Sizes (in bytes) of keys and signatures of ML-DSA.	14
201	Table 3	NIST Security Strength Categories	42
202	Table 4	Estimated gate counts for the optimal key recovery and collision attacks on AES	
203		and SHA-3	43

List of Algorithms

204

205	Algorithm 1	ML-DSA.KeyGen()	15
206	Algorithm 2	ML-DSA.Sign(sk, M)	17
207	Algorithm 3	ML-DSA.Verify(pk, M, σ)	19
208	Algorithm 4	IntegerToBits(x, α)	20
209	Algorithm 5	BitsToInteger(y)	20
210	Algorithm 6	BitsToBytes(y)	21
211	Algorithm 7	BytesToBits(z)	21
212	Algorithm 8	CoeffFromThreeBytes(b_0, b_1, b_2)	21
213	Algorithm 9	CoeffFromHalfByte(b)	22
214	Algorithm 10	SimpleBitPack(w, b)	22
215	Algorithm 11	BitPack(w, a, b)	22
216	Algorithm 12	SimpleBitUnpack(v, b)	23
217	Algorithm 13	BitUnpack(v, a, b)	23
218	Algorithm 14	HintBitPack(\mathbf{h})	24
219	Algorithm 15	HintBitUnpack(y)	24
220	Algorithm 16	pkEncode(ρ, \mathbf{t}_1)	25
221	Algorithm 17	pkDecode(pk)	25
222	Algorithm 18	skEncode($\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0$)	26
223	Algorithm 19	skDecode(sk)	27
224	Algorithm 20	sigEncode($\tilde{c}, \mathbf{z}, \mathbf{h}$)	28
225	Algorithm 21	sigDecode(σ)	28
226	Algorithm 22	w1Encode(\mathbf{w}_1)	28
227	Algorithm 23	SampleInBall(ρ)	30
228	Algorithm 24	RejNTTPoly(ρ)	30
229	Algorithm 25	RejBoundedPoly(ρ)	31
230	Algorithm 26	ExpandA(ρ)	31
231	Algorithm 27	ExpandS(ρ)	32
232	Algorithm 28	ExpandMask(ρ, μ)	32
233	Algorithm 29	Power2Round(r)	34
234	Algorithm 30	Decompose(r)	34
235	Algorithm 31	HighBits(r)	34
236	Algorithm 32	LowBits(r)	35
237	Algorithm 33	MakeHint(z, r)	35
238	Algorithm 34	UseHint(h, r)	35
239	Algorithm 35	NTT(w)	36
240	Algorithm 36	NTT ⁻¹ (\hat{w})	37

241 Algorithm 37 Montgomery_Reduce(a) 44

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 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. NIST SP 800-89, *Recommendation for Obtaining Assurances for Digital Signature Applications* [3], specifies the required assurances and methods for obtaining these assurances.

The digital signature scheme approved in this standard is ML-DSA (Module Lattice Digital Signature Algorithm). It is based on the Module Learning With Errors problem. ML-DSA is believed to be secure even against adversaries in possession of a large-scale 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 (Section 5), signature (Section 6), and verification, (Section 7) and for supporting algorithms used by them (Section 8). ML-DSA is standardized with three possible parameter sets, each corresponding 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. In particular, 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, the National Institute of Standards and Technology (NIST) initiated a public process to select quantum-resistant public-key cryptographic algorithms for standardization. A total of 82 candidate algorithms were submitted to NIST for consideration for standardization.

After three rounds of evaluation and analysis, NIST selected the first four algorithms to standardize as a result of the Post-Quantum Cryptography (PQC) Standardization process. The algorithm in this standard, *ML-DSA*, is derived from one of the selected schemes: CRYSTALS-DILITHIUM [4, 5] and is intended to protect sensitive U.S. Government information well into the foreseeable future, including after the advent of large-scale fault-tolerant quantum computers.

281 **1.3 Differences Between the ML-DSA Standard and CRYSTALS-** 282 **DILITHIUM**

283 ML-DSA is derived from Version 3.1 of CRYSTALS-DILITHIUM [5]. Version 3.1 differs slightly
284 from the most recent version appearing on the NIST website (Version 3 CRYSTALS-DILITHIUM [4].)
285 Sections 1.3.1, and 1.3.2 document, respectively, the differences between Versions 3 and 3.1, and the
286 differences between Version 3.1 and the ML-DSA standard as published in this document.

287 **1.3.1 Differences Between Version 3.1 and the Round 3 Version of CRYSTALS-** 288 **DILITHIUM**

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

293 Additionally, the length of the variable tr (the hash of the public key) was reduced from 384 to 256 bits. In
294 key generation, the variable ζ was relabeled as ρ' and increased in size from 256 bits to 512 bits.

295 **1.3.2 Differences Between the ML-DSA Standard and Version 3.1 of CRYSTALS-** 296 **DILITHIUM**

297 In order to ensure the properties noted in [6], ML-DSA increases the length of tr to 512 bits, and increases
298 the length of \tilde{c} to 384 and 512 bits, respectively, for the parameter sets ML-DSA-65 and ML-DSA-87.

299 In Version 3.1 of the CRYSTALS-DILITHIUM submission, the default version of the signing algorithm
300 is deterministic with ρ' being generated pseudorandomly from the signer's private key and the message,
301 and an optional version of the signing algorithm has ρ' sampled instead as a 512-bit random string. In
302 ML-DSA, ρ' is generated by a "hedged" procedure, where ρ' is pseudorandomly derived from the signer's
303 private key, the message, and a 256-bit string, rnd , which by default **should** be generated by an Approved
304 RBG. The ML-DSA standard also allows for an optional deterministic version, where rnd is instead a
305 256-bit constant string.

2. Glossary of Terms, Acronyms, and Symbols

2.1 Terms and Definitions

308	approved	FIPS-approved and/or NIST-recommended. An algorithm or technique that is
309		either 1) specified in a FIPS or NIST recommendation, 2) adopted in a FIPS
310		or NIST recommendation, or 3) specified in a list of NIST- approved security
311		functions.
312	assurance of	Confidence that an entity possesses a private key and any associated keying
313	possession	material.
314	bit string	An ordered sequence of zeros and ones.
315	byte	An integer from the set $\{0, 1, 2, \dots, 255\}$.
316	byte string	A sequence of bytes.
317	certificate	A set of data that uniquely identifies a public key (which has a corresponding
318		private key) and an owner that is authorized to use the key pair. The certificate
319		contains the owner's public key and possibly other information and is digitally
320		signed by a Certification Authority (i.e., a trusted party), thereby binding the
321		public key to the owner.
322	certification authority	The entity in a public key infrastructure (PKI) that is responsible for issuing
323	(CA)	certificates and exacting compliance with a PKI policy.
324	claimed signatory	From the verifier's perspective, the claimed signatory is the entity that purport-
325		edly generated a digital signature.
326	destroy	An action applied to a key or a piece of secret data. After a key or a piece of
327		secret data is destroyed, no information about its value can be recovered.
328	digital signature	The result of a cryptographic transformation of data that, when properly im-
329		plemented, provides a mechanism for verifying origin authenticity and data
330		integrity, and enforcing signatory non-repudiation.
331	entity	An individual (person), organization, device, or process. Used interchangeably
332		with "party."
333	extendable-output	A function on bit strings in which the output can be extended to any desired
334	function (XOF)	length. Approved XOFs (such as those specified in FIPS 202 [7]) are designed
335		to satisfy the following properties as long as the specified output length is
336		sufficiently long to prevent trivial attacks:
337		1. (One-way) It is computationally infeasible to find any input that maps to
338		any new pre-specified output.
339		2. (Collision-resistant) It is computationally infeasible to find any two
340		distinct inputs that map to the same output.
341	hash function	A function on bit strings in which the length of the output is fixed. Approved
342		hash functions (such as those specified in FIPS 180 [8] and FIPS 202 [7]) are
343		designed to satisfy the following properties:

344		1. (One-way) It is computationally infeasible to find any input that maps to
345		any new pre-specified output.
346		2. (Collision-resistant) It is computationally infeasible to find any two
347		distinct inputs that map to the same output.
348	hash value	See “message digest.”
349	key	A parameter used in conjunction with a cryptographic algorithm that deter-
350		mines its operation. Examples of cryptographic algorithms applicable to this
351		standard include:
352		1. The computation of a digital signature from data and
353		2. The verification of a digital signature.
354	key pair	A public key and its corresponding private key.
355	message	The data that is signed. Also known as “signed data” during the signature
356		verification and validation process.
357	message digest	The result of applying a hash function to a message. Also known as a “hash
358		value.”
359	non-repudiation	A service that is used to provide assurance of the integrity and origin of data in
360		such a way that the integrity and origin can be verified and validated by a third
361		party as having originated from a specific entity in possession of the private
362		key (i.e., the signatory).
363	owner	A key pair owner is the entity authorized to use the private key of a key pair.
364	party	An individual (person), organization, device, or process. Used interchangeably
365		with “entity.”
366	public key	A framework that is established to issue, maintain, and revoke public key
367	infrastructure (PKI)	certificates.
368	private key	A cryptographic key that is used with an asymmetric (public key) cryptographic
369		algorithm. The private key is uniquely associated with the owner and is not
370		made public. The private key is used to compute a digital signature that may
371		be verified using the corresponding public key.
372	pseudorandom	A process or data produced by a process is said to be pseudorandom when the
373		outcome is deterministic yet also effectively random as long as the internal
374		action of the process is hidden from observation. For cryptographic purposes,
375		“effectively random” means “computationally indistinguishable from random
376		within the limits of the intended security strength.”
377	public key	A cryptographic key that is used with an asymmetric (public-key) cryptographic
378		algorithm and is associated with a private key. The public key is associated
379		with an owner and may be made public. In the case of digital signatures, the
380		public key is used to verify a digital signature that was generated using the
381		corresponding private key.
382	security category	A number associated with the security strength of a post-quantum crypto-
383		graphic algorithm as specified by NIST (see Appendix A, Table 3).

384	security strength	A number associated with the amount of work (i.e., the number of operations)
385		that is required to break a cryptographic algorithm or system.
386	seed	A bit string used as input to a pseudorandom process.
387	shall	Used to indicate a requirement of this standard.
388	should	Used to indicate a strong recommendation but not a requirement of this stan-
389		dard. Ignoring the recommendation could lead to undesirable results.
390	signatory	The entity that generates a digital signature on data, using a private key.
391	signature generation	The process of using a digital signature algorithm and a private key to generate
392		a digital signature on data.
393	signature validation	The (mathematical) verification of the digital signature along with obtaining
394		the appropriate assurances (e.g., public-key validity, private-key possession,
395		etc.).
396	signature verification	The process of using a digital signature algorithm and a public key to verify a
397		digital signature on data.
398	signed data	The data or message upon which a digital signature has been computed. Also
399		see “message.”
400	trusted third party	An entity (other than the key pair owner and the verifier) that is trusted by the
401	(TTP)	owner, the verifier, or both. Sometimes shortened to “trusted party.”
402	verifier	The entity that verifies the authenticity of a digital signature, using the public
403		key of the signatory.

404 2.2 Acronyms

405	AES	Advanced Encryption Standard
406	FIPS	Federal Information Processing Standard
407	ML-DSA	Module-Lattice-Based Digital Signature Algorithm
408	MLWE	Module Learning With Errors
409	NIST	National Institute of Standards and Technology
410	NISTIR	NIST Interagency or Internal Report
411	NTT	Number Theoretic Transform
412	PQC	Post-Quantum Cryptography
413	RBG	Random Bit Generator
414	SHA	Secure Hash Algorithm
415	SHAKE	Secure Hash Algorithm KECCAK
416	SP	Special Publication
417	XOF	eXtendable-Output Function

418 2.3 Mathematical Symbols

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

420	\mathbb{B}	The set $\{0, 1, \dots, 255\}$.
421	\mathbb{N}	The set of natural numbers $\{1, 2, 3, \dots\}$.
422	\mathbb{Z}	The ring of integers.
423	$[a, b]$	For two integers $a \leq b$, $[a, b]$ denotes the set of integers $\{a, a + 1, \dots, b\}$.
424	\mathbb{Z}_m	The ring of integers modulo m , also denoted by $\mathbb{Z}/m\mathbb{Z}$.
425	R	The ring of single-variable polynomials over \mathbb{Z} modulo $X^{256} + 1$, also denoted by
426		$\mathbb{Z}[X]/(X^{256} + 1)$.
427	R_m	The ring of single-variable polynomials over \mathbb{Z}_m modulo $X^{256} + 1$, also denoted by
428		$\mathbb{Z}_m[X]/(X^{256} + 1)$.
429	q	The prime number $q = 2^{23} - 2^{13} + 1 = 8380417$.
430	B_τ	The set of all polynomials $p = \sum_{i=0}^{255} p_i X^i$ in R_q that are such that exactly τ of the
431		coefficients of p_i are from the set $\{-1, 1\}$, and all other coefficients are zero. (See
432		subsection 8.3.)
433	Π	Used to denote a direct product of two or more rings, where addition and multiplica-
434		tion are performed componentwise.
435	T_q	The ring $\prod_{j=0}^{255} \mathbb{Z}_q$.
436	\top	If A is a matrix, then A^\top denotes the transpose of A .
437	\log	The base-2 logarithm. For example, $\log 256 = 8$.
438	$\text{bitlen } a$	For a positive integer a , the minimum number of binary digits required to represent
439		a . For example, $\text{bitlen } 32 = 6$ and $\text{bitlen } 31 = 5$.
440	$\lfloor x \rfloor$	The largest integer less than or equal to the real number x , called the floor of x .
441	$\lceil x \rceil$	The least integer greater than or equal to the real number x , called the ceiling of x .
442	mod	If α is a positive integer and $m \in \mathbb{Z}$ or $m \in \mathbb{Z}_\alpha$, then $m \text{ mod } \alpha$ denotes the unique
443		element $m' \in \mathbb{Z}$ in the range $0 \leq m' < \alpha$ such that m and m' are congruent modulo α .
444	mod^\pm	If α is a positive integer and $m \in \mathbb{Z}$ or $m \in \mathbb{Z}_\alpha$, then $m \text{ mod}^\pm \alpha$ denotes the unique
445		element $m' \in \mathbb{Z}$ in the range $-\alpha/2 < m' \leq \alpha/2$ such that m and m' are congruent
446		modulo α .
447	$a!$	The factorial quantity $1 \cdot 2 \cdot 3 \cdot \dots \cdot a$.
448	$\binom{a}{b}$	The quantity $a!/(b!(a-b)!)$.
449	$\text{brv}(r)$	Bit reversal. If $r = r_0 + 2r_1 + 4r_2 + \dots + 128r_7$ is a byte, with $r_i \in \{0, 1\}$, then
450		$\text{brv}(r) = r_7 + 2r_6 + 4r_5 + \dots + 128r_0$.

451	\leftarrow	If S is a set, then $s \leftarrow S$ denotes that s is sampled uniformly at random from S . If D is a probability distribution on S , then $s \leftarrow D$ denotes that s is sampled from S according to D . If R is an algorithm with input z , then $s \leftarrow R(z)$ denotes that s is the recorded output of a single execution of $R(z)$. This notation is used for both probabilistic and deterministic algorithms.
452		
453		
454		
455		
456	$x \in S \leftarrow y$	Type casting. An element x in a set S is constructed from an element y of a 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.
457		
458		
459	$w[i]$	For a bit string w , $w[i]$ denotes the i^{th} byte of w , $w[8i] + 2 \cdot w[8i + 1] + 4 \cdot w[8i + 2] + \dots + 128 \cdot w[8i + 7]$ where $w[j]$ is the j^{th} bit of w . That is, when encoding a byte into a bit string, “little-endian” order is used.
460		
461		
462	$\ \cdot\ _{\infty}$	The infinity norm. For an element $w \in \mathbb{Z}$, $\ w\ _{\infty} = w $, the absolute value of w . For an element $w \in \mathbb{Z}_q$, $\ w\ _{\infty} = w \bmod^{\pm} q$. 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 , $\ \mathbf{w}\ _{\infty} = \max_{0 \leq i < m} \ w[i]\ _{\infty}$.
463		
464		
465		
466	$[[a < b]]$	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.
467		
468		
469	$\langle\langle f(x) \rangle\rangle$	A temporary variable that stores the output of a computation $f(x)$, so that this output can be used many times, without needing to recompute it. 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.
470		
471		
472		
473	$a b$	Concatenation of two bit or byte strings, a and b .
474	$a \circ b$	Multiplication (of a and b) in the ring T_q .
475	$a \cdot b$ or ab	Multiplication in any of the rings $\mathbb{Z}, \mathbb{Z}_d, R, R_d$.
476	$a + b$	Addition of a and b .
477	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.
478		
479	$A \times B$	Cartesian product of two sets A, B .
480	\perp	Blank symbol. (This symbol indicates failure or lack of an output from an algorithm.)

2.4 Notation

Elements of the rings $\mathbb{Z}, \mathbb{Z}_q, \mathbb{Z}_2, R, R_q$, 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}). 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 $\hat{u}[i] \cdot \hat{v}[i] \in \mathbb{Z}_q$.) As noted in subsection 2.3, the multiplication operation in T_q is denoted by the symbol \circ .

487 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
 488 either is an element of \mathbb{Z}_m or R_m), then the product is also understood to be a congruence class modulo m .
 489 Likewise when an element of R or \mathbb{Z} may be used as the input of a function specified to act on an element
 490 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
 491 taking the unique congruence class containing the integer.

492 The coefficients of an element w of R or R_m are denoted by w_i so that $w = w_0 + w_1X + \dots + w_{255}X^{255}$.
 493 If w is in R (respectively, R_m) and t is in \mathbb{Z} (respectively, \mathbb{Z}_d), then $w(t)$ denotes the polynomial $w =$
 494 $w_0 + w_1X + \dots + w_{255}X^{255}$ evaluated at $X = t$.

495 Vectors with elements in R or R_m are denoted by bold lowercase letters, such as, \mathbf{v} . Matrices with elements
 496 in R or R_m are denoted by bold uppercase letters, such as, \mathbf{A} .

497 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

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

498 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
 499 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
 500 length- L vector can also be treated as an $L \times 1$ matrix.

501 2.5 NTT Representation

502 The Number Theoretic Transform (NTT) is a specific isomorphism between the rings R_q and T_q . Let
 503 $\zeta = 1753 \in \mathbb{Z}_q$, which is a 512th root of unity. If $w \in R_q$, then

$$\text{NTT}(w) = (w(\zeta_0), w(-\zeta_0), \dots, w(\zeta_{127}), w(-\zeta_{127})) \in T_q, \quad (2.1)$$

504 where $\zeta_i = \zeta^{\text{brv}(128+i)}$. See section 8.5 for a discussion of the implementation of NTT and NTT^{-1} .

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

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

507 If \mathbf{A} is a matrix with entries from R_q , then $\text{NTT}(\mathbf{A})$ denotes the matrix computed via the entry-wise
 508 application of NTT to \mathbf{A} . The symbol \circ is also used to denote matrix multiplication of matrices with entries
 509 in T_q . Thus, $\text{NTT}(\mathbf{AB}) = \text{NTT}(\mathbf{A}) \circ \text{NTT}(\mathbf{B})$.

3. Overview of the ML-DSA Signature Scheme

ML-DSA is a digital signature scheme based on CRYSTALS-DILITHIUM [5]. 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 [9, 10] and bears the most resemblance to the schemes proposed in [11, 12].

3.1 Security Properties

ML-DSA is designed to be strongly existentially unforgeable under chosen message attack (i.e. 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).

ML-DSA is also designed to satisfy additional security properties beyond unforgeability, which are described in [6].

3.2 Computational Assumptions

Security for lattice-based digital signature schemes is typically related to two central problems: the Learning With Errors (LWE) problem and the Short Integer Solution (SIS) problem. The LWE problem [13] is to recover a vector $\mathbf{s} \in \mathbb{Z}_q^n$ given a set of random noisy linear equations satisfied by \mathbf{s} . The SIS problem is to find, for a given linear system over \mathbb{Z}_q of the form $\mathbf{A}\mathbf{t} = \mathbf{0}$, a solution $\mathbf{t} \in \mathbb{Z}_q^n$ 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 (such as R_q), the resulting problems are called MLWE (Module Learning With Errors [14]) and MSIS (Module Short Integer Solution). The security of ML-DSA is based on the MLWE problem over R_q and a nonstandard variant of MSIS called SelfTargetMSIS [15].

3.3 The 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 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 less than the order of g to the prover.
3. Response: The prover returns $s = r - cx$, 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.

546 The basic idea of ML-DSA and similar lattice signature schemes is to build a signature scheme from an
 547 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 $\mathbf{S}_2 \in \mathbb{Z}_q^{K \times n}$
 548 with short coefficients, demonstrates knowledge of these matrices to a verifier who knows \mathbf{A} and $\mathbf{T} \in$
 549 $\mathbb{Z}_q^{K \times n} = \mathbf{A}\mathbf{S}_1 + \mathbf{S}_2$. Such an interactive protocol would proceed as follows:

- 550 1. Commitment: The prover generates $\mathbf{y} \in \mathbb{Z}_q^L$ with short coefficients and commits to its value by
 551 sending $\mathbf{A}\mathbf{y}$ to the verifier.
- 552 2. Challenge: The verifier sends a vector $\mathbf{c} \in \mathbb{Z}_q^n$ with short coefficients to the prover.
- 553 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
 554 that $\mathbf{A}\mathbf{z} - \mathbf{T}\mathbf{c} \approx \mathbf{A}\mathbf{y}$.

555 As written the above protocol has a security flaw: The response \mathbf{z} will be biased in a direction related to
 556 the private value \mathbf{S}_1 . However, this flaw can be corrected when converting the interactive protocol into
 557 a signature scheme: As with Schnorr signatures, the signer derives the challenge by a pseudorandom
 558 process from a hash of the commitment concatenated with the message. However, to correct the bias, the
 559 signer applies rejection sampling to \mathbf{z} : if coefficients of \mathbf{z} fall outside a specified range, the signing process
 560 is aborted, and the signer starts over from a new value of \mathbf{y} . In the resulting ‘‘Fiat-Shamir with Aborts’’
 561 signature, the public key is (\mathbf{A}, \mathbf{T}) and the private key is $(\mathbf{S}_1, \mathbf{S}_2)$

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

- 564 • To reduce key and signature size and to use fast NTT-based polynomial multiplication, ML-DSA
 565 uses module-structured matrices. That is to say, relative to the basic scheme described above, it
 566 replaces dimension- $n \times n$ blocks of matrices and dimension- n blocks of vectors with polynomials in
 567 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
 568 has $\mathbf{A} \in R_q^{k \times \ell}$, $\mathbf{t} \in R_q^k$, $\mathbf{s}_1 \in R_q^\ell$, $\mathbf{s}_2 \in R_q^k$, $\mathbf{y} \in R_q^\ell$, $c \in R_q$.
- 569 • To further reduce the size of the public key, the matrix \mathbf{A} is pseudorandomly derived from a 256-bit
 570 public seed, ρ which is included in the ML-DSA public key in place of \mathbf{A} .
- 571 • For a still further reduction in public key size, the ML-DSA public key substitutes for \mathbf{t} a compressed
 572 value \mathbf{t}_0 , which drops the d low order bits of each coefficient.
- 573 • To obtain beyond unforgeability (BUFF) properties noted in [6], ML-DSA does not sign the message
 574 M directly, but rather signs a message representative μ obtained by hashing the concatenation of a
 575 hash tr of the public key and M .
- 576 • To reduce signature size, rather than including the commitment $\mathbf{w} = \mathbf{A}\mathbf{y}$ in the signature, the ML-
 577 DSA signature uses a rounded version \mathbf{w}_1 as a commitment, and includes only the hash, \tilde{c} , of \mathbf{w}_1
 578 concatenated with μ .
- 579 To ensure that \mathbf{w}_1 can be reconstructed by the verifier from \mathbf{z} and the compressed value \mathbf{t}_0 , the
 580 signature must also include a *hint* $\mathbf{h} \in R_q^k$ computed by the signer using the signer’s private key.

581 In this document, we use the abbreviations ML-DSA-44, ML-DSA-65, and ML-DSA-87 to refer to
 582 ML-DSA with the parameter choices given in Table 1. (In these abbreviations, the numerical suffix refers
 583 to the dimension of the matrix \mathbf{A} . For example, in ML-DSA-65, the matrix \mathbf{A} is a 6×5 matrix over R_q .)

584 3.4 Use of Digital Signatures

585 Secure key management is an essential requirement for the use of digital signatures. This is context-
586 dependent and involves more than the key generation, signing, and signature verification algorithms in this
587 document. Guidance for key management is provided in the NIST SP 800-57 series [16, 17, 18].

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

593 3.5 Additional Requirements

594 This section describes several required assurances when implementing ML-DSA. These are in addition to
595 the considerations in Section 3.4.

596 3.5.1 Randomness Generation

597 Algorithm 1, implementing key generation for ML-DSA, uses an RBG to generate the 256-bit random
598 value ξ . The seed ξ **shall** be freshly generated using an **approved** RBG, as prescribed in NIST SP 800-90A,
599 SP 800-90B, and SP 800-90C [19, 20, 21]. Moreover, the RBG used **shall** have a security strength of at
600 least 192 bits for ML-DSA-65 and 256 bits for ML-DSA-87. For ML-DSA-44, the RBG **should** have a
601 security strength of at least 192 bits and **shall** have a security strength of at least 128 bits. (If an **approved**
602 RBG with at least 128 bits of security but less than 192 bits of security is used, then the claimed security
603 strength of ML-DSA-44 is reduced from category 2 to category 1.)

604 Additionally, in the default “hedged” variant of Algorithm 2, implementing signing for ML-DSA, the
605 value *rnd* is generated using an RBG. While this value **should** ideally be generated by an **approved** RBG,
606 other methods for generating fresh randomness may be used. The primary purpose of *rnd* is to facilitate
607 countermeasures to side-channel attacks and fault attacks on deterministic signatures, such as [22, 23, 24].¹
608 For this purpose, even a weak RBG may be preferable to the fully deterministic variant of Algorithm 2.

609 3.5.2 Public-Key Validity and Signature Length Checks

610 Algorithm 3, implementing verification for ML-DSA, specifies the length of the signature σ and the public
611 key *pk* in terms of the parameters described in Table 1. If an implementation of ML-DSA can accept
612 inputs for σ or *pk* of any other length, it **shall** return false whenever the lengths of either of these inputs
613 differs from its specified length. Checking the length of *pk* serves as a partial public-key validity check,
614 and failing to do so may interfere with the security properties that ML-DSA is designed to have, like strong
615 unforgeability. ML-DSA is not designed to require any additional public-key validity checks.

616 3.5.3 Intermediate Values

617 Data used internally by the key generation and signing algorithms in intermediate computation steps could
618 be used by an adversary to gain information about the private key, and thereby compromise security. For
619 some applications, including the verification of signatures that are used as bearer tokens (i.e., authentication

¹In addition, when signing is deterministic, there is leakage through timing side-channels of information about the message (but not the private key). In cases where the signer does not want to reveal the message being signed, hedged signatures should be used; see section 3.2 in [5].

620 secrets) or the verification of signatures on plaintext messages that are intended to be confidential, data
621 used internally by verification algorithms is similarly sensitive. (Intermediate values of the verification
622 algorithm may reveal information about its inputs, i.e., the message, signature, and public key, and in some
623 applications security or privacy requires one or more of these inputs to be confidential.) Implementations
624 of ML-DSA **shall**, therefore, ensure that any potentially sensitive intermediate data is destroyed as soon as
625 it is no longer needed.

626 In certain situations, such as deterministic signing (described above), and the verification of confidential
627 messages and signatures (described above), additional care must be taken to protect implementations
628 against side-channel attacks or fault attacks. A cryptographic device may leak critical information through
629 side-channels that allows internal data or keying material to be extracted without breaking the cryptographic
630 primitives.

4. Parameter Sets

Table 1. ML-DSA Parameter sets

Parameters (see sections 5 and 6 of this document)	Values assigned by each parameter set		
	ML-DSA-44	ML-DSA-65	ML-DSA-87
q - modulus [see §5]	8380417	8380417	8380417
d - # of dropped bits from \mathbf{t} [see §5]	13	13	13
τ - # of ± 1 's in polynomial c [see §6]	39	49	60
λ - collision strength of \tilde{c} [see §6]	128	192	256
γ_1 - coefficient range of \mathbf{y} [see §6]	2^{17}	2^{19}	2^{19}
γ_2 - low-order rounding range [see §6]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, ℓ) - dimensions of \mathbf{A} [see §5]	(4,4)	(6,5)	(8,7)
η - private key range [see §5]	2	4	2
$\beta = \tau \cdot \eta$ [see §6]	78	196	120
ω - max # of 1's in the hint \mathbf{h} [see §6]	80	55	75
Challenge entropy $\log \binom{256}{\tau} + \tau$ [see §6]	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 [4]. The names of the parameter sets are of the form “ML-DSA- $k\ell$,” where (k, ℓ) 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 [25]. These security strength categories are explained further in Appendix A.

Using this approach, security strength is not described by a single number, such as “128 bits of security.” Instead, each ML-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 [26].

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 [5]. For additional discussion of the security strength of MLWE-based cryptosystems, see [27].

The sizes of keys and signatures corresponding to each parameter set are given in Table 2. Note that certain optimizations are possible, when storing ML-DSA public and private keys. If additional space is available, one can pre-compute 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 only store the 32-byte seed ξ , which is sufficient to generate the other parts of the private key. For additional details, see Section 3.1 in [5].

	Private Key	Public Key	Signature Size
ML-DSA-44	2528	1312	2420
ML-DSA-65	4000	1952	3293
ML-DSA-87	4864	2592	4595

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

5. 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 begins by using an **approved** RBG to generate a 256-bit random seed ξ , which is expanded as needed using an XOF (namely, SHAKE-256) to produce other random values. In particular:

- A public random seed ρ . Using this seed, a polynomial matrix, $\mathbf{A} \in R_q^{k \times \ell}$ is pseudorandomly sampled² from $R_q^{k \times \ell}$.
- A private random seed ρ' . 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 coefficients are short, (i.e. in the range $[-\eta, \eta]$).
- A 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 thought of as 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 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 ρ and the compressed polynomial vector \mathbf{t}_1 .

The ML-DSA private key sk is a byte encoding of the public random seed ρ ; a 256-bit private random seed K for use during signing; a 512-bit hash of the public key, tr , for use during signing; the secret polynomial 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} .

²More 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 (it could be computed but is not).

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: $\xi \leftarrow \{0, 1\}^{256}$ ▷ Choose random seed
- 2: $(\rho, \rho', K) \in \{0, 1\}^{256} \times \{0, 1\}^{512} \times \{0, 1\}^{256} \leftarrow H(\xi, 1024)$ ▷ Expand seed
- 3: $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ \mathbf{A} is generated and stored in NTT representation as $\hat{\mathbf{A}}$
- 4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$
- 5: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ Compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
- 6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t}, d)$ ▷ Compress \mathbf{t}
- 7: $pk \leftarrow \text{pkEncode}(\rho, \mathbf{t}_1)$
- 8: $tr \leftarrow H(\text{BytesToBits}(pk), 512)$
- 9: $sk \leftarrow \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ K and tr are for use in signing
- 10: **return** (pk, sk)

6. Signing

679

680 ML-DSA.Sign (Algorithm 2) takes as input a private key sk , encoded as a byte string, and a message,
681 M , encoded as a bit string, and it outputs a signature encoded as a byte string. There are two versions
682 of the algorithm: “hedged” and “deterministic.” The default “hedged” version of ML-DSA.Sign uses
683 fresh randomness. In addition, for platforms where a random number generator is unavailable, an optional
684 deterministic variant is specified. However, the lack of randomness in the deterministic variant makes
685 the risk of side-channel attacks more difficult to mitigate. Therefore, this variant **should not** be used on
686 platforms where side-channel attacks are a concern and where they cannot be otherwise mitigated. (See
687 the discussion in Section 3 for more details.)

688 Note that implementing the hedged variant only (without the deterministic variant) is sufficient to guarantee
689 interoperability. The same verification algorithm will work to verify signatures produced by either
690 variant, so implementing the deterministic variant in addition to the hedged variant does not enhance
691 interoperability.

692 In both variants, the signer first extracts the following from the private key: the public random seed ρ ;
693 the 256-bit private random seed K ; the 512-bit hash of the public key, tr ; the secret polynomial vectors
694 \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
695 uncompressed public key polynomial \mathbf{t} . ρ is then expanded to the same matrix \mathbf{A} as in key generation.

696 Before the message, M , is signed, it is concatenated with the public-key hash tr and hashed down to a
697 512-bit message representative, μ , using the hash function H (see section 8.3).

698 The signer produces an additional 512-bit seed ρ' , for private randomness during each signing operation.
699 ρ' is computed as $\rho' \leftarrow H(K || rnd || \mu, 512)$. In the default “hedged” variant, rnd is the output of an RBG,
700 while in the deterministic variant rnd is a 256-bit string consisting entirely of zeroes. This is the only
701 difference between the deterministic and hedged variant of ML-DSA.Sign.

702 The main part of the signing algorithm consists of a rejection sampling loop, where each iteration of the
703 loop either produces a valid signature or an invalid signature whose release would leak information about
704 the private key. The loop is repeated until a valid signature is produced, which can then be encoded as a
705 byte string and output. The rejection sampling loop follows the Fiat-Shamir with aborts paradigm [9] and

706 (aside from the rejection step) is similar in structure to Schnorr signatures [28] (e.g., EdDSA [29]). The
 707 signer first produces a “commitment” \mathbf{w}_1 . Then the signer pseudorandomly derives a “challenge” c from
 708 \mathbf{w}_1 and the message representative μ . Finally, the signer computes a response \mathbf{z} .

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

- 710 • Using the [ExpandMask](#) function (Algorithm 28), the seed ρ' and a counter κ , a polynomial vector
 711 $\mathbf{y} \in R_q^\ell$ is pseudorandomly sampled from the subset of polynomial vectors whose coefficients are
 712 moderately short (i.e. in the range $[-\gamma_1 + 1, \gamma_1]$).
- 713 • From \mathbf{y} , the signer computes the commitment \mathbf{w}_1 by computing $\mathbf{w} = \mathbf{A}\mathbf{y}$ and then rounding to a
 714 nearby multiple of $2\gamma_2$, using [HighBits](#) (Algorithm 31).
- 715 • \mathbf{w}_1 and μ are concatenated and hashed to produce the commitment hash \tilde{c} . This uses the function
 716 [w1Encode](#) (Algorithm 22). Let \tilde{c}_1 denote the first 256 bits of \tilde{c} . The bit string \tilde{c}_1 is used to
 717 pseudorandomly sample a polynomial $c \in R_q$ that has coefficients in $\{-1, 0, 1\}$ and Hamming
 718 weight τ . The sampling is done with the function [SampleInBall](#) (Algorithm 23).³
- 719 • The signer computes the response $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ and performs various validity checks. If any of the
 720 checks fail, the signer will continue the rejection sampling loop.
- 721 • If the checks pass, the signer can compute a hint polynomial, \mathbf{h} , which will allow the verifier to
 722 reconstruct \mathbf{w}_1 using the compressed public key (along with the other components of the signature).
 723 This uses the function [MakeHint](#) (Algorithm 33). The signer will then output the final signature,
 724 which is a byte encoding of the commitment hash \tilde{c} , the response \mathbf{z} , and the hint \mathbf{h} .

725 In addition, there is an alternative way of implementing the validity checks on \mathbf{z} , and the computation of \mathbf{h} ,
 726 which is described in section 5.1 of [5]. This method may also be used in implementations of ML-DSA.

727 In Algorithm 2, variables are sometimes used to store products to avoid recomputing them later in the
 728 signing algorithm. These precomputed products are denoted in the pseudocode by a pair of double angle
 729 brackets enclosing the variables being multiplied (e.g., $\langle\langle c\mathbf{s}_1 \rangle\rangle$).

³The length of \tilde{c}_1 is determined by the targeted security strength against signature forgery attacks, and the required length is only 256 bits for 256 bits of classical security. The length of \tilde{c} is determined by the desired security with respect to the “message-bound signatures” property described in [6]. Here, a length of 2λ bits is required for λ bits of classical security.

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

Generates a signature for a message M .

Input: Private key, $sk \in \mathbb{B}^{32+32+64+32 \cdot ((\ell+k) \cdot \text{bitlen}(2\eta) + dk)}$ and the message $M \in \{0, 1\}^*$.

Output: Signature, $\sigma \in \mathbb{B}^{32+\ell \cdot 32 \cdot (1+\text{bitlen}(\gamma_1-1)) + \omega + k}$.

```

1:  $(\rho, K, tr, s_1, s_2, t_0) \leftarrow \text{skDecode}(sk)$ 
2:  $\hat{s}_1 \leftarrow \text{NTT}(s_1)$ 
3:  $\hat{s}_2 \leftarrow \text{NTT}(s_2)$ 
4:  $\hat{t}_0 \leftarrow \text{NTT}(t_0)$ 
5:  $\hat{A} \leftarrow \text{ExpandA}(\rho)$  ▷  $A$  is generated and stored in NTT representation as  $\hat{A}$ 
6:  $\mu \leftarrow H(tr || M, 512)$  ▷ Compute message representative  $\mu$ 
7:  $rnd \leftarrow \{0, 1\}^{256}$  ▷ For the optional deterministic variant, substitute  $rnd \leftarrow \{0\}^{256}$ 
8:  $\rho' \leftarrow H(K || rnd || \mu, 512)$  ▷ Compute private random seed
9:  $\kappa \leftarrow 0$  ▷ Initialize counter  $\kappa$ 
10:  $(z, h) \leftarrow \perp$ 
11: while  $(z, h) = \perp$  do ▷ Rejection sampling loop
12:    $y \leftarrow \text{ExpandMask}(\rho', \kappa)$ 
13:    $w \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(y))$ 
14:    $w_1 \leftarrow \text{HighBits}(w)$  ▷ Signer's commitment
15:    $\tilde{c} \in \{0, 1\}^{2\lambda} \leftarrow H(\mu || w_1 \text{Encode}(w_1), 2\lambda)$  ▷ Commitment hash
16:    $(\tilde{c}_1, \tilde{c}_2) \in \{0, 1\}^{256} \times \{0, 1\}^{2\lambda-256} \leftarrow \tilde{c}$  ▷ First 256 bits of commitment hash
17:    $c \leftarrow \text{SampleInBall}(\tilde{c}_1)$  ▷ Verifier's challenge
18:    $\hat{c} \leftarrow \text{NTT}(c)$ 
19:    $\langle\langle cs_1 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$ 
20:    $\langle\langle cs_2 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_2)$ 
21:    $z \leftarrow y + \langle\langle cs_1 \rangle\rangle$  ▷ Signer's response
22:    $r_0 \leftarrow \text{LowBits}(w - \langle\langle cs_2 \rangle\rangle)$ 
23:   if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|r_0\|_\infty \geq \gamma_2 - \beta$  then  $(z, h) \leftarrow \perp$  ▷ Validity checks
24:   else
25:      $\langle\langle ct_0 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{t}_0)$ 
26:      $h \leftarrow \text{MakeHint}(-\langle\langle ct_0 \rangle\rangle, w - \langle\langle cs_2 \rangle\rangle + \langle\langle ct_0 \rangle\rangle)$  ▷ Signer's hint
27:     if  $\|\langle\langle ct_0 \rangle\rangle\|_\infty \geq \gamma_2$  or the number of 1's in  $h$  is greater than  $\omega$ , then  $(z, h) \leftarrow \perp$ 
28:     end if
29:   end if
30:    $\kappa \leftarrow \kappa + \ell$  ▷ Increment counter
31: end while
32:  $\sigma \leftarrow \text{sigEncode}(\tilde{c}, z \bmod^\pm q, h)$ 
33: return  $\sigma$ 

```

7. Verification

730

731 The algorithm [ML-DSA.Verify](#) (Algorithm 3) takes as input a public key pk encoded as a byte string, a
 732 message M encoded as a bit string, and a signature σ encoded as a byte string. [ML-DSA.Verify](#) requires
 733 no randomness. It produces as output a Boolean value (i.e., a value that is true if the signature is valid
 734 with respect to the message and public key, and false if the signature is invalid). Algorithm 3 specifies the
 735 length of the signature σ and the public key pk in terms of the parameters described in section 1. If an
 736 implementation of [ML-DSA.Verify](#) can accept inputs for σ or pk of any other length, it **shall** return false
 737 whenever the length of either of these inputs differs from its specified length.

738 The verifier first extracts the public random seed ρ and the compressed polynomial vector \mathbf{t}_1 from the
 739 public key pk ; and extracts the signer’s commitment hash \tilde{c} , response \mathbf{z} , and hint \mathbf{h} from the signature σ .
 740 The verifier may find that the hint was not properly byte encoded, denoted by the symbol “ \perp ,” in which
 741 case the verification algorithm will immediately return false, indicating that the signature is invalid.

742 Assuming that the signature is successfully extracted from its byte encoding, the verifier pseudorandomly
 743 derives \mathbf{A} from ρ , as is done in key generation and signing, and creates a message representative μ , by
 744 hashing the concatenation of tr (the hash of the public key pk) and the message M .

745 The verifier then attempts to reconstruct the signer’s commitment (the polynomial vector \mathbf{w}_1) from
 746 the public key pk and the signature σ . In [ML-DSA.Sign](#), \mathbf{w}_1 is computed by rounding $\mathbf{w} = \mathbf{A}\mathbf{y}$. In
 747 [ML-DSA.Verify](#), the reconstructed value of \mathbf{w}_1 is called \mathbf{w}'_1 , since it may have been computed in a different
 748 way, in the case where the signature is invalid. This \mathbf{w}'_1 is computed through the following process:

- 749 • Derive the challenge polynomial c from the signer’s commitment hash \tilde{c} , as done in [ML-DSA.Sign](#).
- 750 • Use the signer’s response \mathbf{z} to compute

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

751 Note that assuming the signature was computed correctly, as in [ML-DSA.Sign](#), it follows that

$$\mathbf{w} = \mathbf{A}\mathbf{y} = \mathbf{A}\mathbf{z} - c\mathbf{t} + c\mathbf{s}_2 \approx \mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d,$$

752 because c and \mathbf{s}_2 have small coefficients, and $\mathbf{t}_1 \cdot 2^d \approx \mathbf{t}$.

- 753 • Use the signer’s hint \mathbf{h} to obtain \mathbf{w}'_1 from $\mathbf{w}'_{\text{Approx}}$.

754 Finally, the verifier checks that the signer’s response \mathbf{z} and the signer’s hint \mathbf{h} are valid, and that the
 755 reconstructed \mathbf{w}'_1 is consistent with the signer’s commitment hash \tilde{c} . More precisely, the verifier checks
 756 that all of the coefficients of \mathbf{z} are sufficiently small (i.e., in the range $[-(\gamma_1 - \beta), \gamma_1 - \beta]$); that \mathbf{h} contains
 757 no more than ω nonzero coefficients; and that \tilde{c} matches the hash \tilde{c}' of the message representative μ
 758 concatenated with \mathbf{w}'_1 (represented as a bit string). If all of these checks succeed, then [ML-DSA.Verify](#)
 759 returns true. Otherwise it returns false.

Algorithm 3 ML-DSA.Verify(pk, M, σ)

Verifies a signature σ for a message M .

Input: Public key, $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$ and message $M \in \{0, 1\}^*$.

Input: Signature, $\sigma \in \mathbb{B}^{32+\ell \cdot 32 \cdot (1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

Output: Boolean

```

1:  $(\rho, \mathbf{t}_1) \leftarrow \text{pkDecode}(pk)$ 
2:  $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \text{sigDecode}(\sigma)$            ▷ 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:  $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$                        ▷  $\mathbf{A}$  is generated and stored in NTT representation as  $\hat{\mathbf{A}}$ 
6:  $tr \leftarrow \text{H}(\text{BytesToBits}(pk), 512)$ 
7:  $\mu \leftarrow \text{H}(tr || M, 512)$                        ▷ Compute message representative  $\mu$ 
8:  $(\tilde{c}_1, \tilde{c}_2) \in \{0, 1\}^{256} \times \{0, 1\}^{2\lambda-256} \leftarrow \tilde{c}$ 
9:  $c \leftarrow \text{SampleInBall}(\tilde{c}_1)$                    ▷ Compute verifier's challenge from  $\tilde{c}$ 
10:  $\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$    ▷  $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$ 
11:  $\mathbf{w}'_1 \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$            ▷ Reconstruction of signer's commitment
12:  $\tilde{c}' \leftarrow \text{H}(\mu || \text{w1Encode}(\mathbf{w}'_1), 2\lambda)$    ▷ Hash it; this should match  $\tilde{c}$ 
13: return  $[\|\mathbf{z}\|_\infty < \gamma_1 - \beta]$  and  $[\tilde{c} = \tilde{c}']$  and  $[\text{number of 1's in } \mathbf{h} \text{ is } \leq \omega]$ 

```

7.1 Prehash ML-DSA

760

761 For some cryptographic modules that generate ML-DSA signatures, hashing the message in [ML-DSA.Sign](#)
762 (step 6 of Algorithm 2) may have unacceptable performance when the message M is large. For example, the
763 platform may require hardware support for hashing to achieve acceptable performance, but lack hardware
764 support for SHAKE256 specifically. For some use cases, this may be addressed by signing a digest of the
765 message rather than signing the message directly.

766 In order to maintain the same level of security strength, the digest that is signed needs to be generated using
767 an **approved** hash function or extendable-output function (XOF) (e.g., from FIPS 180 [8] or FIPS 202 [7])
768 that provides at least λ bits of classical security strength against both collision and second preimage
769 attacks [7, Table 4].⁴ Note that verification of a signature created in this way will require the verify function
770 to generate a digest from the message in the same way to be used as input for the verification function.

⁴Obtaining at least λ bits of classical security strength against collision attacks requires that the digest to be signed be at least 2λ bits in length.

771 8. Auxiliary Functions

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

774 8.1 Conversion Between Data Types

775 All keys and signatures in ML-DSA are communicated as byte strings. The goal in this subsection is to
776 construct procedures for translating between byte strings and the various other algebraic objects defined in
777 subsection 2.3.

778 Algorithms 4–7 are simple intermediate procedures for converting between bit strings, byte strings, and
779 integers.

Algorithm 4 IntegerToBits(x, α)

Computes the base-2 representation of $x \bmod 2^\alpha$ (using in little-endian order).

Input: A nonnegative integer x and a positive integer α .

Output: A bit string y of length α .

```

1: for  $i$  from 0 to  $\alpha - 1$  do
2:    $y[i] \leftarrow x \bmod 2$ 
3:    $x \leftarrow \lfloor x/2 \rfloor$ 
4: end for
5: return  $y$ 

```

Algorithm 5 BitsToInteger(y)

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

Input: A bit string y of length α .

Output: A nonnegative integer x .

```

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

```

Algorithm 6 BitsToBytes(y)

Converts a string of bits of length c into a string of bytes of length $\lceil c/8 \rceil$.

Input: A bit string y of length c .

Output: A byte string z .

```

1:  $z \leftarrow 0^{\lceil c/8 \rceil}$ 
2: for  $i$  from 0 to  $c - 1$  do
3:    $z[\lfloor i/8 \rfloor] \leftarrow z[\lfloor i/8 \rfloor] + y[i] \cdot 2^{i \bmod 8}$ 
4: end for
5: return  $z$ 

```

Algorithm 7 BytesToBits(z)

Converts a byte string into a bit string.

Input: A byte string z of length d .

Output: A bit string y .

```

1: for  $i$  from 0 to  $d - 1$  do
2:   for  $j$  from 0 to 7 do
3:      $y[8i + j] \leftarrow z[i] \bmod 2$ 
4:      $z[i] \leftarrow \lfloor z[i]/2 \rfloor$ 
5:   end for
6: end for
7: return  $y$ 

```

780 Algorithms 8–9 translate byte strings into coefficients of polynomials in R_q . `CoeffFromThreeBytes`
781 uses a three-byte string to either generate an element of $\{0, 1, \dots, q - 1\}$ or return the blank symbol \perp .
782 `CoeffFromHalfByte` uses an element of $\{0, 1, \dots, 15\}$ to either generate an element of $\{-\eta, -\eta + 1, \dots, \eta\}$
783 or return \perp . These two procedures will be used in the uniform sampling algorithms, `RejNTTPoly` and
784 `RejBoundedPoly`, discussed in subsection 8.3.

Algorithm 8 CoeffFromThreeBytes(b_0, b_1, b_2)

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

Input: Bytes b_0, b_1, b_2 .

Output: An integer modulo q or \perp .

```

1: if  $b_2 > 127$  then
2:    $b_2 \leftarrow b_2 - 128$  ▷ Set the top bit of  $b_2$  to zero
3: end if
4:  $z \leftarrow 2^{16} \cdot b_2 + 2^8 \cdot b_1 + b_0$ 
5: if  $z < q$  then return  $z$ 
6: else return  $\perp$ 
7: end if

```

Algorithm 9 CoeffFromHalfByte(b)

Generates an element of $\{-\eta, -\eta + 1, \dots, \eta\} \cup \{\perp\}$.

Input: Integer $b \in \{0, 1, \dots, 15\}$.

Output: An integer between $-\eta$ and η , or \perp .

```

1: if  $\eta = 2$  and  $b < 15$  then return  $2 - (b \bmod 5)$ 
2: else
3:   if  $\eta = 4$  and  $b < 9$  then return  $4 - b$ 
4:   else return  $\perp$ 
5:   end if
6: end if

```

785 Algorithms 10–13 efficiently translate an element $w \in R$ into a byte string and vice versa under the
786 assumption that the coefficients of w are in a restricted range. [SimpleBitPack](#) assumes that $w_i \in [0, b]$
787 for some positive integer b and packs w into a byte string of length $32 \cdot \text{bitlen } b$. [BitPack](#) allows for the
788 more general restriction $w_i \in [-a, b]$. The [BitPack](#) algorithm works by merely subtracting w from the
789 polynomial $\sum_{i=0}^{255} bX^i$ and then applying [SimpleBitPack](#).

Algorithm 10 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 $32 \cdot \text{bitlen } b$.

```

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

```

Algorithm 11 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 $32 \cdot \text{bitlen } (a + b)$.

```

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

```

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

Algorithm 12 SimpleBitUnpack(v, b)

Reverses the procedure [SimpleBitPack](#).

Input: $b \in \mathbb{N}$ and a byte string v of length $32 \cdot \text{bitlen } b$.

Output: A polynomial $w \in R$, with coefficients in $[0, 2^c - 1]$, where $c = \text{bitlen } b$.

When $b + 1$ is a power of 2, the coefficients are in $[0, b]$.

```

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

```

Algorithm 13 BitUnpack(v, a, b)

Reverses the procedure [BitPack](#).

Input: $a, b \in \mathbb{N}$ and a byte string v of length $32 \cdot \text{bitlen } (a + b)$.

Output: A polynomial $w \in R$, with coefficients in $[b - 2^c + 1, b]$, where $c = \text{bitlen } (a + b)$.

When $a + b + 1$ is a power of 2, the coefficients are in $[-a, b]$.

```

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

```

795 Algorithms [14–15](#) carry out byte-string-to-polynomial conversions for polynomials with sparse binary
796 coefficients. In particular, the signing and verification algorithms (sections [6](#) and [7](#)) make use of a “hint,”
797 which is a vector of polynomials $\mathbf{h} \in R_2^k$ such that the total number of coefficients in $\mathbf{h}[0], \mathbf{h}[1], \dots, \mathbf{h}[k - 1]$
798 that are equal to 1 is no more than ω . This constraint enables encoding and decoding procedures that are
799 more efficient (although more complex) than [BitPack](#) and [BitUnpack](#).

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

Algorithm 14 HintBitPack(**h**)

Encodes a polynomial vector **h** with binary coefficients into a byte string.

Input: A polynomial vector $\mathbf{h} \in R_2^k$ such that at most ω of the coefficients in **h** are equal to 1.

Output: A byte string y of length $\omega + k$.

```

1:  $y \in \mathbb{B}^{\omega+k} \leftarrow 0^{\omega+k}$ 
2: Index  $\leftarrow 0$ 
3: for  $i$  from 0 to  $k - 1$  do
4:   for  $j$  from 0 to 255 do
5:     if  $\mathbf{h}[i]_j \neq 0$  then
6:        $y[\text{Index}] \leftarrow j$             $\triangleright$  Store the locations of the nonzero coefficients in  $\mathbf{h}[i]$ 
7:       Index  $\leftarrow$  Index + 1
8:     end if
9:   end for
10:   $y[\omega + i] \leftarrow$  Index            $\triangleright$  Store the value of Index after processing  $\mathbf{h}[i]$ 
11: end for
12: return  $y$ 

```

Algorithm 15 HintBitUnpack(y)

Reverses the procedure [HintBitPack](#).

Input: A byte string y of length $\omega + k$.

Output: A polynomial vector $\mathbf{h} \in R_2^k$ or \perp .

```

1:  $\mathbf{h} \in R_2^k \leftarrow 0^k$ 
2: Index  $\leftarrow 0$ 
3: for  $i$  from 0 to  $k - 1$  do
4:   if  $y[\omega + i] < \text{Index}$  or  $y[\omega + i] > \omega$  then return  $\perp$ 
5:   end if
6:   while Index  $< y[\omega + i]$  do
7:      $\mathbf{h}[i]_{y[\text{Index}]} \leftarrow 1$ 
8:     Index  $\leftarrow$  Index + 1
9:   end while
10: end for
11: while Index  $< \omega$  do
12:   if  $y[\text{Index}] \neq 0$  then return  $\perp$ 
13:   end if
14:   Index  $\leftarrow$  Index + 1
15: end while
16: return  $\mathbf{h}$ 

```

804 **8.2 Encodings of ML-DSA Keys and Signatures**

805 Algorithms 16–21 translate keys and signatures for ML-DSA into byte strings. These procedures take
806 certain sequences of algebraic objects, encode them (consecutively) into byte strings, and perform the
807 respective decoding procedures.

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

Algorithm 16 `pkEncode`(ρ, \mathbf{t}_1)

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

Input: $\rho \in \{0, 1\}^{256}, \mathbf{t}_1 \in R^k$ 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: $pk \leftarrow \text{BitsToBytes}(\rho)$
 - 2: **for** i **from** 0 **to** $k - 1$ **do**
 - 3: $pk \leftarrow pk \parallel \text{SimpleBitPack}(\mathbf{t}_1[i], 2^{\text{bitlen}(q-1)-d} - 1)$
 - 4: **end for**
 - 5: **return** pk
-

Algorithm 17 `pkDecode`(pk)

Reverses the procedure `pkEncode`.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$.

Output: $\rho \in \{0, 1\}^{256}, \mathbf{t}_1 \in R^k$ with coefficients in $[0, 2^{\text{bitlen}(q-1)-d} - 1]$.

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

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

Algorithm 18 `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: $\rho \in \{0, 1\}^{256}$, $K \in \{0, 1\}^{256}$, $tr \in \{0, 1\}^{512}$, $\mathbf{s}_1 \in R^\ell$ 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, $sk \in \mathbb{B}^{32+32+64+32 \cdot ((k+\ell) \cdot \text{bitlen}(2\eta) + dk)}$.

```

1:  $sk \leftarrow \text{BitsToBytes}(\rho) \parallel \text{BitsToBytes}(K) \parallel \text{BitsToBytes}(tr)$ 
2: for  $i$  from 0 to  $\ell - 1$  do
3:    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{s}_1[i], \eta, \eta)$ 
4: end for
5: for  $i$  from 0 to  $k - 1$  do
6:    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{s}_2[i], \eta, \eta)$ 
7: end for
8: for  $i$  from 0 to  $k - 1$  do
9:    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{t}_0[i], 2^{d-1} - 1, 2^{d-1})$ 
10: end for
11: return  $sk$ 

```

Algorithm 19 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:** $\rho \in \{0, 1\}^{256}$, $K \in \{0, 1\}^{256}$, $tr \in \{0, 1\}^{512}$,
 $s_1 \in R^\ell$, $s_2 \in R^k$, $t_0 \in R^k$ with coefficients in $[-2^{d-1} + 1, 2^{d-1}]$.

- 1: $(f, g, h, y_0, \dots, y_{\ell-1}, z_0, \dots, z_{k-1}, w_0, \dots, w_{k-1}) \in \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{64} \times \left(\mathbb{B}^{32 \cdot \text{bitlen}(2\eta)}\right)^\ell \times \left(\mathbb{B}^{32 \cdot \text{bitlen}(2\eta)}\right)^k \times \left(\mathbb{B}^{32d}\right)^k \leftarrow sk$
 - 2: $\rho \leftarrow \text{BytesToBits}(f)$
 - 3: $K \leftarrow \text{BytesToBits}(g)$
 - 4: $tr \leftarrow \text{BytesToBits}(h)$
 - 5: **for** i **from** 0 **to** $\ell - 1$ **do**
 - 6: $s_1[i] \leftarrow \text{BitUnpack}(y_i, \eta, \eta)$ \triangleright This may lie outside $[-\eta, \eta]$, if input is malformed
 - 7: **end for**
 - 8: **for** i **from** 0 **to** $k - 1$ **do**
 - 9: $s_2[i] \leftarrow \text{BitUnpack}(z_i, \eta, \eta)$ \triangleright This may lie outside $[-\eta, \eta]$, if input is malformed
 - 10: **end for**
 - 11: **for** i **from** 0 **to** $k - 1$ **do**
 - 12: $t_0[i] \leftarrow \text{BitUnpack}(w_i, 2^{d-1} - 1, 2^{d-1})$ \triangleright This is always in the correct range
 - 13: **end for**
 - 14: **return** $(\rho, K, tr, s_1, s_2, t_0)$
-

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

Algorithm 20 `sigEncode`($\tilde{c}, \mathbf{z}, \mathbf{h}$)

Encodes a signature into a byte string.

Input: $\tilde{c} \in \{0, 1\}^{2\lambda}$, $\mathbf{z} \in R^\ell$ with coefficients in $[-\gamma_1 + 1, \gamma_1]$, $\mathbf{h} \in R_2^k$.

Output: Signature, $\sigma \in \mathbb{B}^{\lambda/4 + \ell \cdot 32 \cdot (1 + \text{bitlen}(\gamma_1 - 1)) + \omega + k}$.

- 1: $\sigma \leftarrow \text{BitsToBytes}(\tilde{c})$
 - 2: **for** i **from** 0 **to** $\ell - 1$ **do**
 - 3: $\sigma \leftarrow \sigma \parallel \text{BitPack}(\mathbf{z}[i], \gamma_1 - 1, \gamma_1)$
 - 4: **end for**
 - 5: $\sigma \leftarrow \sigma \parallel \text{HintBitPack}(\mathbf{h})$
 - 6: **return** σ
-

Algorithm 21 `sigDecode`(σ)

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: $\tilde{c} \in \{0, 1\}^{2\lambda}$, $\mathbf{z} \in R_q^\ell$ with coefficients in $[-\gamma_1 + 1, \gamma_1]$, $\mathbf{h} \in R_2^k$ or \perp .

- 1: $(w, x_0, \dots, x_{\ell-1}, y) \in \mathbb{B}^{\lambda/4} \times \mathbb{B}^{\ell \cdot 32 \cdot (1 + \text{bitlen}(\gamma_1 - 1))} \times \mathbb{B}^{\omega + k} \leftarrow \sigma$
 - 2: $\tilde{c} \leftarrow \text{BytesToBits}(w)$
 - 3: **for** i **from** 0 **to** $\ell - 1$ **do**
 - 4: $\mathbf{z}[i] \leftarrow \text{BitUnpack}(x_i, \gamma_1 - 1, \gamma_1)$ \triangleright This is always in the correct range, as γ_1 is a power of 2
 - 5: **end for**
 - 6: $\mathbf{h} \leftarrow \text{HintBitUnpack}(y)$
 - 7: **return** $(\tilde{c}, \mathbf{z}, \mathbf{h})$
-

819 Lastly, `w1Encode` (Algorithm 22) is a specific subroutine that is used in `ML-DSA.Sign` (Algorithm 2).
 820 Algorithm 22 encodes a polynomial vector \mathbf{w}_1 into a string of bits, so that it can be processed by the hash
 821 function, H.

Algorithm 22 `w1Encode`(\mathbf{w}_1)

Encodes a polynomial vector \mathbf{w}_1 into a bit string.

Input: $\mathbf{w}_1 \in R^k$ with coefficients in $[0, (q - 1)/(2\gamma_2) - 1]$.

Output: A bit string representation, $\tilde{\mathbf{w}}_1 \in \{0, 1\}^{32k \cdot \text{bitlen}((q-1)/(2\gamma_2)-1)}$.

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

8.3 Hashing and Pseudorandom Sampling

The ML-DSA scheme makes use of two extendable-output functions (XOFs) that will be denoted by H and H_{128} , each of which accepts a bit string v and a positive integer d and returns a length- d bit string. The functions H and H_{128} **shall** be computed from the procedures in FIPS 202 [7] as follows:

$$H(v, d) \leftarrow \text{SHAKE256}(v, d), \quad (8.1)$$

$$H_{128}(v, d) \leftarrow \text{SHAKE128}(v, d). \quad (8.2)$$

The algorithm SHAKE256 is such that if $c < d$ are positive integers and $\rho \in \{0, 1\}^*$, then $H(\rho, c)$ is exactly equal to the first c bits of $H(\rho, d)$. The same is true of SHAKE128. For convenience, the expression $H(\rho)[k]$ may be used to denote the bit $H(\rho, k+1)[k]$ for any nonnegative integer k . Similarly, $H(\rho)[j]$ denotes the byte expressed by the bit string

$$H(\rho, 8(j+1))[8j], H(\rho, 8(j+1))[8j+1], \dots, H(\rho, 8(j+1))[8j+7] \quad (8.3)$$

(in little endian-order). The expressions $H_{128}(\rho)[k]$ and $H_{128}(\rho)[j]$ are similarly defined.

The notation $H(\rho)[k]$ is used in loops where an unknown number of bits of the form $H(\rho)[k]$ will be needed to compute a pseudorandom value (for the same ρ and consecutive, increasing values of k). It is expected that implementations will avoid recomputation by keeping track of the internal state of the SHAKE256 computation throughout the loop, and will only completely destroy that information once the the loop ends. Similar implementation considerations apply when the XOF is H_{128} and when the output is parsed in bytes.

When H is used with a fixed length output, this standard sometimes refers to H as a hash function. Note that, while H used with a fixed output length is a hash function, it is not an **approved** hash function for general use. This standard only approves the use of H as a hash function where it is explicitly specified as part of the algorithms herein, or as part of a mathematically equivalent set of steps being performed in place of the steps of these algorithms. In other contexts, the fact that $H(\rho, c)$ is a prefix of $H(\rho, d)$ for any $d > c$, may interfere with desired security properties, but it is believed that when H is used as described in this standard, it is overwhelmingly unlikely that H will be used with the same input string but a different output length.

In addition, this subsection specifies various algorithms for generating algebraic objects pseudorandomly from a seed ρ . The length of the bit string ρ varies depending on the algorithm.

The first procedure to be defined is [SampleInBall](#) in Algorithm 23. Let B_τ denote the set of all polynomials $c \in R_q$ such that

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

[SampleInBall](#) generates an element of B_τ pseudorandomly using the XOF of a seed ρ . The procedure is based on the Fisher-Yates shuffle. The first 8 bytes of $H(\rho)$ are used to choose the signs of the nonzero entries of c ,⁵ and subsequent bytes of $H(\rho)$ are used to choose the positions of those nonzero entries.

⁵The parameter τ is always less than or equal to 64, and thus 8 bytes are sufficient to choose the signs for all τ nonzero entries of c .

Algorithm 23 SampleInBall(ρ)

Samples a polynomial $c \in R_q$ with coefficients from $\{-1, 0, 1\}$ and Hamming weight τ .

Input: A seed $\rho \in \{0, 1\}^{256}$

Output: A polynomial c in R_q .

```

1:  $c \leftarrow 0$ 
2:  $k \leftarrow 8$ 
3: for  $i$  from  $256 - \tau$  to  $255$  do
4:   while  $H(\rho)[[k]] > i$  do
5:      $k \leftarrow k + 1$ 
6:   end while
7:    $j \leftarrow H(\rho)[[k]]$   $\triangleright j$  is a pseudorandom byte that is  $\leq i$ 
8:    $c_i \leftarrow c_j$ 
9:    $c_j \leftarrow (-1)^{H(\rho)[i+\tau-256]}$ 
10:   $k \leftarrow k + 1$ 
11: end for
12: return  $c$ 

```

853 Algorithms 24–28 are the pseudorandom procedures [RejNTTPoly](#), [RejBoundedPoly](#), [ExpandA](#), [ExpandS](#),
854 and [ExpandMask](#). Each generates elements of R_q or T_q under different input and output conditions.
855 [RejNTTPoly](#) and [ExpandA](#) make use of the more efficient XOF H_{128} , whereas the other three procedures
856 use the XOF H .

857 The procedure [ExpandMask](#) (Algorithm 28) generates a polynomial vector \mathbf{s} in R_q^k that disguises the secret
858 key in the Sign procedure (Algorithm 2). In addition to the seed ρ , [ExpandMask](#) also accepts an integer
859 input μ that is incorporated into the pseudorandom procedure that generates \mathbf{s} .

Algorithm 24 RejNTTPoly(ρ)

Samples a polynomial $\hat{a} \in T_q$.

Input: A seed $\rho \in \{0, 1\}^{272}$.

Output: An element $\hat{a} \in T_q$.

```

1:  $j \leftarrow 0$ 
2:  $c \leftarrow 0$ 
3: while  $j < 256$  do
4:    $\hat{a}[j] \leftarrow \text{CoeffFromThreeBytes}(H_{128}(\rho)[[c]], H_{128}(\rho)[[c+1]], H_{128}(\rho)[[c+2]])$ 
5:    $c \leftarrow c + 3$ 
6:   if  $\hat{a}[j] \neq \perp$  then
7:      $j \leftarrow j + 1$ 
8:   end if
9: end while
10: return  $\hat{a}$ 

```

Algorithm 25 RejBoundedPoly(ρ)

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

Input: A seed $\rho \in \{0, 1\}^{528}$.

Output: A polynomial $a \in R_q$.

```

1:  $j \leftarrow 0$ 
2:  $c \leftarrow 0$ 
3: while  $j < 256$  do
4:    $z \leftarrow H(\rho)[c]$ 
5:    $z_0 \leftarrow \text{CoeffFromHalfByte}(z \bmod 16, \eta)$ 
6:    $z_1 \leftarrow \text{CoeffFromHalfByte}(\lfloor z/16 \rfloor, \eta)$ 
7:   if  $z_0 \neq \perp$  then
8:      $a_j \leftarrow z_0$ 
9:      $j \leftarrow j + 1$ 
10:  end if
11:  if  $z_1 \neq \perp$  and  $j < 256$  then
12:     $a_j \leftarrow z_1$ 
13:     $j \leftarrow j + 1$ 
14:  end if
15:   $c \leftarrow c + 1$ 
16: end while
17: return  $a$ 

```

Algorithm 26 ExpandA(ρ)

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

Input: $\rho \in \{0, 1\}^{256}$.

Output: Matrix $\hat{\mathbf{A}}$.

```

1: for  $r$  from 0 to  $k - 1$  do
2:   for  $s$  from 0 to  $\ell - 1$  do
3:      $\hat{\mathbf{A}}[r, s] \leftarrow \text{RejNTTPoly}(\rho || \text{IntegerToBits}(s, 8) || \text{IntegerToBits}(r, 8))$ 
4:   end for
5: end for
6: return  $\hat{\mathbf{A}}$ 

```

Algorithm 27 ExpandS(ρ)

Samples vectors $\mathbf{s}_1 \in R_q^\ell$ and $\mathbf{s}_2 \in R_q^k$, each with coefficients in the interval $[-\eta, \eta]$.

Input: $\rho \in \{0, 1\}^{512}$

Output: Vectors $\mathbf{s}_1, \mathbf{s}_2$ of polynomials in R_q .

- 1: **for** r **from** 0 **to** $\ell - 1$ **do**
 - 2: $\mathbf{s}_1[r] \leftarrow \text{RejBoundedPoly}(\rho \parallel \text{IntegerToBits}(r, 16))$
 - 3: **end for**
 - 4: **for** r **from** 0 **to** $k - 1$ **do**
 - 5: $\mathbf{s}_2[r] \leftarrow \text{RejBoundedPoly}(\rho \parallel \text{IntegerToBits}(r + \ell, 16))$
 - 6: **end for**
 - 7: **return** $(\mathbf{s}_1, \mathbf{s}_2)$
-

Algorithm 28 ExpandMask(ρ, μ)

Samples a vector $\mathbf{s} \in R_q^\ell$ such that each polynomial \mathbf{s}_j has coefficients between $-\gamma_1 + 1$ and γ_1 .

Input: A bit string $\rho \in \{0, 1\}^{512}$ and a nonnegative integer μ .

Output: Vector $\mathbf{s} \in R_q^\ell$.

- 1: $c \leftarrow 1 + \text{bitlen}(\gamma_1 - 1)$ $\triangleright \gamma_1$ is always a power of 2
 - 2: **for** r **from** 0 **to** $\ell - 1$ **do**
 - 3: $n \leftarrow \text{IntegerToBits}(\mu + r, 16)$
 - 4: $v \leftarrow (\text{H}(\rho \parallel n)[[32rc]], \text{H}(\rho \parallel n)[[32rc + 1]], \dots, \text{H}(\rho \parallel n)[[32rc + 32c - 1]])$
 - 5: $\mathbf{s}[r] \leftarrow \text{BitUnpack}(v, \gamma_1 - 1, \gamma_1)$
 - 6: **end for**
 - 7: **return** \mathbf{s}
-

8.4 High Order / Low Order Bits and Hints

This specification uses the auxiliary functions [Power2Round](#), [Decompose](#), [HighBits](#), [LowBits](#), [MakeHint](#), and [UseHint](#). This document 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, the 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.

That is:

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

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

- 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 $\mathbf{r}_1 = \text{HighBits}(\mathbf{r})$, so that:

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

- For $\mathbf{r} \in R_q^k$ define $\mathbf{r}_0 = \text{LowBits}(\mathbf{r})$, 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. The basic idea is to drop 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 obtains \mathbf{w}_1 directly using the function [HighBits](#), and the part rounded off, \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 $r \bmod q$ 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). This is because unlike for other unequal pairs of values of $r_1 \cdot 2^d$ and $r'_1 \cdot 2^d$, the distance between $\lfloor q/2^d \rfloor \cdot 2^d$ and 0 may be very small.

888 To avoid this problem, this specification defines [Decompose](#), which is similar to [Power2Round](#) except:

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

892 The functions [HighBits](#) and [LowBits](#)– which only return r_1 and r_0 , respectively – and [MakeHint](#) and
 893 [UseHint](#) use [Decompose](#). For additional discussion of the mathematical properties of these functions that
 894 are relevant to the correctness and security of ML-DSA, see Section 2.4 in [5].

Algorithm 29 [Power2Round](#)(r)

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

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 30 [Decompose](#)(r)

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

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: $r_1 \leftarrow 0$
 - 5: $r_0 \leftarrow r_0 - 1$
 - 6: **else** $r_1 \leftarrow (r^+ - r_0)/(2\gamma_2)$
 - 7: **end if**
 - 8: **return** (r_1, r_0)
-

Algorithm 31 [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 32 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 33 MakeHint(z, r)

Compute 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 34 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
-

8.5 NTT and NTT⁻¹

The following algorithms implement the NTT and its inverse (NTT⁻¹). Using the NTT is important for efficiency. There are other important optimizations that are not included in this standard. In particular, mod and mod^\pm are expensive operations whose use can be minimized by using Montgomery reduction (see Appendix B).

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 for 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.

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

$$\text{NTT}(w) = (w(\zeta_0), w(-\zeta_0), \dots, w(\zeta_{127}), w(-\zeta_{127})) \in T_q, \quad (8.4)$$

where $\zeta_i = \zeta^{\text{brv}(128+i)} \bmod q$. The values $\zeta^{\text{brv}(k)} \bmod q$ for $k = 1, \dots, 255$ used in line 10 of Algorithm 35 and line 10 of Algorithm 36 are typically pre-computed. That optimization is not included in this document.

Algorithm 35 NTT(w)

Computes the Number-Theoretic Transform.

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

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

```

1: for  $j$  from 0 to 255 do
2:    $\hat{w}[j] \leftarrow w_j$ 
3: end for
4:  $k \leftarrow 0$ 
5:  $len \leftarrow 128$ 
6: while  $len \geq 1$  do
7:    $start \leftarrow 0$ 
8:   while  $start < 256$  do
9:      $k \leftarrow k + 1$ 
10:     $zeta \leftarrow \zeta^{\text{brv}(k)} \bmod q$ 
11:    for  $j$  from  $start$  to  $start + len - 1$  do
12:       $t \leftarrow zeta \cdot \hat{w}[j + len]$ 
13:       $\hat{w}[j + len] \leftarrow \hat{w}[j] - t$ 
14:       $\hat{w}[j] \leftarrow \hat{w}[j] + t$ 
15:    end for
16:     $start \leftarrow start + 2 \cdot len$ 
17:  end while
18:   $len \leftarrow \lfloor len/2 \rfloor$ 
19: end while
20: return  $\hat{w}$ 

```

Algorithm 36 $\text{NTT}^{-1}(\hat{w})$

Computes the inverse of the Number-Theoretic Transform.

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

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

```

1: for  $j$  from 0 to 255 do
2:    $w_j \leftarrow \hat{w}[j]$ 
3: end for
4:  $k \leftarrow 256$ 
5:  $len \leftarrow 1$ 
6: while  $len < 256$  do
7:    $start \leftarrow 0$ 
8:   while  $start < 256$  do
9:      $k \leftarrow k - 1$ 
10:     $zeta \leftarrow -\zeta^{\text{brv}(k)} \bmod q$ 
11:    for  $j$  from  $start$  to  $start + len - 1$  do
12:       $t \leftarrow w_j$ 
13:       $w_j \leftarrow t + w_{j+len}$ 
14:       $w_{j+len} \leftarrow t - w_{j+len}$ 
15:       $w_{j+len} \leftarrow zeta \cdot w_{j+len}$ 
16:    end for
17:     $start \leftarrow start + 2 \cdot len$ 
18:  end while
19:   $len \leftarrow 2 \cdot len$ 
20: end while
21:  $f \leftarrow 8347681$   $\triangleright f = 256^{-1} \bmod q$ 
22: for  $j$  from 0 to 255 do
23:    $w_j \leftarrow f \cdot w_j$ 
24: end for
25: return  $w$ 

```

References

- 911
- 912 [1] National Institute of Standards and Technology. Digital signature standard (DSS). (U.S. Department
913 of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 186-5,
914 February 2023. <https://doi.org/10.6028/NIST.FIPS.186-5>.
- 915 [2] Elaine Barker. Guideline for using cryptographic standards in the federal government: Cryptographic
916 mechanisms. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special
917 Publication (SP) 800-175B, Rev. 1, March 2020. <https://doi.org/10.6028/NIST.SP.800-175Br1>.
- 918 [3] Elaine B. Barker. Recommendation for obtaining assurances for digital signature applications.
919 National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication (SP)
920 800-89, November 2006. <https://doi.org/10.6028/NIST.SP.800-89>.
- 921 [4] Shi Bai, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor
922 Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting docu-
923 mentation. Submission to the NIST’s post-quantum cryptography standardization process, 2020.
924 <https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>.
- 925 [5] Shi Bai, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor
926 Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting docu-
927 mentation (Version 3.1), 2021. <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>.
- 928
- 929 [6] C. Cremers, S. Düzlü, R. Fiedler, C. Janson, and M. Fischlin. BUFFing signature schemes beyond
930 unforgeability and the case of post-quantum signatures. In *2021 IEEE Symposium on Security and*
931 *Privacy (SP)*, pages 1696–1714, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.
- 932 [7] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and
933 extendable-output functions. (U.S. Department of Commerce, Washington, DC), Federal Information
934 Processing Standards Publication (FIPS) 202, August 2015. <https://doi.org/10.6028/NIST.FIPS.202>.
- 935 [8] National Institute of Standards and Technology. Secure hash standard (SHS). (U.S. Department of
936 Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 180-4,
937 August 2015. <https://doi.org/10.6028/NIST.FIPS.180-4>.
- 938 [9] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures.
939 In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin,
940 Heidelberg, 2009. Springer Berlin Heidelberg.
- 941 [10] Vadim Lyubashevsky. Lattice signatures without trapdoors. In *EUROCRYPT*, volume 7237 of *Lecture*
942 *Notes in Computer Science*, pages 738–755. Springer, 2012.
- 943 [11] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography:
944 A signature scheme for embedded systems. In *CHES*, volume 7428, pages 530–547. Springer, 2012.
- 945 [12] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on
946 learning with errors. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, pages 28–47,
947 Cham, 2014. Springer International Publishing.
- 948 [13] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings*
949 *of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, STOC ’05*, page 84–93,
950 New York, NY, USA, 2005. Association for Computing Machinery.

- 951 [14] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices.
952 *Designs, Codes and Cryptography*, 75(3):565–599, 2015.
- 953 [15] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir
954 signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors,
955 *Advances in Cryptology – EUROCRYPT 2018*, pages 552–586, Cham, 2018. Springer International
956 Publishing.
- 957 [16] Elaine B. Barker. Recommendation for key management: part 1 - general. National Institute of
958 Standards and Technology, Gaithersburg, MD. NIST Special Publication (SP) 800-57 Part 1 Revision
959 5, May 2020. <https://doi.org/10.6028/NIST.SP.800-57pt1r5>.
- 960 [17] Elaine B. Barker and William C. Barker. Recommendation for key management: Part 2 - best
961 practices for key management organizations. National Institute of Standards and Technology,
962 Gaithersburg, MD. NIST Special Publication (SP) 800-57 Part 2 Revision 1, May 2019. <https://doi.org/10.6028/NIST.SP.800-57pt2r1>.
- 964 [18] Elaine B. Barker and Quynh Dang. Recommendation for key management: Part 3 - application-
965 specific key management guidance. National Institute of Standards and Technology, Gaithersburg,
966 MD. NIST Special Publication (SP) 800-57 Part 3 Revision 1, May 2019. [http://dx.doi.org/10.6028](http://dx.doi.org/10.6028/NIST.SP.800-57pt3r1)
967 [/NIST.SP.800-57pt3r1](http://dx.doi.org/10.6028/NIST.SP.800-57pt3r1).
- 968 [19] Elaine B. Barker and John M. Kelsey. Recommendation for random number generation using
969 deterministic random bit generators. (National Institute of Standards and Technology, Gaithersburg,
970 MD), NIST Special Publication (SP) 800-90A, Rev. 1, June 2015. [https://doi.org/10.6028/NIST.SP.](https://doi.org/10.6028/NIST.SP.800-90Ar1)
971 [800-90Ar1](https://doi.org/10.6028/NIST.SP.800-90Ar1).
- 972 [20] Meltem Sönmez Turan, Elaine B. Barker, John M. Kelsey, Kerry A. McKay, Mary L. Baish, and
973 Mike Boyle. Recommendation for the entropy sources used for random bit generation. (National
974 Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B,
975 January 2018. <https://doi.org/10.6028/NIST.SP.800-90B>.
- 976 [21] Elaine B. Barker, John M. Kelsey, Kerry McKay, Allen Roginsky, and Meltem Sönmez Turan.
977 Recommendation for random bit generator (RBG) constructions. (National Institute of Standards
978 and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90C (Third Public Draft),
979 September 2022. <https://csrc.nist.gov/publications/detail/sp/800-90c/draft>.
- 980 [22] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures.
981 *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 21–43, 2018.
- 982 [23] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler.
983 Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium*
984 *on Security and Privacy (EuroS&P)*, pages 338–352. IEEE, 2018.
- 985 [24] Niels Samwel, Lejla Batina, Guido Bertoni, Joan Daemen, and Ruggero Susella. Breaking ed25519
986 in wolfssl. In *Topics in Cryptology–CT-RSA 2018: The Cryptographers’ Track at the RSA Conference*
987 *2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 1–20. Springer, 2018.
- 988 [25] National Institute of Standards and Technology. Submission requirements and evaluation criteria for
989 the post-quantum cryptography standardization process, 2016. [https://csrc.nist.gov/CSRC/media/Pro](https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf)
990 [jects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf](https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf).

- 991 [26] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger,
992 Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel
993 Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization
994 process. Technical Report NIST Interagency or Internal Report (IR) 8413, National Institute of
995 Standards and Technology, Gaithersburg, MD, July 2022. <https://doi.org/10.6028/NIST.IR.8413-upd>
996 [1](#).
- 997 [27] Robert Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M.
998 Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifica-
999 tions and supporting documentation. 3rd Round submission to the NIST’s post-quantum cryptography
1000 standardization process, 2020. <https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-sub>
1001 [missions](#).
- 1002 [28] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor,
1003 *Advances in Cryptology — CRYPTO’ 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer
1004 New York.
- 1005 [29] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC
1006 8032, January 2017.
- 1007 [30] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. Implementing Grover
1008 oracles for quantum key search on AES and LowMC. In Anne Canteaut and Yuval Ishai, editors,
1009 *Advances in Cryptology – EUROCRYPT 2020*, pages 280–310, Cham, 2020. Springer International
1010 Publishing.
- 1011 [31] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the*
1012 *Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC ’96, page 212–219, New
1013 York, NY, USA, 1996. Association for Computing Machinery.

1014 **Appendix A — Security Strength Categories**

1015 NIST understands that there are significant uncertainties in estimating the security strengths of post-
1016 quantum cryptosystems. These uncertainties come from two sources: first, the possibility that new
1017 cryptanalytic attacks are discovered, based on classical or quantum computation; and second, our limited
1018 ability to predict the performance characteristics of future quantum computers, such as their cost, speed,
1019 and memory size.

1020 In order to address these uncertainties, NIST proposed the following approach in its original Call for
1021 Proposals [25]. Instead of defining the strength of an algorithm using precise estimates of the number
1022 of “bits of security,” NIST defined a collection of broad security strength categories. Each category is
1023 defined by a comparatively easy-to-analyze reference primitive whose security serves as a floor for a wide
1024 variety of metrics that NIST deems potentially relevant to practical security. A given cryptosystem may
1025 be instantiated using different parameter sets in order to fit into different categories. The goals of this
1026 classification are:

- 1027 • To facilitate meaningful performance comparisons between various post-quantum algorithms by
1028 ensuring – insofar as possible – that the parameter sets being compared provide comparable security
- 1029 • To allow NIST to make prudent future decisions regarding when to transition to longer keys
- 1030 • To help submitters make consistent and sensible choices regarding what symmetric primitives to use
1031 in padding mechanisms or other components of their schemes that require symmetric cryptography
- 1032 • To better understand the security/performance trade-offs involved in a given design approach

1033 In accordance with the second and third goals above, NIST based its classification on the range of security
1034 strengths offered by the existing NIST standards in symmetric cryptography, which NIST expects to offer
1035 significant resistance to quantum cryptanalysis. In particular, NIST defined a separate category for each of
1036 the following security requirements (listed in order of increasing strength):

- 1037 1. Any attack that breaks the relevant security definition must require computational resources com-
1038 parable to or greater than those required for key search on a block cipher with a 128-bit key (e.g.,
1039 AES-128).
- 1040 2. Any attack that breaks the relevant security definition must require computational resources compa-
1041 rable to or greater than those required for collision search on a 256-bit hash function (e.g., SHA-256/
1042 SHA3-256).
- 1043 3. Any attack that breaks the relevant security definition must require computational resources com-
1044 parable to or greater than those required for key search on a block cipher with a 192-bit key (e.g.,
1045 AES-192).
- 1046 4. Any attack that breaks the relevant security definition must require computational resources compa-
1047 rable to or greater than those required for collision search on a 384-bit hash function (e.g., SHA-384/
1048 SHA3-384).
- 1049 5. Any attack that breaks the relevant security definition must require computational resources com-
1050 parable to or greater than those required for key search on a block cipher with a 256-bit key (e.g.,
1051 AES-256).

Table 3. NIST Security Strength Categories

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

1052 When considering the five categories, computational resources may be measured using a variety of
 1053 different metrics (e.g., number of classical elementary operations, quantum circuit size). In order for a
 1054 cryptosystem to satisfy one of the above security requirements, any attack must require computational
 1055 resources comparable to or greater than the stated threshold with respect to all metrics that NIST deems to
 1056 be potentially relevant to practical security.

1057 NIST intends to consider a variety of possible metrics that reflect different predictions about the future
 1058 development of quantum and classical computing technology and the cost of different computing resources
 1059 (such as the cost of accessing extremely large amounts of memory).⁶ NIST will also consider input from
 1060 the cryptographic community regarding this question.

1061 In an example metric provided to submitters, NIST suggested an approach where quantum attacks are
 1062 restricted to a fixed running time or circuit depth. Call this parameter MAXDEPTH. This restriction is moti-
 1063 vated by the difficulty of running extremely long serial computations. Plausible values for MAXDEPTH
 1064 range from 2^{40} logical gates (the approximate number of gates that presently envisioned quantum comput-
 1065 ing architectures are expected to serially perform in a year) through 2^{64} logical gates (the approximate
 1066 number of gates that current classical computing architectures can perform serially in a decade), to no
 1067 more than 2^{96} logical gates (the approximate number of gates that atomic scale qubits with speed-of-light
 1068 propagation times could perform in a millennium). The most basic version of this cost metric ignores
 1069 costs associated with physically moving bits or qubits so that they are physically close enough to per-
 1070 form gate operations. This simplification may result in an underestimate of the cost of implementing
 1071 memory-intensive computations on real hardware.

1072 The complexity of quantum attacks can then be measured in terms of circuit size. These numbers can be
 1073 compared to the resources required to break AES and SHA-3. During the post-quantum standardization
 1074 process, NIST gave the following estimates for the classical and quantum gate counts⁷ for the optimal
 1075 key recovery and collision attacks on AES and SHA-3, respectively, where circuit depth is limited to
 1076 MAXDEPTH.⁸

⁶See the discussion in [26, Appendix B].

⁷Quantum circuit sizes are based on the work in [30].

⁸NIST believes that the above estimates are accurate for the majority of values of MAXDEPTH that are relevant to its security analysis, but the above estimates may understate the security of SHA for very small values of MAXDEPTH and may understate the quantum security of AES for very large values of MAXDEPTH.

Table 4. Estimated gate counts for the optimal key recovery and collision attacks on AES and SHA-3

Algorithm	Estimated number of gates
AES-128	$2^{157}/\text{MAXDEPTH}$ quantum gates or 2^{143} classical gates
SHA3-256	2^{146} classical gates
AES-192	$2^{221}/\text{MAXDEPTH}$ quantum gates or 2^{207} classical gates
SHA3-384	2^{210} classical gates
AES-256	$2^{285}/\text{MAXDEPTH}$ quantum gates or 2^{272} classical gates
SHA3-512	2^{274} classical gates

1077 It is worth noting that the security categories based on these reference primitives provide substantially
 1078 more quantum security than a naïve analysis might suggest. For example, categories 1, 3, and 5 are defined
 1079 in terms of block ciphers, which can be broken using Grover’s algorithm [31] with a quadratic quantum
 1080 speedup. However, Grover’s algorithm requires a long-running serial computation, which is difficult to
 1081 implement in practice. In a realistic attack, one has to run many smaller instances of the algorithm in
 1082 parallel, which makes the quantum speedup less dramatic.

1083 Finally, for attacks that use a combination of classical and quantum computation, one may use a cost metric
 1084 that rates logical quantum gates as being several orders of magnitude more expensive than classical gates.
 1085 Presently envisioned quantum computing architectures typically indicate that the cost per quantum gate
 1086 could be billions or trillions of times the cost per classical gate. However, especially when considering
 1087 algorithms that claim a high security strength (e.g., equivalent to AES-256 or SHA-384), it is likely prudent
 1088 to consider the possibility that this disparity will narrow significantly or even be eliminated.

1089 Appendix B — Montgomery Reduction

1090 This document uses modular multiplication. This is an expensive operation that is, in practice, often
 1091 avoided. One way of achieving this is through the use of Montgomery multiplication. If a is an integer
 1092 modulo q , then its *Montgomery form with multiplier 2^{32}* is $r \equiv a \cdot 2^{32} \pmod{q}$.

1093 Suppose two integers a and b modulo q are in Montgomery form. Their product modulo q is $c = a \cdot b \cdot 2^{-32}$,
 1094 also in Montgomery form. If a and b have absolute value less than q , one can compute c by first performing
 1095 the integer multiplication $a \cdot b$ and then “reducing” the product by multiplying by 2^{-32} modulo q . This last
 1096 operation can be done efficiently as follows.

1097 The `Montgomery_Reduce` function takes as input an integer a with absolute value at most $2^{31}q$. It returns
 1098 an integer r with absolute value strictly less than q and such that $r = a \cdot 2^{-32} \pmod{q}$. The output is in
 1099 Montgomery form with multiplier $2^{32} \pmod{q}$. An implementation would typically use a 64-bit input and
 1100 return a 32-bit output. The “modulo 2^{32} ” operation simply extracts the 32 least significant bits of a 64-bit
 1101 value. The value $(a - t \cdot q)$ on line 3 is an integer divisible by 2^{32} . Therefore, the division consists simply
 1102 of taking the most significant 32 bits of a 64-bit value.

Algorithm 37 `Montgomery_Reduce(a)`

Converts from Montgomery form to regular form.

Input: integer a with $-2^{31}q \leq a \leq 2^{31}q$.

Output: $r \equiv a \cdot 2^{-32} \pmod{q}$ such that $-q < r < q$.

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

1103 With this algorithm, the modular product of a and b is `Montgomery_Reduce(a · b)`.

1104 Converting an integer modulo q to Montgomery form by multiplying by 2^{32} modulo q is an expensive
 1105 operation. When a sequence of modular operations is to be performed, as in Algorithms 35 and 36, the
 1106 operands are converted once to Montgomery form, the operations are performed, and the factor 2^{32} is
 1107 extracted from the final results.