Withdrawn Draft

Warning Notice

The attached draft document has been withdrawn and is provided solely for historical purposes.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

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

The attached draft document is followed by:

Status	Final
Series/Number	NIST FIPS 205
Title	Stateless Hash-Based Digital Signature Standard
Publication Date	August 13, 2024
DOI	https://doi.org/10.6028/NIST.FIPS.205
CSRC URL	https://csrc.nist.gov/pubs/fips/205/final
Additional Information	https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

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

FIPS 205 (Draft)

Federal Information Processing Standards Publication

Stateless Hash-Based Digital Signature Standard

Category: Computer Security **Subcategory: Cryptography**

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

This publication is available free of charge from:
https://doi.org/10.6028/NIST.FIPS.205.ipd

Published: August 24, 2023

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

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

Foreword

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

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

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

Abstract

This standard specifies the stateless hash-based digital signature algorithm (SLH-DSA). Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of the signatory. In addition, the recipient of signed data can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation since the signatory cannot easily repudiate the signature at a later time. SLH-DSA is based on SPHINCS$^+$, which was selected for standardization as part of the NIST Post-Quantum Cryptography Standardization process.

Keywords: computer security; cryptography; digital signatures; Federal Information Processing Standards; hash-based signatures; public-key cryptography

Federal Information Processing Standards Publication 205

Published: August 24, 2023

Announcing the

Stateless Hash-Based Digital Signature Standard

Federal Information Processing Standards Publications (FIPS) 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.** Stateless Hash-Based Digital Signature Standard (FIPS 205).

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

3. **Explanation.** This standard specifies a stateless hash-based digital signature scheme, SLH-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 private and public key pair. 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 signature generation.

   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 SLH-DSA that are **approved** for use. Additional parameter sets may be specified and approved in future NIST Special Publications.

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

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

6. **Applicability.** This standard is applicable to all federal departments and agencies for the protection of sensitive unclassified information that is not subject to section 2315 of Title 10, United States Code, or section 3502 (2) of Title 44, United States Code. Either this standard, FIPS 204, FIPS 186-5, or NIST Special Publication 800-208 **shall** be used in designing and implementing public-key-based signature systems that federal departments and agencies operate or that are operated for them under contract. In the future, additional digital signature schemes may be specified and approved in FIPS publications or in NIST Special Publications.

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

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

8. **Implementations.** A digital signature algorithm may be implemented in software, firmware, hardware, or any combination thereof. NIST will develop a validation program to test implementations for conformance to the algorithms in this standard. For every computational procedure that is specified in this standard, a conforming implementation may replace the given set of steps with any mathematically equivalent set of steps. In other words, different procedures that produce the correct output for every input are permitted. Information about validation programs is available at https://csrc.nist.gov/projects/cmvp. Examples for digital signature algorithms are available at https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

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

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

   a. Specified in a Federal Information Processing Standard (FIPS),

   b. Adopted in a FIPS or NIST recommendation, or

   c. Specified in the list of **approved** security functions for FIPS 140-3.

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

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

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

13. **Specifications.** Federal Information Processing Standard (FIPS) 205, Stateless Hash-Based Digital Signature Standard (affixed).

14. **Qualifications.** The security of a digital signature system is dependent on the secrecy of the signatory's private keys. Signatories **shall**, therefore, guard against the disclosure of their private keys. While it is the intent of this standard to specify general security requirements for generating digital signatures, conformance to this standard does not ensure that a particular

implementation is secure. It is the responsibility of an implementer to ensure that any module that implements a digital signature capability is designed and built in a secure manner.

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

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

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

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

17. **How to Cite this Publication.** NIST has assigned **NIST FIPS 205 ipd** as the publication identifier for this FIPS, per the NIST Technical Series Publication Identifier Syntax. NIST recommends that it be cited as follows:

    National Institute of Standards and Technology (2023) Stateless Hash-Based Digital Signature Standard. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 205 ipd. https://doi.org/10.6028/NIST.FIPS.205.ipd

18. **Inquiries and Comments.** Inquiries and comments about this FIPS may be submitted to fips-205-comments@nist.gov.

Call for Patent Claims

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

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

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

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

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

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

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

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

Such statements should be addressed to: fips-205-comments@nist.gov

Federal Information Processing Standards Publication 205

Specification for the Stateless Hash-Based Digital Signature Standard

Table of Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

1.1 Purpose and Scope

This standard defines a method for digital signature generation that can be used for the protection of binary data (commonly called a message) and 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.) The security of SLH-DSA relies on the presumed difficulty of finding preimages for hash functions as well as several related properties of the same hash functions. Unlike the algorithms specified in FIPS 186-5 [1], SLH-DSA is expected to provide resistance to attacks from a large-scale quantum computer.

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.

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. These algorithms are intended to protect sensitive U.S. Government information well into the foreseeable future, including after the advent of quantum computers. This standard includes the specification for one of the algorithms selected: SPHINCS$^+$, a stateless hashed-based digital signature scheme. Throughout this standard, SPHINCS$^+$ will be referred to as *SLH-DSA* for stateless hash-based digital signature algorithm.

1.3 Differences From the SPHINCS$^+$ Submission

This standard is based on version 3.1 of the SPHINCS$^+$ specification [4], and contains several minor modifications compared to version 3 [5], which was submitted at the beginning of round three of the NIST PQC Standardization process:

- Two new address types were defined, `WOTS_PRF` and `FORS_PRF`, which are used for WOTS$^+$ and FORS secret key value generation.
- **PK**.seed was added as an input to **PRF** in order to mitigate multi-key attacks.

- For the category 3 and 5 parameter sets that use SHA-2, SHA-256 was replaced with SHA-512 in $\mathbf{H}_{msg}$, $\mathbf{PRF}_{msg}$, $\mathbf{H}$, and $\mathbf{T}_l$ based on weaknesses that were discovered when using SHA-256 to obtain category 5 security [6, 7, 8].
- R and **PK**.seed were added as inputs to MGF1 when computing $\mathbf{H}_{msg}$ for the SHA-2 parameter sets in order to mitigate against multi-target long-message second preimage attacks.

In addition to the changes that appear in version 3.1 of the SPHINCS$^+$ specification, this standard differs from the version 3 specification in its method for extracting bits from the message digest for selecting a forest of random subsets (FORS) key. This change was made in order to align with the reference implementation that was submitted along with the round three specification. The description of the method for extracting indices for FORS signature generation and verification from the message digest was also changed due to ambiguity in the submitted specification. The method described in this standard is not compatible with the method used in the reference implementation that was submitted along with the round three specification. Also, step 9 in both wots_sign and wots_PKFromSig were changed the match the reference implementation, as the pseudocode in [4, 5] will sometimes shift *csum* by the incorrect amount when lg_w is not 4.

This standard **approves** the use of only 12 of the 36 parameter sets defined in [4, 5]. As specified in Section 10, only the ‘simple’ instances in which the cryptographic functions are instantiated with SHA-2 or SHAKE are **approved**.

2. Glossary of Acronyms, Terms, and Mathematical Symbols

2.1 Acronyms

ADRS	Address
$\mathrm{ADRS^c}$	Compressed Address
AES	Advanced Encryption Standard
FIPS	Federal Information Processing Standard
FORS	Forest of Random Subsets
ITL	Information Technology Laboratory
MGF	Mask Generation Function
NIST	National Institute of Standards and Technology
PQC	Post-Quantum Cryptography
PRF	Pseudorandom Function
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm KECCAK
SP	Special Publication
RFC	Request for Comments
WOTS^+	Winternitz One-Time Signature Plus
XMSS	eXtended Merkle Signature Scheme
XOF	eXtendable-Output Function

2.2 Terms and Definitions

approved	FIPS-approved and/or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST recommendation, 2) adopted in a FIPS or NIST recommendation, or 3) specified in a list of NIST-**approved** security functions. [1]
big-endian	The property of a byte string having its bytes positioned in order of decreasing significance. In particular, the leftmost (first) byte is the most significant, and the rightmost (last) byte is the least significant. The term “big-endian” may also be applied in the same manner to bit strings. [9, adapted]
byte string	An array of integers in which each integer is in the set $\{0,\ldots,255\}$.
claimed signatory	From the verifier’s perspective, the claimed signatory is the entity that purportedly generated a digital signature. [1]

destroy	An action applied to a key or a piece of secret data. After a key or a piece of secret data is destroyed, no information about its value can be recovered. [1]
digital signature	The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity, and signatory non-repudiation. [1]
entity	An individual (person), organization, device, or process. Used interchangeably with "party." [1]
equivalent process	Two processes are equivalent if the same output is produced when the same values are input to each process (either as input parameters, as values made available during the process, or both). [1]
extendable-output function	A function on bit strings in which the output can be extended to any desired length. **Approved** XOFs (such as those specified in FIPS 202 [10]) are designed to satisfy the following properties as long as the specified output length is sufficiently long to prevent trivial attacks: 1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output. 2. (Collision-resistant) It is computationally infeasible to find any two distinct inputs that map to the same output. [11, adapted]
hash function	A function on bit strings in which the length of the output is fixed. **Approved** hash functions (such as those specified in FIPS 180 [12] and FIPS 202 [10]) are designed to satisfy the following properties: 1. (One-way) It is computationally infeasible to find any input that maps to any new pre-specified output 2. (Collision-resistant) It is computationally infeasible to find any two distinct inputs that map to the same output. [1]
hash value	See "message digest." [1]
key	A parameter used in conjunction with a cryptographic algorithm that determines its operation. Examples applicable to this standard include: 1. The computation of a digital signature from data, and 2. The verification of a digital signature. [1]
key pair	A public key and its corresponding private key. [1]
message	The data that is signed. Also known as "signed data" during the signature verification and validation process. [1]
message digest	The result of applying a hash function to a message. Also known as a "hash value." [1]
non-repudiation	A service that is used to provide assurance of the integrity and origin of data in such a way that the integrity and origin can be verified and

	validated by a third party as having originated from a specific entity in possession of the private key (i.e., the signatory). [1]
owner	A key pair owner is the entity authorized to use the private key of a key pair. [1]
party	An individual (person), organization, device, or process. Used interchangeably with “entity.” [1]
private key	A cryptographic key that is used with an asymmetric (public-key) cryptographic algorithm. The private key is uniquely associated with the owner and is not made public. The private key is used to compute a digital signature that may be verified using the corresponding public key. [1]
pseudorandom	A process or data produced by a process is said to be pseudorandom when the outcome is deterministic yet also effectively random as long as the internal action of the process is hidden from observation. For cryptographic purposes, “effectively random” means “computationally indistinguishable from random within the limits of the intended security strength.” [1]
public key	A cryptographic key that is used with an asymmetric (public-key) cryptographic algorithm and is associated with a private key. The public key is associated with an owner and may be made public. In the case of digital signatures, the public key is used to verify a digital signature that was generated using the corresponding private key. [1]
security category	A number associated with the security strength of a post-quantum cryptographic algorithm as specified by NIST (see Appendix A, Table 2).
security strength	A number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system. [1]
shall	Used to indicate a requirement of this standard. [1]
should	Used to indicate a strong recommendation but not a requirement of this standard. Ignoring the recommendation could result in undesirable results. [1]
signatory	The entity that generates a digital signature on data using a private key. [1]
signature generation	The process of using a digital signature algorithm and a private key to generate a digital signature on data. [1]
signature validation	The (mathematical) verification of the digital signature and obtaining the appropriate assurances (e.g., public-key validity, private-key possession, etc.). [1]

signature verification — The process of using a digital signature algorithm and a public key to verify a digital signature on data. [1]

signed data — The data or message upon which a digital signature has been computed. Also see "message." [1]

verifier — The entity that verifies the authenticity of a digital signature using the public key. [1]

2.3 Mathematical Symbols

The following notation is used in this standard.

$X \parallel Y$ — The concatenation of two arrays X and Y. If X is an array of length ℓ_x and Y is an array of length ℓ_y, then $Z = X \parallel Y$ is an array of length $\ell_x + \ell_y$ such that

$$Z[i] = \begin{cases} X[i] & \text{if } 0 \le i < \ell_x \\ Y[i - \ell_x] & \text{if } \ell_x \le i < \ell_x + \ell_y \end{cases}$$

$X[i:j]$ — A subarray of X. If X is an array of length ℓ_x, $0 \le i < j \le \ell_x$, and $Y = X[i:j]$, then Y is an array of length $j - i$ such that $Y[k] = X[i+k]$ for $0 \le k < j - i$.

$\mathsf{Trunc}_\ell(X)$ — A truncation function that outputs the most significant ℓ bytes of the input byte string X. If $Y = \mathsf{Trunc}_\ell(X)$, then Y is a byte string (array) of length ℓ such that $Y[i] = X[i]$ for $0 \le i < \ell$ (i.e., $Y = X[0:\ell]$).

$|X|$ — The length (in bytes) of byte string X.

$\lceil a \rceil$ — The ceiling of a; the smallest integer that is greater than or equal to a. For example, $\lceil 5 \rceil = 5$, $\lceil 5.3 \rceil = 6$, and $\lceil -2.1 \rceil = -2$. [1]

$\lfloor a \rfloor$ — The floor of a; the largest integer that is less than or equal to a. For example, $\lfloor 5 \rfloor = 5$, $\lfloor 5.3 \rfloor = 5$, and $\lfloor -2.1 \rfloor = 3$. [1]

$a \bmod n$ — The unique remainder r, $0 \le r \le (n-1)$, when integer a is divided by the positive integer n. For example, $23 \bmod 7 = 2$. [1]

$a \cdot b$ — The product of a and b. For example, $3 \cdot 5 = 15$.

a^b — a raised to the power b. For example, $2^5 = 32$.

$\log_2 x$ — The base 2 logarithm of x. For example, $\log_2(16) = 4$.

0b — The prefix to a number that is represented in binary.

0x — The prefix to a number that is represented in hexadecimal. [1, adapted]

$a \gg b$ — The logical right shift of a by b positions (i.e., $a \gg b = \lfloor a/2^b \rfloor$). For example. $0x73 \gg 4 = 7$. [4, adapted]

$a \ll b$ — The logical left shift of a by b positions (i.e., $a \ll b = a \cdot 2^b$). For example. $0x73 \ll 4 = 0x730$. [4, adapted]

$a \oplus b$	The bitwise exclusive-or of a and b. For example, $115 \oplus 1 = 114$ ($115 \oplus 1 = 0\text{b}01110011 \oplus 0\text{b}00000001 = 0\text{b}01110010 = 114$).
$s \leftarrow x$	In pseudocode, this notation means that the variable s is set to the value of the expression x.
$s \xleftarrow{\$} \mathbb{B}^n$	In pseudocode, this notation means that the variable s is set to a byte string of length n chosen at random. A fresh random value must be generated for each time this step is performed.

3. Overview of the SLH-DSA Signature Scheme

SLH-DSA is a stateless hash-based signature scheme that is constructed using other hash-based signature schemes as components: a few-time signature scheme, forest of random subsets (FORS), and a multi-time signature scheme, the eXtended Merkle Signature Scheme (XMSS). XMSS is constructed using the hash-based one-time signature scheme Winternitz One-Time Signature Plus (WOTS$^+$) as a component.[1]

Conceptually, an SLH-DSA key pair consists of a very large set of FORS key pairs.[2] The few-time signature scheme FORS allows each key pair to safely sign a small number of messages (about 10 for the parameter sets in this standard). An SLH-DSA signature is created by computing a randomized hash of the message, using part of the resulting message digest to (pseudorandomly) select a FORS key, and signing the remaining part of the message digest with that key. An SLH-DSA signature consists of the FORS signature along with information that authenticates the FORS public key. The authentication information is created using XMSS signatures.

XMSS is a multi-time signature scheme that is created using a combination of WOTS$^+$ one-time signatures and Merkle hash trees [15]. An XMSS key consists of $2^{h'}$ WOTS$^+$ keys and can sign $2^{h'}$ messages. The WOTS$^+$ public keys are formed into a Merkle hash tree, and the root of the tree is the XMSS public key. (The Merkle hash tree formed from the WOTS$^+$ keys is also referred to as an XMSS tree.) An XMSS signature consists of a WOTS$^+$ signature and an authentication path within the Merkle hash tree for the WOTS$^+$ public key. In Figure 1, each triangle represents an XMSS tree with squares representing the WOTS$^+$ public keys and circles representing the interior nodes of the hash tree. The square and circles that are filled in represent the authentication path for the WOTS$^+$ public key needed to verify the signature.

The authentication information for a FORS public key is a hypertree signature. A hypertree is a tree of XMSS trees, as depicted in Figure 1. The tree consists of d layers,[3] with the top layer (layer $d-1$) consisting of a single XMSS tree, the next layer down (layer $d-2$) consisting of $2^{h'}$ XMSS trees, and the lowest layer (layer 0) consisting of $2^{(d-1)h'}$ XMSS trees. The public key of each XMSS key at layers 0 through $d-2$ is signed by an XMSS key at the next higher layer. The XMSS keys at layer 0 collectively have $2^{dh'} = 2^h$ WOTS$^+$ keys, which are used to sign the 2^h FORS public keys in the SLH-DSA key pair. The sequence of d XMSS signatures needed to authenticate a FORS public key when starting with the public key of the XMSS key at layer $d-1$ is a hypertree signature. An SLH-DSA signature consists of a FORS signature along with a hypertree signature.

An SLH-DSA public key contains two n-byte components: **PK**.root, which is the public key of the XMSS key at layer $d-1$; and **PK**.seed, which is used to provide domain separation between different SLH-DSA key pairs. An SLH-DSA private key consists of an n-byte seed **SK**.seed, which is used to pseudorandomly generate all of the secret values for the WOTS$^+$ and FORS keys, and an n-byte key **SK**.prf, which is used in the generation of the randomized hash of the message. An SLH-DSA private key also includes copies of **PK**.root and **PK**.seed, as these values

[1]The WOTS$^+$ and XMSS schemes that are used as components of SLH-DSA are not the same as the WOTS$^+$ and XMSS schemes in RFC 8391 [13] and NIST SP 800-208 [14].

[2]For the parameter sets in this standard, an SLH-DSA key pair contains 2^{63}, 2^{64}, 2^{66}, or 2^{68} FORS keys, which are pseudorandomly generated from a single seed.

[3]For the parameter sets in this standard, d is 7, 8, 17, or 22.

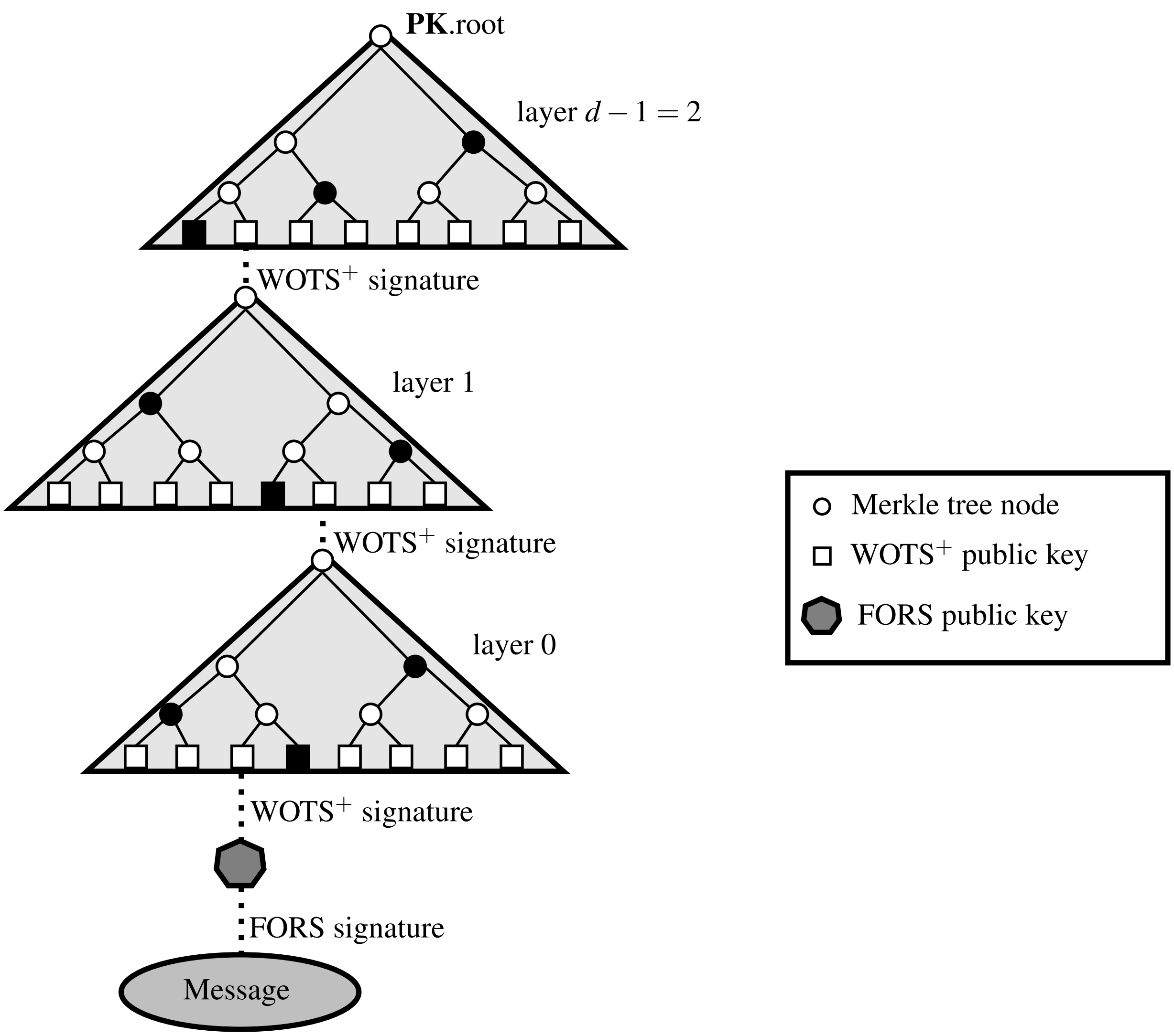

Figure 1. An SLH-DSA signature

are needed during both signature generation and signature verification.

The WOTS$^+$ one-time signature scheme is specified in Section 5, and the XMSS multi-time signature scheme is specified in Section 6. Section 7 specifies the generation and verification of hypertree signatures. The FORS few-time signature scheme is specified in Section 8. Finally, Section 9 specifies the SLH-DSA key generation, signature, and verification functions. As the WOTS$^+$, XMSS, hypertree, and FORS schemes described in this standard are not intended for use as stand-alone signature schemes, only the components of the schemes necessary to implement SLH-DSA are described. In particular, these sections do not include functions for key pair generation, and a signature verification function is only specified for hypertree signatures.

When used in this standard, WOTS$^+$, XMSS, and FORS signatures are verified implicitly using functions to generate public keys from messages and signatures (see Sections 5.3, 6.3, and 8.4). When verifying an SLH-DSA signature, the randomized hash of the message and the FORS signature are used to compute a candidate FORS public key. The candidate FORS public key and the WOTS$^+$ signature from the layer 0 XMSS key are used to compute a candidate WOTS$^+$ public key, and this candidate public key is then used in conjunction with the corresponding authentication path to compute a candidate XMSS public key. The candidate layer 0 XMSS public key is used along with the layer 1 XMSS signature to compute a candidate layer 1 XMSS public key, and this process is repeated until a candidate layer $d-1$ public key has been computed. SLH-DSA signature verification succeeds if the computed candidate layer $d-1$ XMSS public key is the same as the SLH-DSA public key root **PK**.root.

3.1 Additional Requirements

This section specifies requirements for cryptographic modules that implement SLH-DSA. Appendix B discusses issues that implementers of cryptographic modules should take into consideration, but that are not requirements. NIST SP 800-89, *Recommendation for Obtaining Assurances for Digital Signature Applications* [3], specifies requirements that apply to the use of digital signature schemes.

Randomness generation. SLH-DSA key generation (Algorithm 17) requires the generation of three random n-byte values, **PK**.seed, **SK**.seed, and **SK**.prf (where n is 16, 24, or 32, depending on the parameter set). For each invocation of key generation each of these values **shall** be freshly generated using an **approved** random bit generator (RBG), as prescribed in NIST SP 800-90A, SP 800-90B, and SP 800-90C [16, 17, 18]. Moreover, the RBG used **shall** have a security strength of at least $8n$ bits.

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

Key validation. NIST SP 800-89 imposes requirements for assurance of public-key validity and private-key possession. In the case of SLH-DSA, where public-key validation is required implementations **shall** verify that the public key is $2n$ bytes in length. When assurance of private key possession is obtained via regeneration, the owner of the private key **shall** check that the private key is $4n$ bytes in length and **shall** use **SK**.seed and **PK**.seed to recompute **PK**.root and compare the newly-generated value with the value in the private key currently held.

4. Functions and Addressing

4.1 Hash Functions and Pseudorandom Functions

The specification of SLH-DSA makes use of six functions — $\mathbf{PRF}_{msg}$, $\mathbf{H}_{msg}$, **PRF**, $\mathbf{T}_\ell$, **H**, and **F** — that are all implemented using hash functions (or XOFs with fixed output lengths). The inputs and output of each function are byte strings. In the following definitions, $\mathbb{B} = \{0, \ldots, 255\}$ denotes the set of all bytes, $\mathbb{B}^n$ denotes the set of byte strings of length n bytes, and $\mathbb{B}^*$ denotes the set of all byte strings. The **ADRS** input is described in Section 4.2.

- $\mathbf{PRF}_{msg}(\mathbf{SK}.\text{prf}, opt\_rand, M)$ $(\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^n)$ is a pseudorandom function (PRF) that generates the randomizer (R) for the randomized hashing of the message to be signed.
- $\mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M)$ $(\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \to \mathbb{B}^m)$ is used to generate the digest of the message to be signed.
- $\mathbf{PRF}(\mathbf{PK}.\text{seed}, \mathbf{SK}.\text{seed}, \mathbf{ADRS})$ $(\mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^{32} \to \mathbb{B}^n)$ is a PRF that is used to generate the secret values in WOTS$^+$ and FORS private keys.
- $\mathbf{T}_\ell(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_\ell)$ $(\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n} \to \mathbb{B}^n)$ is a hash function that maps an ℓn-byte message to an n-byte message.
- $\mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_2)$ $(\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n} \to \mathbb{B}^n)$ is a special case of $\mathbf{T}_\ell$ that takes a $2n$-byte message as input.
- $\mathbf{F}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_1)$ $(\mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n \to \mathbb{B}^n)$ is a hash function that takes an n-byte message as input and produces an n-byte output.

The specific instantiations for these functions differ for different parameter sets and are specified in Section 10.

4.2 Addresses

Four of the functions described in Section 4.1 take a 32-byte address (**ADRS**) as input. An **ADRS** consists of public values that indicate the position of the value being computed by the function. A different **ADRS** value is used for each call to each function. In the case of **PRF**, this is in order to generate a large number of different secret values from a single seed. In the case of $\mathbf{T}_\ell$, **H**, and **F**, it is used to mitigate multi-target attacks.

The structure of an **ADRS** conforms to word boundaries, with each word being 4 bytes long, and with values being encoded as unsigned integers in big-endian byte order. The first word of **ADRS** specifies the layer address, which is the height of an XMSS tree within the hypertree. Trees on the bottom layer have a height of zero, and the single XMSS tree at the top has a height of $d-1$ (see Figure 1). The next three words of **ADRS** specify the tree address, which is the position of an XMSS tree within a layer of the hypertree. The leftmost XMSS tree in a layer has a tree address of zero, and the rightmost XMSS tree in layer L has a tree address of $2^{(d-1-L)h'} - 1$. The next word is used to specify the type of the address, which differs depending on the use case. There are seven different types of address used in SLH-DSA, as described below.[4] The type of the

[4]The *type* word will have a value of 0, 1, 2, 3, 4, 5, or 6. In order to improve readability, these values will be referred to in this standard by the constants `WOTS_HASH`, `WOTS_PK`, `TREE`, `FORS_TREE`, `FORS_ROOTS`, `WOTS_PRF`,

address determines how the final 12 bytes of the address are to be interpreted. The algorithms in this standard are written based on the assumption that whenever the type in an **ADRS** is changed, the final 12 bytes of address are initialized to zero.

The type is set to WOTS_HASH (0) for a WOTS$^+$ hash address (see Figure 2), which is used when computing hash chains in WOTS$^+$. When type is WOTS_HASH, the next word encodes the key pair address, which is the index of the WOTS$^+$ key pair within the XMSS tree specified by the layer and tree addresses, with the leftmost WOTS$^+$ key having an index of zero and the rightmost WOTS$^+$ key having an index of $2^{h'} - 1$. Next is the chain address, which encodes the index of the chain within WOTS$^+$, followed by the hash address, which encodes the address of the hash function within the chain.

layer address	4 bytes
tree address	12 bytes
$type = 0$ (WOTS_HASH)	4 bytes
key pair address	4 bytes
chain address	4 bytes
hash address	4 bytes

Figure 2. WOTS$^+$ hash address

layer address	4 bytes
tree address	12 bytes
$type = 1$ (WOTS_PK)	4 bytes
key pair address	4 bytes
$padding = 0$	8 bytes

Figure 3. WOTS$^+$ public key compression address

The type is set to WOTS_PK (1) when compressing WOTS$^+$ public keys (see Figure 3). As when the type is WOTS_HASH, the next word encodes the index of the WOTS$^+$ key pair within the XMSS tree specified by the layer and tree addresses. The remaining two words of **ADRS** are not needed and are set to zero.

The type is set to TREE (2) when computing the hashes within the XMSS tree (see Figure 4). For this type of address, the next word is always set to zero. The following word encodes the height of the node within the tree that is being computed, and the final word encodes the index of the node at that height.

layer address	4 bytes
tree address	12 bytes
$type = 2$ (TREE)	4 bytes
$padding = 0$	4 bytes
tree height	4 bytes
tree index	4 bytes

Figure 4. Hash tree address

The type is set to FORS_TREE (3) when computing hashes within the FORS tree (see Figure 5). The next word is the key pair address, which encodes the FORS key that is used and is the same as

and FORS_PRF, respectively.

the key pair address in WOTS$^+$ addresses (see Figure 2 and Figure 3). The next two words — the tree height and tree index — encode the node within the FORS tree that is being computed. The tree height starts with zero for the leaf nodes. The tree index is counted continuously across the k different FORS trees. The leftmost node in the leftmost tree has an index of zero and rightmost node in the rightmost tree at level j has an index of $k \cdot 2^{(a-j)} - 1$, where a is the height of the tree.

layer address = 0	4 bytes
tree address	12 bytes
$type = 3$ (FORS_TREE)	4 bytes
key pair address	4 bytes
tree height	4 bytes
tree index	4 bytes

Figure 5. FORS tree address

layer address = 0	4 bytes
tree address	12 bytes
$type = 4$ (FORS_ROOTS)	4 bytes
key pair address	4 bytes
$padding = 0$	8 bytes

Figure 6. FORS tree roots compression address

The type is set to FORS_ROOTS (4) when compressing the k FORS tree roots (see Figure 6). The next word is the key pair address, which has the same meaning as it does in the FORS_TREE address. The remaining two words of **ADRS** are not needed and are set to zero.

The type is set to WOTS_PRF (5) when generating secret values for WOTS$^+$ keys (see Figure 7). The values for the other words in the address are set to the same values as for the WOTS_HASH address (Figure 2) used for the chain. The hash address is always set to zero.

layer address	4 bytes
tree address	12 bytes
$type = 5$ (WOTS_PRF)	4 bytes
key pair address	4 bytes
chain address	4 bytes
hash address = 0	4 bytes

Figure 7. WOTS$^+$ key generation address

layer address = 0	4 bytes
tree address	12 bytes
$type = 6$ (FORS_PRF)	4 bytes
key pair address	4 bytes
tree height = 0	4 bytes
tree index	4 bytes

Figure 8. FORS key generation address

The type is set to FORS_PRF (6) when generating secret values for FORS keys (see Figure 8). The values for the other words in the address are set to the same values as for the FORS_TREE address (Figure 5) used for the same leaf node.

The instantiations of the functions in Section 4.1 that are based on SHA-2 (Section 10.2 and Section 10.3) make use of a compressed version of **ADRS**. A compressed address (**ADRS**^{c}) is a 22-byte string that is the same as an **ADRS** with the exceptions that the encodings of the layer address and type are reduced to one byte each and the encoding of the tree address is reduced to eight bytes (i.e., $\mathbf{ADRS}^c = \mathbf{ADRS}[3] \,\|\, \mathbf{ADRS}[8:16] \,\|\, \mathbf{ADRS}[19] \,\|\, \mathbf{ADRS}[20:32]$).

4.3 Member Functions

The algorithms in this standard make use of member functions. If a complex data structure, such as an **ADRS**, contains a component X, then **ADRS**.getX() returns the value of X, and **ADRS**.setX(Y) sets the component X in **ADRS** to the value held by Y. If a data structure s contains multiple instances of X, then s.getX(i) returns the value of the i^{th} instance of X in s. For example, if s is a FORS signature (Figure 13), then s.getAUTH(i) returns the authentication path for the i^{th} tree.

As noted in Section 4.2, whenever the *type* in an address changes, the final 12 bytes of the address are initialized to zero. The member function **ADRS**.setTypeAndClear(Y) for addresses sets the $type$ of the **ADRS** to Y and sets the final 12 bytes of the **ADRS** to zero.

4.4 Arrays, Byte Strings, and Integers

If X is an array of length n, then $X[i]$ (for $i \in \{0,\ldots,n-1\}$) will refer to the i^{th} element in the string X. If X is an array of m n-byte strings, then $X[i]$ (for $i \in \{0,\ldots,m-1\}$) will refer to the i^{th} n-byte string in X, and X will refer to the $m \cdot n$-byte string $X[0] \parallel X[1] \parallel \ldots X[m-1]$.

A byte string may be interpreted as the big-endian representation of an integer. In such cases, a byte string X of length n is converted to the integer

$$X[0] \cdot 256^{n-1} + X[1] \cdot 256^{n-2} + \ldots X[n-2] \cdot 256 + X[n-1].$$

Similarly, an integer x may be converted to a byte string of length n by finding coefficients $x_0, x_1, \ldots x_{n-1}, x_{n-2} \in \{0,\ldots,255\}$ such that

$$x = x_0 \cdot 256^{n-1} + x_1 \cdot 256^{n-2} + \ldots x_{n-2} \cdot 256 + x_{n-1}$$

and then setting the byte string to be $x_0 x_1 \ldots x_{n-2} x_{n-1}$.

Algorithm 1 is a function that converts a byte string X of length n to an integer, and Algorithm 2 is a function that converts an integer x to a byte string of length n.

Algorithm 1 toInt(X, n)

Convert a byte string to an integer.

Input: n-byte string X.
Output: Integer value of X.

$total \leftarrow 0$

for i **from** 0 **to** $n-1$ **do**
    $total \leftarrow 256 \cdot total + X[i]$
end for
return $total$

Algorithm 2 toByte(x, n)

Convert an integer to a byte string.

Input: Integer x, string length n.
Output: Byte string of length n containing binary representation of x in big-endian byte-order.

$total \leftarrow x$

for i **from** 0 **to** $n-1$ **do**
  $S[n-1-i] \leftarrow total \bmod 256$ ▷ Least significant 8 bits of $total$
  $total \leftarrow total \gg 8$
end for
return S

For the WOTS$^+$ and FORS schemes, the messages to be signed need to be split into a sequence of b-bit strings, where each b-bit string is interpreted as an integer between 0 and $2^b - 1$.[5] (This is the equivalent of creating the base-2^b representation of the message.) The base_2^b function (Algorithm 3) takes as input a byte string X, a bit string length b, and an output length out_len and returns an array of base-2^b integers that represent the first $out\_len \cdot b$ bits of X (if the individual bytes in X are encoded as 8-bit strings in big-endian bit order). X must be at least $\lceil out\_len \cdot b/8 \rceil$ bytes in length.

Algorithm 3 base_2^b(X, b, out_len)

Compute the base 2^b representation of X.

Input: Byte string X of length at least $\left\lceil \frac{out\_len \cdot b}{8} \right\rceil$, integer b, output length out_len.
Output: Array of out_len integers in the range $[0, \ldots, 2^b - 1]$.

$in \leftarrow 0$
$bits \leftarrow 0$
$total \leftarrow 0$

for out **from** 0 **to** $out\_len - 1$ **do**
  while $bits < b$ **do**
    $total \leftarrow (total \ll 8) + X[in]$
    $in \leftarrow in + 1$
    $bits \leftarrow bits + 8$
  end while
  $bits \leftarrow bits - b$
  $baseb[out] \leftarrow (total \gg bits) \bmod 2^b$
end for
return $baseb$

[5] b will be the value of lg_w when the base_2^b function is used in WOTS$^+$, and b will be the value of a when the base_2^b function is used in FORS. For the parameter sets in this standard, lg_w is 4, and a is 6, 8, 9, 12, or 14.

5. One-Time Signatures

This section describes the WOTS$^+$ one-time signature scheme that is a component of SLH-DSA.

WOTS$^+$ uses two parameters. The security parameter n is the length in bytes of the messages that may be signed, as well as the length of the private key elements, public key elements, and signature elements. For the parameter sets specified in this standard, n may be 16, 24, or 32 (see Table 1). The second parameter, lg_w, indicates the number of bits that are encoded by each hash chain that is used.[6] lg_w is 4 for all parameter sets in this standard. These parameters are used to compute four additional values:

$$w = 2^{lg_w} \tag{5.1}$$

$$len_1 = \left\lceil \frac{8n}{lg_w} \right\rceil \tag{5.2}$$

$$len_2 = \left\lfloor \frac{\log_2(len_1 \cdot (w-1))}{lg_w} \right\rfloor + 1 \tag{5.3}$$

$$len = len_1 + len_2 \tag{5.4}$$

When $lg_w = 4$, $w = 16$, $len_1 = 2n$, $len_2 = 3$, and $len = 2n+3$.

A WOTS$^+$ private key consists of len secret values of length n. In SLH-DSA, these are all generated from an n-byte seed **SK**.seed using a PRF. Chains of length w are then created from the secret values using a chaining function, and the end values from each of the chains are public values. The WOTS$^+$ public key is computed as the hash of these public values. In order to create a signature, the $8n$-bit message is first converted into an array of len_1 base-w integers. A checksum is then computed for this string, and the checksum is converted into an array of len_2 base-w integers. The signature consists of the appropriate entries from the chains for each of the integers in the message and checksum arrays.

The WOTS$^+$ functions make use of two helper functions: base_2^b and chain. The base_2^b function (Section 4.4) is used to break the message to be signed and the checksum value into arrays of base-w integers. The chain function (Algorithm 4) is used to compute the hash chains.

The chain function takes as input an n-byte string X and integers s and i and returns the result of iterating a hash function **F** on the input s times, starting from an index of i. The chain function also requires as input **PK**.seed, which is part of the SLH-DSA public key, and an address **ADRS**. The *type* in **ADRS** must be set to `WOTS_HASH`, and the layer address, tree address, key pair address, and chain address must be set to the address of the chain being computed. The chain function updates the hash address in **ADRS** with each iteration to specify the current position in the chain prior to **ADRS**'s use in **F**.

[6] In [4], the Winternitz parameter w is used at the second WOTS$^+$ parameter, where w indicates the length of the hash chains that are used. This standard uses the parameter $lg_w = \log_2(w)$ instead, in order to simplify computations.

Algorithm 4 chain(X, i, s, **PK**.seed, **ADRS**)

Chaining function used in WOTS$^+$.

Input: Input string X, start index i, number of steps s, public seed **PK**.seed, address **ADRS**.
Output: Value of **F** iterated s times on X.

```
if (i + s) ≥ w then
    return NULL
end if

tmp ← X

for j from i to i + s − 1 do
    ADRS.setHashAddress(j)
    tmp ← F(PK.seed, ADRS, tmp)
end for
return tmp
```

5.1 WOTS$^+$ Public-Key Generation

The wots_PKgen function (Algorithm 5) generates WOTS$^+$ public keys. It takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key and an address. The *type* in the address **ADRS** must be set to `WOTS_HASH`, and the layer address, tree address, and key pair address must encode the address of the WOTS$^+$ public key to be generated.

Lines 4 through 9 in Algorithm 5 generate the public values, as described in Section 5. For each of the *len* public values, the corresponding secret value is generated in lines 5 and 6, and the chain function is called to compute the end value of the chain of length w. Once the *len* public values are computed, they are compressed into a single n-byte value in lines 10 through 13.

Algorithm 5 wots_PKgen(**SK**.seed, **PK**.seed, **ADRS**)

Generate a WOTS$^+$ public key.

Input: Secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**.
Output: WOTS$^+$ public key pk.

```
skADRS ← ADRS                                ▷ Copy address to create key generation key address
skADRS.setTypeAndClear(WOTS_PRF)
skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
for i from 0 to len − 1 do
    skADRS.setChainAddress(i)
    sk ← PRF(PK.seed, SK.seed, skADRS)            ▷ Compute secret value for chain i
    ADRS.setChainAddress(i)
    tmp[i] ← chain(sk, 0, w − 1, PK.seed, ADRS)   ▷ Compute public value for chain i
end for
wotspkADRS ← ADRS                            ▷ Copy address to create WOTS+public key address
wotspkADRS.setTypeAndClear(WOTS_PK)
wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
pk ← T_len(PK.seed, wotspkADRS, tmp)                        ▷ Compress public key
return pk
```

5.2 WOTS$^+$ Signature Generation

A WOTS$^+$ signature is an array of len byte strings of length n, as shown in Figure 9. The wots_sign function (Algorithm 6) generates the signature by converting the n-byte message M[7] into an array of len_1 base-w integers (line 3). A checksum is computed over M (lines 5 through 7). The checksum is converted to a byte string, which is then converted into an array of len_2 base-w integers (lines 9 and 10). The len_2 integers that represent the checksum are appended to the len_1 integers that represent the message (line 10).[8] For each of the len base-w integers, the signature consists of the corresponding node in one of the hash chains. For each of these integers, lines 16 and 17 compute the secret value for the hash chain, and lines 18 and 19 compute the node in the hash chain that corresponds to the integer. The selected nodes are concatenated to form the WOTS$^+$ signature.

$\text{sig}_{\text{ots}}[0]$	n bytes
$\cdots$	
$\text{sig}_{\text{ots}}[len-1]$	n bytes

Figure 9. WOTS$^+$ signature data format

In addition to the n-byte message to be signed, wots_sign takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key and an address. The *type* in the address **ADRS** must be set to

[7] In SLH-DSA, the message M that is signed using WOTS$^+$ is either an XMSS public key or a FORS public key.

[8] In the case that $lg_w = 4$, the n-byte message is converted into an array of $2n$ base-16 integers (i.e., hexadecimal digits). The checksum is encoded as 2 bytes with the least significant 4 bits being zeros, and the most significant 12 bits are appended to the message as an array of three base-16 integers.

WOTS_HASH, and the layer address, tree address, and key pair address must encode the address of the WOTS$^+$ key that is used to sign the message.

Algorithm 6 wots_sign(M, **SK**.seed, **PK**.seed, **ADRS**)

Generate a WOTS$^+$ signature on an n-byte message.

Input: Message M, secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**.
Output: WOTS$^+$ signature sig.

$csum \leftarrow 0$

$msg \leftarrow \text{base\_2}^{\text{b}}(M, lg_w, len_1)$ ▷ Convert message to base w

for i **from** 0 **to** $len_1 - 1$ **do** ▷ Compute checksum
    $csum \leftarrow csum + w - 1 - msg[i]$
end for

$csum \leftarrow csum \ll ((8 - ((len_2 \cdot lg_w) \bmod 8)) \bmod 8)$ ▷ For $lg_w = 4$ left shift by 4
$msg \leftarrow msg \parallel \text{base\_2}^{\text{b}}\left(\text{toByte}\left(csum, \left\lceil \frac{len_2 \cdot lg_w}{8} \right\rceil\right), lg_w, len_2\right)$ ▷ Convert $csum$ to base w

skADRS ← **ADRS**
skADRS.setTypeAndClear(WOTS_PRF)
skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
for i **from** 0 **to** $len - 1$ **do**
    skADRS.setChainAddress(i)
    $sk \leftarrow$ **PRF**(**PK**.seed, **SK**.seed, skADRS) ▷ Compute secret value for chain i
    ADRS.setChainAddress(i)
    $sig[i] \leftarrow$ chain(sk, 0, $msg[i]$, **PK**.seed, **ADRS**) ▷ Compute signature value for chain i
end for
return sig

5.3 Computing a WOTS$^+$ Public Key From a Signature

As noted in Section 3, verifying a WOTS$^+$ signature involves computing a public-key value from a message and signature value. Verification succeeds if the correct public-key value is computed, which is determined by using the computed public-key value along with other information to compute a candidate **PK**.root value and then comparing that value to the known value of **PK**.root from the SLH-DSA public key. This section describes wots_PKFromSig (Algorithm 7), a function that computes a candidate WOTS$^+$ public key from a WOTS$^+$ signature and corresponding message.

In addition to an n-byte message M and a $len \cdot n$-byte signature sig, which is interpreted as an array of len n-byte strings, the wots_PKFromSig function takes as input **PK**.seed from the SLH-DSA public key and an address. The *type* of the address **ADRS** must be set to WOTS_HASH, and the layer address, tree address, and key pair address must encode the address of the WOTS$^+$ key that was used to sign the message.

Lines 1 through 10 of wots_PKFromSig are the same as lines 1 through 10 of wots_sign (Algorithm 6). Lines 11 through 14 of wots_PKFromSig compute the end nodes for each of the chains using the signature value as the starting point and the message value to determine the number of iterations that need to be performed to get to the end node. Finally, as with lines 10 through 13 of Algorithm 5, the computed public-key values are compressed in lines 15 through 18.

Algorithm 7 wots_PKFromSig(*sig*, *M*, **PK**.seed, **ADRS**)

Compute a WOTS$^+$ public key from a message and its signature.

Input: WOTS$^+$ signature *sig*, message *M*, public seed **PK**.seed, address **ADRS**.
Output: WOTS$^+$ public key pk_{sig} derived from *sig*.

1: $csum \leftarrow 0$
2:
3: $msg \leftarrow \text{base\_2}^\text{b}(M, lg_w, len_1)$ ▷ Convert message to base w
4:
5: **for** i **from** 0 **to** $len_1 - 1$ **do** ▷ Compute checksum
6: $\quad csum \leftarrow csum + w - 1 - msg[i]$
7: **end for**
8:
9: $csum \leftarrow csum \ll ((8 - ((len_2 \cdot lg_w) \bmod 8)) \bmod 8)$ ▷ For $lg_w = 4$ left shift by 4
10: $msg \leftarrow msg \,\|\, \text{base\_2}^\text{b}\left(\text{toByte}\left(csum, \left\lceil \frac{len_2 \cdot lg_w}{8} \right\rceil\right), lg_w, len_2\right)$ ▷ Convert *csum* to base w
11: **for** i **from** 0 **to** $len - 1$ **do**
12: $\quad$ **ADRS**.setChainAddress(i)
13: $\quad tmp[i] \leftarrow \text{chain}(sig[i], msg[i], w - 1 - msg[i], \mathbf{PK}.\text{seed}, \mathbf{ADRS})$
14: **end for**
15: wotspkADRS ← **ADRS**
16: wotspkADRS.setTypeAndClear(`WOTS_PK`)
17: wotspkADRS.setKeyPairAddress(**ADRS**.getKeyPairAddress())
18: $pk_{sig} \leftarrow \mathbf{T}_{len}(\mathbf{PK}.\text{seed}, \text{wotspkADRS}, tmp)$
19: **return** pk_{sig}

6. The eXtended Merkle Signature Scheme (XMSS)

XMSS extends the WOTS$^+$ signature scheme into one that can sign multiple messages. A Merkle tree [15] of height h' is used to allow $2^{h'}$ WOTS$^+$ public keys to be authenticated using a single n-byte XMSS public key, which is the root of the Merkle tree.[9] As each WOTS$^+$ key may be used to sign one message, the XMSS key may be used to sign $2^{h'}$ messages.

An XMSS signature is $(h' + len) \cdot n$ bytes in length and consists of a WOTS$^+$ signature and an authentication path (see Figure 10). The authentication path is an array of nodes from the Merkle tree — one from each level of the tree (except the root) — that allows the verifier to compute the root of the tree when used in conjunction with the WOTS$^+$ public key that can be computed from the WOTS$^+$ signature.

$\mathbf{SIG}_{\mathrm{WOTS}^+}$	$len \cdot n$ bytes
AUTH[0]	n bytes
$\cdots$	
AUTH$[h'-1]$	n bytes

Figure 10. XMSS signature data format

6.1 Generating a Merkle Hash Tree

The xmss_node function (Algorithm 8) computes the nodes of an XMSS tree. The xmss_node function takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key; a target node index i, which is the index of the node being computed; a target node height z, which is the height within the Merkle tree of the node being computed; and an address. The address **ADRS** must have the layer address and tree address set to the XMSS tree within which the node is being computed.

Each node in an XMSS tree is the root of a subtree, and Algorithm 8 computes the root of the subtree recursively. If the subtree consists of a single leaf node, then the function simply returns the value of the node's WOTS$^+$ public key (lines 5 through 7). Otherwise, the function computes the roots of the left subtree (line 9) and right subtree (line 10) and hashes them together (lines 11 through 14).

[9] The Merkle tree formed from the $2^{h'}$ WOTS$^+$ keys of an XMSS key is referred to in this standard as an XMSS tree.

Algorithm 8 xmss_node(**SK**.seed, i, z, **PK**.seed, **ADRS**)

Compute the root of a Merkle subtree of WOTS$^+$ public keys.

Input: Secret seed **SK**.seed, target node index i, target node height z, public seed **PK**.seed, address **ADRS**.
Output: n-byte root *node*.

if $z > h'$ **or** $i \geq 2^{(h'-z)}$ **then**
    return NULL
end if
if $z = 0$ **then**
    ADRS.setTypeAndClear(`WOTS_HASH`)
    ADRS.setKeyPairAddress(i)
    $node \leftarrow$ wots_PKgen(**SK**.seed, **PK**.seed, **ADRS**)
else
    $lnode \leftarrow$ xmss_node($\mathbf{SK}$.seed$, 2i, z-1,$ $\mathbf{PK}$.seed$,$ $\mathbf{ADRS}$)
    $rnode \leftarrow$ xmss_node($\mathbf{SK}$.seed$, 2i+1, z-1,$ $\mathbf{PK}$.seed$,$ $\mathbf{ADRS}$)
    ADRS.setTypeAndClear(`TREE`)
    ADRS.setTreeHeight(z)
    ADRS.setTreeIndex(i)
    $node \leftarrow \mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, lnode \,\|\, rnode)$
end if
return *node*

6.2 Generating an XMSS Signature

The xmss_sign function (Algorithm 9) creates an XMSS signature on an n-byte message M[10] by first creating an authentication path (lines 1 through 4) and then signing M with the appropriate WOTS$^+$ key (lines 6 through 8). In addition to M, xmss_sign takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key, an address, and an index. The address **ADRS** must have the layer address and tree address set to the XMSS key that is being used to sign the message, and the index *idx* must be the index of the WOTS$^+$ key within the XMSS tree that will be used to sign the message.

The authentication path consists of the sibling nodes of each node that is on the path from the WOTS$^+$ key used to the root. For example, in Figure 11, if the message is signed with K_2, then K_2, $n_{1,1}$, and $n_{2,0}$ are the on path nodes, and the authentication path consists of K_3, $n_{1,0}$, and $n_{2,1}$. In line 2 of Algorithm 9, $\lfloor idx/2^j \rfloor$ is the on path node, and $\lfloor idx/2^j \rfloor \oplus 1$ is the authentication path node. Line 3 computes the value of the authentication path node.

[10] In SLH-DSA, the message M that is signed using XMSS is either an XMSS public key or a FORS public key.

Algorithm 9 xmss_sign(M, **SK**.seed, idx, **PK**.seed, **ADRS**)

Generate an XMSS signature.

Input: n-byte message M, secret seed **SK**.seed, index idx, public seed **PK**.seed, address **ADRS**.
Output: XMSS signature $\text{SIG}_{XMSS} = (sig \parallel \text{AUTH})$.

for j **from** 0 **to** $h' - 1$ **do** ▷ Build authentication path
  $k \leftarrow \lfloor idx/2^j \rfloor \oplus 1$
  $\text{AUTH}[j] \leftarrow$ xmss_node(**SK**.seed, k, j, **PK**.seed, **ADRS**)
end for

ADRS.setTypeAndClear(WOTS_HASH)
ADRS.setKeyPairAddress(idx)
$sig \leftarrow$ wots_sign(M, **SK**.seed, **PK**.seed, **ADRS**)
$\text{SIG}_{XMSS} \leftarrow sig \parallel \text{AUTH}$
return SIG_{XMSS}

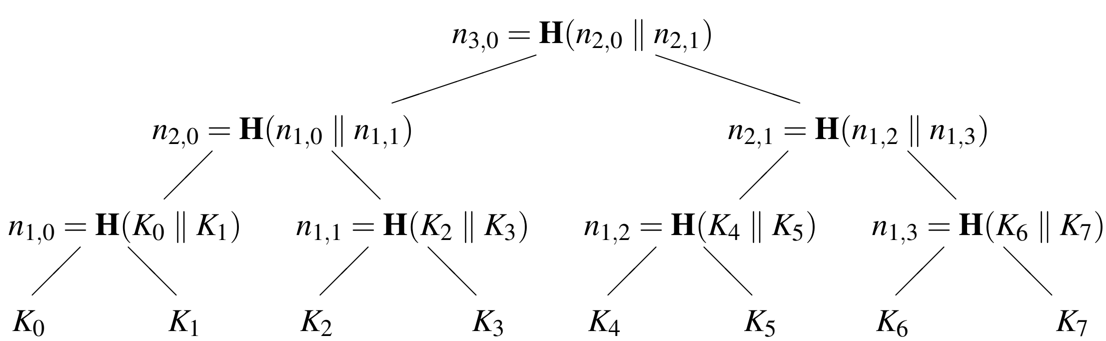

Figure 11. Merkle Hash Tree

6.3 Computing an XMSS Public Key From a Signature

As noted in Section 3, verifying an XMSS signature involves computing a public-key value from a message and a signature value. Verification succeeds if the correct public-key value is computed, which is determined by using the computed public-key value along with other information to compute a candidate **PK**.root value and then comparing that value to the known value of **PK**.root from the SLH-DSA public key. This section describes xmss_PKFromSig (Algorithm 10), a function that computes a candidate XMSS public key from an XMSS signature and corresponding message.

In addition to an n-byte message M and an $(len+h')\cdot n$-byte signature SIG_{XMSS}, xmss_PKFromSig takes as input **PK**.seed from the SLH-DSA public key, an address, and an index. The address **ADRS** must be set to the layer address and tree address of the XMSS key that was used to sign the message, and the index idx must be the index of the WOTS^+ key within the XMSS tree that was used to sign the message.

Algorithm 10 begins by computing the WOTS^+ public key in lines 1 through 5. The root is then computed in lines 7 through 19. Starting with the leaf node (the WOTS^+ public key), a node at each level of the tree is computed by hashing together the node computed in the previous iteration with the corresponding authentication path node. In lines 13 and 16, AUTH is interpreted as an array of h' n-byte strings.

Algorithm 10 xmss_PKFromSig(idx, SIG_{XMSS}, M, **PK**.seed, **ADRS**)

Compute an XMSS public key from an XMSS signature.

Input: Index idx, XMSS signature $\mathrm{SIG}_{XMSS} = (sig \parallel \mathrm{AUTH})$, n-byte message M, public seed **PK**.seed, address **ADRS**.
Output: n-byte root value $node[0]$.

ADRS.setTypeAndClear(`WOTS_HASH`) ▷ Compute WOTS$^+$ pk from WOTS$^+$ sig
ADRS.setKeyPairAddress(idx)
$sig \leftarrow \mathrm{SIG}_{XMSS}$.getWOTSSig() ▷ $\mathrm{SIG}_{XMSS}[0 : len \cdot n]$
$\mathrm{AUTH} \leftarrow \mathrm{SIG}_{XMSS}$.getXMSSAUTH() ▷ $\mathrm{SIG}_{XMSS}[len \cdot n : (len + h') \cdot n]$
$node[0] \leftarrow$ wots_PKFromSig($sig, M,$ **PK**.seed, **ADRS**)

ADRS.setTypeAndClear(`TREE`) ▷ Compute root from WOTS$^+$ pk and AUTH
ADRS.setTreeIndex(idx)
for k **from** 0 **to** $h' - 1$ **do**
    ADRS.setTreeHeight($k + 1$)
    if $\lfloor idx/2^k \rfloor$ is even **then**
        ADRS.setTreeIndex(**ADRS**.getTreeIndex()$/2$)
        $node[1] \leftarrow$ **H**(**PK**.seed, **ADRS**, $node[0] \parallel \mathrm{AUTH}[k]$)
    else
        ADRS.setTreeIndex((**ADRS**.getTreeIndex()$- 1)/2$)
        $node[1] \leftarrow$ **H**(**PK**.seed, **ADRS**, $\mathrm{AUTH}[k] \parallel node[0]$)
    end if
    $node[0] \leftarrow node[1]$
end for
return $node[0]$

7. The SLH-DSA Hypertree

As noted in Section 3, SLH-DSA requires a very large number of WOTS$^+$ keys to sign FORS public keys. As it would not be feasible for the parameter sets in this standard to have a single XMSS key with so many WOTS$^+$ keys, SLH-DSA uses a hypertree to sign the FORS keys. As depicted in Figure 1, a hypertree is a tree of XMSS trees. The XMSS keys at the lowest layer are used to sign FORS public keys (Section 8), and the XMSS keys at every other layer are used to sign the XMSS public keys at the layer below.

The hypertree has d layers of XMSS trees with each XMSS tree being a Merkle tree of height h', so the total height of the hypertree is $h = d \cdot h'$ (see Table 1). The top layer (layer $d-1$) is a single XMSS tree, and the public key of this XMSS key pair (i.e., the root of the Merkle tree) is the public key of the hypertree (**PK**.root). The next layer down has $2^{h'}$ XMSS trees, and the public key of each of these XMSS keys is signed by one of the $2^{h'}$ WOTS$^+$ keys that is part of the top layer's XMSS key. The lowest layer has $2^{h-h'}$ XMSS trees, providing 2^h WOTS$^+$ keys to sign FORS keys.

7.1 Hypertree Signature Generation

A hypertree signature is $(h + d \cdot len) \cdot n$ bytes in length and consists of a sequence of d XMSS signatures, starting with one generated using an XMSS key at the lowest layer and ending with one generated using the XMSS key at the top layer (see Figure 12).

XMSS signature $\mathbf{SIG}_{\mathrm{XMSS}}$ (layer 0)	$(h' + len) \cdot n$ bytes
XMSS signature $\mathbf{SIG}_{\mathrm{XMSS}}$ (layer 1)	$(h' + len) \cdot n$ bytes
...	
XMSS signature $\mathbf{SIG}_{\mathrm{XMSS}}$ (layer $d-1$)	$(h' + len) \cdot n$ bytes

Figure 12. HT signature data format

In addition to the n-byte message M,[11] the ht_sign function (Algorithm 11) takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key, the index of the XMSS tree at the lowest layer that will sign the message idx_{tree}, and the index of the WOTS$^+$ key within the XMSS tree that will sign the message idx_{leaf}.

Algorithm 11 begins in lines 1 through 4 by signing M with the specified XMSS key using the WOTS$^+$ key within that XMSS key specified by idx_{leaf}. The XMSS public key is obtained (line 6 or 15) for each successive layer and signed by the appropriate key at the next higher level (lines 8 through 12).

[11] In SLH-DSA, the message M that is provided to ht_sign is a FORS public key.

Algorithm 11 ht_sign(M, **SK**.seed, **PK**.seed, idx_{tree}, idx_{leaf})

Generate a hypertree signature.

Input: Message M, private seed **SK**.seed, public seed **PK**.seed, tree index idx_{tree}, leaf index idx_{leaf}.
Output: HT signature SIG_{HT}.

$\textbf{ADRS} \leftarrow \text{toByte}(0, 32)$

$\textbf{ADRS}.\text{setTreeAddress}(idx_{tree})$
$\text{SIG}_{tmp} \leftarrow \text{xmss\_sign}(M, \textbf{SK}.\text{seed}, idx_{leaf}, \textbf{PK}.\text{seed}, \textbf{ADRS})$
$\text{SIG}_{HT} \leftarrow \text{SIG}_{tmp}$
$root \leftarrow \text{xmss\_PKFromSig}(idx_{leaf}, \text{SIG}_{tmp}, M, \textbf{PK}.\text{seed}, \textbf{ADRS})$
for j **from** 1 **to** $d-1$ **do**
    $idx_{leaf} \leftarrow idx_{tree} \bmod 2^{h'}$ ▷ h' least significant bits of idx_{tree}
    $idx_{tree} \leftarrow idx_{tree} \gg h'$ ▷ Remove least significant h' bits from idx_{tree}
    $\textbf{ADRS}.\text{setLayerAddress}(j)$
    $\textbf{ADRS}.\text{setTreeAddress}(idx_{tree})$
    $\text{SIG}_{tmp} \leftarrow \text{xmss\_sign}(root, \textbf{SK}.\text{seed}, idx_{leaf}, \textbf{PK}.\text{seed}, \textbf{ADRS})$
    $\text{SIG}_{HT} \leftarrow \text{SIG}_{HT} \parallel \text{SIG}_{tmp}$
    if $j < d-1$ **then**
        $root \leftarrow \text{xmss\_PKFromSig}(idx_{leaf}, \text{SIG}_{tmp}, root, \textbf{PK}.\text{seed}, \textbf{ADRS})$
    end if
end for
return SIG_{HT}

7.2 Hypertree Signature Verification

Hypertree signature verification works by making d calls to xmss_PKFromSig (Algorithm 10) and comparing the result to the public key of the hypertree.

In addition to the n-byte message M and the $(h + d \cdot len) \cdot n$-byte signature SIG_{HT}, ht_verify (Algorithm 12) takes as input **PK**.seed and **PK**.root from the SLH-DSA public key, the index of the XMSS tree at the lowest layer that signed the message idx_{tree}, and the index of the WOTS$^+$ key within the XMSS tree that signed the message idx_{leaf}.

At each layer, either the message M or the computed public key of the XMSS key at the lower layer is provided along with the appropriate XMSS signature to xmss_PKFromSig in order to obtain the layer's computed XMSS public key. If the computed XMSS public key of the top layer tree is the same as the known hypertree public key, **PK**.root, then verification succeeds.

Algorithm 12 ht_verify(M, SIG_{HT}, **PK**.seed, idx_{tree}, idx_{leaf}, **PK**.root)

Verify a hypertree signature.

Input: Message M, signature SIG_{HT}, public seed **PK**.seed, tree index idx_{tree}, leaf index idx_{leaf}, HT public key **PK**.root.
Output: Boolean.

ADRS $\leftarrow$ toByte$(0, 32)$

ADRS.setTreeAddress(idx_{tree})
$\text{SIG}_{tmp} \leftarrow \text{SIG}_{HT}$.getXMSSSignature(0) ▷ $\text{SIG}_{HT}[0 : (h' + len) \cdot n]$
$node \leftarrow$ xmss_PKFromSig(idx_{leaf}, SIG_{tmp}, M, **PK**.seed, **ADRS**)
for j **from** 1 **to** $d - 1$ **do**
    $idx_{leaf} \leftarrow idx_{tree} \bmod 2^{h'}$ ▷ h' least significant bits of idx_{tree}
    $idx_{tree} \leftarrow idx_{tree} \gg h'$ ▷ Remove least significant h' bits from idx_{tree}
    ADRS.setLayerAddress(j)
    ADRS.setTreeAddress(idx_{tree})
    $\text{SIG}_{tmp} \leftarrow \text{SIG}_{HT}$.getXMSSSignature($j$) ▷ $\text{SIG}_{HT}[j \cdot (h' + len) \cdot n : (j+1)(h' + len) \cdot n]$
    $node \leftarrow$ xmss_PKFromSig(idx_{leaf}, SIG_{tmp}, $node$, **PK**.seed, **ADRS**)
end for
if $node =$ **PK**.root **then**
    return true
else
    return false
end if

---

8. Forest of Random Subsets (FORS)

FORS is a few-time signature scheme that is used to sign the digests of the actual messages. Unlike WOTS$^+$, for which forgeries become feasible if a key is used twice [19], the security of a FORS key degrades gradually as the number of signatures increases.

FORS uses two parameters: k and $t = 2^a$ (see Table 1). A FORS private key consists of k sets of t n-byte strings, all of which are pseudorandomly generated from the seed **SK**.seed. Each of the k sets is formed into a Merkle tree, and the roots of the trees are hashed together to form the FORS public key. A signature on a ka-bit message digest consists of k elements from the private key, one from each set selected using a bits of the message digest, along with the authentication paths for each of these elements (see Figure 13).

private key value (tree 0)	n bytes
AUTH (tree 0)	$a \cdot n$ bytes
...	
private key value (tree $k-1$)	n bytes
AUTH (tree $k-1$)	$a \cdot n$ bytes

Figure 13. FORS signature data format

8.1 Generating FORS Secret Values

The fors_SKgen function (Algorithm 13) generates the n-byte strings of the FORS private key. The function takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key, an address, and an index. The *type* in the address **ADRS** must be set to `FORS_TREE`, and the tree address and key pair address must be set to the index of the WOTS$^+$ key within the XMSS tree that signs the FORS key. The layer address must be set to zero. The index *idx* is the index of the FORS secret value within the sets of FORS trees.

Algorithm 13 fors_SKgen(**SK**.seed, **PK**.seed, **ADRS**, *idx*)

Generate a FORS private-key value.

Input: Secret seed **SK**.seed, public seed **PK**.seed, address **ADRS**, secret key index *idx*.
Output: n-byte FORS private-key value.

```
skADRS ← ADRS                                  ▷ Copy address to create key generation address
skADRS.setTypeAndClear(FORS_PRF)
skADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
skADRS.setTreeIndex(idx)
return PRF(PK.seed, SK.seed, skADRS)
```

8.2 Generating a Merkle Hash Tree

The fors_node function (Algorithm 14) computes the nodes of a Merkle tree. It is the same as xmss_node, except that the leaf nodes are the hashes of the FORS secret values instead of WOTS$^+$

public keys.

The fors_node function takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key; a target node index i, which is the index of node being computed; a target node height z, which is the height within the Merkle tree of the node being computed; and an address. The address **ADRS** must have the layer address set to zero (since the XMSS tree that signs a FORS key is always at layer 0), the tree address set to the XMSS tree that signs the FORS key, the *type* set to `FORS_TREE`, and the key pair address set to the index of the WOTS$^+$ key within the XMSS tree that signs the FORS key.

Each node in the Merkle tree is the root of a subtree, and Algorithm 14 computes the root of a subtree recursively. If the subtree consists of a single leaf node, then the function simply returns a hash of the node's private n-byte string (lines 5 through 8). Otherwise, the function computes the roots of the left subtree (line 10) and right subtree (line 11) and hashes them together (lines 12 through 14).

Algorithm 14 fors_node(**SK**.seed, i, z, **PK**.seed, **ADRS**)

Compute the root of a Merkle subtree of FORS public values.

Input: Secret seed **SK**.seed, target node index i, target node height z, public seed **PK**.seed, address **ADRS**.
Output: n-byte root *node*.

1: **if** $z > a$ **or** $i \geq k \cdot 2^{(a-z)}$ **then**
2:     **return** NULL
3: **end if**
4: **if** $z = 0$ **then**
5:     $sk \leftarrow$ fors_SKgen($\mathbf{SK}.\text{seed}, \mathbf{PK}.\text{seed}, \mathbf{ADRS}, i$)
6:     **ADRS**.setTreeHeight(0)
7:     **ADRS**.setTreeIndex(i)
8:     $node \leftarrow \mathbf{F}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, sk)$
9: **else**
10:     $lnode \leftarrow$ fors_node($\mathbf{SK}.\text{seed}, 2i, z-1, \mathbf{PK}.\text{seed}, \mathbf{ADRS}$)
11:     $rnode \leftarrow$ fors_node($\mathbf{SK}.\text{seed}, 2i+1, z-1, \mathbf{PK}.\text{seed}, \mathbf{ADRS}$)
12:     **ADRS**.setTreeHeight(z)
13:     **ADRS**.setTreeIndex(i)
14:     $node \leftarrow \mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, lnode \parallel rnode)$
15: **end if**
16: **return** *node*

8.3 Generating a FORS Signature

The fors_sign function (Algorithm 15) signs a ka-bit message digest md.[12] In addition to the message digest, fors_sign takes as input **SK**.seed and **PK**.seed from the SLH-DSA private key and an address. The address **ADRS** must have the layer address set to zero (since the XMSS tree that signs a FORS key is always at layer 0), the tree address set to the XMSS tree that signs the

[12]For convenience, fors_sign takes as input a $\lceil \frac{k \cdot a}{8} \rceil$ byte message digest and then extracts $k \cdot a$ bits to sign.

FORS key, the *type* set to `FORS_TREE`, and the key pair address set to the index of the WOTS$^+$ key within the XMSS tree that signs the FORS key.

The fors_sign function splits ka bits of md into k a-bit strings (line 2), each of which is interpreted as an integer between 0 and $t-1$. Each of these integers is used to select a secret value from one of the k sets (line 4). For each secret value selected, an authentication path is computed and added to the signature (lines 6 through 10).

Algorithm 15 fors_sign(md, **SK**.seed, **PK**.seed, **ADRS**)

Generate a FORS signature.

Input: Message digest md, secret seed **SK**.seed, address **ADRS**, public seed **PK**.seed.
Output: FORS signature SIG_{FORS}.

1: $\text{SIG}_{FORS} = \text{NULL}$ ▷ Initialize SIG_{FORS} as a zero-length byte string
2: $indices \leftarrow \text{base\_2}^{\text{b}}(md, a, k)$
3: **for** i **from** 0 **to** $k-1$ **do** ▷ Compute signature elements
4: $\quad \text{SIG}_{FORS} \leftarrow \text{SIG}_{FORS} \parallel \text{fors\_SKgen}(\textbf{SK}.\text{seed}, \textbf{PK}.\text{seed}, \textbf{ADRS}, i \cdot 2^a + indices[i])$
5:
6: $\quad$ **for** j **from** 0 **to** $a-1$ **do** ▷ Compute auth path
7: $\quad\quad s \leftarrow \lfloor indices[i]/2^j \rfloor \oplus 1$
8: $\quad\quad \text{AUTH}[j] \leftarrow \text{fors\_node}(\textbf{SK}.\text{seed}, i \cdot 2^{a-j} + s, j, \textbf{PK}.\text{seed}, \textbf{ADRS})$
9: $\quad$ **end for**
10: $\quad \text{SIG}_{FORS} \leftarrow \text{SIG}_{FORS} \parallel \text{AUTH}$
11: **end for**
12: **return** SIG_{FORS}

8.4 Computing a FORS Public Key From a Signature

As noted in Section 3, verifying a FORS signature involves computing a public-key value from a message digest and a signature value. Verification succeeds if the correct public-key value is computed, which is determined by verifying the hypertree signature on the computed public-key value using the SLH-DSA public key. This section describes fors_pkFromSig (Algorithm 16), a function that computes a candidate FORS public key from a FORS signature and corresponding message digest.

In addition to a message digest md and a $k \cdot (a+1) \cdot n$-byte signature SIG_{FORS}, fors_pkFromSig takes as input **PK**.seed from the SLH-DSA public key and an address.[13] The address **ADRS** must have the layer address set to zero (since the XMSS tree that signs a FORS key is always at layer 0), the tree address set to the XMSS tree that signs the FORS key, the *type* set to `FORS_TREE`, and the key pair address set to the index of the WOTS$^+$ key within the XMSS tree that signs the FORS key.

The fors_pkFromSig function begins by computing the roots of each of the k Merkle trees (lines 2 through 21). As in fors_sign, ka bits of the message digest are split into k a-bit strings (line 1), each of which is interpreted as an integer between 0 and $t-1$. The integers are used to determine

[13]As with fors_sign, fors_pkFromSig takes as input a $\lceil \frac{k \cdot a}{8} \rceil$ byte message digest and then extracts $k \cdot a$ bits.

the locations in the Merkle trees of the secret values from the signature (lines 3 through 5). The hashes of the secret values are computed (line 6), and the hash values are used along with the corresponding authentication paths from the signature to compute the Merkle tree roots (lines 8 through 20). Once all of the Merkle tree roots have been computed, they are hashed together to compute the FORS public key (lines 22 through 25).

Algorithm 16 fors_pkFromSig(SIG_{FORS}, *md*, **PK**.seed, **ADRS**)

Compute a FORS public key from a FORS signature.

Input: FORS signature SIG_{FORS}, message digest *md*, public seed **PK**.seed, address **ADRS**.
Output: FORS public key.

1: $indices \leftarrow$ base_2^b(md, a, k)
2: **for** i **from** 0 **to** $k-1$ **do**
3: $\quad sk \leftarrow \mathrm{SIG}_{FORS}.\mathrm{getSK}(i)$ ▷ $\mathrm{SIG}_{FORS}[i\cdot(a+1)\cdot n : (i\cdot(a+1)+1)\cdot n]$
4: $\quad$ **ADRS**.setTreeHeight(0) ▷ Compute leaf
5: $\quad$ **ADRS**.setTreeIndex($i \cdot 2^a + indices[i]$)
6: $\quad node[0] \leftarrow \mathbf{F}(\mathbf{PK}.\mathrm{seed}, \mathbf{ADRS}, sk)$
7:
8: $\quad auth \leftarrow \mathrm{SIG}_{FORS}.\mathrm{getAUTH}(i)$ ▷ $\mathrm{SIG}_{FORS}[(i\cdot(a+1)+1)\cdot n : (i+1)\cdot(a+1)\cdot n]$
9: $\quad$ **for** j **from** 0 **to** $a-1$ **do** ▷ Compute root from leaf and AUTH
10: $\quad\quad$ **ADRS**.setTreeHeight($j+1$)
11: $\quad\quad$ **if** $\lfloor indices[i]/2^j \rfloor$ is even **then**
12: $\quad\quad\quad$ **ADRS**.setTreeIndex(**ADRS**.getTreeIndex()$/2$)
13: $\quad\quad\quad node[1] \leftarrow \mathbf{H}(\mathbf{PK}.\mathrm{seed}, \mathbf{ADRS}, node[0] \parallel auth[j])$
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ **ADRS**.setTreeIndex((**ADRS**.getTreeIndex()$-1)/2$)
16: $\quad\quad\quad node[1] \leftarrow \mathbf{H}(\mathbf{PK}.\mathrm{seed}, \mathbf{ADRS}, auth[j] \parallel node[0])$
17: $\quad\quad$ **end if**
18: $\quad\quad node[0] \leftarrow node[1]$
19: $\quad$ **end for**
20: $\quad root[i] \leftarrow node[0]$
21: **end for**
22: forspkADRS $\leftarrow$ **ADRS** ▷ Compute the FORS public key from the Merkle tree roots
23: forspkADRS.setTypeAndClear(`FORS_ROOTS`)
24: forspkADRS.setKeyPairAddress(**ADRS**.getKeyPairAddress())
25: $pk \leftarrow \mathbf{T}_k(\mathbf{PK}.\mathrm{seed}, \mathrm{forspkADRS}, root)$
26: **return** *pk*;

9. SLH-DSA

SLH-DSA uses the hypertree and the FORS keys to create a stateless hash-based signature scheme. The SLH-DSA private key contains a secret seed value and a secret PRF key. The public key consists of a key identifier **PK**.seed and the root of the hypertree. A signature is created by hashing the message, using part of the message digest to select a FORS key, signing other bits from the message digest with the FORS key, and generating a hypertree signature for the FORS key. The parameters for SLH-DSA are those specified previously for WOTS$^+$, XMSS, the SLH-DSA hypertree, and FORS, which are given in Table 1.

SLH-DSA uses one additional parameter m, which is the length in bytes of the message digest. It is computed as:

$$m = \left\lceil \frac{h - h'}{8} \right\rceil + \left\lceil \frac{h'}{8} \right\rceil + \left\lceil \frac{k \cdot a}{8} \right\rceil$$

SLH-DSA uses h bits of the message digest to select a FORS key: $h - h'$ bits to select an XMSS tree at the lowest layer and h' bits to select a WOTS$^+$ key (and corresponding FORS key) from that tree. $k \cdot a$ bits of the digest are signed by the selected FORS key. While only $h + k \cdot a$ bits of the message digest are used, implementation is simplified by extracting the necessary bits from a slightly larger digest.

9.1 SLH-DSA Key Generation

SLH-DSA public keys contain two elements (see Figure 15). The first is an n-byte public seed **PK**.seed, which is used in many hash function calls to provide domain separation between different SLH-DSA key pairs. The second value is the hypertree public key (i.e., the root of the top layer XMSS tree). **PK**.seed **shall** be generated using an **approved** random bit generator (see the NIST SP 800-90 series of publications [16, 17, 18]), where the instantiation of the random bit generator supports at least $8n$ bits of security strength.

The SLH-DSA private key contains two random, secret n-byte values (see Figure 14). **SK**.seed is used to generate all of the WOTS$^+$ and FORS private key elements. **SK**.prf is used to generate a randomization value for the randomized hashing of the message in SLH-DSA. The private key also includes a copy of the public key. Both **SK**.seed and **SK**.prf **shall** be generated using an **approved** random bit generator, where the instantiation of the random bit generator supports at least $8n$ bits of security strength.

Algorithm 17 generates an SLH-DSA key pair. Lines 1 through 3 generate the random values for the private and public keys using an instantiation of an **approved** random bit generator that

SK.seed	n bytes
SK.prf	n bytes
PK.seed	n bytes
PK.root	n bytes

Figure 14. SLH-DSA private key

PK.seed	n bytes
PK.root	n bytes

Figure 15. SLH-DSA public key

supports at least $8n$ bits of security strength. Lines 5 through 7 then compute the root of the top layer XMSS tree.

Algorithm 17 slh_keygen()

Generate an SLH-DSA key pair.

Input: (none)
Output: SLH-DSA key pair (SK, PK).

```
1: SK.seed ←$ B^n            ▷ Set SK.seed, SK.prf, and PK.seed to random n-byte
2: SK.prf ←$ B^n             ▷ strings using an approved random bit generator
3: PK.seed ←$ B^n
4:
5: ADRS ← toByte(0, 32)      ▷ Generate the public key for the top-level XMSS tree
6: ADRS.setLayerAddress(d − 1)
7: PK.root ← xmss_node(SK.seed, 0, h′, PK.seed, ADRS)
8:
9: return ( (SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root) )
```

9.2 SLH-DSA Signature Generation

An SLH-DSA signature consists of a randomization string, a FORS signature, and a hypertree signature, as shown in Figure 16.

Generating an SLH-DSA signature (Algorithm 18) begins by creating an m-byte message digest (lines 3 through 10). A PRF is used to create a message randomizer (line 7), and it is hashed along with the message to create the digest (line 10). Bits are then extracted from the message digest to be signed by the FORS key (line 11), to select an XMSS tree (lines 12 and 15), and to select a WOTS$^+$ key and corresponding FORS key within that XMSS tree (lines 13 and 16). Next, the FORS signature is computed (lines 18 through 21) and the corresponding FORS public key is obtained (line 24). Finally, the FORS public key is signed (line 26).

The message randomizer may be set in either a deterministic or non-deterministic way, depending on whether *opt_rand* is set to a fixed value (line 3) or a random value (line 5). If *opt_rand* is set to **PK**.seed, then signing will be deterministic — signing the same message twice will result in the same signature. For devices that are vulnerable to side-channel attacks and for which deterministic signing would be a problem, *opt_rand* may be set to a random value. The generation of a random value for *opt_rand* does not require the use of an **approved** random bit generator.

Randomness **R**	n bytes
FORS signature $\mathbf{SIG}_{\text{FORS}}$	$k(1+a) \cdot n$ bytes
HT signature $\mathbf{SIG}_{\text{HT}}$	$(h + d \cdot len) \cdot n$ bytes

Figure 16. SLH-DSA signature data format

Algorithm 18 slh_sign(M, SK)

Generate an SLH-DSA signature.

Input: Message M, private key SK = (**SK**.seed, **SK**.prf, **PK**.seed, **PK**.root).
Output: SLH-DSA signature SIG.

$\mathbf{ADRS} \leftarrow \text{toByte}(0, 32)$
$opt\_rand \leftarrow \mathbf{PK}.\text{seed}$ ▷ Set opt_rand to either **PK**.seed
if (RANDOMIZE) **then** ▷ or to a random n-byte string
  $opt\_rand \xleftarrow{\$} \mathbb{B}^n$
end if
$R \leftarrow \mathbf{PRF}_{msg}(\mathbf{SK}.\text{prf}, opt\_rand, M)$ ▷ Generate randomizer
$\text{SIG} \leftarrow R$
$digest \leftarrow \mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M)$ ▷ Compute message digest
$md \leftarrow digest\left[0 : \left\lceil \frac{k \cdot a}{8} \right\rceil\right]$ ▷ first $\left\lceil \frac{k \cdot a}{8} \right\rceil$ bytes
$tmp\_idx_{tree} \leftarrow digest\left[\left\lceil \frac{k \cdot a}{8} \right\rceil : \left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h - h/d}{8} \right\rceil\right]$ ▷ next $\left\lceil \frac{h - h/d}{8} \right\rceil$ bytes
$tmp\_idx_{leaf} \leftarrow digest\left[\left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h - h/d}{8} \right\rceil : \left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h - h/d}{8} \right\rceil + \left\lceil \frac{h}{8d} \right\rceil\right]$ ▷ next $\left\lceil \frac{h}{8d} \right\rceil$ bytes
$idx_{tree} \leftarrow \text{toInt}\left(tmp\_idx_{tree}, \left\lceil \frac{h - h/d}{8} \right\rceil\right) \bmod 2^{h - h/d}$
$idx_{leaf} \leftarrow \text{toInt}\left(tmp\_idx_{leaf}, \left\lceil \frac{h}{8d} \right\rceil\right) \bmod 2^{h/d}$
$\mathbf{ADRS}.\text{setTreeAddress}(idx_{tree})$
$\mathbf{ADRS}.\text{setTypeAndClear}(\texttt{FORS\_TREE})$
$\mathbf{ADRS}.\text{setKeyPairAddress}(idx_{leaf})$
$\text{SIG}_{FORS} \leftarrow \text{fors\_sign}(md, \mathbf{SK}.\text{seed}, \mathbf{PK}.\text{seed}, \mathbf{ADRS})$
$\text{SIG} \leftarrow \text{SIG} \parallel \text{SIG}_{FORS}$
$\text{PK}_{FORS} \leftarrow \text{fors\_pkFromSig}(\text{SIG}_{FORS}, md, \mathbf{PK}.\text{seed}, \mathbf{ADRS})$ ▷ Get FORS key
$\text{SIG}_{HT} \leftarrow \text{ht\_sign}(\text{PK}_{FORS}, \mathbf{SK}.\text{seed}, \mathbf{PK}.\text{seed}, idx_{tree}, idx_{leaf})$
$\text{SIG} \leftarrow \text{SIG} \parallel \text{SIG}_{HT}$
return SIG

9.3 SLH-DSA Signature Verification

As with signature generation, SLH-DSA signature verification (Algorithm 19) begins by computing a message digest (line 9) and then extracting *md* (line 10), idx_{tree} (lines 11 and 14), and idx_{leaf} (lines 12 and 15) from the digest. A candidate FORS public key is then computed (line 21), and the signature on the FORS key is verified (line 23). If this signature verification succeeds, then the correct FORS public key was computed, and the signature SIG on message M is valid.

Algorithm 19 slh_verify(M, SIG, PK)

Verify an SLH-DSA signature.

Input: Message M, signature SIG, public key PK = (**PK**.seed, **PK**.root).
Output: Boolean.

1: **if** $|SIG| \neq (1+k(1+a)+h+d \cdot len) \cdot n$ **then**
2:     **return** false
3: **end if**
4: **ADRS** $\leftarrow$ toByte$(0, 32)$
5: $R \leftarrow$ SIG.getR() ▷ SIG$[0:n]$
6: $\text{SIG}_{FORS} \leftarrow$ SIG.getSIG_FORS() ▷ SIG$[n:(1+k(1+a)) \cdot n]$
7: $\text{SIG}_{HT} \leftarrow$ SIG.getSIG_HT() ▷ SIG$[(1+k(1+a)) \cdot n : (1+k(1+a)+h+d \cdot len) \cdot n]$
8:
9: $digest \leftarrow \mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M)$ ▷ Compute message digest
10: $md \leftarrow digest\left[0 : \left\lceil \frac{k \cdot a}{8} \right\rceil\right]$ ▷ first $\left\lceil \frac{k \cdot a}{8} \right\rceil$ bytes
11: $tmp\_idx_{tree} \leftarrow digest\left[\left\lceil \frac{k \cdot a}{8} \right\rceil : \left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h-h/d}{8} \right\rceil\right]$ ▷ next $\left\lceil \frac{h-h/d}{8} \right\rceil$ bytes
12: $tmp\_idx_{leaf} \leftarrow digest\left[\left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h-h/d}{8} \right\rceil : \left\lceil \frac{k \cdot a}{8} \right\rceil + \left\lceil \frac{h-h/d}{8} \right\rceil + \left\lceil \frac{h}{8d} \right\rceil\right]$ ▷ next $\left\lceil \frac{h}{8d} \right\rceil$ bytes
13:
14: $idx_{tree} \leftarrow$ toInt$\left(tmp_idx_{tree}, \left\lceil \frac{h-h/d}{8} \right\rceil\right) \bmod 2^{h-h/d}$
15: $idx_{leaf} \leftarrow$ toInt$\left(tmp_idx_{leaf}, \left\lceil \frac{h}{8d} \right\rceil\right) \bmod 2^{h/d}$
16:
17: **ADRS**.setTreeAddress(idx_{tree}) ▷ Compute FORS public key
18: **ADRS**.setTypeAndClear(FORS_TREE)
19: **ADRS**.setKeyPairAddress(idx_{leaf})
20:
21: $\text{PK}_{FORS} \leftarrow$ fors_pkFromSig(SIG_{FORS}, md, **PK**.seed, **ADRS**)
22:
23: **return** ht_verify(PK_{FORS}, SIG_{HT}, **PK**.seed, idx_{tree}, idx_{leaf}, **PK**.root)

9.4 Prehash SLH-DSA

For some cryptographic modules that generate SLH-DSA signatures, performing lines 7 and 10 of Algorithm 18 may be infeasible if the message M is large. This may, for example, be the result of the module having limited memory to store the message to be signed. Similarly, for some cryptographic modules that verify SLH-DSA signatures, performing step 9 of Algorithm 19 may be infeasible if the message M is large. For some use cases, these issues may be addressed by signing a digest of the message rather than signing the message directly. In order to maintain the same level of security strength, the digest that is signed needs to be generated using an **approved** hash function or extendable-output function (XOF) (e.g., from FIPS 180-4 [12] or FIPS 202 [10]) that provides at least $8n$ bits of classical security strength against both collision and second preimage attacks [10, Table 4].[14] Note that verification of a signature created in this way will require the verify function to generate a digest from the message in the same way for input to the verification function.

It should be noted that even if it is feasible to compute collisions on the hash functions (or XOF) used to instantiate $\mathbf{H}_{msg}$, **PRF**, $\mathbf{PRF}_{msg}$, **F**, **H**, and $\mathbf{T}_l$, there is believed to be no adverse effect on the security of SLH-DSA.[15] However, if the input to the signing function is a digest of the message, then collisions on the function used to compute the digest can result in forged messages.

[14] Obtaining at least $8n$ bits of classical security strength against collision attacks requires that the digest to be signed is at least $2n$ bytes in length.

[15] As noted in Section 10, applications that require message-bound signatures may be adversely affected if it is feasible to compute collisions on $\mathbf{H}_{msg}$.

10. Parameter Sets

This standard approves 12 parameter sets for use with SLH-DSA. A parameter set consists of parameters for WOTS$^+$ (n and lg_w), XMSS and the SLH-DSA hypertree (h and d), and FORS (k and a), as well as instantiations for the functions $\mathbf{H}_{msg}$, **PRF**, $\mathbf{PRF}_{msg}$, **F**, **H**, and $\mathbf{T}_l$.

Table 1 lists the parameter sets that are **approved** for use. Each parameter set name indicates the hash function family (SHA2 or SHAKE) that is used to instantiate the hash functions, the length in bits of the security parameter n, and whether the parameter set was designed to create relatively small signatures ('s') or to have relatively fast signature generation ('f'). There are six sets of values for n, lg_w, h, d, k, and a that are **approved** for use.[16] For each of the six sets of values, the functions $\mathbf{H}_{msg}$, **PRF**, $\mathbf{PRF}_{msg}$, **F**, **H**, and $\mathbf{T}_l$ may be instantiated using either SHAKE [10] or SHA-2 [12]. For the SHAKE parameter sets, the functions **shall** be instantiated as specified in Section 10.1. For the SHA2 parameter sets, the functions **shall** be instantiated as specified in Section 10.2 if $n = 16$ and **shall** be instantiated as specified in Section 10.3 if $n = 24$ or $n = 32$.

Table 1. SLH-DSA parameter sets

	n	h	d	h'	a	k	lg_w	m	sec level	pk bytes	sig bytes
SLH-DSA-SHA2-128s SLH-DSA-SHAKE-128s	16	63	7	9	12	14	4	30	1	32	7 856
SLH-DSA-SHA2-128f SLH-DSA-SHAKE-128f	16	66	22	3	6	33	4	34	1	32	17 088
SLH-DSA-SHA2-192s SLH-DSA-SHAKE-192s	24	63	7	9	14	17	4	39	3	48	16 224
SLH-DSA-SHA2-192f SLH-DSA-SHAKE-192f	24	66	22	3	8	33	4	42	3	48	35 664
SLH-DSA-SHA2-256s SLH-DSA-SHAKE-256s	32	64	8	8	14	22	4	47	5	64	29 792
SLH-DSA-SHA2-256f SLH-DSA-SHAKE-256f	32	68	17	4	9	35	4	49	5	64	49 856

In Sections 10.2 and 10.3, the functions MGF1-SHA-256 and MGF1-SHA-512 are MGF from Section 7.2.2.2 of NIST SP 800-56B Revision 2 [9], where *hash* is SHA-256 or SHA-512, respectively. The functions HMAC-SHA-256 and HMAC-SHA-512 are the HMAC function from FIPS 198-1 [20], where H is SHA-256 or SHA-512, respectively.

The 12 parameter sets included in Table 1 were designed to meet certain security strength categories defined by NIST in its original Call for Proposals [21] with respect to existential unforgeability under chosen message attack (EUF-CMA) when each key pair is used to sign at most 2^{64} messages.[17] These security strength categories are explained further in Appendix A.

[16] In addition to n, lg_w, h, d, k, and a, Table 1 also lists values for parameters that may be computed from these values (h', m, public-key size, and signature size). The security level is the security category in which the parameter set is claimed to be [4].

[17] If a key pair were used to sign 10 billion (10^{10}) messages per second it would take over 58 years to sign 2^{64} messages.

Using this approach, security strength is not described by a single number, such as "128 bits of security." Instead, each parameter set is claimed to be at least as secure as a generic block cipher with a prescribed key size. More precisely, it is claimed that the computational resources needed to break SLH-DSA are greater than or equal to the computational resources needed to break the block cipher 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 [22].

Concretely, the parameter sets with $n = 16$ are claimed to be in security category 1, the parameter sets with $n = 24$ are claimed to be in security category 3, and the parameter sets with $n = 32$ are claimed to be in security category 5 [4]. For additional discussion of the security strength of SLH-DSA, see [4, 23].

Some applications require a property known as message-bound signatures [24, 25], which intuitively requires that it be infeasible for anyone to create a public key and a signature that are valid for two different messages. Signature schemes are not required to have this property under the EUF-CMA security definition used in assigning security categories. In the case of SLH-DSA, the key pair owner could create two messages with the same signature by finding a collision on $\mathbf{H}_{msg}$. Due to the length of the output of $\mathbf{H}_{msg}$, finding such a collision would be expected to require fewer computational resources than specified for the parameter sets' claimed security levels in all cases except SLH-DSA-SHA2-128f and SLH-DSA-SHAKE-128f. Therefore, applications that require message-bound signatures should either take the expected cost of finding collisions on $\mathbf{H}_{msg}$ into account when choosing an appropriate parameter set or apply a technique, such as the BUFF transformation [25], in order to obtain the message-bound signatures property.

10.1 SLH-DSA Using SHAKE

$\mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M) = \text{SHAKE256}(R \parallel \mathbf{PK}.\text{seed} \parallel \mathbf{PK}.\text{root} \parallel M, 8m)$

$\mathbf{PRF}(\mathbf{PK}.\text{seed}, \mathbf{SK}.\text{seed}, \mathbf{ADRS}) = \text{SHAKE256}(\mathbf{PK}.\text{seed} \parallel \mathbf{ADRS} \parallel \mathbf{SK}.\text{seed}, 8n)$

$\mathbf{PRF}_{msg}(\mathbf{SK}.\text{prf}, opt\_rand, M) = \text{SHAKE256}(\mathbf{SK}.\text{prf} \parallel opt\_rand \parallel M, 8n)$

$\mathbf{F}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_1) = \text{SHAKE256}(\mathbf{PK}.\text{seed} \parallel \mathbf{ADRS} \parallel M_1, 8n)$

$\mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_2) = \text{SHAKE256}(\mathbf{PK}.\text{seed} \parallel \mathbf{ADRS} \parallel M_2, 8n)$

$\mathbf{T}_\ell(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_\ell) = \text{SHAKE256}(\mathbf{PK}.\text{seed} \parallel \mathbf{ADRS} \parallel M_\ell, 8n)$

10.2 SLH-DSA Using SHA2 for Security Category 1

$\mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M) = \text{MGF1-SHA-256}(R \parallel \mathbf{PK}.\text{seed} \parallel \text{SHA-256}(R \parallel \mathbf{PK}.\text{seed} \parallel \mathbf{PK}.\text{root} \parallel M), m)$

$\mathbf{PRF}(\mathbf{PK}.\text{seed}, \mathbf{SK}.\text{seed}, \mathbf{ADRS}) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64-n) \parallel \mathbf{ADRS}^c \parallel \mathbf{SK}.\text{seed}))$

$\mathbf{PRF}_{msg}(\mathbf{SK}.\text{prf}, opt\_rand, M) = \text{Trunc}_n(\text{HMAC-SHA-256}(\mathbf{SK}.\text{prf}, opt\_rand \parallel M))$

$\mathbf{F}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_1) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64-n) \parallel \mathbf{ADRS}^c \parallel M_1))$

$\mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_2) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64-n) \parallel \mathbf{ADRS}^c \parallel M_2))$

$\mathbf{T}_\ell(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_\ell) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64-n) \parallel \mathbf{ADRS}^c \parallel M_\ell))$

10.3 SLH-DSA Using SHA2 for Security Categories 3 and 5

$\mathbf{H}_{msg}(R, \mathbf{PK}.\text{seed}, \mathbf{PK}.\text{root}, M) = \text{MGF1-SHA-512}(R \parallel \mathbf{PK}.\text{seed} \parallel \text{SHA-512}(R \parallel \mathbf{PK}.\text{seed} \parallel \mathbf{PK}.\text{root} \parallel M), m)$

$\mathbf{PRF}(\mathbf{PK}.\text{seed}, \mathbf{SK}.\text{seed}, \mathbf{ADRS}) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64 - n) \parallel \mathbf{ADRS}^c \parallel \mathbf{SK}.\text{seed}))$

$\mathbf{PRF}_{msg}(\mathbf{SK}.\text{prf}, opt\_rand, M) = \text{Trunc}_n(\text{HMAC-SHA-512}(\mathbf{SK}.\text{prf}, opt\_rand \parallel M))$

$\mathbf{F}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_1) = \text{Trunc}_n(\text{SHA-256}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 64 - n) \parallel \mathbf{ADRS}^c \parallel M_1))$

$\mathbf{H}(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_2) = \text{Trunc}_n(\text{SHA-512}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 128 - n) \parallel \mathbf{ADRS}^c \parallel M_2))$

$\mathbf{T}_\ell(\mathbf{PK}.\text{seed}, \mathbf{ADRS}, M_\ell) = \text{Trunc}_n(\text{SHA-512}(\mathbf{PK}.\text{seed} \parallel \text{toByte}(0, 128 - n) \parallel \mathbf{ADRS}^c \parallel M_\ell))$

References

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

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

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

[4] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS$^+$ – submission to the NIST post-quantum project, v.3.1, 2022.

[5] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS$^+$ – submission to the NIST post-quantum project, v.3, 2020.

[6] Morgan Stern. Re: Diversity of signature schemes. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/2LEoSpskELs/m/LkUdQ5mKAwAJ, 2021.

[7] Sydney Antonov. Round 3 official comment: SPHINCS+. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/FVItvyRea28/m/mGaRi5iZBwAJ, 2022.

[8] Ray Perlner, John Kelsey, and David Cooper. Breaking category five SPHINCS$^+$ with SHA-256. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography*, pages 501–522, Cham, 2022. Springer International Publishing.

[9] Elaine B. Barker, Lily Chen, Allen L. Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. Recommendation for pair-wise key-establishment using integer factorization cryptography. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-56B Revision 2, March 2019. https://doi.org/10.6028/NIST.SP.800-56Br2.

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

[11] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash. (National Institute of Standards and Technology, Gaithersburg,

MD), NIST Special Publication (SP) 800-185, December 2016. https://doi.org/10.6028/NIST.SP.800-185.

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

[13] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle signature scheme, Internet Research Task Force (IRTF) request for comments (RFC) 8391. https://doi.org/10.17487/RFC8391, May 2018.

[14] David A Cooper, Daniel Apon, Quynh H Dang, Michael S Davidson, Morris J Dworkin, and Carl A Miller. Recommendation for stateful hash-based signature schemes. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-208, October 2020. https://doi.org/10.6028/NIST.SP.800-208.

[15] Ralph Charles Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford university, 1979.

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

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

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

[19] Leon Groot Bruinderink and Andreas Hülsing. “Oops, i did it again” – security of one-time signatures under two-message attacks. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 299–322, Cham, 2018. Springer International Publishing.

[20] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 198-1, July 2008. https://doi.org/10.6028/NIST.FIPS.198-1.

[21] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016.

[22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-

quantum cryptography standardization process. Technical Report NIST Interagency or Internal Report (IR) 8413, National Institute of Standards and Technology, Gaithersburg, MD, July 2022.

[23] Andreas Hülsing and Mikhail Kudinov. Recovering the tight security proof of SPHINCS$^+$. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 3–33, Cham, 2022. Springer Nature Switzerland.

[24] Jacques Stern, David Pointcheval, John Malone-Lee, and Nigel P. Smart. Flaws in applying proof methodologies to signature schemes. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 93–110, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[25] C. Cremers, S. Düzlü, R. Fiedler, C. Janson, and M. Fischlin. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1696–1714, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

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

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

[28] Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. Differential power analysis of XMSS and SPHINCS. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 168–188, Cham, 2018. Springer International Publishing.

[29] Laurent Castelnovi, Ange Martinelli, and Thomas Prest. Grafting trees: A fault attack against the SPHINCS framework. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 165–184, Cham, 2018. Springer International Publishing.

[30] Aymeric Genêt, Matthias J. Kannwischer, Hervé Pelletier, and Andrew McLauchlan. Practical fault injection attacks on SPHINCS. Cryptology ePrint Archive, Paper 2018/674, 2018. https://eprint.iacr.org/2018/674.

[31] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. FPGA-based SPHINCS+ implementations: Mind the glitch. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 229–237, 2020.

[32] Aymeric Genêt. On protecting SPHINCS+ against fault attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):80–114, Mar. 2023.

Appendix A — Security Strength Categories

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

In order to address these uncertainties, NIST proposed the following approach in its original Call for Proposals [21]. Instead of defining the strength of an algorithm using precise estimates of the number of "bits of security," NIST defined a collection of broad security strength categories. Each category is defined by a comparatively easy-to-analyze reference primitive whose security will serve as a floor for a wide variety of metrics that NIST deems potentially relevant to practical security. A given cryptosystem may be instantiated using different parameter sets in order to fit into different categories. The goals of this classification are:

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

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

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

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

Table 2. NIST Security Strength Categories

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

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

NIST intends to consider a variety of possible metrics, reflecting different predictions about the future development of quantum and classical computing technology, and the cost of different computing resources (such as the cost of accessing extremely large amounts of memory).[18] NIST will also consider input from the cryptographic community regarding this question.

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

The complexity of quantum attacks can then be measured in terms of circuit size. These numbers can be compared to the resources required to break AES and SHA-3. During the post-quantum standardization process, NIST gave the estimates in Table 3 for the classical and quantum gate counts[19] for the optimal key recovery and collision attacks on AES and SHA-3, respectively, where circuit depth is limited to MAXDEPTH.[20]

[18] See the discussion in [22, Appendix B].

[19] Quantum circuit sizes are based on the work in [26].

[20] NIST believes the above estimates are accurate for the majority of values of MAXDEPTH that are relevant to its

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

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

It is worth noting that the security categories based on these reference primitives provide substantially more quantum security than a naïve analysis might suggest. For example, categories 1, 3, and 5 are defined in terms of block ciphers, which can be broken using Grover's algorithm [27] with a quadratic quantum speedup. However, Grover's algorithm requires a long-running serial computation, which is difficult to implement in practice. In a realistic attack, one has to run many smaller instances of the algorithm in parallel, which makes the quantum speedup less dramatic.

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

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.

Appendix B — Implementation Considerations

This appendix discusses some implementation considerations for SLH-DSA.

Don't support component use. As WOTS$^+$, XMSS, FORS, and hypertree signature schemes are not approved for use as standalone signature schemes, cryptographic modules **should not** make interfaces to these components available to applications. NIST SP 800-208 [14] specifies **approved** stateful hash-based signature schemes.

Side-channel and fault attacks. For signature schemes, secrecy of the private key is critical. Care must be taken to protect implementations against attacks, such as side-channel attacks or fault attacks [28, 29, 30, 31, 32]. A cryptographic device may leak critical information with side-channel analysis or attacks that allow internal data or keying material to be extracted without breaking the cryptographic primitives.

Floating-point arithmetic. Implementations of SLH-DSA **should not** use floating-point arithmetic, as rounding errors in floating point operations may lead to incorrect results in some cases. In all pseudocode in this standard in which division is performed (e.g., x/y), and y may not divide x, either $\lfloor x/y \rfloor$ or $\lceil x/y \rceil$ is used. Both of these may be computed without floating-point arithmetic as ordinary integer division x/y computes $\lfloor x/y \rfloor$, and $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$.

While the value of len_2 (see Equation 5.3) may be computed without using floating-point arithmetic (see Algorithm 20), it is recommended that this value be precomputed. When $lg_w = 4$ and $9 \le n \le 136$, the value of len_2 will be 3.

Algorithm 20 gen_len$_2$(n, lg_w)

Compute len_2 (Equation 5.3).

Input: Security parameter n, bits per hash chain lg_w.
Output: len_2.

```
w ← 2^{lg_w}                                   ▷ Equation 5.1
len_1 ← ⌈(8·n + lg_w − 1) / lg_w⌉              ▷ Equation 5.2
max_checksum = len_1 · (w − 1)                 ▷ Maximum checksum value that may need to be signed

len_2 ← 1                                      ▷ Maximum value that may be signed using
capacity ← w                                   ▷ len_2 hash chains is w^{len_2} − 1 = capacity − 1
while capacity ≤ max_checksum do
    len_2 ← len_2 + 1
    capacity ← capacity · w
end while
return len_2
```
