ETSI TR 103 692 V1.1.1 (2021-11)

CYBER;
State management for stateful authentication mechanisms

Reference
DTR/CYBER-QSC-0016

Keywords
digital signature, Quantum Safe Cryptography

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights 5
Foreword 5
Modal verbs terminology 5
Introduction 5
1 Scope 6
2 References 6
2.1 Normative references 6
2.2 Informative references 6
3 Definition of terms, symbols and abbreviations 7
3.1 Terms 7
3.2 Symbols 8
3.3 Abbreviations 8
4 Background 9
4.1 Hash-based Signatures 9
4.1.1 Introduction 9
4.1.2 One-time signature schemes 10
4.1.2.1 General 10
4.1.2.2 The Winternitz OTS 10
4.1.3 Few-time signature schemes 10
4.1.4 Many-time signature schemes 11
4.1.4.1 Introduction 11
4.1.4.2 Binary hash trees 11
4.1.4.3 Authentication paths 12
4.1.4.4 Many-time signatures 13
4.1.4.5 Tree traversal 14
4.2 Hierarchical systems 14
4.3 Stateful vs stateless 16
4.4 The state index 17
4.5 Notational differences between IRTF RFC 8391 and IRTF RFC 8554 17
5 The state object 18
5.1 Contents of the state object 18
5.2 Characteristics of the state object 20
5.2.1 Size of the state object 20
5.2.2 Format of the state object 21
5.2.3 Sensitivity and access of the state object 21
6 State index reuse 22
6.1 Secure state index reuse 22
6.2 Insecure state index reuse 22
6.3 Avoiding and detecting insecure state reuse 24
7 Operational considerations 25
7.1 Storage of the state object 25
7.2 Number of signatures generated 25
7.3 Compatibility with existing APIs 26
7.4 Multi-component systems 26
8 Comparisons between HSS and XMSS-MT 26
8.1 Performance comparison 26
8.2 Security comparison 27
8.3 Selecting a S-HBS scheme 28
9 Applications of S-HBS schemes 29
9.1 NIST intended applications for S-HBS schemes 29
9.2 Additional applications for S-HBS schemes 30

9.3 Suitable applications....30
9.3.1 Applications conformable to NIST SP 800-208....30
9.3.2 Applications not conformable to NIST SP 800-208....31
9.4 Non-suitable applications....32
History....34

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Cyber Security (CYBER).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

Implementations of Stateful Hash-Based Signature (S-HBS) schemes require exceptional care to ensure they are done securely. Existing specifications are complex and prioritize the interoperability of implementations at the cost of certain security and operational considerations. An implementor of a S-HBS scheme, using only the existing specifications, is likely to experience unforeseen security vulnerabilities and operational problems due to the under-specification of the state object and its management.

State management is not only about ensuring state is not reused. There are operational considerations of state as well, such as the capabilities of the physical systems the algorithms are run on, and the inherent suitability of S-HBS solutions for specific applications.

1 Scope

The present document is limited to discussion of the characteristics of the state object, the reuse of the state index, and of architectural and operational considerations for deploying stateful hash-based signatures. First, it discusses characteristics of the state object for S-HBS schemes and identifies potential security vulnerabilities and operational problems associated with its management. Second, it gives guidance on mitigating the issues identified. And third, it helps a prospective implementor determine if a S-HBS solution is suitable for their given application; examples of suitable and non-suitable applications are given.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] IRTF RFC 8391: "XMSS: eXtended Merkle Signature Scheme", 2018.

[i.2] IRTF RFC 8554: "Leighton-Micali Hash-Based Signatures", 2019.

[i.3] D. A. Cooper, D. C. Apon, Q. H. Dang, M. S. Davidson, Morris J. Dworkin and Carl A. Miller. "Recommendation for Stateful Hash-Based Signature Schemes", NISTIR 8240, SP 800-208.

NOTE: Available at https://csrc.nist.gov/publications/detail/sp/800-208/final.

[i.4] P. Kampanakis and S. Fluhrer: "LMS vs XMSS", IACR ePrint Archive 2016/085, 2017.

[i.5] L. Lamport: "Constructing digital signatures from a one way function", Technical Report SRI-CSL-98. SRI International Computer Science Laboratory, 1979.

[i.6] J.-P. Aumasson, D.J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe and B. Westerbaan: "SPHINCS+ Submission to the NIST post-quantum project, v.3". October 1, 2020 .

NOTE: Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[i.7] D. J. Bernstein, J. Buchmann and E. Damen, (eds.): "Post-Quantum Cryptography", Springer-Verlag Berlin Heidelberg, 2009.

[i.8] J. Buchmann, E. Dahmen, and M. Schneider: "Merkle tree traversal revisited", LNCS vol. 5299, pages 63-78, 2008.

[i.9] M. Jakobsson, F. T. Leighton, S. Micali, and M. Szydlo: "Fractal Merkle Tree Representation and Traversal". LNCS vol. 2612, pages 314-326, 2003.

[i.10] A. Genêt, M. J. Kannwischer, H. Pelletier, and A. McLauchlan: "Practical Fault Injection Attacks on SPHINCS", IACR ePrint Archive 2018/674, 2018.

[i.11] D. McGrew, P. Kampanakis, S. Fluhrer, S. Gazdag, D. Butin, and J. Buchmann: "State Management for Hash-Based Signatures", IACR ePrint Archive 2016/357, 2016.

[i.12] J. Katz: "Analysis of a Proposed Hash-Based Signature Standard". Security Standardisation Research: Third International Conference, SSR 2016. LNCS, vol. 10074, pp. 261-273. Springer, 2016.

[i.13] S. Fluhrer: "Further Analysis of a Proposed Hash-Based Signature Standard", IACR ePrint Archive 2017/533, 2017.

[i.14] E. Eaton: "Leighton-Micali Hash-Based Signatures in the Quantum Random-Oracle Model", IACR ePrint Archive 2017/607, 2017.

[i.15] A. Hülsing, J. Rijneveld, F. Song: "Mitigating Multi-Target Attacks in Hash-based Signatures", IACR ePrint Archive 2015/1256, 2015.

[i.16] F. Campos, T. Kohlstadt, S. Reith, and M. Stoettinger: "LMS vs XMSS: Comparison of Stateful Hash-Based Signature Schemes on ARM Cortex-M4", IACR ePrint Archive 2020/470, 2020.

[i.17] NIST: "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process", December 2016.

NOTE: Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[i.18] National Institute of Standards and Technology (2001): "Security Requirements for Cryptographic Modules" (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-2, Change Notice 2 December 03, 2002.

NOTE: Available at https://doi.org/10.6028/NIST.FIPS.140-2.

[i.19] National Institute of Standards and Technology (2019): "Security Requirements for Cryptographic Modules" (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-3.

NOTE: Available at https://doi.org/10.6028/NIST.FIPS.140-3.

[i.20] P. Kampanakis, P. Panburana, M. Curcio, C. Shroff, and M. Alam: "Post-Quantum LMS and SPHINCS+ Hash-Based Signatures for UEFI Secure Boot", IACR ePrint Archive 2021/041, 2021.

[i.21] A. Hülsing, C. Busold, and J. Buchmann: "Forward Secure Signatures on Smart Cards", IACR ePrint Archive 2018/924, 2018.

[i.22] M. J. Kannwischer, A. Genêt, D. Butin, J. Krämer, and J. Buchmann: "Differential Power Analysis of XMSS and SPHINCS", IACR ePrint Archive 2018/673, 2018.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

asymmetric cryptography: cryptographic system that utilizes a pair of keys, a private key known only to one entity, and a public key which can be openly distributed without loss of security

authentication path: ordered collection of sibling nodes along a leaf-to-root path in a Merkle tree used to compute the root node of the Merkle tree

binary hash tree: tree structure created by iteratively pairing and hashing leaf nodes

cryptographic hash function: function that maps a bit string of arbitrary length to a fixed length bit string (*message digest* or *digest* for short), and that fulfils some specific security properties

height: number of nodes in a leaf-to-root path in a Merkle tree, inclusive. Equivalently, number of levels in a Merkle tree

hyper-tree: hierarchical structure of binary trees, divided into layers

intermediate signature: signature on a Merkle tree root node within a hyper-tree

layer: index of the vertical position of a Merkle tree in a hyper-tree structure

level: vertical index within a Merkle tree

NOTE: The level of a node is given by the number of nodes along a path from the leaf-level to the given node, non-inclusive.

Merkle tree: binary hash tree used as a component of a Hash-Based Signature (HBS) scheme

message digest/digest: fixed-length output of a cryptographic hash function over a variable length input

node (root, leaf, internal, child, sibling, parent): octet string, serving as a minimal component of a binary hash tree

octet string: ordered sequence of octets/bytes consisting of 8 bits each

private key: key in an asymmetric cryptographic scheme that is kept secret

public key: key in an asymmetric cryptographic scheme that can be made public without loss of security

public key cryptography: See asymmetric cryptography.

security level: measure of the strength of a cryptographic algorithm

NOTE: If 2^n operations are required to break the cryptographic algorithm/scheme/method, then the security level is *n*. Sometimes also referred to as *bit-strength*.

stale state: index corresponding to an OTS key pair which cannot, or can no longer, be used to securely sign a message

NOTE: An index can become stale for a variety of reasons, such as having already been used to sign a message, or as a security mechanism after a system restart.

state index: non-negative integer representing the position of the next unused one-time signature signing key within a S-HBS scheme instance

state object: collection of data related to a stateful hash-based signature scheme instance that is required to compute or verify signatures from that instance

3.2 Symbols

For the purposes of the present document, the following symbols apply:

A || B	The concatenation of binary strings A followed by B.
$H(\,)$	A cryptographic hash function.
N_i^j	The i^{th} node on the j^{th} level of a binary hash tree.
L_i	The i^{th} leaf node of a binary hash tree.
h	The height of a binary hash tree.
w	The Winternitz parameter.
$\mathcal{L}$	The number of layers in a hyper-tree hierarchy.
$\mathcal{H}$	The total height of a hyper-tree hierarchy.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Programming Interface
BDS	Buchmann-Dahmen-Schneider

CA	Certificate Authority
CFRG	Crypto Forum Research Group
ECDSA	Elliptic Curve Digital Signature Algorithm
FIPS	Federal Information Processing Standard
FTS	Few-time Signature
HBS	Hash-Based Signature
HS-HBS	Hierarchical Stateful Hash-Based Signature
HSM	Hardware Security Module
HSS	Hierarchical Signature System
IRTF	Internet Research Task Force
LM-OTS	Leighton-Micali One-Time Signature
LMS	Leighton-Micali Signature
MTS	Many-Time Signature
NIST	National Institute of Standards and Technology
OTS	One-Time Signature
PKI	Public Key Infrastructure
PQC	Post-Quantum Cryptography
RFC	Request For Comments
RSA	Rivest-Shamir-Adleman
S-HBS	Stateful Hash-Based Signature
SP	Special Publication
TLS	Transport Layer Security
US	United States
VM	Virtual Machine
W-OTS	Winternitz One-Time Signature
W-OTS+	Winternitz One-Time Signature Plus
XMSS	eXtended Merkle Signature Scheme
XMSS-MT	Multi-tree eXtended Merkle Signature Scheme

4 Background

4.1 Hash-based Signatures

4.1.1 Introduction

The theoretical security of a cryptographic signature scheme is typically derived from the difficulty of solving an instance of some underlying mathematical problem, such as the Discrete Logarithm Problem or the Integer Factorization Problem. Hash-Based Signature (HBS) schemes are atypical in this sense as their theoretical security is based on the security of an underlying hash function.

Although the security of hash functions can themselves be modelled after mathematical problems, making a distinction between these two cases is useful. If an HBS implementation uses a hash function which is believed to be resistant to attacks from quantum-capable adversaries, then the resulting signature scheme will also be (believed to be) quantum-resistant. Further, if the underlying hash function is eventually broken, or if a different hash function is desired for any reason, the scheme can be repaired by switching out and replacing the hash function. With signature schemes such as RSA or ECDSA, a break in the underlying mathematical problem is not generally fixable.

NOTE: Although their theoretical security is based on that of their underlying hash functions, the specific security requirements of the underlying hash functions are not the same for all HBS schemes. Different HBS schemes sometimes require different security properties of the hash function they employ. For this reason, the present document does not explicitly discuss hash function security properties, but instead uses the term "cryptographic hash function" to imply a hash function with the relevant security properties for the HBS scheme under consideration.

The issues of state management are particular to a specific class of HBS schemes, namely, *stateful hash-based signature schemes*. Such schemes are built from a variety of cryptographic algorithms and mathematical techniques. The following clauses describe the main categories of HBS schemes, including the components that comprise such schemes and their related mathematical concepts. Clause 5 and thereafter specifically discuss stateful hash-based signature schemes, the management of state, and applications for stateful hash-based signature schemes.

4.1.2 One-time signature schemes

4.1.2.1 General

A One-Time Signature (OTS) scheme is a cryptographic signature scheme where each instance is secure if at most one message is signed with the signing key of that instance. Re-using a one-time signing key to sign multiple messages greatly degrades the security of the scheme instance, allowing forgeries to be feasibly computed. The reduction in security is because a one-time signature leaks partial information of the private signing materials of the scheme.

EXAMPLE: Signatures from the original hash-based OTS scheme by Lamport [i.5] revealed an expected 50 % of the private key. Provably, an attacker with 50 % of the private key cannot feasibly forge a signature. However, if two signatures are computed under the same instance of the scheme, 75 % of the private key is expected to be revealed, and making forgeries becomes feasible.

In some OTS schemes, a signature directly reveals components of the private key, and in other schemes a signature reveals sensitive information derived from the private key. In either case, by seeing multiple signatures computed under the same OTS signing key, an attacker likely has acquired enough sensitive information to feasibly forge signatures.

OTS schemes are used as building blocks for the more general schemes described in clause 4.1.4.

The present document is primarily concerned with the HBS schemes specified in IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2], and the corresponding profiles described in NIST SP 800-208 [i.3]. IRTF RFC 8391 and IRTF RFC 8554 define the OTS schemes WOTS+ and LM-OTS respectively. As both of these OTS schemes are based on the Winternitz OTS (W-OTS) scheme [i.7], the following clause gives a brief overview of the W-OTS scheme.

4.1.2.2 The Winternitz OTS

In a digital signature scheme, the public key is required to verify signatures that were generated using the corresponding private key. The Winternitz one-time signature scheme, and variants thereof, have the property that their public key can be recomputed from a valid signature by running the signature verification algorithm. At first, this property does not seem particularly useful as the verifier does not have any way of knowing that the recovered public key is the authentic public key of the signer, unless the verifier is also supplied with a copy of the signer's public key. In a stand-alone OTS scheme, it is not generally practical to pre-distribute the public key in this way. However, the W-OTS scheme is used as a building block for the more general signature schemes discussed throughout clause 4.1.4. In the context of these more general schemes, this property of W-OTS signatures can be used effectively. The property allows the sender to not include their OTS public key with the signature, thereby reducing communication overhead. This process is described further in clause 4.1.4.

In the original W-OTS scheme [i.7], a public key is a collection of n-byte octet strings. IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2] specify OTS schemes that are variants of the original W-OTS design. The WOTS+ scheme specified in IRTF RFC 8391 has public keys that are also a collection of n-byte octet strings. When used as a component of the XMSS or XMSS-MT signature schemes (clauses 4.1.4, 4.2 and 4.3) the WOTS+ public keys are compressed into a single n-byte octet string by using structures known as L-trees. Public keys in the LM-OTS scheme specified in IRTF RFC 8554 are similar except that they are compressed into a single n-byte octet string by concatenating together each public key component and computing a hash value of the resulting concatenation.

Throughout the rest of the present document, OTS public keys are implicitly assumed to be n-bytes, where n is the output length of the employed hash function, in bytes.

4.1.3 Few-time signature schemes

A Few-Time Signature (FTS) scheme is a cryptographic signature scheme where each instance is secure if some, but not too many, messages are signed with the signing key of that instance. FTS schemes can be thought of as a slight generalization of OTS schemes.

The core difference between an OTS scheme and an FTS scheme is that FTS signatures leak a smaller fraction of private information relative to OTS signatures. In this way, an attacker needs to see a few signatures before they have enough information to compute a forgery. The number of signatures that can be computed securely in an FTS scheme is dependent on the parameters selected and the specifics of the signature scheme itself.

FTS schemes are used in stateless HBS schemes (clause 4.3) such as the SPHINCS+ NIST PQC Round 3 Alternate Candidate [i.6]. Guidance on FTS schemes is outside the scope of the present document.

4.1.4 Many-time signature schemes

4.1.4.1 Introduction

A many-time signature (MTS) scheme is a cryptographic signature scheme where each instance is capable of signing many, but not unlimited messages under the signing key of that instance. MTS schemes are constructed from instances of one- or few-time schemes by using binary hash tree structures (clause 4.1.4.2). The security of MTS schemes is largely based on the security of the underlying OTS or FTS schemes and the hash function used to construct the binary hash tree, as discussed throughout clause 4.1.4.

MTS schemes are sometimes called *full HBS schemes*; the present document uses the two terms interchangeably.

Clauses 4.1.4.2 to 4.1.4.5 describe the generic components of an MTS scheme.

4.1.4.2 Binary hash trees

Let H be a hash function with n-byte outputs. A *binary hash tree* is a data structure built from iterative invocations of H on an ordered collection of n-byte octet strings, called *leaf nodes*, as described below.

The present document assumes a binary hash tree has 2^h leaf nodes, where the exponent, h, is the *height of the tree*. Each leaf node is indexed on the 0^{th} *level* of the tree. The tree is constructed via an iterative process on the leaf nodes. There is a single node at the topmost level of the tree, called the *root node*. Nodes that are neither leaves nor the root are called *internal nodes*. Nodes used to compute nodes on the level directly above them are called *child nodes*, where those pairs of child nodes are called *sibling nodes*. A node computed directly from two sibling nodes is called the *parent node* of the two child nodes.

Level 1 of the hash tree is constructed by concatenating pairs of sibling leaf nodes and applying H to each of the resulting $2n$-byte octet strings. The outputs of these computations are the nodes of the first level, where the ordering is maintained; there are 2^{h-1} nodes on the first level. When used as a component of an HBS scheme, additional data can be included in the hash computations. The process is iterated to exhaustion. The final output of H is the root node.

Nodes in Figure 1 are indexed with a superscript to denote the level of the node, and a subscript to denote the position of the node on that level, where indexing is done from left to right starting from 0, except for the leaf nodes which are notated differently for readability-the leaf nodes can alternatively be indexed as $L_i = N_i^0$. Level 0 is also called the *leaf level*.

EXAMPLE 1: In Figure 1, $N_0^1 = H(L_0 \,||\, L_1)$, $N_1^1 = H(L_2 \,||\, L_3)$, $N_0^2 = H(N_0^1 \,||\, N_1^1)$, and so forth.

The *height of a node* is the level the node is indexed on within the tree. Equivalently, it is the number of nodes along a path from that node to the leaf level, excluding the node being measured from. Therefore, the root node has height h, leaf nodes have height 0, and internal nodes have heights between 1 and $h - 1$ inclusively.

NOTE: Because the number of leaf nodes is assumed to be a power of 2, all paths from a given node to the leaf level will be of the same length.

EXAMPLE 2: In Figure 1, the height of node N_0^2 is 2 because all paths from it to the leaf level contain two nodes. One such path is N_0^1, L_1. Similarly, the height of the root node N_0^3 is 3.

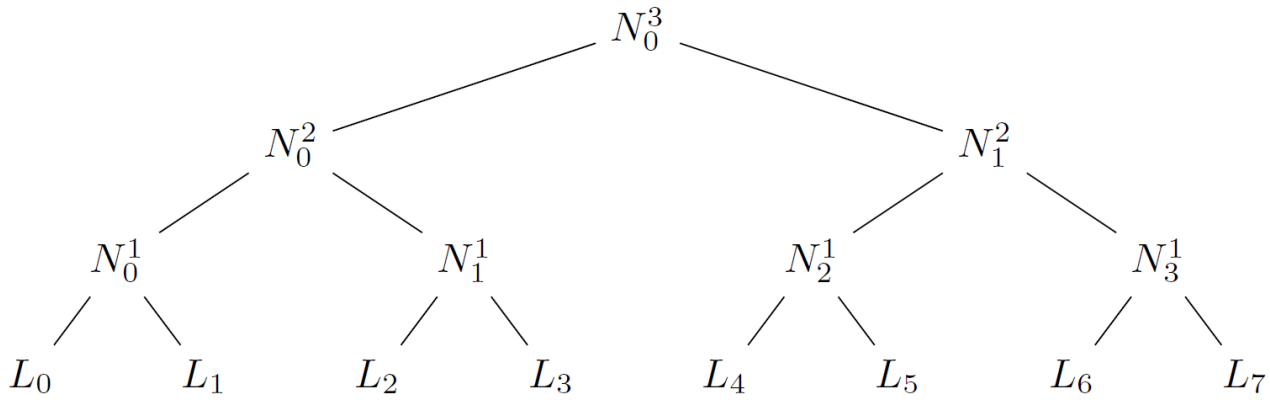

Figure 1: A height 3 binary hash tree

Assuming a cryptographic hash function is used, the root node is determined by the leaf nodes. If even a single bit is flipped in one of the leaf nodes, the resulting root will, with very high probability, be computationally independent of the authentic root. That is, the bit-wise exclusive OR of the two root nodes will be computationally indistinguishable from a uniformly random string of the same length. This is one of the properties of binary hash trees that makes them so well-suited for constructing many-time HBS schemes. This property is discussed further in clause 4.1.4.4.

A binary hash tree used in an MTS scheme is called a *Merkle tree.*

Any tree node is itself the root node of a *sub-tree* embedded in the full tree structure. For a given node N, the corresponding sub-tree has the same height as N. If two sub-trees have no nodes in common, they are said to be *disjoint*.

EXAMPLE 3: In Figure 1, the sub-tree corresponding to node N_1^2 is the height 2 tree constructed from leaf nodes L_4, L_5, L_6, and L_7. The sub-tree corresponding to N_0^1 is the height 1 tree constructed from L_0 and L_1. The sub-trees corresponding to N_1^2 and N_0^1 are disjoint.

A common algorithm for computing a Merkle tree is the TreeHash algorithm. Pseudocode for the TreeHash algorithm is given below. The pseudocode is presented as Algorithm 2.1 in [i.7], from which it is taken directly, except H is used here instead of g as the cryptographic hash function, and h for the tree height, for consistency. As stated above, the output of the TreeHash algorithm is technically the Merkle tree root node. However, to compute the root, each node in the full tree is calculated. By using a different height input, it is not difficult to see how TreeHash can be used to compute any desired tree node.

Further, Leafcalc(j) is simply a sub-routine which calculates the j^{th} leaf node of the Merkle tree by generating the j^{th} OTS key pair, and Stack.push() and Stack.pop() are the typical push and pop operations for a data stack.

TreeHash

Input: Height $h \geq 2$
Output: Root of the Merkle tree
1. **for** $j = 0, \ldots, 2h-1$ **do**
   a) Compute the j^{th} leaf: $Node1 \leftarrow$ Leafcalc(j)
   b) **While** $Node1$ has the same height as the top node on Stack **do**
      i. Pop the top node from the stack: $Node2 \leftarrow$ Stack.pop()
      ii. Compute their parent node: $Node1 \leftarrow H(Node2 \,||\, Node1)$
   c) Push the parent node on the stack: Stack.push$(Node1)$
2. Let R be the single node stored on the stack: $R \leftarrow$ Stack.pop()
3. **Return** R

Algorithm 1: TreeHash

4.1.4.3 Authentication paths

An *authentication path* in a binary hash tree, corresponding to node N, is the ordered collection of the siblings of the nodes on the path starting from N and ending at the root node.

Authentication paths corresponding to internal nodes are interpreted as *partial authentication paths*. For the remainder of the present document, unless stated otherwise, the term *authentication path* is taken to mean an authentication path corresponding to a leaf node.

EXAMPLE: In Figure 2, the authentication path nodes corresponding to leaf L_2 are L_3, N_0^1, and N_1^2, shown in blue and bolded. The regular path nodes from L_2 to the root node are shown in red and with hatched edges. Observe that the root node does not have any siblings and is thus not included in any authentication path.

The authentication path of a node is determined by its position in the tree structure. No two nodes share the exact same authentication path; however, different authentication paths can share some nodes.

If N_1 and N_2 are two distinct nodes in a Merkle tree, then the number of nodes their authentication paths differ by is equal to the height of the smallest sub-tree containing both N_1 and N_2. In the case of sibling nodes, the height 1 sub-tree whose root is the parent node is the smallest such tree. Therefore, the authentication paths of siblings differ by exactly one node, namely the respective sibling. If the smallest sub-tree containing N_1 and N_2 is the full Merkle tree itself, then the two corresponding authentication paths have no nodes in common.

Authentication paths are used as components of signatures in MTS schemes, as they allow for an efficient method to re-compute the root node, which acts as the public key. This concept is further discussed in clause 4.1.4.4; clause 4.1.4.5 discusses algorithms for computing authentication paths.

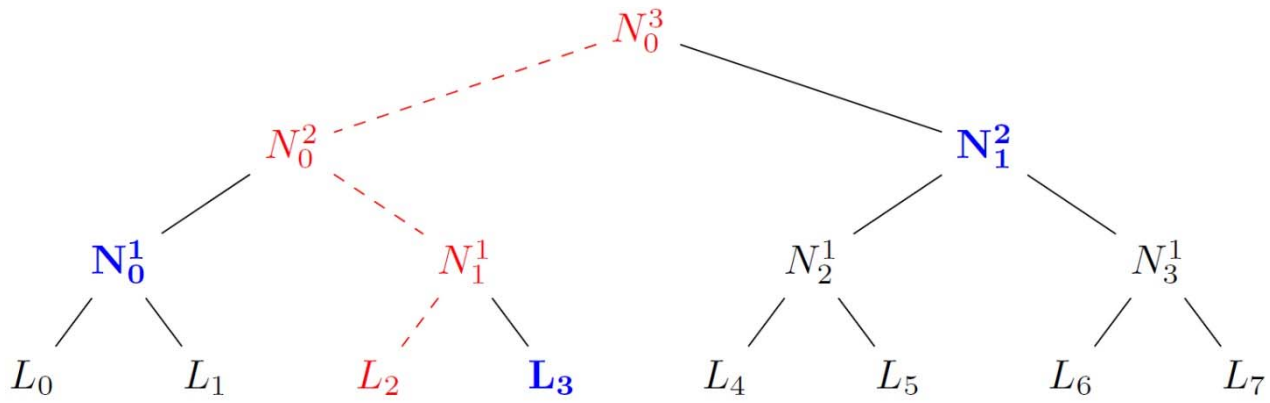

Figure 2: Authentication path for L_2

4.1.4.4 Many-time signatures

A generic MTS scheme can be constructed by generating 2^h W-OTS key pairs and interpreting the public keys as the leaf nodes of a height h Merkle tree. The tree is generated as described in clause 4.1.4.2. Each W-OTS instance, and hence each W-OTS key pair, is associated with an index of a leaf node.

The public key of the MTS scheme is the root node of the Merkle tree. To help ensure a distinction between this public key and the W-OTS public keys, the root node is referred to as the *long-term*, or *global public key*. In practice, the global public key can contain additional data.

A signature from the MTS scheme described above contains a W-OTS signature and the authentication path corresponding to the index of that W-OTS instance.

Verification of a signature in the generic MTS scheme involves computing the (candidate) W-OTS public key from the W-OTS signature by running the W-OTS verification algorithm. Next, by interpreting the W-OTS public key as a leaf node, the verifier uses the nodes from the authentication path to compute the (candidate) root node of the Merkle tree. If the resulting root node matches that from the (published) global public key, then the MTS signature is accepted as valid.

As per clause 4.1.4.1, binary hash trees are determined by their leaf nodes. Therefore, assuming a cryptographic hash function is used, if the W-OTS signature is invalid, then an invalid W-OTS public key (candidate leaf node) is computed by the W-OTS verification algorithm. An invalid leaf node yields a root node that does not match the global public key, with very high probability. Hence, the full signature is rejected as invalid. This means that a verifier does not require a copy of the W-OTS public key prior to verifying the many-time signature, they only require the signer's global public key.

4.1.4.5 Tree traversal

Typically, the OTS instances associated with the leaf nodes of a Merkle tree are used to sign messages sequentially in the natural left-to-right ordering. That is, a leaf node L_i will be used to sign before L_{i+1}, which is used before L_{i+2}, and so on, where $0 \leq i \leq 2^h - 1$. The general reason to adopt this simple ordering is that more complex orderings require more complex logic and management. The present document assumes the simple sequential ordering.

As different nodes have different authentication paths, the authentication path required for the next MTS signature will be different from those included in any previous signature. Implementations have many choices for how to calculate authentication paths. The present document generically refers to any algorithm by which authentication paths are calculated as *tree traversal algorithms*. The present document also uses the terms *strategy* and *algorithm* interchangeably in the context of tree traversal.

The choice of tree traversal algorithm does not, in general, affect interoperability, i.e. it does not affect the format of the signature or any of the outputs from the signing or verification algorithms as specified in IRTF RFC 8391 [i.1], IRTF RFC 8554 [i.2], or NIST SP 800-208 [i.3]. The chosen tree traversal algorithm is simply one possible method for generating the data that is to be used in signing or verification; the actual method of generating that data does not generally affect interoperability, and so there is little discussion of tree traversal algorithms in the current specifications. However, different tree traversal algorithms have different properties, and yield different trade-offs (such as time vs storage). Some of these trade-offs are discussed below.

One possible tree traversal strategy is to recompute each authentication path node when that node is required. This means that the authentication path nodes are not pre-computed and retrieved from storage when they are needed, rather, they are computed as they are needed. This method requires minimal storage but has a high computational cost. Additionally, different nodes can require different amounts of time and resources to calculate, which can lead to undesirably variable signature generation times. One reason for this is because there is a dependent order in which certain nodes can be computed. Recall that each node in a tree can be interpreted as a root node of some sub-tree. To calculate a root node, every node below it in the sub-tree is calculated first, as non-leaf nodes are iteratively calculated from their children, and children's children, etc. Further, no two nodes in any given authentication path have the same height (for example, observe that in Figure 2 each of L_3, N_0^1, and N_1^2 are on a different level), and therefore each node in any given authentication path has a different number of prerequisite nodes to be calculated.

Another tree traversal strategy is to store the entire Merkle tree in storage and select and load authentication path nodes as required. This second strategy requires enough storage for the entire tree but does not require any node re-computation.

Other tree traversal algorithms, such as the BDS and Fractal algorithms [i.8], [i.9], achieve trade-offs somewhere in-between the two mentioned above. The basic idea for such algorithms is to amortize the cost of node computation over some number of signature calculations. Yet another possibility is to store a fraction of the Merkle tree, such as all levels up to and including some level k, where $k < h$ (and where h is the total height of the tree). In this way, only the final $h - k$ nodes of an authentication path need be recomputed as the rest are available from storage; nodes above the k^{th} level can be computed quickly.

4.2 Hierarchical systems

The maximum number of signatures an MTS scheme, which signs messages using an OTS algorithm, can generate over the lifetime of its global key pair is a function of the height of the Merkle tree used. Concretely, a height-h Merkle tree has 2^h leaf nodes, and each leaf node corresponds to an OTS instance, and therefore the scheme can produce at most 2^h signatures. After 2^h signatures have been computed, the global signing key is considered exhausted and cannot be used to securely generate any more signatures. Depending on the application, it can be difficult, and possibly infeasible, to transition to a new global key pair once the current global signing key has been exhausted. Therefore, it can be highly important to understand the number of signatures the system will be expected to generate over the global key pair lifetime prior to selecting the height parameter.

Further, to generate a key pair for such an MTS scheme, the entire Merkle tree is generated. Key generation time and computational cost are exponential in the height of the Merkle tree. If the system is required to generate 2^{40} signatures, then key generation requires generating 2^{40} OTS instances and combining them into the Merkle tree construction. The resources required to generate such a tree are not likely to be available on many platforms, and for some applications the cost of such key generation is prohibitive.

In fact, current specifications for many-time HBS schemes [i.1], [i.2], [i.3] do not define parameter sets for height-40 trees; the largest tree permitted by any of the current specifications is of height 25, which corresponds to approximately 33 million possible signatures. For some applications, 2^{25} is a suitable upper bound for the total number of possible signatures. For some other applications, this upper bound is prohibitively low.

To increase the total number of possible signatures without introducing unreasonable overhead, *hyper-trees* are used. A hyper-tree, often called a *multi-tree,* is a hierarchical structure of binary trees, divided into *layers*. Each layer in a hyper-tree contains some number of disjoint binary trees, where that number equals the total number of leaf nodes among all trees on the layer above.

A 2-layer hyper-tree is a construction where there is a single binary tree of height h on the top layer (layer 1), and 2^h disjoint trees, of some height possibly different from h on the bottom layer (layer 0). Each tree on layer 0 corresponds to a unique leaf index on layer 1. If another layer were added to the bottom of the hierarchy, it would contain one tree for each of the leaves on the layer directly above it. The reader's attention is drawn to the distinction between the use of the terms *layer* and *level* (clause 4.5). The system can generate at most one signature for each leaf node on the bottom layer of the hierarchy. The number of layers in a hyper-tree is denoted as $\mathcal{L}$ in the present document. Figure 3 visualizes a 3-layer hyper-tree.

In the context of MTS schemes, it is worth emphasizing that every single tree in a hyper-tree is itself an instance of a single-tree MTS scheme. That is, the leaf nodes of every tree in the hierarchy correspond to OTS instances, not just those on the bottommost layer. In the 1-layer (single-tree) construction, the leaf nodes are used to sign messages. In the multi-layer construction, only the leaf nodes on the bottommost layer of the hierarchy are used to sign messages. The other leaf nodes, the leaf nodes of trees that are not on the bottom layer, are used to sign the root nodes of the trees they uniquely correspond to on the layer directly below them.

Signature generation requires signing the message with a leaf node of a tree on the bottom layer and computing the authentication path up to the root, R_0, of that tree just as in the 1-layer construction. However, in the hyper-tree setting, R_0 is then interpreted as a message and is signed with the leaf node corresponding to R_0 on the layer directly above R_0. An authentication path is constructed from that leaf (the leaf that signed R_0) to the root of its tree, R_1, and the process is iterated to exhaustion. The resulting signature consists of the collection of one-time signature and authentication path pairs. Similarly, verification is performed iteratively.

The OTS signatures computed on Merkle tree root nodes are referred to by the present document as *intermediate signatures.*

EXAMPLE 1: Suppose that in a 2-layer hyper-tree, the single tree on layer 1 has height 25 and each of the 2^{25} trees on layer 0 have height 15. Then there are $(2^{25})(2^{15}) = 2^{40}$ total leaf nodes on layer 0, and thus, the system is capable of producing at most 2^{40} signatures. If another layer, say with trees of height 10, were added to the bottom of the hierarchy, then that layer would hold 2^{40} disjoint height 10 trees and the full system would be capable of generating up to 2^{50} signatures.

The use of hyper-trees allows an implementation to exponentially increase the total number of possible signatures a MTS scheme can generate over the global key pair lifetime, at the cost of larger signatures and longer signature generation and verification times. Key generation requires only the single tree on the top layer be generated, as the global public key corresponds to the root node of the topmost tree. However, to generate any signatures, further trees are required to be generated. This is because a signature from an MTS scheme consists of multiple authentication path and one-time signature pairs, one pair for each layer in the hierarchy.

EXAMPLE 2: Suppose that a 2-layer hyper-tree is to have the single tree on layer 1 of height 25 and each of the 2^{25} trees on layer 0 have height 15. Then key generation requires only generating the height 25 tree on the top layer. However, a signature consists of two OTS signatures and two authentication paths, one path with 25 nodes and the other with 15, which is notably larger than in the non-hierarchical case. In order to compute the full signature, a tree on layer 0 will have to be generated.

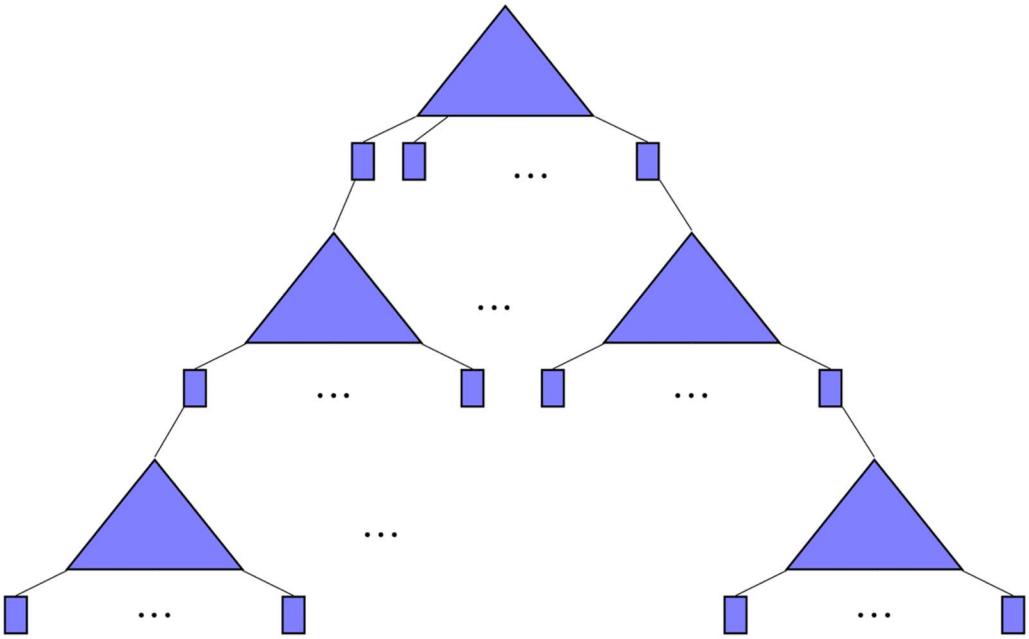

Figure 3: A 3-layer hyper-tree

Suppose that node N is the root node of some height h tree in the hierarchy, where that tree is not the topmost tree. Then N corresponds to the public key of an instance of an MTS scheme embedded in the full hierarchy, and that instance can itself generate up to 2^h signatures. However, in the full hierarchical scheme, N is signed by a leaf node on the layer above it, and this signing is possibly done up to 2^h times, if the signature on N and its corresponding authentication path will be included in the full signature. Therefore, a signature on N is potentially computed many times over the lifetime of the full scheme by the same OTS signing key.

As previously stated, reusing an OTS signing key can break the security of the full scheme. Unless the OTS signing algorithm is deterministic (i.e. each time a fixed message is signed by a given OTS signing key, an identical signature is computed), information about that OTS signing key will be leaked and the system will lose security. If the OTS signing algorithm is deterministic, then the intermediate signatures can be securely recomputed each time, as any information leaked is identical each time and no additional information is leaked, excepting faults or fault-injection attacks (clause 6.1). If the OTS signing algorithm relies on randomization, then repeated signing is equivalent to OTS key reuse.

In the case where the OTS signing algorithm relies on randomization, the intermediate signatures are stored to avoid repeated signing. In the deterministic case, intermediate signatures can be stored or recomputed depending on the trade-offs desired, and compliance obligations (see clause 6.1). As the same leaf is used to sign each time, the corresponding authentication path will not change until the entire lower tree is exhausted and the next tree is used.

4.3 Stateful vs stateless

Signature schemes such as the MTS schemes described above require that no OTS signing key be used to sign more than one message. As per clause 4.1.2, one-time signatures leak partial sensitive data. If an OTS signing key signs more than one message, then forgeries become feasible, and security of the full scheme is broken.

The MTS schemes described above use the OTS signing algorithm to sign messages on the bottommost leaf level. Other many-time hash-based signature schemes, such as SPHINCS+ [i.6], sign the messages with a few-time signature scheme.

In the former style of scheme, re-using an OTS signing key even once breaks security. Therefore, it is critical to the security of such schemes that the used leaf indices be tracked in some way so that they are not reused. In the latter style of scheme, because the FTS key pairs can be used securely more than once, a probabilistic argument yields that the system does not require keeping track of which FTS keys have been used to sign, assuming suitable parameters are used.

Many-time HBS schemes whose security requires keeping track of which OTS instances have been used are called *stateful schemes*. Conversely, many-time HBS schemes whose security does not require keeping track of used indices are called *stateless schemes.* The present document is concerned with Stateful Hash-Based Signature (S-HBS) schemes.

S-HBS schemes that use hyper-trees are referred to by the present document as *hierarchical S-HBS (HS-HBS)* schemes.

The Crypto Forum Research Group (CFRG) has published two RFCs specifying S-HBS schemes, IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2]. IRTF RFC 8391 [i.1] defines the *eXtended Merkle Signature Scheme (XMSS)*, and the corresponding hierarchical variant, the *Multi-Tree eXtended Merkle Signature Scheme (XMSS-MT)*. IRTF RFC 8554 [i.2] defines the *Leighton-Micali Signature (LMS)* scheme, and the corresponding hierarchical variant, the *Hierarchical Signature System (HSS)*.

Importantly, IRTF RFC 8554 [i.2] makes a distinction between a stand-alone LMS instance and a single-layer HSS instance. IRTF RFC 8554 [i.2] restricts the number of layers in a hierarchy, denoted as L in IRTF RFC 8554 [i.2], to a value between 1 and 8 inclusive. The parameter L is defined as an unsigned integer (a 4-byte string) and is included in the HSS global public key. Therefore, the main difference between LMS and HSS with $L = 1$ is the additional 4 bytes in the public key. To maximize interoperability, the authors of IRTF RFC 8554 [i.2] decided that the overhead associated with using $L = 1$ is small enough that all compliant implementations of LMS are required to support it. Therefore, RFC compliant implementations of LMS are technically implementations of single-tree HSS.

4.4 The state index

In S-HBS schemes, one option for keeping track of the used indices is to keep track of all indices that have already been used. However, maintaining such a list is burdensome. A simpler approach is to keep track of the next leaf index to be used, where the leaf nodes are used in the simple sequential ordering described in clause 4.1.4.5. In this way, the indices that have already been used are implicitly tracked by the next unused index. The present document refers to the next unused leaf index as the *state index.*

A leaf index is said to be a *stale state* if it corresponds to an OTS key pair which cannot, or can no longer, be used to securely sign a message. That is, a leaf index that has already been used to sign, or which cannot be securely used to sign for some other reason. Some situations where an unused index can become stale are described throughout clause 6.

In a HS-HBS scheme, each layer in the system has its own state index. That is, there is a state index for the bottom layer (the layer that signs messages), a state index for the first layer (whose leaves sign the root nodes on the bottom layer), and so on up to and including the tree on the topmost layer. Further, for each layer, an implementation has the option of maintaining a state index only for the current tree on that layer, or for maintaining an index for the entire layer. In the former case, the state index can be as large as the number of leaves on any one tree, and in the latter case, the state index can be as large as the total number of leaves among all trees on that layer. There are possibly other ways to maintain state indices as well. Depending on the specific application of the HS-HBS scheme, different methods of keeping track of state can have different advantages and disadvantage. This is discussed further in clause 5.2.1.

4.5 Notational differences between IRTF RFC 8391 and IRTF RFC 8554

There are numerous conflicts of terminology and notation between IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2]. Often, the same notation or terminology is used to describe different things, making it difficult to compare the specifications. The present clause collects and lists these differences and selects unified terminology and notation to use throughout the rest of the present document. Further comparisons between the two RFCs are discussed in clause 8.

NOTE: For the reason given above, notation and terminology used in the present document can be inconsistent with that used the RFC specifications.

Winternitz parameter: In IRTF RFC 8391 [i.1], the Winternitz parameter is defined as the length of a *Winternitz chain* (not discussed in the present document). In IRTF RFC 8554 [i.2], the Winternitz parameter is defined as the number of message digest bits that can be encoded in a single Winternitz chain. A length-16 Winternitz chain can encode $\log_2 16 = 4$ bits of the message digest. Similarly, if w bits are encoded by a Winternitz chain, then the chain has length 2^w. The present document defines w as the total length of a Winternitz chain as in IRTF RFC 8391 [i.1]. For clarity, $w = 16$ in the terminology of IRTF RFC 8391 is equivalent to $w = 4$ in the IRTF RFC 8554 [i.2] terminology. IRTF RFC 8391 [i.1] only defines parameter sets where $w = 16$, and IRTF RFC 8554 [i.2] defines parameter sets with $w = 1, 2, 4, 8$.

Number of Winternitz chains: In IRTF RFC 8391 [i.1], the total number of Winternitz chains in a given W-OTS instance is denoted by *len*. In IRTF RFC 8554, this value is denoted as p. The present document uses the notation len from IRTF RFC 8391 [i.1].

Merkle Tree height: Both of IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2] denote the height of a single Merkle tree by h. The present document also denotes the height of a single Merkle tree (or binary hash tree) as h.

Level: IRTF RFC 8391 [i.1] uses the term *level* to describe a height position within a single Merkle tree. IRTF RFC 8554 [i.2] uses the term to describe the height position of a Merkle tree within a hyper-tree. The present document uses the term as described in IRTF RFC 8391 [i.1].

Layer: IRTF RFC 8391 [i.1] uses the term *layer* to describe a height position of a single Merkle tree within a hyper-tree and denotes the total number of layers as d. IRTF RFC 8554 [i.2] does not use the term explicitly but uses the notation L to denote the total number of layers in a hierarchy. The present document uses the term *layers* as in IRTF RFC 8391 [i.1] but denotes the total number of layers in a hierarchy as $\mathcal{L}$, to avoid a conflict of notation with the i^{th} leaf node of a tree. In both RFCs, all trees in a given layer of a hyper-tree have the same height. It therefore makes sense, and is often convenient, to refer to the *height of a layer* as the height of each Merkle tree within that layer.

Total hierarchy height: IRTF RFC 8391 [i.1] also uses h to denote the total height of a hyper-tree, and IRTF RFC 8554 [i.2] does not define the term explicitly. To avoid a conflict of notation between the hash function and the total height of a hierarchy, the present document uses $\mathcal{H}$ to denote the sum-total height of a hyper-tree structure. Equivalently, $\mathcal{H}$ is the total sum of the heights of each layer in a hyper-tree structure.

EXAMPLE 1: A height h Merkle tree has $h + 1$ levels (where the leaves are indexed on level 0 and the root node is on level h) and consists of a single layer. A two-layer hyper-tree, where both layers are of height h, has total height $\mathcal{H} = 2h$.

EXAMPLE 2: A 2-layer hyper-tree, where the top layer has height h_1 and the bottom layer has height h_2 is typically denoted as an h_1/h_2 scheme, so that $\mathcal{H} = h_1 + h_2$.

Seed: IRTF RFC 8391 [i.1] generally uses the term *seed* to refer to public data that is included in each hash function call (to prevent multitarget preimage attacks), where IRTF RFC 8554 [i.2] uses the public value I to serve that same purpose. Both IRTF RFC 8391 [i.1] and IRTF RFC 8554[i.2] uses the term *seed* to refer to secret data that is used to pseudo randomly derive OTS key pairs. Although IRTF RFC 8391 [i.1] uses the term *seed* to refer to both public and private values, it denotes public seeds as SEED and private seeds as S. The present document simply uses the terms *private seed* and *public seed* to describe the relevant data objects.

5 The state object

5.1 Contents of the state object

The *state object*, as described in the present document, is not explicitly described in the existing S-HBS specifications. Further, the present document defines the state object in a flexible way and does not rigorously prescribe the precise format or contents of the state object. It is useful to think of the state object as a collection of data related to a S-HBS scheme instance that is required to compute or verify signatures from that scheme instance. There are many choices available to an implementor regarding the content, format, and other characteristics of the state object, such as how it can or cannot change over time.

Given the inherent flexibility of the state object, it is unlikely that its representation in one implementation will be interoperable with that of another implementation. The format and content of a signer's state object is not a concern for those performing verification operations (as the verifier need only know the signer's global public key). However, lack of interoperability between different implementations of state objects is potentially an issue for systems performing back-up, restore, or exportation of the state object. Interoperability between state object representations can also be a problem in systems comprised of cryptographic components from multiple vendors, or where signing is done in a distributed way (clause 7.4).

Implementations are not required to define a state object, as that term is defined in the present document. State objects are simply convenient ways to collect, store, and access S-HBS scheme data. Importantly, even if an implementation chooses not to define a state object as such, the implementation will still be required to track and maintain the state index, or possibly multiple state indices in the cases of hierarchical or distributed systems. Therefore, it can be useful to think of the state index (or state indices) as the only strictly necessary component of a state object.

Typical components of the state object are described below.

State index: The state index is the index of the next OTS instance to be used for signing, as discussed in clause 4.4. The state index is the most important component of the state object in terms of security. The value of the state index can be as large as the number of leaves in the Merkle tree the state index corresponds to. Further, in a HS-HBS scheme, there is one current state index per hierarchy layer. Depending on the specific S-HBS scheme used, the maximum value of a state index can be different on different hierarchy layers.

Pre-computed nodes: Full MTS scheme signatures include OTS signatures and their corresponding authentication paths. The authentication path nodes can be calculated before they are required to be included in a signature, as discussed in clause 4.1.4.5. Therefore, an implementation can include current authentication path nodes as well as nodes for future authentication paths within the state object. The entire Merkle tree or hyper-tree can be stored in the state object if desired and if resources are available to do so.

Stored signatures: If a hierarchical scheme is used, intermediate OTS signatures can be stored in the state object. If the OTS scheme used relies on randomization, then it is expected that the intermediate signatures are stored as recomputing and re-releasing them would equate to OTS key reuse. If the OTS scheme is deterministic, the implementations can still choose to store the intermediate signatures as a method to reduce signing time and cost, or as a mitigation to fault-injection attacks [i.10]. An implementation can choose to store OTS signatures with their corresponding authentication paths.

Parameters: Per IRTF RFC 8554 [i.2], LMS public keys include a 32-bit identifier for the LMS parameters, a 32-bit identifier for the LM-OTS parameters, a 128-bit LMS instance Identifier, and an $8n$-bit Merkle tree root node (where the hash function has n-byte outputs). In the HSS signing algorithm, intermediate signatures are computed on LMS public keys and not just on the Merkle tree root nodes; the LMS public keys contain additional data over just the value of the root node. The signed public keys are included with the corresponding signatures (and authentication paths) in the full HSS signature. Therefore, the LMS public keys (including algorithm parameters) are included in the full HSS signature. Per IRTF RFC 8391 [i.1], XMSS signatures do not similarly include algorithm parameters. In either case, an implementation can find it useful to store algorithm parameters separately in the state object, possibly to reduce the cost of recovering the parameters from signatures or for simple convenience.

Long-term public key: The long-term public key of the S-HBS scheme instance can be derivable from other data within the state object such as from authentication path nodes or stored signatures or be publicly available such as in a digital certificate, and so the inclusion of the long-term public key in the state object is a choice of convenience for the implementation.

Long-term private key: The long-term private key of the S-HBS scheme instance can be stored as part of the state object, but as discussed in clause 5.2.3, implementations can find it more useful to consider the long-term private key as distinct from the state object.

NOTE: Some HSS implementations treat the state object as a component of the private key. Generally, in such implementations the state object only contains the state indices. This is discussed further in clause 5.2.3.

5.2 Characteristics of the state object

5.2.1 Size of the state object

As mentioned in clause 5.1, the contents of the state object are at the discretion of the implementor. Therefore, the size of the state object is highly variable. In the following, the data sizes of the potential state object components identified in clause 5.1 are discussed. To facilitate the discussion, parameter sets from IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2] are considered.

State index: The smallest Merkle trees defined by IRTF RFC 8391 [i.1] are of height 10, and the smallest trees defined by IRTF RFC 8554 [i.2] are of height 5. Thus, the state index can theoretically be made as small as 5 or 10 bits depending on the specification being adhered to. However, IRTF RFC 8391 [i.1] defines XMSS-MT parameter sets that allow up to 2^{60} signatures, and IRTF RFC 8554 [i.2] permits up to 8 layers in a hierarchy where each layer can have a height of at most 25, thereby allowing up to 2^{200} possible signatures. Also, of consideration is how the implementation maintains the state indices per layer; this was briefly mentioned in clause 4.4. For smaller parameter sets, regardless of the choice of how state indices are maintained, the state indices can likely be stored in a 32-bit unsigned integer. If an implementation maintains a single state index for an entire layer (as opposed to a state index per individual tree), and if the layer enables more than 2^{32} total signatures, other datatypes can be required to store the state index.

NOTE 1: The HSS parameter set permitted by IRTF RFC 8554 [i.2] that enables up to 2^{200} signatures is a largely theoretical one. Although the parameter set is technically permitted by the RFC, it is not expected that such a parameter set would be used, or usable, in practice.

Pre-computed nodes: The number of nodes in an authentication path equals the height of the tree the path is in. The more layers there are in a hierarchy, the more authentication paths there are. Further, nodes that are not a part of the current authentication path can be pre-computed by a tree traversal algorithm. Using the smallest defined parameter sets of IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2], authentication paths are of length 5 or 10, equating to 160 or 320 bytes respectively, using a 32-byte hash function. Using the largest defined parameter sets, the authentication path nodes total 1 920 or 6 400 bytes respectively, using a 32-byte hash function. Again, additional nodes corresponding to future authentication paths can also be included in the state object.

The full Merkle tree or Merkle tree hierarchy can be stored in the state object as well. Considering again the largest and smallest parameter sets defined by IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2], the full structure is 2 016 or 4 032 bytes respectively at the smallest and $(2^{61} - 1)$x32 or $(2^{201} - 1)$x32 bytes respectively at the largest. As parameter sets grow larger, storing the associated data structures becomes less feasible.

NOTE 2: The current publications on tree traversal algorithms, such as the BDS and Fractal algorithms [i.8], [i.9] do not define mechanisms for storing pre-computed nodes into the state object. Additional mechanisms are required to achieve such behaviour.

Stored signatures: If a HS-HBS scheme is used, then signatures will include intermediate one-time signatures (that are computed on public keys of LMS or XMSS trees internal to the hyper-tree). As discussed in clause 4.2, such signatures can be stored to reduce computational overhead, because the one-time signatures rely on randomization or to mitigate fault-injection attacks (clause 6.1). IRTF RFC 8391 [i.1] defines parameter sets with up to 12 layers, in which case up to 11 intermediate signatures can be stored. IRTF RFC 8554 [i.2] permits up to 8 layers in a hierarchy, in which case up to 7 intermediate signatures can be stored. IRTF RFC 8391 [i.1] defines only a single required OTS parameter set (*WOTSP-SHA2_256*) which yields signatures of approximately 2 144 bytes each; storing 11 such signatures requires 23 584 bytes. The smallest one-time signature defined in IRTF RFC 8554 [i.2] is 2 180 bytes; storing 7 such signatures requires 15 260 bytes. These values do not include the data sizes for the authentication paths.

NOTE 3: IRTF RFC 8391 [i.1] requires that all layers in an XMSS-MT instance use the same public parameters. Specifically, every Merkle tree in an XMSS-MT hierarchy has the same height. IRTF RFC 8554 [i.2] does not have this restriction, and in fact explicitly defines HSS parameter sets where different layers have different heights. This is discussed further in clause 8.

Parameters: As described in clause 5.1, the parameters of each LMS instance used to generate an HSS signature are included in the corresponding LMS public keys. The LMS public keys are included in the HSS signature, and therefore, all LMS parameters are embedded in the HSS signature. XMSS and XMSS-MT signatures do not have a similar property. LMS algorithm parameters include a 4-byte identifier for the LMS parameter set, a 4-byte identifier for the LM-OTS parameter set, and a 16-byte LMS instance identifier, equalling 24 bytes per hierarchy layer, or $24\mathcal{L}$ bytes of total storage if the implementor chooses to explicitly store the HSS parameters.

XMSS algorithm parameters include a 4-byte identifier for the WOTS+ parameter set, a 4-byte identifier for the XMSS parameter set, and a public n-byte seed value (where n equals the length of the hash function output in bytes). However, the XMSS parameter set identifier and public seed are included in the XMSS public key, and all layers in an XMSS-MT hierarchy are required to have the same XMSS parameter sets. An XMSS-MT public key includes an XMSS type identifier, a Merkle tree root, and an XMSS-MT public seed. In practice, each XMSS public seed is derived deterministically from the single XMSS-MT public seed. Therefore, if the implementor chooses to store XMSS-MT parameters, it suffices to store the 4-byte WOTS+ parameter set identifier, a single 4-byte XMSS type identifier, and the n-byte XMSS-MT public seed as part of the state object.

Long-term public key: Using an n-byte hash function, HSS public keys are $(28 + n)$-bytes, and LMS public keys are also $(28 + n)$-bytes when including the additional 4-byte identifier for the number of layers. Both XMSS and XMSS-MT public keys are $(4 + 2n)$-bytes.

Long-term private key: Private key formats are not defined by the RFCs for LMS and XMSS because such definitions are not generally required for interoperability. Further, the sizes of the private keys depend on the methods used to generate them. If a deterministic method is used to derive XMSS private seeds from an XMSS-MT private seed, and to derive WOTS+ private seeds from XMSS private seeds, then only a single n-byte private seed is required in the XMSS-MT private key (or XMSS private key in the single-layer case). HSS private keys can be similarly compressed. Additionally, implementors can choose to include additional data such as the 16-byte LMS identifier I for the topmost tree in an HSS instance (or a seed used to generate each LMS identifier), or any implementation specific metadata with the long-term private key.

NOTE 4: It is conceivable that for systems comprised of components from multiple vendors that interoperability of private key formats can be required. However, for general purposes of signing and verifying, such interoperability is not required.

5.2.2 Format of the state object

As mentioned in clause 5.1, because the state object is not explicitly defined in the existing specifications there are no restrictions on the structure or format of the state object or how it behaves over time. If an implementation changes the structure or format of the state object over time, then the implementation should ensure that the resulting state object can still be properly parsed and handled by the dependent systems.

As discussed in clause 7.1, implementations can in some situations load components of the state object from persistent storage into volatile memory, thereby allowing safer and easier use of state object data.

5.2.3 Sensitivity and access of the state object

Some implementations interpret the state object as a part of the global private key, some consider the global private key to be a component of the state object, and others consider the two as being distinct. The present document interprets all private keys (global or otherwise) to be optional components of the state object. Regardless of the interpretation chosen by the implementation, disclosure of any state object data except for the private keys does not affect the security of the system. However, unauthorized modifications to the state object data can have devastating consequences to system security or operability. If the state index is modified to a stale state, then OTS signing keys can be insecurely reused, resulting in a complete loss of system security. If authentication path nodes or pre-computed nodes are modified, then signature verification can fail, or signature generation can produce malformed signatures (which will also fail verification).

For systems performing signing operations, it should be ensured that their state object is in protected and persistent storage and that any state object data loaded into volatile memory is appropriately synchronized with that in persistent storage; this is discussed further in clause 6.1. For signature verification, the required state object data is included as part of the signature.

6 State index reuse

6.1 Secure state index reuse

In certain situations, reuse of an OTS signing key does not degrade security of the S-HBS scheme. The present clause outlines some such situations.

Deterministic signatures: One-time signatures can be made deterministic by generating signatures from pseudorandom data such as seeds and addresses (specific locations of nodes or Merkle trees) within the (hyper-)tree structure. If an OTS instance is made deterministic, then computing two signatures on the exact same message results in two identical signatures. In this way, the same sensitive data is leaked by each recomputed signature and an attacker is not able to acquire enough information to forge signatures.

Fault hardening: Implementors should be aware of the fault-injection attack [i.10] against hierarchical HBS schemes that use deterministic OTS algorithms. The attack works roughly as follows. Each time an attacker requests a signature, an OTS instance signs the requested message, the authentication path is constructed, and the root node of the Merkle tree is computed. That root node is then signed by a leaf node on the next higher layer. If the lower tree is of height h, then the leaf node above will sign the lower root up to 2^h times. After repeated signature requests it is possible that the lower Merkle tree root is miscomputed-through machine error or some action by the attacker. The faulted root node is different from the authentic root node and is also signed by the higher-layer leaf node. Thus, two distinct signatures will have been generated by the same OTS instance on the higher layer, resulting in a break of the system. The larger the lower Merkle tree is, the more opportunities there are to miscompute the root node.

Therefore, even if OTS signatures are deterministic, the potential remains that a signature will be computed on a faulted message. One possible mitigation is to store signatures on root nodes so that they are never recomputed, which requires extra storage as well as the confidence that those stored values are not modified, either accidentally or maliciously. Another mitigation is to compute each signature twice before releasing it-provided the OTS signatures are deterministic. In this way, if two different signatures are produced the signer should refuse to release the faulted signatures. In other words, if the signature algorithm is deterministic, then a state index can be used multiple times as a check to see if any faults occurred.

Current S-HBS specifications, such as NIST SP 800-208 [i.3] require the state index be updated in storage after the computation of an OTS signature, but before the signature is released. In particular, the Special Publication requires that the state index be incremented (in storage) after the cryptographic module has been given a signing request. Further, the Special Publication specifically disallows signing Merkle tree root nodes more than once, even if the signature scheme is deterministic, as a defence against fault-injection attacks. This is discussed further in clause 6.2. Therefore, use of the OTS recomputation strategy described above results in an implementation not compliant to NIST SP 800-208 [i.3]. The recomputation strategy is included here for completeness, but the simpler and more efficient mitigation strategy against these fault-injection attacks is to store signatures on root nodes and to not ever recompute them.

6.2 Insecure state index reuse

Apart from scenarios such as those described in clause 6.1, reuse of OTS signing keys is a critical security risk in S-HBS systems. Insecure reuse of even a single OTS signing key can completely break the security of the scheme. The present clause non-exhaustively discusses situations wherein an OTS signing key (or state index) can be insecurely reused and suggests mitigations to those scenarios.

NOTE 1: The insecure state index reuse scenarios described in the present clause are not necessarily mutually exclusive to each other. Some of the scenarios described throughout the present clause are closely related but are described separately for the sake of clarity and readability.

NOTE 2: For the sake of convenience, the present clause assumes that the state index is maintained in persistent storage. This assumption does not preclude the possibility that the state index or other state object data also exists in volatile memory.

Unexpected system restart: If the state index is updated in storage after the corresponding stateful signature has been released, there exists a risk of state index reuse if the system reboots for any reason before the update is completed. In such a case, the system can reboot to the previous, now stale, state. If the system is restored to the previous state index, then the next signature generated will insecurely reuse that stale state, resulting in a loss of security.

To mitigate this issue, the updated state index should be written to storage before the signature is released (or even before the signature is generated). System interruption and reboot can occur due to a variety of reasons including power loss or power fluctuations, human intervention, and hardware or software failures. To further mitigate this issue, implementors should implement appropriate safeguards to prevent such system failure and reboot from occurring in the first place.

Scheduled write to storage: There is a delay between when the command to write to storage is given and when the actual write occurs. In particular, some systems schedule a write to storage at a future time instead of immediately performing the write operation. This issue should be considered in tandem with the system reboot issue described above to prevent situations where a state index update has been scheduled and the signature released and the system reboots before the write to storage occurs. Write to storage can be further delayed by things such as caching or interference by other system processes.

This issue becomes more complex as signature throughput increases. If a new signature is generated before the state index is properly updated from the previous signature, then the system is at risk of reusing the previous, now stale, state index.

To mitigate the high-throughput issue, implementors should be aware of the characteristics of the relevant systems and processes and take appropriate action to guarantee the state index has been updated in storage prior to releasing the signature. Solutions to this issue can vary from system to system. However, one possible mitigation to this high-throughput problem is for implementors to reserve a range of state indices (sometimes referred to as a *state reservation strategy* [i.11]) in volatile memory and advance the state index in permanent storage to beyond the end of the reserved range. In this way, the state index in volatile memory is not required to be synchronized to permanent storage, and if the system reboots for any reason, the state index in volatile memory will be discarded, leaving the system to resume signing from the state index stored in permanent storage. However, such a solution is at the possible cost of a number of signatures. That is, the state indices that were reserved but not used will not be able to be securely used in the future.

In applications where signatures can be generated with low frequency, such as by an offline root Certificate Authority (CA), it is possible that procedural controls can be used to track and manage the state index (e.g. by manually writing down the state index and checking it prior to the next signature generation.) However, depending on how they are implemented such controls can result in non-compliance to the relevant S-HBS scheme specifications. For example, NIST SP 800-208 [i.3] requires that the state index be updated in permanent storage after a signature request has been given to the cryptographic module.

State synchronization: As mentioned in clause 5.2.2 as well as above, implementations can in some situations load components of the state object from persistent storage into volatile memory. The system can perform operations on the data in volatile memory (possibly including updating authentication path nodes, updating the state index, and so forth), and synchronize this data with that in persistent storage at a later time. This is the more general case of the high throughput issue discussed above. If the contents of volatile memory are lost for any reason (such as power loss, system reboot, or hardware failure), then the system can be restored to a previous state object, and hence, be susceptible to insecure state index reuse.

To mitigate this issue, a similar approach as above can be taken. The implementor can reserve a range of state indices in volatile memory and advance the state index in permanent storage to beyond that range. However, due to the nature of tree traversal algorithms, it may not be possible or reasonable to advance the precomputed or authentication path nodes in volatile memory to reflect an advanced state index in permanent storage. The costs associated with advancing storage should be taken into consideration. In such a case, the implementor should be aware of the discrepancy between the state index and other data in the state object in permanent storage. Multiple solutions to this state synchronization problem exist, though not all are compliant with the current S-HBS scheme specifications. Further analysis of state synchronization issues and mitigations can be found in [i.11].

Multiple signers: It is possible that in a distributed system, multiple signers can have access to the same S-HBS global private key and can sign independently and possibly at the same time as each other. This further complicates the high-throughput and synchronization issues discussed above. The present document does not propose explicit solutions to this issue, as solutions will be dependent on the specific requirements and desired features of the system.

Back-up and restore: The nature of the state index complicates typical back-up and restore procedures. If a S-HBS private key without state index data is backed-up and then later restored, then the system it is restored to cannot know which state indices have already been used and thus, cannot safely use any index. Further, the entire tree structure of the instance can be regenerated from the private key, and consequently all state indices become at risk of insecure reuse. If the private key is backed-up with the state index, then the restored-to system cannot know which indices were used after the back-up was performed, and consequently the restored-to system is at risk of insecure state reuse again.

Another issue with back-up and restore is that even if the state or private data is encrypted before it is exported from the source machine, once it is restored it is still at risk of insecure state index reuse. This means that an attacker does not need to know the plaintext content of the encrypted data to open the possibility of insecure state index reuse through restore mechanisms. In short, any time the state data is restored, regardless of how it was protected outside of the source machine, there is a possibility of insecure state index reuse.

It is because of the high risk for insecure state index reuse that back-up of S-HBS state or private data is not recommended. The NIST Special Publication on stateful signature schemes [i.3] expressly forbids the export of state or private data from the machine(s) that generate said data and requires that said data be only used on the machines within which they are generated.

It is possible that secure mechanisms for back-up and restore for S-HBS schemes can be developed, or that there are specific contexts or applications wherein secure back-up and restore can be achieved. Such situations and solutions are not explored in the present document. The present document does not suggest mitigations to the issue of insecure state reuse stemming from back-up and restore. However, if an implementor wishes to implement such a system, they should carefully take into consideration the various issues described above.

State cloning: There is a significant risk of insecure state reuse if a S-HBS private key is cloned. This issue is distinct from back-up and restore as discussed above. In such a case, multiple machines or processes can perform signing operations on different messages using the same private data (including seed data), resulting in complete loss of system security. Such cloning can happen, for example, in Virtual Machine (VM) environments that support live cloning.

While it can be possible to mitigate this issue using techniques similar to those described above, the present document recommends that such contexts should be completely avoided. S-HBS scheme instances should not be run in cloneable VMs, or other environments that support live cloning; at least in situations where security is required. For example, if one is simply testing algorithms or investigating proof of concept projects, then usage of VMs can be acceptable. For further analysis of the VM cloning issue, see [i.11].

State object changes: This issue is not unique to S-HBS schemes and was briefly discussed in clause 5.2.3 and is repeated here for the sake of completeness. If unauthorized changes are made to the state object data-intentional or accidental-, such as decrementing the state index or mutating the current authentication path, then the system is at risk of insecure state reuse or of producing malformed or incorrect signatures. Such unauthorized changes can occur anywhere the state object or private key data is stored, including persistent storage and volatile memory.

To mitigate this issue, systems performing signing operations should ensure that the medium where the state object is recorded (persistent storage or volatile memory) is protected and tamper resistant, and that any state object data loaded into volatile memory is appropriately synchronized with that in persistent storage.

6.3 Avoiding and detecting insecure state reuse

The simplest way to avoid insecure state index reuse is to maintain an unambiguous list of which indices have already been used. This is easiest when the S-HBS implementation maintains a single state index, whereby each of the used indices are implicitly tracked by the current state index. In this case, if a request is given for a new signature with a specific index, then the implementation can check the value of that index against the system's current state index. If the requested signature's index is lower than the current state index, then the system should infer that the signature is requesting to reuse a state index, and to not compute the signature.

The simplest way to detect if a given signature has reused a state index is to maintain an unambiguous list of which indices have already been used to sign along with hash digests of the corresponding messages that were signed, and by checking against that list as appropriate. However, this technique can become burdensome to maintain, depending on the system. Methods to detect previous state index reuse are likely not necessary if appropriate steps are taken to prevent state index reuse in the first place. Methods to detect previous state reuse are not discussed in the current S-HBS specifications.

As briefly mentioned in clause 6.2, if the frequency of signature requests is reasonably low, and if the relevant compliance obligations can be met, then procedural controls for state management can possibly be used; such as manually writing down and checking state indices.

Tracking stale state indices becomes more difficult if the implementation maintains multiple state indices; possibly due to provisioning sub-trees, state reservation strategies, multi-component systems such as those described in clause 7.4, the use of hyper-trees, or other reasons. One possible solution for such situations is to perform checking (as described above) against multiple current state indices, or multiple lists of index and message digest pairs. However, such a solution can require additional software and mechanisms not described in the current specifications.

7 Operational considerations

7.1 Storage of the state object

Prospective implementors of S-HBS schemes should have a detailed understanding of the requirements and expected usage of the implementation before selecting the physical systems to run it on. The present clause discusses physical memory and storage considerations for S-HBS scheme that arise from the requirements and expected usage of the state object.

Wearing out memory: Issues arise when the physical system wears out due to frequent re-writes. The expected lifetime and general capabilities of the hardware should be understood and taken into consideration while designing the implementation. If many re-writes are required, then implementors should ensure the underlying hardware and technology can support that number of re-writes at the appropriate speed.

Persistent storage: Another factor to consider is if the implementation requires the current state index to persist after system shutdown or power loss. Implementations will likely require the state index to reside in persistent storage. However, it is conceivable that some (short-lived) implementations will not require the state index to persist, and in the case of an unexpected system restart, such implementation can simply generate a new global key pair.

Volatile memory: A S-HBS implementation can hold the state object in persistent storage, copy the state index and other data into volatile memory, and perform operations on or with the data in volatile memory-including updates to the state index and current authentication path nodes. At some point, the implementation will update the contents of permanent storage using the data in volatile memory. Implementations using this approach can allow multiple state index updates in volatile memory before synchronizing with the permanent storage. If the system shuts down for any reason and the contents of the volatile memory are lost, state reuse can be prevented by advancing the state index in permanent storage by one more than however much was provisioned to the volatile memory. This technique mitigates the state and authentication path synchronization issues outlined above, at the potential cost of some signatures. If this technique is used, implementors should again ensure that the underlying hardware and technology are suitable to support it. This technique was discussed in clause 6.2.

Storage capacity and buffer allocation: Factors such as how the state object is constructed and maintained, or how much state object data is loaded into volatile memory and how that data is processed and synchronized, influence the amount of storage required by the supporting hardware. In general, implementors should be aware of the expected data sizes, buffer allocation requirements, read/write speeds, and other characteristics of the storage mediums.

It is worth noting that large amounts of memory and storage are increasingly more affordable. Consequently, issues surrounding lack of memory or storage space, and other resource constraints, are more easily overcome than the other issues identified throughout clauses 6 and 7. It is in general more important that durable mediums and appropriate technologies are selected to avoid situations as described above where the memory or storage wears or fails out from use.

7.2 Number of signatures generated

Prospective implementors of S-HBS schemes should understand how many signatures are expected to be generated over the lifetime of the global S-HBS key pair. If the implementor underestimates the number of signatures, then at some point they may have to renew the global key pair to make more signatures available to them. Key renewal can be a difficult and expensive process. If the implementor overestimates the number of signatures, then the implementation will use larger parameter sets and require more resources to operate than is strictly necessary. By better estimating the number of signatures, the implementor can select more suitable parameters and make better decisions about resource allocation.

Parameter selection for HBS schemes can be difficult due to the large variety of approved parameter sets and lack of clarity in the trade-offs between different sets. The present document does not provide explicit guidance on HBS parameter selection (in fact, there are currently no publications known that do provide such guidance), but rather gives high-level guidance on possible trade-offs between different parameter sets (and tree traversal algorithm selection) and various considerations to make when selecting parameters.

7.3 Compatibility with existing APIs

A state index (and state object) is not traditionally a typical component of a cryptographic scheme. Consequently, cryptographic Application Programming Interfaces (APIs) do not in general understand how to accept or process state data. Thus, S-HBS schemes may not be compatible with current APIs. This lack of compatibility inherently limits the scope of applications S-HBS schemes can currently be used for. Future versions of current APIs may be able to accommodate state data. In general, an implementor should be aware of the APIs the system is expected to interface with, and of how state data operates with said APIs prior to implementation.

7.4 Multi-component systems

Implementations of S-HBS schemes can utilize components (both hardware and software) from multiple vendors and suppliers. In such implementations, it is possible that issues of interoperability will arise between separate components. As briefly mentioned in clause 5.2.1, the current HBS specifications do not define private key formats, and thereby require the implementor to define their own private key formats. Other factors of an HBS implementation, such as the state object, can similarly be defined proprietarily.

One possible example of an implementation requiring interoperability across separate components is suggested in section 7 of NIST SP 800-208 [i.3]. The document describes, at a high-level, implementations of HSS or XMSS-MT comprised of multiple cryptographic modules.

If an implementor chooses to implement a system such as those described in section 7 of [i.3], they should be aware of possible interoperability problems between the constituent devices. Again, such an implementation can require additional software and mechanisms not defined in NIST SP 800-208 [i.3] or in IRTF RFC 8391 [i.1] or IRTF RFC 8554 [i.2].

8 Comparisons between HSS and XMSS-MT

8.1 Performance comparison

In general, HSS performs faster than XMSS-MT, but has larger signature and key sizes. However, this is not always the case. For each scheme, the time it takes for key generation, signing, or verification depends on many factors. Similarly, data sizes also vary. Some of these factors are as follows:

- choice of underlying OTS algorithm (see note 1);
- choice of OTS parameters;
- choice of tree traversal algorithm;
- choice of hash function;
- total height of the tree hierarchy;
- use of state reservation strategies;
- use of hardware acceleration;
- use of distributed computing; and
- other physical and technical characteristics of the supporting hardware.

NOTE 1: There exist variants of the Winternitz OTS scheme other than those described in IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2]. Use of such alternatives can impact security and performance characteristics of the corresponding MTS scheme.

NOTE 2: The above list of factors is not assumed to be exhaustive.

Therefore, the main differences in the performance characteristics between HSS and XMSS-MT come from the choice of parameters, supporting algorithms (e.g. tree traversal strategy or storage techniques), and the underlying hardware and environment, more so than the technical differences between the algorithms themselves.

Consider the HSS profiles described in NIST SP 800-208 [i.3]. By considering the number of choices of LM-OTS parameter sets, LMS parameter sets, number of layers, and choice of hash function, NIST SP 800-208 [i.3] permits over 107 billion different HSS parameter sets.

There are different ways to compare parameter sets. For example, a 10/5 scheme has the same signature size as a 5/10 scheme but has different performance characteristics (see example 2 of clause 4.5). Most of the computation for LMS is in the computation of the LM-OTS signature, where the choice of Winternitz parameter determines that space-time trade-off. Again, the present document does not give explicit recommendations on parameter set selection.

Another factor which potentially impacts performance characteristics between XMSS-MT and HSS is the fact that HSS allows different LMS (and LM-OTS) parameter sets to be used in each layer of the hierarchy. This includes tree heights.

For example, key generation (in general) only requires that the topmost tree in the hierarchy be generated. The larger the tree, the more resources are required to generate the global public key. Similarly, the smaller the tree, the less resource intensive key generation becomes. Therefore, HSS implementations can achieve trade-offs by using a larger tree at the top of the hierarchy, using smaller trees on the lower layer(s), and storing any intermediate signatures (so they only need to be computed once). Observe that the authentication path of the topmost tree is larger than that of the lower trees in this case. The top authentication path can be stored, and only needs to be updated whenever a tree on the lower level is exhausted. This results in a longer and more computationally expensive key generation time but allows for faster and less resource intensive signature generation. In an XMSS-MT instance, all trees in the hierarchy have the same height, making this trade-off impossible. However, XMSS-MT does have a storage trade-off over HSS as the XMSS parameters for each layer are identical and stored once in the XMSS-MT global key pair. For HSS, these layer parameters are stored in each LMS public key, potentially increasing the amount of storage required over XMSS-MT.

8.2 Security comparison

Annex C.5 of NIST SP 800-208 [i.3] provides a comparison of the cryptographic security believed to be provided by LMS and XMSS, and observes that (under commonly used cryptographic assumptions) both schemes are shown to be existentially unforgeable under chosen message attacks even against an attacker in possession of a large-scale quantum computer.

Therefore, both schemes can be considered theoretically secure, and their concrete security-assuming a secure implementation, proper state management, and so on-relies on the security of the specific parameter sets utilized, which in turn is based on the security of the underlying hash function. Importantly, NIST SP 800-208 [i.3] requires that the same hash function be used in all layers of an HSS hierarchy (different layers can use different LMS parameter sets), and that each LMS instance in a given HSS layer use the same parameter set and hash function (including LM-OTS parameter set and hash function). A similar requirement is given in NIST SP 800-208 [i.3] for XMSS-MT, but XMSS-MT has the additional restriction that each layer in an XMSS-MT hierarchy (i.e. each XMSS instance in the system) use the same parameter set and hash function. Further, NIST SP 800-208 [i.3] permits the same four hash functions for the IRTF RFC 8391 [i.1] schemes as it does the IRTF RFC 8554 [i.2] schemes. Namely, SHA-256/192, SHA-256, SHAKE256/192, and SHAKE256/256. Therefore, the security of either scheme is almost entirely based on a single choice of one of these hash functions.

It is important to note that the security assumptions made in IRTF RFC 8391 are different from those assumed in IRTF RFC 8554 [i.2]. For example, the different schemes make different assumptions about the properties of the underlying hash function. Moreover, even though both schemes are proven to be existentially unforgeable under chosen message attacks, the two security proofs are demonstrated in somewhat different models. Consequently, the security guarantees of LMS are more conservative than XMSS (that is, LMS has stronger security assumptions than XMSS), but XMSS enjoys a tighter security reduction. The merits of the different proof models are not discussed here, but further investigations and discussions of the theoretical security of these S-HBS schemes can be found, for example, in [i.4], [i.12], [i.13], [i.14] and [i.15].

Therefore, the present document does not assume that the same hash function provides precisely the same security for LMS as it does XMSS. However, the concrete security difference is not believed to be substantial. Further discussion can be found in Annex C.5 of NIST SP 800-208 [i.3].

8.3 Selecting a S-HBS scheme

As discussed in clause 8.2, the security of IRTF RFC 8391 [i.1] and IRTF RFC 8554 [i.2] are approximately the same. Therefore, the decision of which scheme to employ for a given application is largely based on other factors such as performance requirements, functionality requirements, other security considerations, and the need for FIPS validation or conformance to NIST SP 800-208 [i.3]. The present clause gives high-level guidance on S-HBS scheme selection.

Performance requirements: In terms of differences in performance characteristics, XMSS-MT is generally more computationally expensive than HSS as it requires approximately three times the amount of hash function calls per OTS signature generation (this number can be greater or less depending on the parameters and any optimizations used [i.16]). Again, HSS signatures and key sizes are slightly larger than those for XMSS-MT. These differences can be mitigated somewhat through careful consideration of the items given in clause 8.1, but if hash function calls are a bottleneck for the implementation's application, then the implementor should consider the less resource intensive HSS. However, if storage and bandwidth requirements are a severely limiting factor for the application, XMSS-MT may be the more suitable choice of signature scheme. Finally, different implementors value the different security reductions (and the models the reductions are demonstrated in) differently, and this preference can influence the choice of S-HBS scheme. For example, some implementors prefer XMSS due to its tighter proof of security.

Functionality requirements: All the signature schemes discussed in the present document have the usual, and expected, functionality of key generation, signature generation, and signature verification. However, there are some slight differences in the overall capabilities and flexibility of the different schemes which can impact their suitability for certain applications. For instance, there is much greater flexibility in parameter selection for HSS than there is for XMSS-MT. As previously mentioned, HSS allows for different LMS parameters to be used on different layers of the HSS hierarchy, whereas XMSS-MT is restricted to the same parameter set across all hierarchy layers, which can lead to certain trade-offs. However, performance trade-offs (in terms of speed and storage costs) are not the only potential benefit of using different parameter sets across different layers.

For example, suppose that in a two-layer HSS hierarchy a large tree is used on the top layer and small trees populate the bottom layer. Now suppose that each bottom-layer tree corresponds to a different identity, such as for a user or device. In such a case, the number of signatures made available to an identity can be selected to approximate the number of signatures they require over the instance lifetime more closely than with XMSS-MT. Thus, fewer signatures and resources are wasted, and resource allocation can be made more efficient than if uniform parameters are required across the hierarchy. Similar situations can be conceived where the bottom-layer trees are considered ephemeral or short-lived.

Difficulty of implementation: It is not obvious which of XMSS-MT or HSS is easier to implement. Again, the XMSS-MT specification IRTF RFC 8391 [i.1] requires all XMSS instances to use identical parameters, and the HSS specification IRTF RFC 8554 [i.2] allows different LMS parameters per HSS layer. Leveraging this flexibility of HSS potentially complicates the code and makes the implementation more difficult, whereas the inherent rigidity in the XMSS-MT specification can induce an easier implementation. However, the LMS algorithms themselves have fewer internal complexities than the XMSS algorithms, such as the use of bitmasks and L-trees. The difficulty of the implementation is likely more impacted by the intended application and target platforms, the secure management of state, and other design choices than by differences between the S-HBS schemes themselves.

Other security considerations: As with any cryptographic system, the security provided by the parameter sets is only one of many aspects of security that should be considered. Other considerations include operational and managerial security, such as the security of the physical environment in which the S-HBS system will operate, including access restrictions and permissions enabling modification or changes. For S-HBS systems, secure management of state is of paramount concern. These general security considerations are highly similar between HSS and XMSS-MT.

Conformance requirements: Section 7 of NIST SP 800-208 [i.3] gives high-level descriptions of how HSS or XMSS-MT can be implemented in a distributed manner across multiple Hardware Security Modules (HSMs). Distributed implementations of stateful signature schemes are attractive as they enable increased functionality and flexibility in the scheme design. To achieve such an implementation of HSS, the instances of LMS within the HSS hierarchy do not require any modifications from the specification in IRTF RFC 8554 [i.2]. However, the same is not true for a distributed implementation of XMSS-MT. To implement XMSS-MT across multiple HSMs, modified XMSS key and signature generation algorithms are required-the modified algorithms are described in section 7.2.1 of NIST SP 800-208 [i.3]. The NIST Special Publication gives high-level guidance on how to implement such distributed schemes. However, NIST SP 800-208 [i.3] observes that external mechanisms are required for such distributed implementations (for both HSS and XMSS-MT) and gives limited guidance on how to design or implement such mechanisms. Further, the external mechanisms required are potentially different for HSS than they are for XMSS-MT. Therefore, the present document does not give specific guidance on how to implement S-HBS schemes in a NIST-compliant, distributed fashion. Rather, if a prospective implementor requires a distributed implementation of a S-HBS scheme for their application, they should investigate which scheme and external mechanisms are most appropriate for their needs.

As discussed in clause 6.2, implementations can choose to reserve a range of state indices, maintain a separate state in volatile memory, and maintain a state index in permanent storage that is advanced past the range reserved for volatile memory. In such a setting, after each new signature is generated the state index in volatile memory is incremented. Importantly, the state index in permanent storage is not incremented after each signature is generated. Such state reservation strategies can be used to achieve certain performance optimizations. Unfortunately, such techniques appear to be non-conformant with the NIST Special Publication. Section 8.1 of NIST SP 800-208 [i.3] specifically requires that the index in nonvolatile storage be incremented before the signature can be exported. However, NIST is aware of these techniques, and have expressed agreement that it is compliant with the intention, if not the letter, of NIST SP 800-208 [i.3] and intend to work with the validation labs to permit this technique. Implementations wishing to take advantage of a state reservation strategy should investigate if their selected strategy prevents compliance with NIST SP 800-208 [i.3], when compliance with the NIST Special Publication is necessary.

9 Applications of S-HBS schemes

9.1 NIST intended applications for S-HBS schemes

NIST is in the process of selecting and standardizing a suite of post quantum cryptographic algorithms for public-key encryption, key encapsulation, and digital signatures through the NIST PQC Standardization Process [i.17]. Stateful signature algorithms were purposely excluded from the NIST PQC Standardization Process, as they were already well-understood and under development by the CFRG at the time NIST published the Call for Proposals. Further, the digital signature algorithms standardized through this process are expected to be suitable for a broader range of applications than are S-HBS schemes; largely because those algorithms do not have the problems of managing state.

The profiles of S-HBS schemes defined in NIST SP 800-208 [i.3] are considered appropriate for use by the US Federal Government. Section 1.1 of NIST SP 800-208 [i.3] gives three characteristics of applications that are intended for S-HBS schemes. These characteristics are as follows:

- it is necessary to implement a digital signature scheme in the near future;
- the implementation will have a long lifetime; and
- it would not be practical to transition to a different digital signature scheme once the implementation has been deployed.

NOTE: NIST SP 800-208 [i.3] makes no assertions that it is sufficient for an application to have the three characteristics listed above for a S-HBS scheme profile to be used in that application by the US Federal Government. Rather, section 1.1 of [i.3] simply gives general, non-prescriptive, guidance on the types of applications that NIST believes S-HBS schemes are appropriate for (for use by the US Federal Government).

9.2 Additional applications for S-HBS schemes

For applications not for use by the US Federal Government, or more generally, for applications that do not require FIPS 140-2 [i.18] or FIPS 140-3 [i.19] validation, greater flexibility is available for designing and implementing the system; potentially at the cost of increased security risks stemming from state management issues. Such implementations do not necessarily conform to the profiles described in NIST SP 800-208 [i.3].

Implementations can introduce mechanisms or techniques which are either not defined in the current specifications or which deviate from the current specifications. One potential reason for introducing such changes is to enable the system to be used for an application it would not otherwise be suitable for. Although such modifications are possible, the implementor should be cautious in deploying them, and consider if other signature solutions are more appropriate to implement instead. Whenever such changes are introduced, there is a possibility of creating new security or operational vulnerabilities which have not been previously accounted for in the security analyses.

Therefore, if it is determined that additional mechanisms or deviations from the specifications are required to enable a S-HBS solution to be used in a given application, the security and operational consequences of those changes should be carefully analysed and considered prior to implementation.

The present clause gives additional characteristics of applications over those listed in clause 9.1 which can be taken into consideration when deciding if a S-HBS scheme is an appropriate and suitable solution for said applications. These additional characteristics are as follows:

- if the number of signatures required over the global key pair lifetime can be reasonably estimated;
- if the frequency of signature generation required is reasonably low;
- if the implementation requires deviation from the specification(s); and
- if the implementation requires the use of additional mechanisms not defined in the specification(s).

NOTE 1: The above list of additional characteristics is not assumed to be exhaustive.

NOTE 2: The present document does not encourage that implementations of S-HBS schemes deviate from, or introduce mechanisms not defined in, the relevant specifications. However, the present document does acknowledge that implementors (especially those not beholden to certain compliance or audit requirements) have the freedom to make their own design decisions. As such, some of the dangers and consequences of deviating from the relevant specifications are addressed herein.

9.3 Suitable applications

9.3.1 Applications conformable to NIST SP 800-208

NIST SP 800-208 [i.3] describes basic characteristic of applications for which S-HBS schemes are primarily intended (see clause 8.1). The present clause gives examples of applications which appear to meet those basic characteristics, and which appear to either conform to, or can be reasonably made to conform to, the requirements of the NIST Special Publication. It is important to recall that NIST SP 800-208 [i.3] does not explicitly approve any specific application for S-HBS schemes, rather, NIST SP 800-208 [i.3] profiles specific cryptographic algorithms which are appropriate for use by the US Federal Government.

For constrained devices or devices that are expected to have long in-field lives, firmware updates can be required over time. For such devices, it is not always practical or feasible to change the public keys embedded within them or alter the code for signature verification after the devices have been deployed. It is expected that the ongoing NIST PQC process will result in the standardization of signature schemes that are more appropriate than S-HBS schemes for certain applications. However, for some deployments, waiting for those standards to appear is not a viable option. Section 1.1 of NIST SP 800-208 [i.3] explicitly mentions this application and describes it as potentially suitable for the S-HBS schemes profiled therein.

An application related to the one above, which appears to meet the requirements of NIST SP 800-208 [i.3], is image signing for secure device boot; in particular, where signing takes place on secure cryptographic modules (meeting the requirements of NIST SP 800-208 [i.3]), and verification takes place on the devices to be booted. A 2021 analysis by Kampanakis et. al. [i.20] investigated the usage of HSS and SPHINCS+ for such applications. Their analysis concluded that the impact to signature verifiers from adopting HBS schemes for these applications is insignificant when compared to the RSA algorithms conventionally used today. Further, the authors of [i.20] concluded that the impact to signers is greater, but still acceptable.

It is worth highlighting that the parameter sets proposed in [i.20] for HSS are consistent with those approved in NIST SP 800-208 [i.3], but that the parameters proposed for the stateless scheme SPHINCS+ are not necessarily consistent with those proposed for the SPHINCS+ NIST PQC Process Round 3 Alternate Candidate [i.6].

A further potential benefit of using S-HBS schemes in this application, as discussed in section 6 of [i.20], is that they can be used alongside classical signature algorithms to facilitate a backwards-compatible migration to post-quantum algorithms.

One more example of a suitable application for a S-HBS scheme, which appears to be conformable to NIST SP 800-208 [i.3], is that of certificate signing and issuance by a Certificate Authority; specifically, a root CA or an intermediary CA who issues certificates with manageably low frequency. In fact, this application is perhaps the one for which S-HBS solutions are thought to be most suitable, because the secure management of state is more easily achieved in such environments.

S-HBS solutions are not appropriate for use in all end-entity certificates (see clause 9.4), and therefore, end-entities will likely require different post-quantum solutions. There is some concern about the practicality of maintaining multiple post-quantum algorithms within the same PKI chain (e.g. one for the root CA, and another for end-entities or intermediary CAs). Again, the NIST PQC Standardization Process is not yet concluded, and there are no other standardized post-quantum signature solutions besides those described in NIST SP 800-208 [i.3]. As mentioned above, for some applications, a post-quantum solution is needed before the publication of the final NIST standards. Therefore, by using a NIST-approved S-HBS solution, CAs can introduce post-quantum security into their PKIs now and use a phased approach over time to upgrade the rest of the PKI to use other post-quantum solutions, as they become available.

NOTE: The present document does not make any guarantees that the applications described in the present clause meet all requirements of NIST SP 800-208 [i.3], nor that NIST will approve S-HBS profiles for those applications.

9.3.2 Applications not conformable to NIST SP 800-208

For some applications, the use of a S-HBS scheme can be technically feasible and favourable, but not in a way which either wholly complies with the requirements of NIST SP 800-208 [i.3] or that meets the three general criteria described in section 1.1 of NIST SP 800-208 [i.3] (see clause 9.1). There are many possible reasons for this, including a need to export state data, the use of parameters or hash functions not approved by NIST SP 800-208 [i.3], the increased (but manageable) difficulty of securely managing state data, the ability to wait for the conclusion of the NIST PQC process, and so on.

One potentially suitable application for a S-HBS solution is in PKI-enabled smart cards. While research has shown the feasibility of performing S-HBS key generation on constrained devices such as smart cards-as in the 2018 analysis by Hülsing et. al. [i.21]-, on-card key generation does not meet the requirements of NIST SP 800-208 [i.3]. However, on-card key generation is not necessary in the applications where digital certificates are stored on the smart cards. Notably, stateful hash-based signatures are not suitable for use in all smart cards.

The PKI-enabled smart card example is one which appears to be able to meet the conformance requirements of NIST SP 800-208 [i.3] but does not meet the three criteria described earlier, namely, that it is relatively simple to transition to a different signature algorithm in the future (the third of the three criteria). However, it is arguable that a post-quantum signature solution does need to be deployed in the near future for this application (the first of the three criteria), and that hybrid solutions can be used in the short-term to help transition the PKI to a fully post-quantum state. S-HBS solutions appear to be a suitable candidate for hybrid deployment in that context. Further, smart cards can be vulnerable to fault-injection attacks, so, there is an increased risk of insecure state index reuse if one uses S-HBS schemes on smart cards. Although such implementations appear to have practical utility, implementors should exercise extreme caution to avoid or mitigate insecure state index reuse issues. If the signatures are deterministic, then one possible mitigation against faults is to run the verification algorithm twice (as the probability of two successive faults is very low). Notably, this mitigation is at the cost of more time and resources.

There has been little published research on side-channel attacks against HBS schemes. Hash-based signature schemes are often described as being "naturally resistant to side-channel attacks" (see for example [i.1] and [i.2]), and the 2018 cryptanalytic results of Kannwischer et. al. appears to support that claim [i.22]. However, further investigation is warranted. In general, side-channel resistance is a topic that needs further analysis for many of the candidate post-quantum algorithms.

9.4 Non-suitable applications

The applications described in the present clause are believed to be non-suitable for the S-HBS schemes described in NIST SP 800-208 [i.3]. It is conceivable that by modifying the algorithms, S-HBS schemes can be made suitable for some applications they would otherwise be non-suitable for, and the present document does not preclude such possibilities. However, it is believed that the trade-offs and efforts required to enable S-HBS schemes in the applications discussed in the present clause outweigh the benefits of using S-HBS schemes in said applications. Stateful hash-based signature schemes are not suitable drop-in replacements for all applications, and so it is to be expected that there are more non-suitable applications than suitable ones for S-HBS schemes.

An application can be considered non-suitable for a S-HBS solution for a variety of reasons. Possible reasons include excessive difficulty of state management, prohibitive performance characteristics (e.g. signatures being too large, signing or key generation taking too long, etc.), or situations where the S-HBS solutions are outperformed by alternative choices.

Clause 6.2 discussed issues surrounding the secure management of state due to a high frequency of signing requests. As signing requests become more frequent, it becomes more difficult to update and synchronize the state object data. Another problem with high-frequency signing environments is that the global key pairs become exhausted more quickly (as compared to in low-frequency environments). The lifetime of a S-HBS key pair is dependent on the number of messages it is expected to be used to sign. This is as opposed to, say, RSA key pairs which can effectively sign an unbounded number of messages during their expected lifetime (i.e. the time between the *Not Before* and *Not After* dates in a digital certificate). Again, while it may be possible to mitigate these issues, S-HBS solutions are not recommended in high-frequency, or high-throughput, environments in general.

An example of such an environment is a Transport Layer Security (TLS) server. During the Handshake portion of the TLS session negotiation, the server will sign some challenge data and send the resulting signature to the client, who then verifies the signature using the information in the digital certificate of the server. A TLS server potentially facilitates many connections over a short period of time (e.g. over a minute or an hour), and hence, premature S-HBS key exhaustion can become a concern.

The above example is highly simplified-it considers only a single TLS server-, and omits discussion of things such as load balancers, server clusters, and other networking devices or middleware software. In a real-world setting, these other components of the network infrastructure add complexity to the operation of the signature scheme and TLS protocol. These complexities can make compliance to NIST SP 800-208 [i.3] more difficult. S-HBS signing keys are not allowed to be shared among other servers or the other networking infrastructure components if the system is to be compliant with NIST SP 800-208 [i.3]. Therefore, issues such as maintaining resiliency and redundancy arise. Further, to be compliant with NIST SP 800-208 [i.3], the message signing (and key generation) is done in a secure cryptographic module operating in specific environments, and a public internet-facing TLS server likely does not meet those requirements, nor can keys be generated in a separate environment and injected into the server. These issues raise further operational problems, such as how to facilitate the signing requests and resultant signatures securely between the server and cryptographic module and dealing with any latency issues this can induce.

NIST SP 800-208 prescribes a relatively strict environment for generating S-HBS keys and signatures, namely in FIPS 140-2 [i.18] or FIPS 140-3 [i.19] Level 3 or higher validated cryptographic modules where the operating environment is non-modifiable or limited, with no bypass mode enabled (see section 8.1 of NIST SP 800-208 [i.3]). Therefore, for example, a FIPS 140-2 Level 2 validated cryptographic module, in a limited operating environment, does not meet the requirements of NIST SP 800-208 [i.3]. Therefore, it is possible that there are applications which are non-compliant with NIST SP 800-208 [i.3], but which are nearly compliant (for lack of a better term).

Consider the example of document signing on a user device, such as a laptop or smart phone, specifically where the signature is generated on the device. The user's device probably does not meet the cryptographic module requirements of NIST SP 800-208 [i.3]. Moreover, this application does not seem to meet all three general application characteristics described in NIST SP 800-208 [i.3].

The user document signing application is non-suitable for S-HBS schemes for other reasons as well. Key generation for S-HBS schemes can be resource intensive and take a relatively long time. If the application does not require compliance with NIST SP 800-208 [i.3], and if it is acceptable to use small Merkle trees in the application (small enough that keys can be reasonably generated on the user's device), then the application is possibly suitable technically for a S-HBS scheme. However, such applications are uncommon. Further, even in such an application, state management on a user device is difficult. Consequently, third parties might have reasonable doubts as to the security of the user's state management program and refuse to engage with the user.

As previously stated, S-HBS schemes are not suitable drop-in replacements for all applications. As such, many applications for digital signatures are likely not appropriate for stateful-hash based signature schemes. The examples given above serve to highlight some of the non-suitable applications for S-HBSs, but it is not practical to construct an exhaustive list of all such applications. In general, applications for which candidate NIST PQC algorithms are expected to be superior solutions, and for which deploying a solution can wait until NIST has concluded their standardization process, are considered non-suitable for S-HBS solutions by the present document. Similarly, applications for which the signing mechanisms are not adequately protected from things such as faults, tampering, unauthorized access, and so on are not recommended. Finally, applications where the state index cannot be reasonably managed are not recommended for S-HBS solutions.

History

Document history		
V1.1.1	November 2021	Publication