

# Notes on hash-based signatures

arnaucube

December 2025

## Abstract

Notes taken while studying hash-based signatures.

Usually while reading books and papers I take handwritten notes in a notebook, this document contains some of them re-written to *LaTeX*.

## Contents

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>1</b> | <b>Preliminaries</b>                  | <b>1</b>  |
| 1.1      | Lamport signatures (1979) . . . . .   | 1         |
| 1.2      | Adding Merkle Trees (1979) . . . . .  | 2         |
| 1.3      | Goldreich's approach (1986) . . . . . | 3         |
| 1.4      | WOTS+ . . . . .                       | 3         |
| 1.5      | HORST . . . . .                       | 4         |
| <b>2</b> | <b>SPHINCS (2015)</b>                 | <b>5</b>  |
| 2.1      | High level view . . . . .             | 5         |
| 2.2      | SPHINCS construction . . . . .        | 6         |
| 2.2.1    | Parameters . . . . .                  | 6         |
| 2.2.2    | key generation: . . . . .             | 7         |
| 2.2.3    | Signature algorithm . . . . .         | 8         |
| 2.2.4    | Verification . . . . .                | 9         |
| 2.2.5    | Sizes of SPHINCS-256 . . . . .        | 10        |
| <b>3</b> | <b>XMSS (2011)</b>                    | <b>10</b> |

## 1 Preliminaries

### 1.1 Lamport signatures (1979)

(OTS: One Time Signature scheme)

**key gen** sk:

$$((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \dots, (s_{255,0}, s_{255,1}))$$

where each  $s_{i,j} \in \{0, 2^{256} - 1\}$ .

pk:

$$((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), (h(s_{2,0}), h(s_{2,1})), \dots, (h(s_{255,0}), h(s_{255,1})))$$

**sign** hash the 256 bits of the msg:  $(m_0, \dots, m_{255})$

and the signature is

$$(s_{0,m_0}, s_{1,m_1}, s_{2,m_2}, \dots, s_{255,m_{255}})$$

**verify** compare hashes of signature components to elements of the public key.

Sizes: 16 kB sk and pk, 8 kB signature.

## 1.2 Adding Merkle Trees (1979)

idea: put a binary hash tree on top of all pk's: leaves are the hashes of pks.

Public key: root of the tree (256 bits)

Signature: OTS plus the authentication path (merkle proofs)

**some numbers** Fix to  $2^{32}$  signatures.

- key gen: compute the tree:  $2^{33} - 1$  hashes
- pk size: 32 bytes
- sk: the seed for the one-time signature (OTS) secret keys: 32 bytes
- signature size:  $\approx 25$  kB:
  - 8 kB Lamport signature
  - 16 kB Lamport pk
  - $32 \cdot 32 = 1024$  B of authentication path (merkle proofs)
  - 4 B for the index of the leaf node

Main issue: have to remember the state, to update the secret key after each signing.

### 1.3 Goldreich's approach (1986)

- make the tree huge (eg. height  $h=256$ ), such that one can pick leaves at random
- each node corresponds to an OTS key pair
- leaf nodes are used to sign messages
- all OTS secret keys are generated from a seed

Numbers

- pk and sk are still small, 32 bytes
- key gen is fast (only generate root OTS key pair)
- signing requires  $2h=512$  OTS key generations and  $h=256$  OTS signatures
- signature is very large, eg. with Lamport OTS:
  - $256 \cdot 24$  kB for Lamport signatures and pks
  - $256 \cdot 32$  B for authentication paths
  - 32 B for the index of the leaf node

total size: 6 MB

### 1.4 WOTS+

Winternitz One-Time Signature.

**Chaining function**  $c^i(x, r)$

input values:  $x \in \{0, 1\}^n$ , iteration counter:  $i \in \mathbb{N}$ ,  
bitmasks:  $r = (r_1, \dots, r_j) \in \{0, 1\}^{n \times j}$ , where  $j \geq i$ .

- if  $i = 0$ , return  $x$ ; ie.  $c^0(x, r) = x$
- if  $i > 0$ ,

$$c^i(x, r) = F(c^{i-1}(x, r) \oplus r_i)$$

(where  $\oplus$  is the bitwise XOR)

**key generation**

$$sk, pk \leftarrow WOTS.keygen(S, r)$$

input seed:  $S \in \{0, 1\}^n$

bitmask:  $r \in \{0, 1\}^{n \times (w-1)}$

compute sk:

( $G_i$  is a pseudorandom generator)

$$sk = (sk_1, \dots, sk_l) \leftarrow G_l(S)$$

ie. the  $n$  bit seed is expanded to  $l$  values of  $n$  bits

$$pk = (pk_1, \dots, pk_l) = (c^{w-1}(sk_1, r), \dots, c^{w-1}(sk_l, r))$$

**signature:**  $\sigma \leftarrow WOTS.sign(M, S, r)$

for a message  $M \in \{0,1\}^n$

compute a base- $w$  representation of  $M$ :

$$M = (M_1, \dots, M_l), \quad M_i \in \{0, \dots, w-1\}$$

compute checksum:

$$C = \sum_{i=1}^{l_1} (w-1-M_i)$$

and its base  $w$  representation

$$C = (C_1, \dots, C_{l_2})$$

set  $B = (b_1, \dots, b_l) = M || C$  (concatenation)

obtain  $sk$  with  $sk \leftarrow WOTS.keygen(S, r)$

signature:

$$\sigma = (\sigma_1, \dots, \sigma_l) = (C^{b_1}(sk_1, r), \dots, C^{b_l}(sk_l, r))$$

**verification:**  $pk' \leftarrow WOTS.vf(M, \sigma, r)$

compute  $b_i$  for  $1 \leq i \leq l$  as described above

return:

$$pk' = (pk'_1, \dots, pk'_l) = (C^{w-1-b_1}(\sigma_1, r_{b+1, w-1}), \dots, C^{2-1-b_l}(\sigma_l, r_{b+1, w-1}))$$

and check that  $pk \stackrel{?}{=} pk'$ .

## 1.5 HORST

**parameters**  $k, t = 2^\tau$ , with  $k\tau = m$  (length of the message signed).

in SPHINCS-256:  $t = 2^{16}$ ,  $k = 32$ .

**key generation:**  $pk \leftarrow HORST.keygen(S, Q)$

input seed:  $S \in \{0,1\}^n$

bitmasks:  $Q \in \{0,1\}^{2n \times \log t}$

$$sk = (sk_1, \dots, sk_t) \leftarrow G_t(S)$$

Leaves of the tree are computed as  $L_i = F(sk_i)$  for  $i \in [t-1]$ , and the tree is constructed using bitmasks  $Q$ .

pk: root node of a binary tree of height  $\log t$ .

**signature:**  $(\sigma, pk) \leftarrow \text{HORST.sign}(M, S, Q)$

$M \in \{0, 1\}^m$ ,  $S \in \{0, 1\}^n$ ,  $Q \in \{0, 1\}^{2n \times \log t}$

Let  $M = (M_0, \dots, M_{k-1})$  be the  $k$  numbers obtained by splitting  $M$  into  $k$  strings of length  $\log t$  bits each, and interpreting each as an unsigned integer.

Signature:

$$\sigma = ( \underbrace{\sigma_0, \dots, \sigma_{k-1}}_{\substack{\sigma_i = (sk_{M_i}, Auth_{M_i}) \\ \text{for } i \in [k-1]}}, \sigma_k )$$

$Auth_{M_i}$ : lower  $\tau - x$  elements of the authentication path of the corresponding leaf  $(A_0, \dots, A_{\tau-1-x})$

$\sigma_k$ : all the  $2^x$  nodes of the binary tree on level  $\tau - x$ ,  $(N_{0, \tau-x}, \dots, N_{2^x-1, \tau-x})$ .

**verification:**  $pk' \leftarrow \text{HORST.vf}(M,$

$sigma, Q)$

first compute  $M_i$  as described in the signature step.

Then, for  $i \in [k-1]$ ,  $y_i = \lfloor M_i / 2^{\tau-x} \rfloor$ , it computes  $N'_{y_i, \tau-x}$  using root computation with index  $M_i$ ,  $L_{M_i} F(\sigma_i^1)$ ,  $Auth_{M_i} = \sigma_i^2$ .

Then check  $\forall i \in [k-1]$ :

$$N'_{y_i, \tau-x} \stackrel{?}{=} N_{y_i, \tau-x}$$

ie. computed nodes match those in  $\sigma_k$ .

If ok, use  $\sigma_k$  to compute the root  $(= N_{0,h})$ , and return the root as  $pk'$ .

## 2 SPHINCS (2015)

### 2.1 High level view

- hyper-tree of total height  $h$
- each tree has height  $h/d$
- inside the tree, use Merkle approach
- between trees use Goldreich approach
- sign messages with a "few-time" signature scheme (FTS)

**SPHINCS-256** (128 bits of security)

- 12 trees of height 5 each
- use WOTS as one-time-signature (OTS) scheme
- use HORST (HORS with tree) as few-time signature (FTS) scheme
- fix  $n = 256$  as bitlength of hashes in WOTS and HORST
- fix  $m = 512$  as size of the message hash (Blake-512)
- use ChaCha12 as pseudorandom generation

## 2.2 SPHINCS construction

### 2.2.1 Parameters

Security parameter:  $n \in \mathbb{N}$

Hash functions:

$$\begin{aligned} F &: \{0, 1\}^n \longrightarrow \{0, 1\}^n \\ H &: \{0, 1\}^{2n} \longrightarrow \{0, 1\}^n \\ \mathcal{H} &: \{0, 1\}^n \times \{0, 1\}^* \longrightarrow \{0, 1\}^m \end{aligned}$$

for  $m = \text{poly}(n)$

Height of the hyper-tree:  $h \in \mathbb{N}$  (and  $h|d$ ):

$d$  layers of trees, each of height  $h/d$ .

Each tree: leaves are  $2^{h/d}$  L-Tree root nodes, that each is a pk of WOTS+.

$\implies$  a tree is a keypair that can be used to sign  $2^{h/d}$  messages.

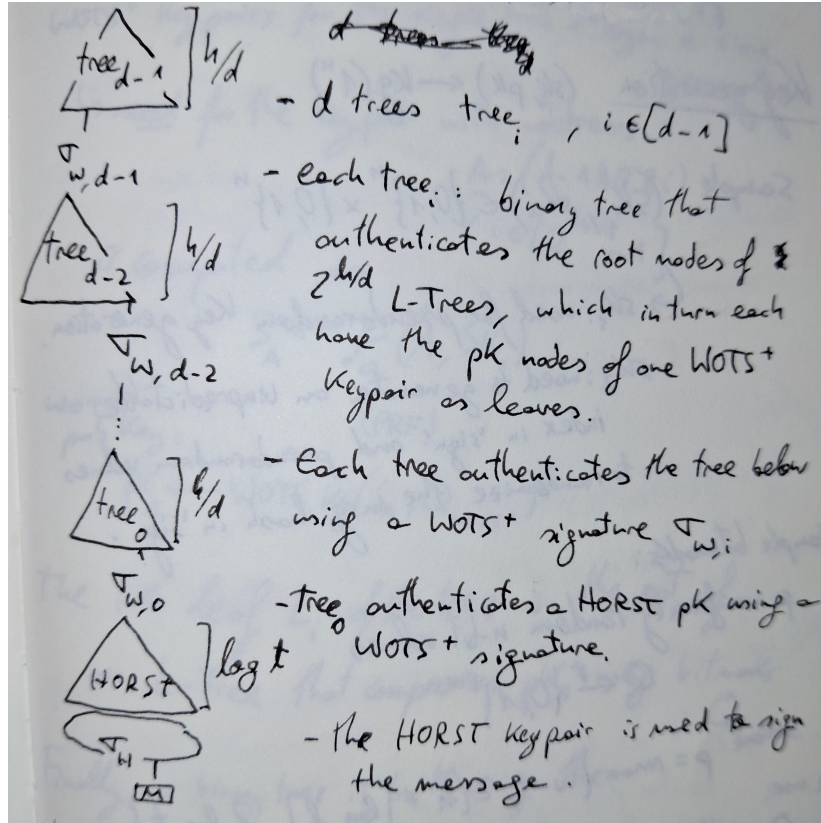
The hyper-tree has  $d$  layers

- on layer  $d - 1$  it has a single tree
- on layer  $d - 2$  it has  $2^{h/d}$  trees
  - roots signed using WOTS+ key pairs of the tree on layer  $d - 1$

In general: layer  $i$ :  $2^{(d-1-i)(h/d)}$  trees, and the roots of these trees are signed using the WOTS+ key pairs of the trees on layer  $i + 1$ .

Layer 0: each WOTS+ keypair is used to sign a HORST pk.

$\longrightarrow$  Full tree structure is never computed, values are determined by the seed and the bit masks.



An address is a bit string of length

$$a = \underbrace{\lceil \log(d+1) \rceil}_{\text{WOTS+ keypair addr}} + \underbrace{(d-1)(h/d)}_{\text{index of the tree in the layer}} + \underbrace{(h/d)}_{\text{index of the WOTS+ key pair within the tree}}$$

$$= \lceil \log(d+1) \rceil + h$$

→ in SPHINCS-256, an address needs 64 bits.

### 2.2.2 key generation:

$(sk, pk) \leftarrow \text{keygen}(1^n)$

sample  $(sk_1, pk_1) \in \{0, 1\}^n \times \{0, 1\}^n$

$sk_1$ : used for pseudorandom key generation

$sk_2$ : used to generate an unpredictable index in 'sign' and pseudorandom values to randomize the msg hash in 'sign'.

Sample bitmasks:  $p$  uniformly random  $n$ -bit values

$$Q \leftarrow^R \{0, 1\}^{p \times n}$$

where  $p = \max\{w - 1, 2(h + \lceil \log(l) \rceil), 2 \log t\}$ .

We use

- $Q_{WOTS+}$  for the first  $w - 1$  bitmasks (of length  $n$ ) in  $Q$
- $Q_{HORST}$  for the first  $2 \log t$
- $Q_{L-Tree}$  for the first  $2 \lceil \log l \rceil$
- $Q_{Tree}$  for the  $2h$  strings of length  $n$  in  $Q$  that follow  $Q_{L-Tree}$

Remaining logic in keygeneration: generate the root node of the tree on layer  $d - 1$ .

WOTS+R keypairs for the single tree at layer  $d - 1$  are generated  
 $\rightarrow$  seed for the keypair with address  $A = (d - 1 || 0 || i)$  with  $i \in [2^{h/d} - 1]$  is computed as

$$S_A \leftarrow \mathcal{F}_a(A, sk_1)$$

where  $\mathcal{F}$  is a PRF.

WOTS+ pub key:

$$pk_A \leftarrow WOTS.keygen(S_A, Q_{WOTS+})$$

The  $i$ -th leaf  $L_i$  of the tree is the root of an L-Tree that compresses  $pk_A$  using bitmasks  $Q_{L-Tree}$ .

Finally, a binary tree is build using the constructed leaves and its root node becomes  $pk_1$ .

SPHINCS  $sk = (sk_1, sk_2, Q)$ ,  $pk = (pk_1, Q)$ .

### 2.2.3 Signature algorithm

$\Sigma \leftarrow sign(M, sk)$

$M \in \{0, 1\}^*$

Compute a randomized message digest  $D \in \{0, 1\}^m$ :

compute pseudorandom  $R = (R_1, R_2) \in \{0, 1\}^n \times \{0, 1\}^n$

$$R \leftarrow \mathcal{F}(M, sk_2)$$

then

$$D \leftarrow \mathcal{H}(R_1, M)$$

ie. randomized hash of  $M$ ; where  $R_1$  are the first  $n$  bits of  $R$ .

$R_2$  (later  $n$  bits of  $R$ ) are used to select a HORST keypair:

compute an  $h$  bit index  $i \leftarrow CHOP(R_2, h)$  as the first  $h$  bits of  $R_2$ .

Given index  $i$ , HORST key pair with address

$$A_{HORST} = (d || i(9, (d - 1)h/d) || i((d - 1)h/d, h/d))$$

is used to sign the message digest  $D$ .

ie. first  $(d-1)h/d$  bits of  $i$  are used as tree index and the remaining bits for the index within the tree.

The HORST signature and public key

$$(\sigma_H, pk_H) \leftarrow (D, S_{A_{HORST}}, Q_{HORST})$$

are computed using the HORST bitmasks and the seed

$$S_{A_{HORST}} \leftarrow \mathcal{F}_a(A_{HORST}, sk_1)$$

Signature:

$$\Sigma = (i, R_1, \sigma_H, \sigma_{W,0}, Auth_{A_0}, \sigma_{W,d-1}, Auth_{A_{d-1}})$$

where

$\sigma_H$ : HORST signature

$\sigma_{W,0}$ : WOTS+ signature

$Auth_{A_0}$ : one auth path per layer.

#### 2.2.4 Verification

$b \leftarrow vf(M, \Sigma, pk)$

compute message digest  $D \leftarrow \mathcal{H}(R_1, M)$  ( $R_1$  comes from  $\Sigma$ )

$$pk_H \leftarrow HORST.vf(D, \sigma_H, Q_{HORST})$$

where  $Q_{HORST}$  comes from  $pk$ .

$$pk_{W,0} \leftarrow WOTS.vf(pk_H, \sigma_{W,0}, Q_{WOTS+})$$

where  $\sigma_{W,0}$  is the WOTS+ signature, and  $Q_{WOTS+}$  is the WOTS+ bitmask.

An L-Tree is used to compute  $L_{i((d-1)h/d, h/d)}$ , ie. the leaf corresponding to  $pk_{W,0}$ .

Then, compute  $Root_0$  with index  $i((d-1)h/d, h/d)$ , leaf  $L_{i((d-1)h/d, h/d)}$  and authentication path  $Auth_0$ .

This procedure is repeated for layers 1 to  $d-1$  with the following differences:

1. on layer  $1 \leq j < d$ , the root of the previous tree,  $Root_{j-1}$ , is used to compute the WOTS+  $pk_{w,j}$ .
2. the leaf from  $pk_{w,j}$  using an L-Tree is  $L_{i((d-1)h/d, h/d)}$ ; ie. the index of the leaf within the tree can be computed cutting off the last  $j(h/d)$  bits of  $i$  and then using the last  $(h/d)$  bits of the resulting bit string.

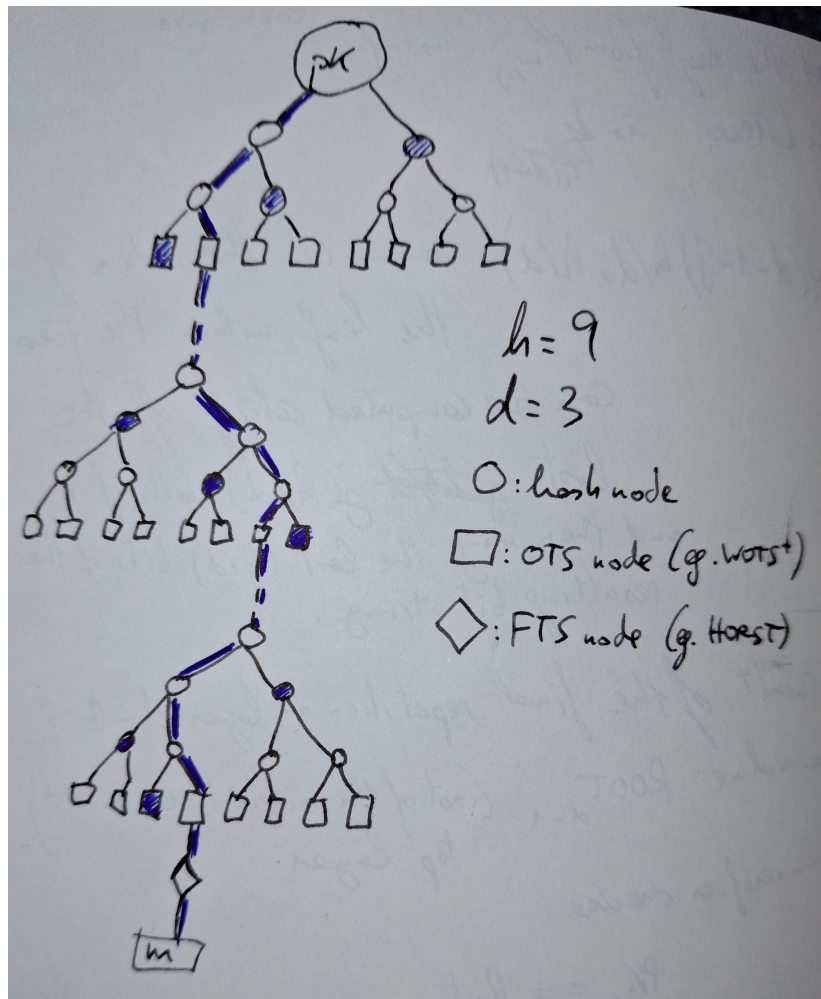
The result of the final repetition on layer  $d-1$  is a value  $Root_{d-1}$  (root of the single tree on the top layer).

Then verifier checks

$$pk_1 \stackrel{?}{=} Root_{d-1}$$

### 2.2.5 Sizes of SPHINCS-256

- pk: 1056 bytes
- sk: 1088 bytes
- sig: 41000 bytes



## 3 XMSS (2011)

After seeing the previous schemes of this document, XMSS signatures are quite straight forward:

MerkleTree which leaves are WOTS+ public keys. Each leaf has a corresponding one-time key  $(sk_i, pk_i)$

To sign message number  $i$ :

$$\sigma_i = (i, R, \sigma_{WOTS+,i}, AuthPath_i)$$

Verifier reconstructs  $pk_i$  from the WOTS+ signature, and then follows the authentication path to obtain the root.

Important: Never use the same WOTS+ leaf more than once. Thus it is important to keep track of the used leaves. ie. keep a state.

## References

- [1] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O’Hearn. SPHINCS: practical stateless hash-based signatures. Cryptology ePrint Archive, Paper 2014/795, 2014.
- [2] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018.