Hash Based Signatures

Halil İbrahim Kaplan

TDD / Kripto Analiz Laboratuvarı

halil.kaplan@tubitak.gov.tr

2023

Overview

1. Hash Functions and Security
2. Lamport One-Time Signiture
3. Merkle’s tree-based signature
4. WOTS/WOTS+
5. XMSS
6. HORS/FORS
7. SPHINCS+

Digital signatures assures

Authenticity

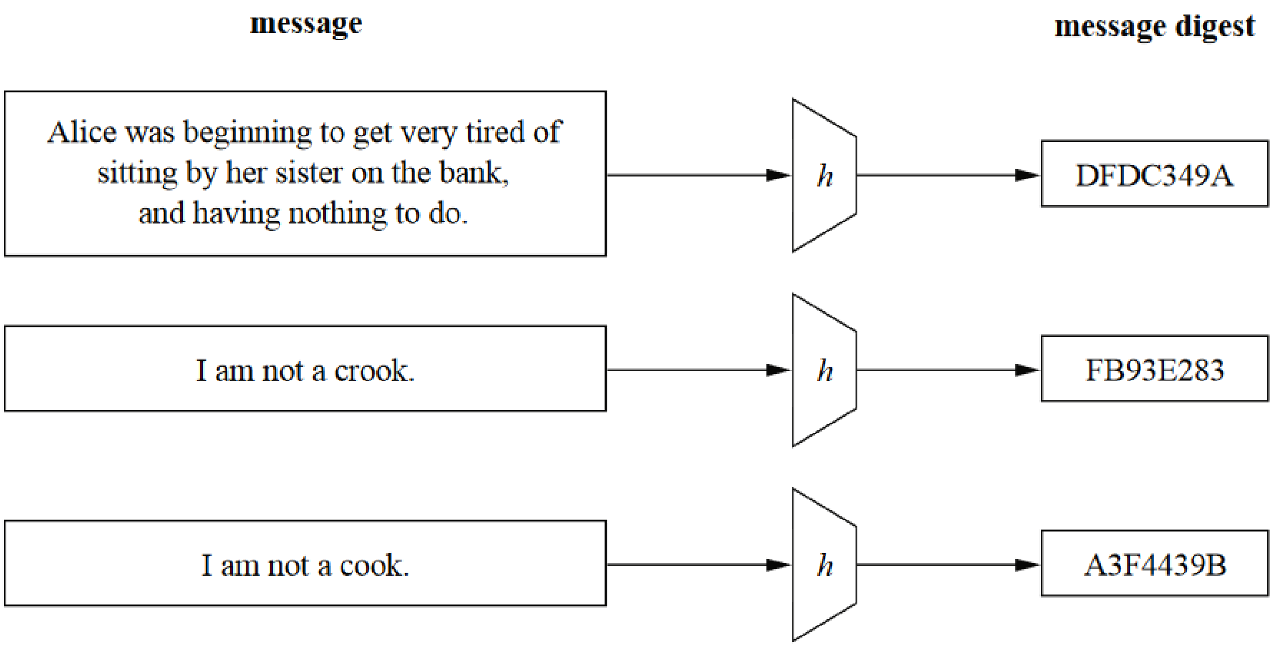
The identity of the organization that sent the message (the message signer) is confirmed.

Integrity

The message content was not changed or tampered with since it was digitally signed.

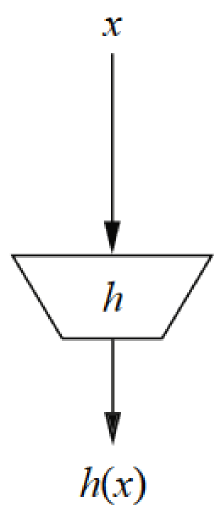
Nonrepudiation

The origin of the signed content is verified to all parties so the message signer cannot deny association with the signed content.


message
message digest
Alice was beginning to get very tired of sitting by her sister on the bank, and having nothing to do.
h
DFDC349A
I am not a crook.
h
FB93E283
I am not a cook.
h
A3F4439B

Preimage resistance:

Given h(x) it should be infeasible to find x

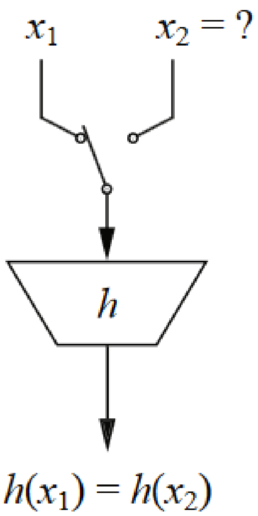


preimage resistance

Second-preimage resistance(weak collision):

Given any first input x_1 , it should be infeasible to find any distinct second input x_2 such that

$$h(x_1) = h(x_2)$$

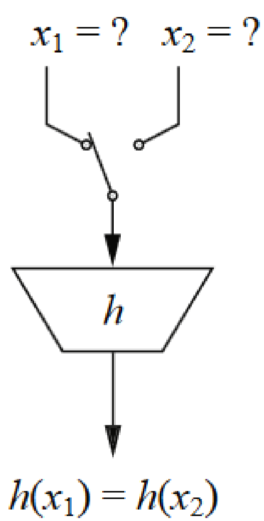


second preimage
resistance

Collision resistance:

It should be infeasible to find any pair of distinct inputs x_1 and x_2 , such that

$$h(x_1) = h(x_2).$$



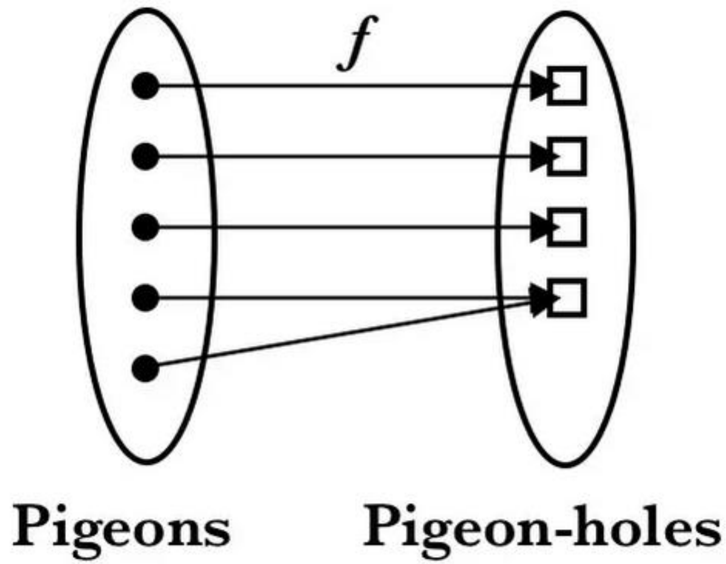
collision resistance

Hash Functions and Security

Alice		**Oscar**		**Bob**
			$\xleftarrow{k_{pub,B}}$	
				$z = h(x_1)$ $s = \mathrm{sig}_{k_{pr,B}}(z)$
	$\xleftarrow{(x_2,s)}$	$\lightning$ substitute	$\xleftarrow{(x_1,s)}$	
$z = h(x_2)$ $\mathrm{ver}_{k_{pub,B}}(s,z) = \mathrm{true}$				

Ideally, we would like to have a hash function for which weak collisions do not exist.

This is, unfortunately, impossible due to the pigeonhole principle



So we say infeasible

How common collisions for 80 bit hash output ?

Expectation: Try 2^{80} message

Reality: Try 2^{40} message

Reason: Birthday paradox

How many people are needed such that
there is a reasonable chance that at least two people have the same birthday?

$$P(\text{no collision among 2 people}) = \left(1 - \frac{1}{365}\right)$$

If a third person joins the party, he or she can collide with both of the people already there, hence:

$$P(\text{no collision among 3 people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Consequently, the probability for t people having no birthday collision is given by:

$$P(\text{no collision among } t \text{ people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

For $t = 366$ people we will have a collision with probability 1 since a year has only 365 days. We return now to our initial question: how many people are needed to have a 50% chance of two colliding birthdays? Surprisingly—following from the equations above—it only requires 23 people to obtain a probability of about 0.5 for a birthday collision since:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= 1 - \left(1 - \frac{1}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\%. \end{aligned}$$

```python
import random

def generate_random_birthday(n):
    days = list(range(1, 32))
    months = list(range(1, 13))
    birthdays = []
    for i in range(n):
        day = random.choice(days)
        month = random.choice(months)
        birthday = (day, month)
        birthdays.append(birthday)
    return birthdays

def check_collision(birthdays):
    collision = False
    for i in range(len(birthdays)):
        for j in range(i + 1, len(birthdays)):
            if birthdays[i] == birthdays[j]:
                collision = True
    return collision

# Take input for number of people
n = int(input("Enter the number of people: "))

# Set number of trials
num_trials = 100

# Initialize collision count
collision_count = 0

# Run 100 trials
for trial in range(num_trials):
    birthdays = generate_random_birthday(n)
    collision = check_collision(birthdays)
    if collision:
        collision_count += 1

print(f"Total collisions in {num_trials} trials: {collision_count}")
```

```
Enter the number of people: 23
Total collisions in 100 trials: 54
```

```
Enter the number of people: 23
Total collisions in 100 trials: 50
```

```
Enter the number of people: 23
Total collisions in 100 trials: 52
```

```
Enter the number of people: 40
Total collisions in 100 trials: 88
```

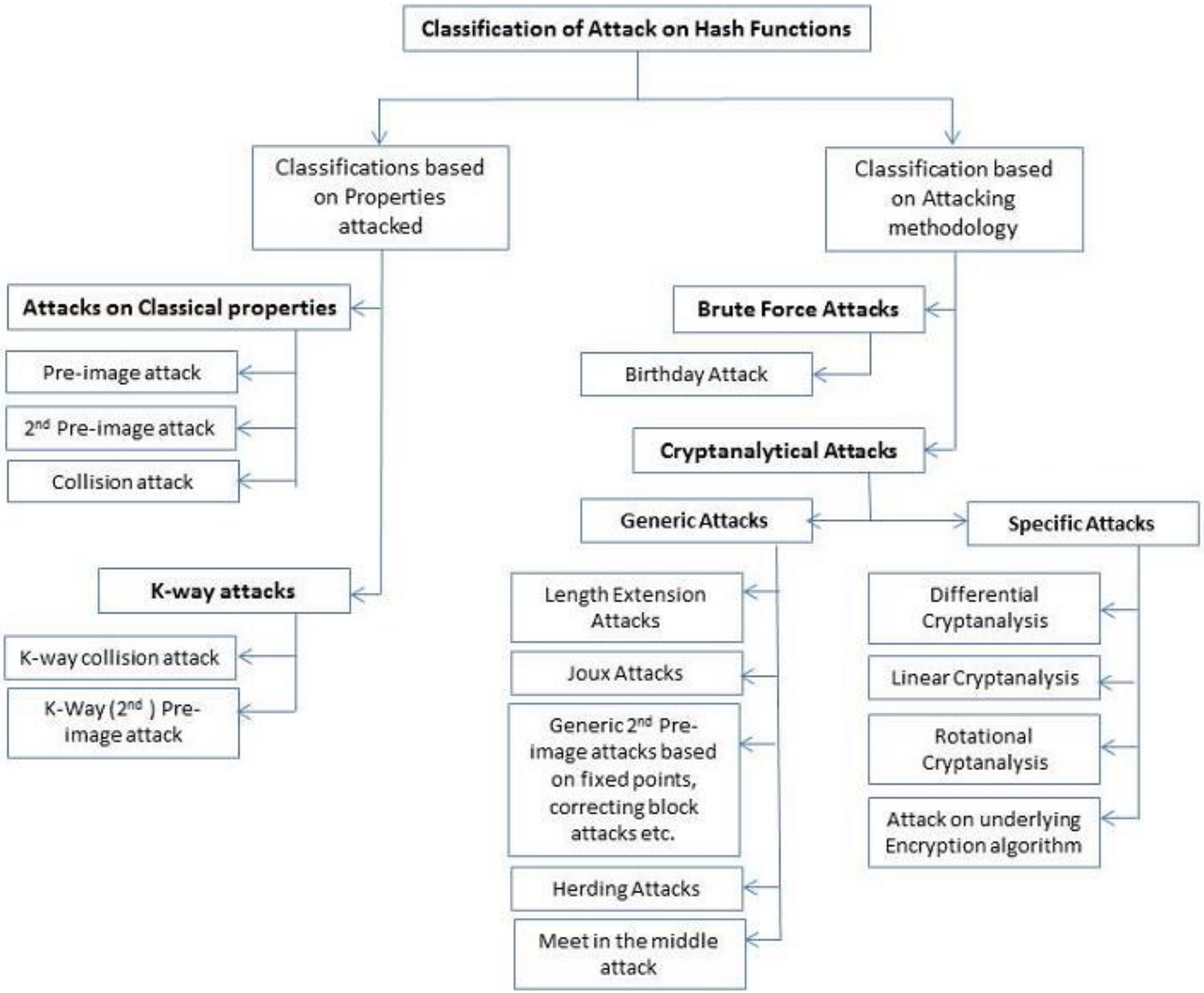
```
Enter the number of people: 40
Total collisions in 100 trials: 91
```

```
Enter the number of people: 40
Total collisions in 100 trials: 90
```

Number of messages we need to hash to find a collision is roughly equal to the

$$\sqrt{2^n} = 2^{\frac{n}{2}}$$

λ	Hash output length 128 bit	160 bit	256 bit	384 bit	512 bit
0.5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}


Classification of Attack on Hash Functions
Classifications based on Properties attacked
Classification based on Attacking methodology
Attacks on Classical properties
Pre-image attack
2nd Pre-image attack
Collision attack
K-way attacks
K-way collision attack
K-Way (2nd) Pre-image attack
Brute Force Attacks
Birthday Attack
Cryptanalytical Attacks
Generic Attacks
Specific Attacks
Length Extension Attacks
Joux Attacks
Generic 2nd Pre-image attacks based on fixed points, correcting block attacks etc.
Herding Attacks
Meet in the middle attack
Differential Cryptanalysis
Linear Cryptanalysis
Rotational Cryptanalysis
Attack on underlying Encryption algorithm

What about finding collision with Quantum Computer ?

Quantum Algorithm for the Collision Problem

Gilles Brassard *
Université de Montréal †

Peter Høyer ‡
Odense University §

Alain Tapp ¶
Université de Montréal †

1 May 1997

Abstract

In this note, we give a quantum algorithm that finds collisions in arbitrary r-to-one functions after only $O(\sqrt[3]{N/r}\,)$ expected evaluations of the function. Assuming the function is given by a black box, this is more effi-

What about finding collision with Quantum Computer ?

Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?

Daniel J. Bernstein *

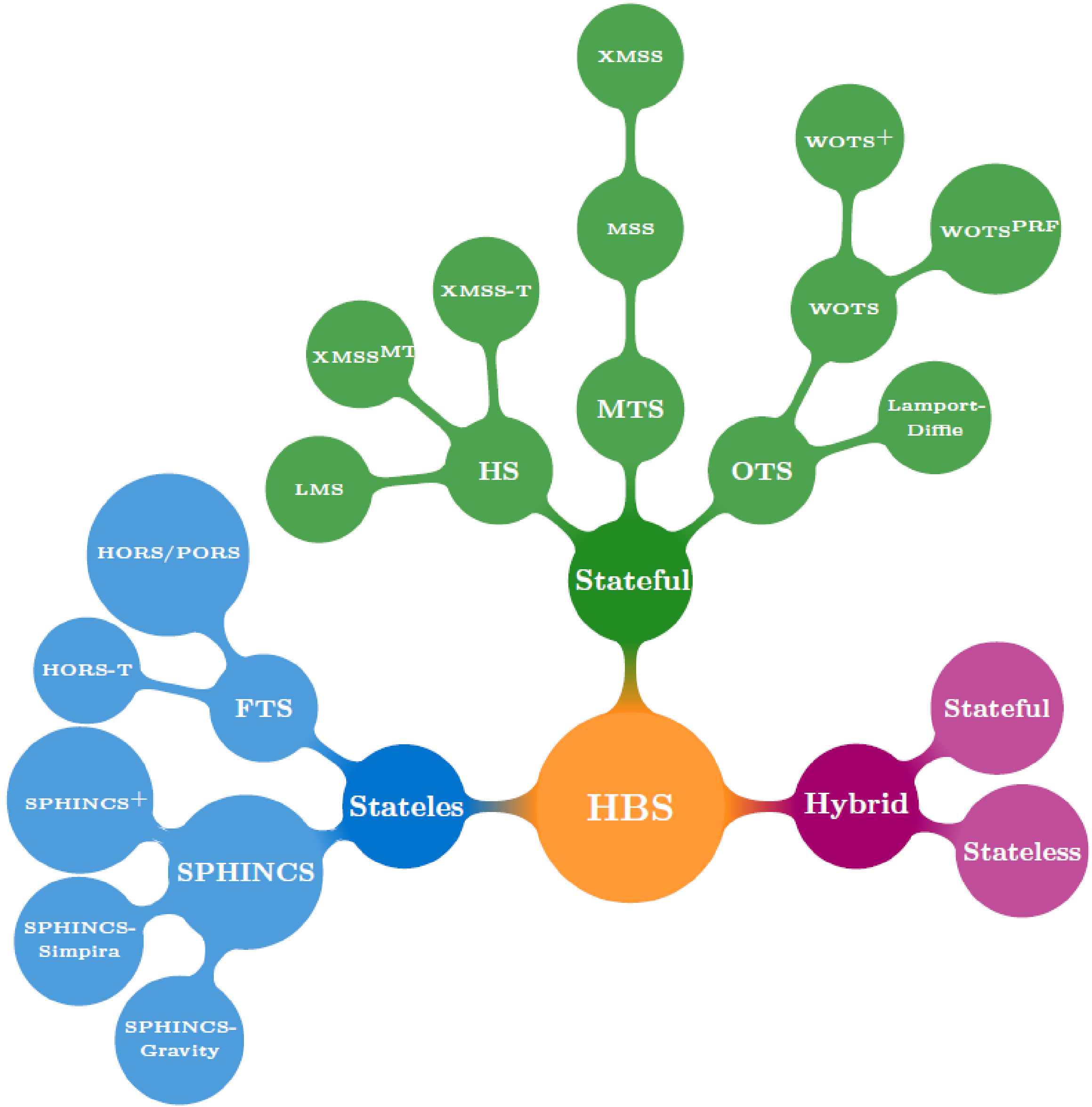
Department of Computer Science (MC 152)
The University of Illinois at Chicago
Chicago, IL 60607–7053
djb@cr.yp.to

Abstract. Current proposals for special-purpose factorization hardware will become obsolete if large quantum computers are built: the number-field sieve scales much more poorly than Shor's quantum algorithm for factorization. Will *all* special-purpose cryptanalytic hardware become obsolete in a post-quantum world?

A quantum algorithm by Brassard, Høyer, and Tapp has frequently been claimed to reduce the cost of b-bit hash collisions from $2^{b/2}$ to $2^{b/3}$. This paper analyzes the Brassard–Høyer–Tapp algorithm and shows that it has fundamentally worse price-performance ratio than the classical van Oorschot–Wiener hash-collision circuits, even under optimistic assumptions regarding the speed of quantum computers.

Hash Functions and Security

Family name	Year	Output size bitlength	bytes	Alternate names and notes
SHA-3	2015	224, 256	28, 32	SHA3-224, SHA3-256
		384, 512	48, 64	SHA3-384, SHA3-512 (NOTE 1)
SHA-2	2001	256, 512	32, 64	SHA-256, SHA-512
SHA-1	1995	160	20	Deprecated (2017) for browser certificates
MD5	1992	128	16	Widely deprecated, for many applications



XMSS
WOTS+
MSS
WOTS^PRF
XMSS-T
WOTS
XMSS^MT
MTS
Lamport-Diffie
LMS
HS
OTS
Stateful
HORS/PORS
HORS-T
FTS
Stateful
SPHINCS+
Stateles
HBS
Hybrid
SPHINCS
Stateless
SPHINCS-Simpira
SPHINCS-Gravity

Lamport One-Time Signature

Lamport One-Time Signature

Hash function : H(256 bits output)
Message : M=(256 bits)
Signature : S=(256 bits)

1. Generate 512 seperate random bitstring, each of 256 bits of length

2. Index them:

$$sk_0 = sk_0^1, sk_0^2 \dots sk_0^{256}$$
$$sk_1 = sk_1^1, sk_1^2 \dots sk_1^{256}$$

Lamport One-Time Signature

Hash function : H(256 bits output)
Message : M=(256 bits)
Signature : S=(256 bits)

1. Generate 512 seperate random bitstring, each of 256 bits of length

2. Index them:

$$sk_0 = sk_0^1, sk_0^2 \dots sk_0^{256}$$
$$sk_1 = sk_1^1, sk_1^2 \dots sk_1^{256}$$

3. Generate public key:

$$pk_0 = H(sk_0^1), H(sk_0^2) \dots H(sk_0^{256})$$
$$pk_1 = H(sk_1^1), H(sk_1^2) \dots H(sk_1^{256})$$

Lamport One-Time Signature

Hash function : H(256 bits output)
Message : M=(256 bits)
Signature : S=(256 bits)

1. Generate 512 seperate random bitstring, each of 256 bits of length

2. Index them:

$$sk_0 = sk_0^1, sk_0^2 \dots sk_0^{256}$$
$$sk_1 = sk_1^1, sk_1^2 \dots sk_1^{256}$$

3. Generate public key:

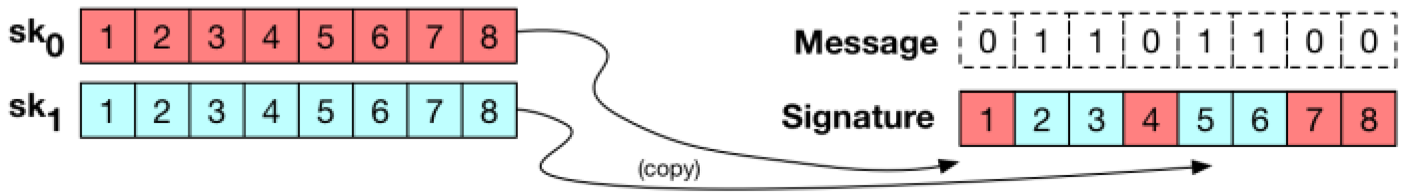
$$pk_0 = H(sk_0^1), H(sk_0^2) \dots H(sk_0^{256})$$
$$pk_1 = H(sk_1^1), H(sk_1^2) \dots H(sk_1^{256})$$

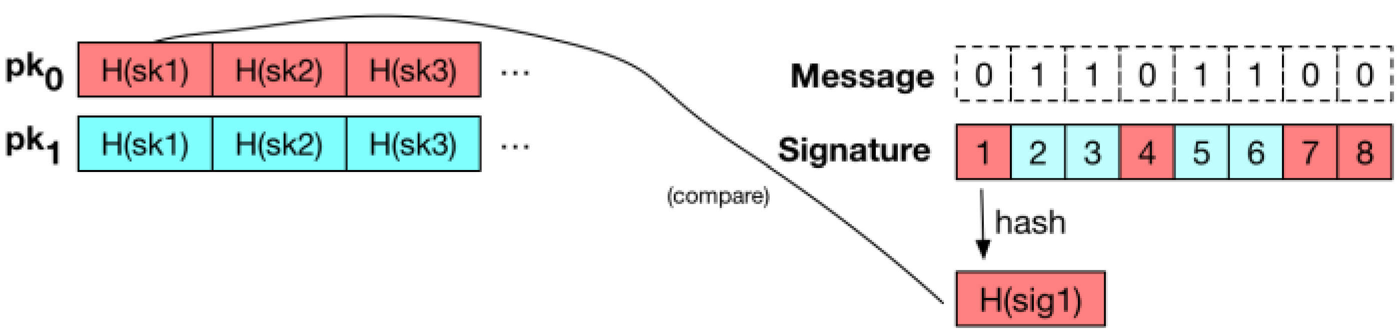
4. For each bit in message

   if m_i = 0 then $s_i = sk_0^i$

   if m_i = 1 then $s_i = sk_1^i$

Lamport One-Time Signature

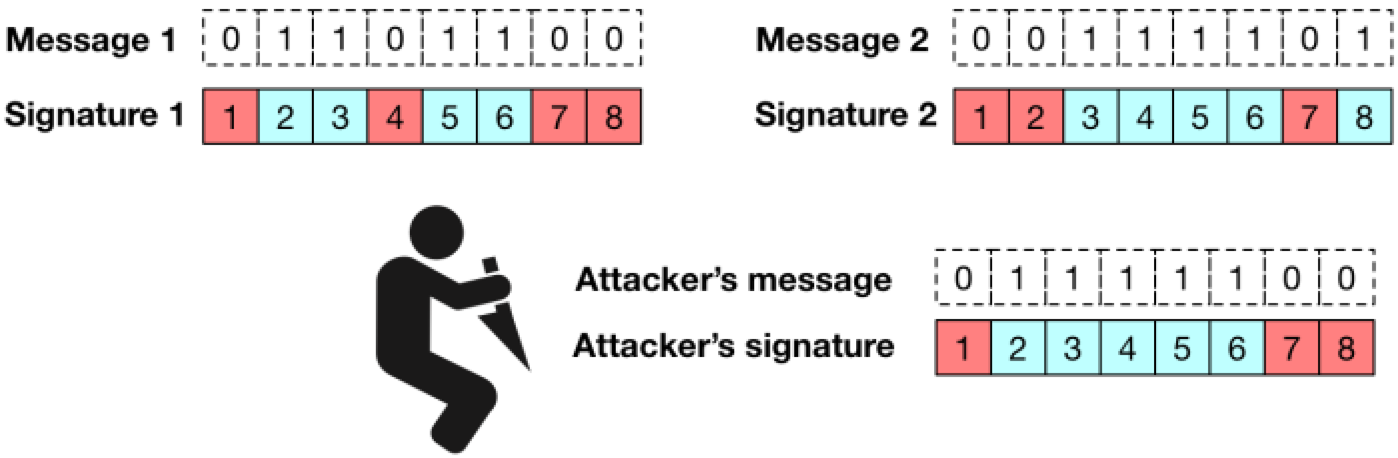




Lamport One-Time Signature

Each key can only be used to sign one message(OTP).

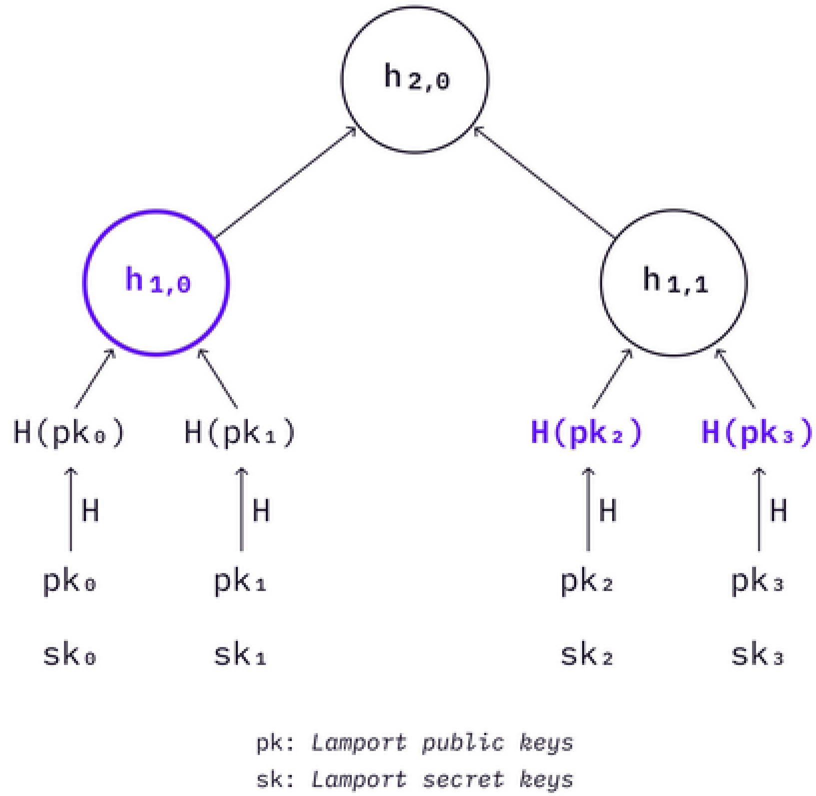
If not, mix and match forgery attack can be done.



Merkle’s tree-based signature

Tranforms one-time signatures to many-time signatures.

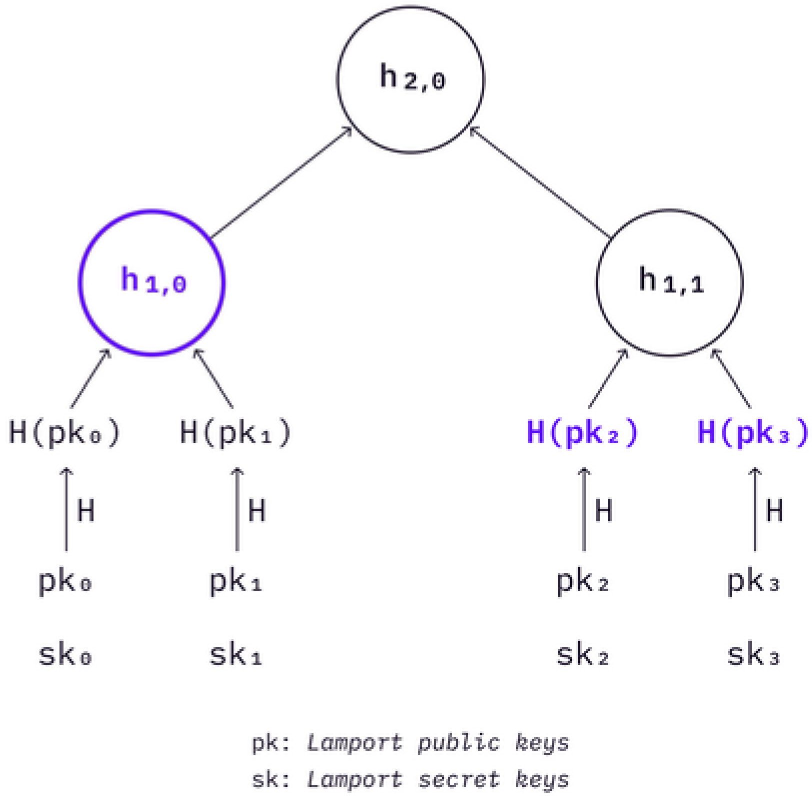
1. Generate N separate Lamport keypairs.
   We can call those $(sk_0), \ldots, (sk_N)$.



Merkle's tree-based signature

Tranforms one-time signatures to many-time signatures.

1. Generate N separate Lamport keypairs. We can call those (sk_0), … , (sk_N).
2. Place each public key at one leaf of a Merkle hash tree (see below), and compute the root of the tree. This root will become the "master" public key of the new Merkle signature scheme.

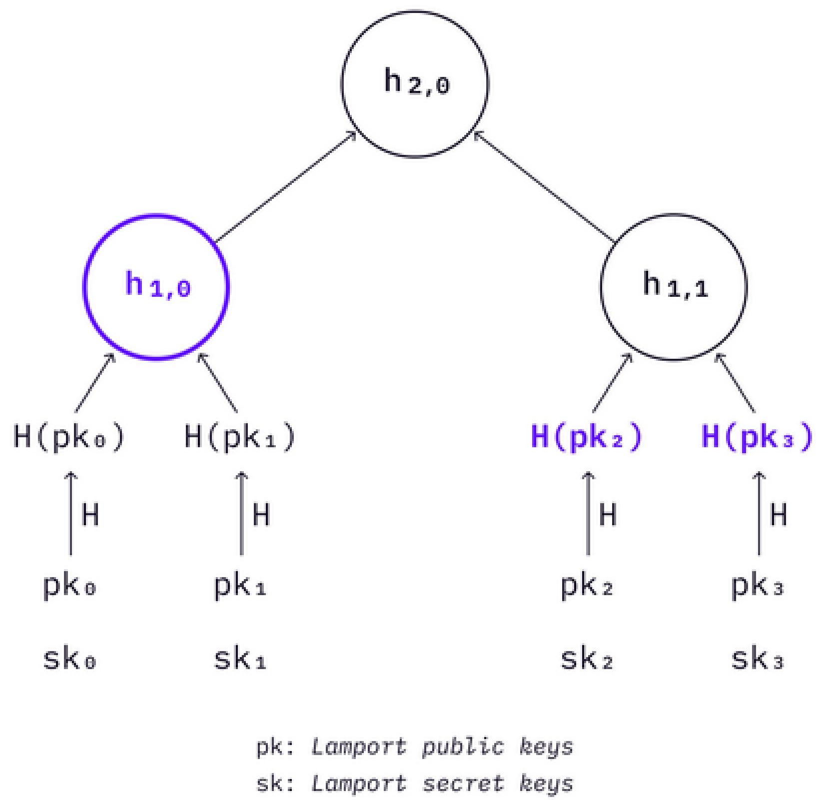


Tranforms one-time signatures to many-time signatures.

1. Generate N separate Lamport keypairs. We can call those (sk_0), … , (sk_N).

2. Place each public key at one leaf of a Merkle hash tree (see below), and compute the root of the tree. This root will become the "master" public key of the new Merkle signature scheme.

3. The signer retains all of the Lamport public and secret keys for use in signing.



Merkle's tree-based signature

1. To sign the i^{th} bit, select the i^{th}public key from the tree, and sign the message using the corresponding Lamport secret key.

2. Concatenate the resulting signature to the Lamport public key and tacks on a "Merkle proof" that shows that this specific Lamport public key is contained within the tree identified by the root (i.e., the public key of the entire scheme).

3. Transmit this whole collection as the signature of the message.

Merkle’s tree-based signature

To verify a signature of this form,

1. Unpack this “signature” as a Lamport signature, Lamport public key, and Merkle Proof.

2. Verify the Lamport signature against the given Lamport public key

3. Use the Merkle Proof to verify that the Lamport public key is really in the tree.

4. If these three objectives achieved, Trust the signature as valid

Disadvantage : Signature size

Advantage : Public key is just 1 hash value

Merkle’s tree-based signature

IDEA: What if we don’t sign all of the message bits

Let’s say we sign only the message bits equal to one.

This would cut the public and secret key sizes in half.

VERY INSECURE

A

111…	abcd...
Message	Signature

C

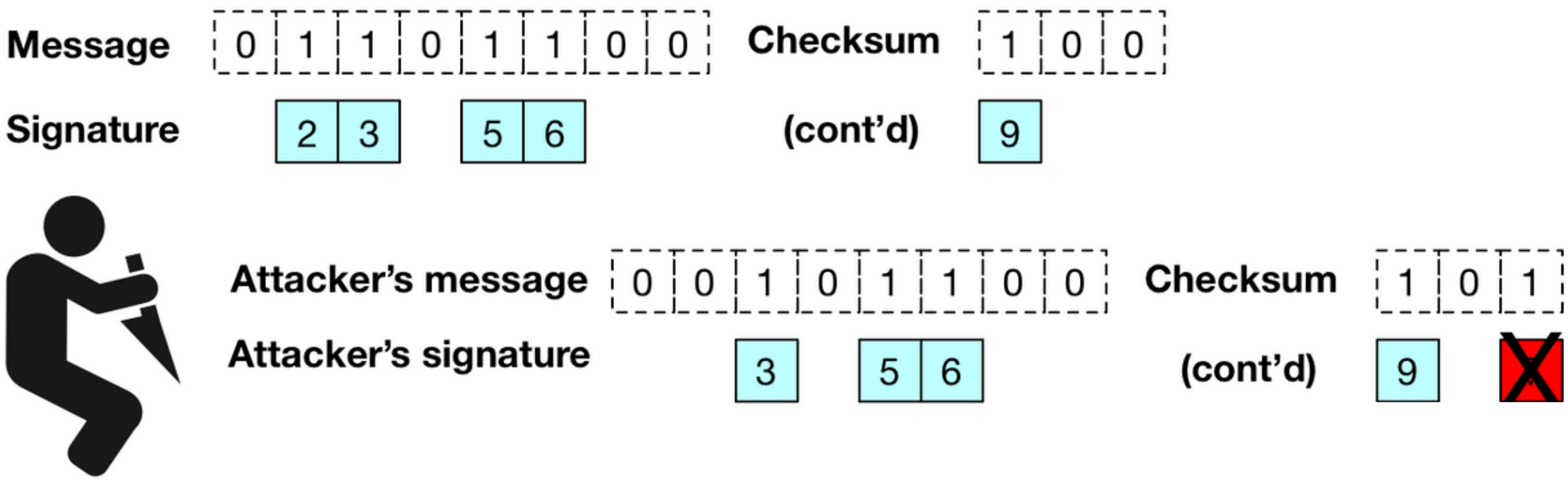
~~111~~…	~~abc~~d...
Message	Signature

B

000…	d...
Message	Signature

Merkle's tree-based signature

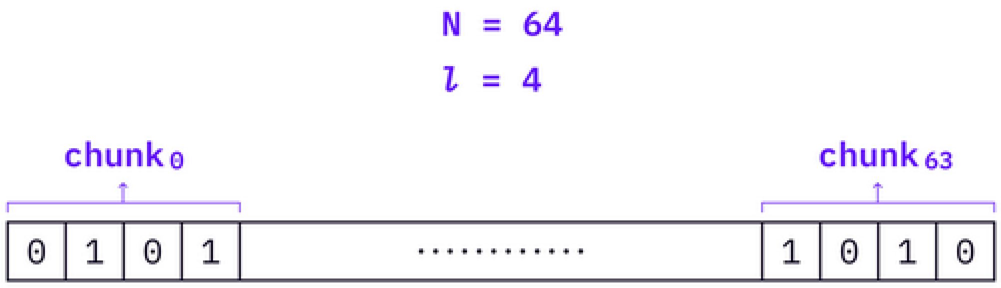
We need to add checksum and sign checksum with message



WOTS

IDEA: Divide message into N pieces of length l and sign each piece

We need secret and public key (For each possible value of piece)



IDEA: Divide message into N pieces of length l and sign each piece

We need secret and public key (For each possible value of piece)

Reduces : The size of the signature by a factor of N

Cost: Increasing the public and secret key size by a factor of 2^l

NEW IDEA: What if we generated those key lists programatically only when we needed them?

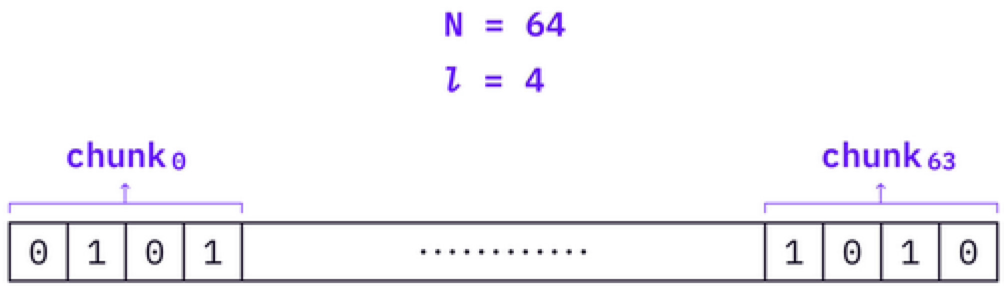
WOTS

- Generate single list of random seeds for our initial secret key.
- Rather than generating additional lists randomly, he proposed to use the hash function H(.) on each element of that initial secret key.
- Public key could be derived by applying the hash function one more time to the final secret key list

$$
\begin{array}{ccccccccc}
sk_0 & \longrightarrow & h(sk_0) & \longrightarrow & h^2(sk_0) & \longrightarrow & \cdots & \longrightarrow & h^{2^l-1}(sk_0) \\
sk_1 & \longrightarrow & h(sk_1) & \longrightarrow & h^2(sk_1) & \longrightarrow & \cdots & \longrightarrow & h^{2^l-1}(sk_1) \\
\vdots & & \vdots & & \vdots & & & & \vdots \\
sk_{N-1} & \longrightarrow & h(sk_{N-1}) & \longrightarrow & h^2(sk_{N-1}) & \longrightarrow & \cdots & \longrightarrow & h^{2^l-1}(sk_{N-1})
\end{array}
$$

A WOTS signature is generated as follows,

1. Compute the decimal values of each piece from the message digest. For example, the decimal value of the first piece is 5(m).



2. Compute signature of ith piece by hashing the corresponding private key $2^l - 1 - m$ times. In the above example, will be hashed $2^4 - 1 - 5 = 10$ times.

$$sk_0 \rightarrow h(sk_0) \rightarrow h^2(sk_0) \rightarrow \cdots \rightarrow h^{10}(sk_0)$$

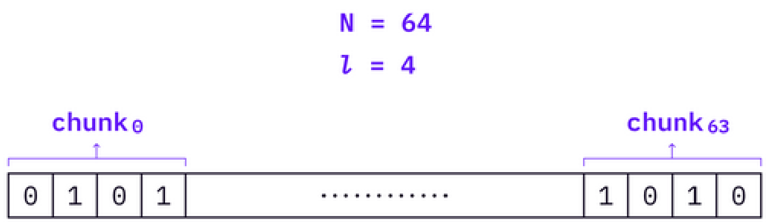
3. Apply (1) and (2) on N piece and produce signature .

$$\sigma = [h^{2^l-1-m_0}(sk_0), \ldots, h^{2^l-1-m_{N-1}}(sk_{N-1})]$$

WOTS

To verify the signature,

1. Hash the message and get digest m



2. Divide m into pieces, obtaining decimal values of each piece.

3. The verifier then hash each value the signature 2^{m_i} times, get

$$Pk_v = (Pk_{v0}, Pk_{v1}, \dots, Pk_{v(N-1)})$$

4. If it is identical to the public key set , signature is valid, otherwise the signature is rejected.

WOTS+

- Adds randomization to WOTS

- There is additional random public key

$$Pk_N = \left(r_1, \ldots, r_{2^l-1}\right)$$

- There is new hash function c to replace h in WOTS s.t

$$c^i(x, PK_N) = \begin{cases} x & \text{i=0} \\ H(c^{i-1}(x, pk_N) \oplus pk_N^i) & \text{i>0} \end{cases}$$

$$c^i(x, PK_N) = \begin{cases} x & \text{i=0} \\ H(c^{i-1}(x, pk_N) \oplus pk_N^i) & \text{i>0} \end{cases}$$

$$(r_1, r_2, \cdots, r_{2^l-1}) = pk_N$$

$$\begin{array}{lllllll}
sk_0 & \longrightarrow c^0(sk_0, pk_N) & \longrightarrow c^1(sk_0, pk_N) & \longrightarrow & \cdots & \longrightarrow c^{2^l-1}(sk_0, pk_N) & = pk_0 \\
sk_1 & \longrightarrow c^0(sk_1, pk_N) & \longrightarrow c^1(sk_1, pk_N) & \longrightarrow & \cdots & \longrightarrow c^{2^l-1}(sk_1, pk_N) & = pk_1 \\
\vdots & \vdots & \vdots & & & \vdots & \vdots \\
sk_{N-1} & \longrightarrow c^0(sk_{N-1}, pk_N) & \longrightarrow c^1(sk_{N-1}, pk_N) & \longrightarrow & \cdots & \longrightarrow c^{2^l-1}(sk_{N-1}, pk_N) & = pk_{N-1}
\end{array}$$

WOTS+

$$c^i(x, PK_N) = \begin{cases} x & \text{i=0} \\ H(c^{i-1}(x, pk_N) \oplus pk^i_N) & \text{i>0} \end{cases}$$

$$(r_1, r_2, \cdots, r_{2^l-1}) = pk_N$$

$$sk_0 \longrightarrow c^0(sk_0, pk_N) \longrightarrow c^1(sk_0, pk_N) \longrightarrow \cdots \longrightarrow c^{2^l-1}(sk_0, pk_N) = pk_0$$

$$sk_0$$

$$H(sk_0 \oplus r_1)$$

$$H(H(sk_0 \oplus r_1) \oplus r_2)$$

A WOTS+ signature is generated as follows,

1. Divide message m into N pieces ($m_0 \dots m_{N-1}$) and compute checksum

$$C = \sum_{i=1}^{N-1} (2^l - 1 - m_i)$$

then produce b = m || c

2. Compute signature

$$\sigma = (\sigma_0 \dots \sigma_{N-1}) = [H^{b_0}(sk_0, pk_N), \dots, H^{b_{N-1}}(sk_{N-1}, pk_N)]$$

3. Send signature with random pk_N (pk_N, σ) to the verifier.

XMSS

The eXtended Merkle Tree Signature Scheme

Internet Research Task Force (IRTF)
Request for Comments: 8391
Category: Informational
ISSN: 2070-1721

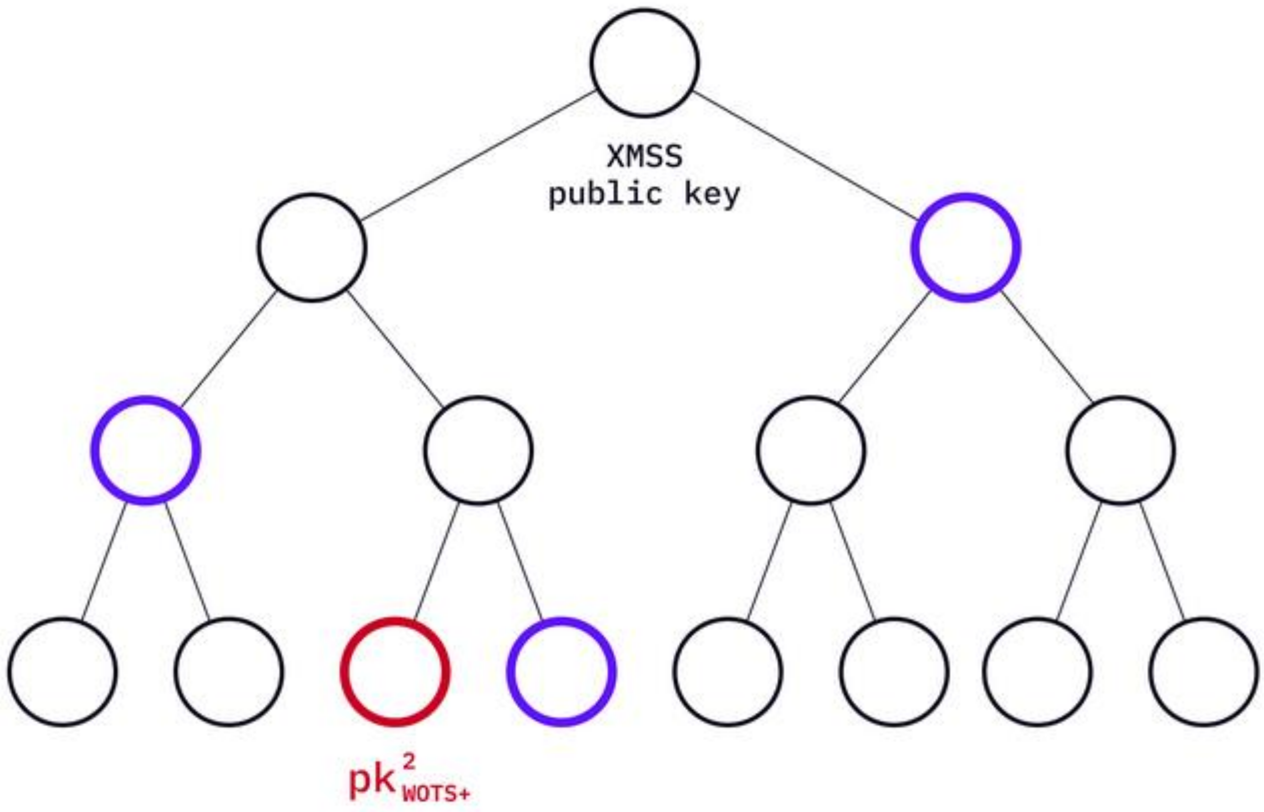
A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua GmbH
J. Rijneveld
Radboud University
A. Mohaisen
University of Central Florida
May 2018

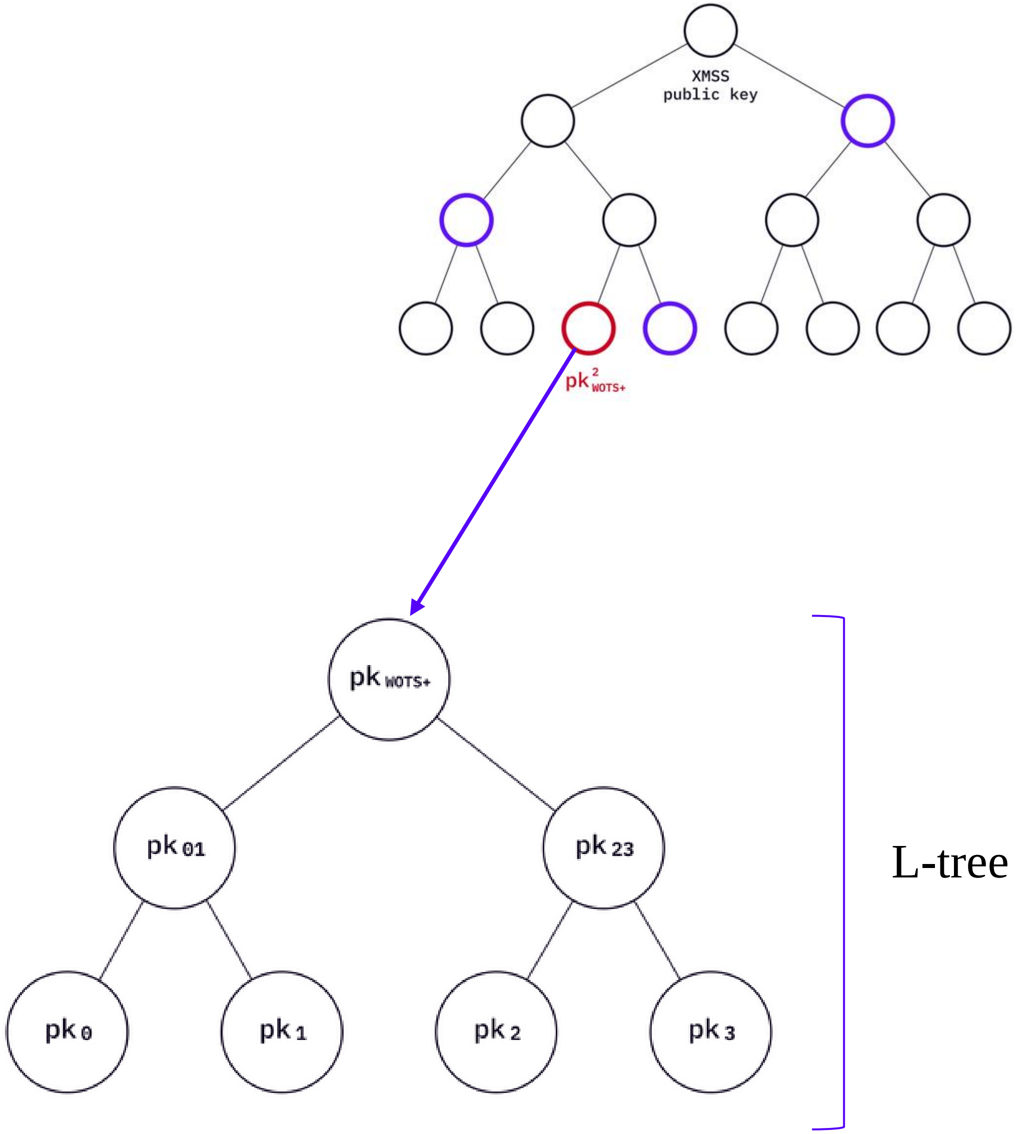
XMSS: eXtended Merkle Signature Scheme

Abstract

This note describes the eXtended Merkle Signature Scheme (XMSS), a hash-based digital signature system that is based on existing descriptions in scientific literature. This note specifies Winternitz One-Time Signature Plus (WOTS+), a one-time signature scheme; XMSS, a single-tree scheme; and XMSS^MT, a multi-tree variant of XMSS. Both XMSS and XMSS^MT use WOTS+ as a main building block. XMSS provides cryptographic digital signatures without relying on the conjectured hardness of mathematical problems. Instead, it is proven that it only relies on the properties of cryptographic hash functions. XMSS provides strong security guarantees and is even secure when the collision resistance of the underlying hash function is broken. It is suitable for compact implementations, is relatively simple to implement, and naturally resists side-channel attacks. Unlike most other signature systems, hash-based signatures can so far withstand known attacks using quantum computers.

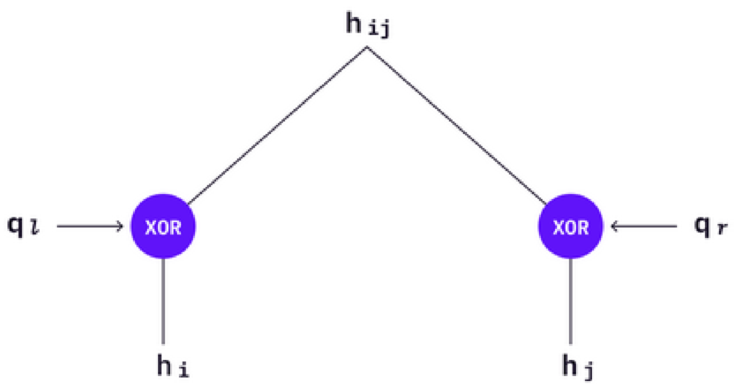
Uses Merkle tree to manage WOTS+ keys



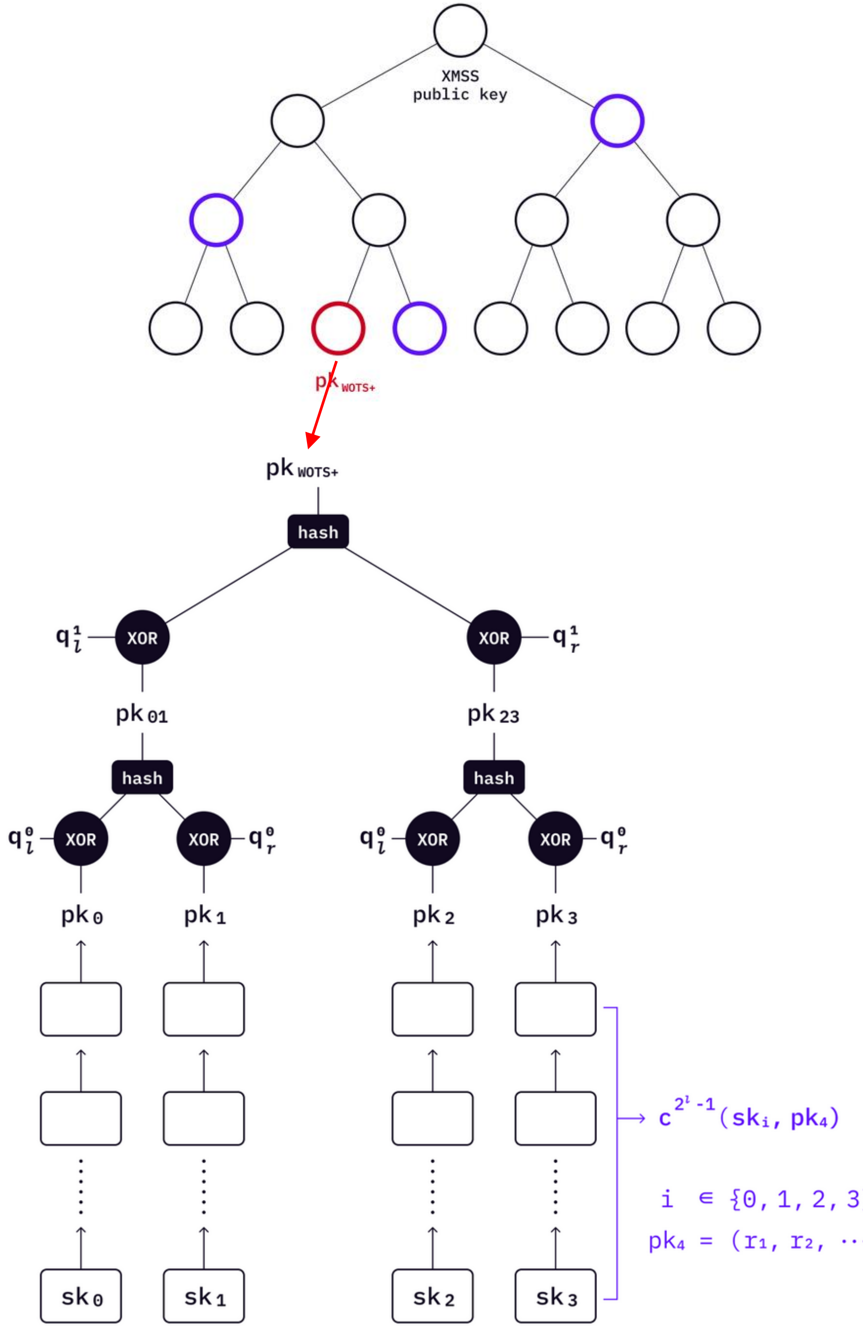

XMSS
public key
pk 2 WOTS+
pk WOTS+
pk 01
pk 23
pk 0
pk 1
pk 2
pk 3
L-tree

- Bitmasks are used in L-trees.

- Before applying the hash function to two nodes each time, two bottom values will XOR with their corresponding bitmasks.



- The leaves of a L-tree are WOTS+ public keys

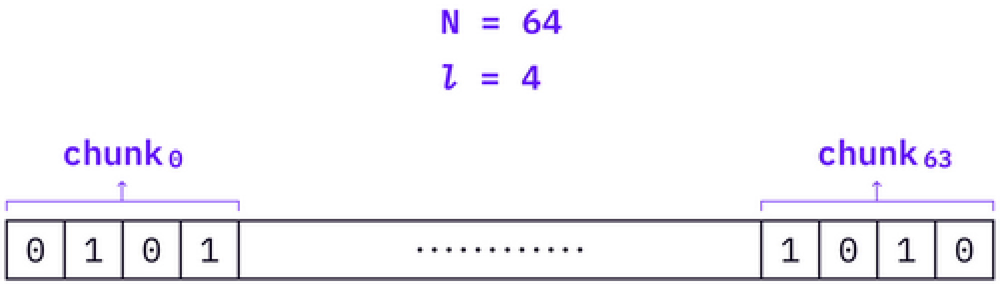

XMSS
public key
pk WOTS+
pk WOTS+
hash
XOR
XOR
pk 01
pk 23
hash
hash
XOR
XOR
XOR
XOR
pk 0
pk 1
pk 2
pk 3
sk 0
sk 1
sk 2
sk 3
i ∈ {0, 1, 2, 3}
pk4 = (r1, r2, ···

HORS - HORST

IDEA: Divide message into N pieces of length l and sign each piece

Generate private keys with PRG

Hash sk and obtain



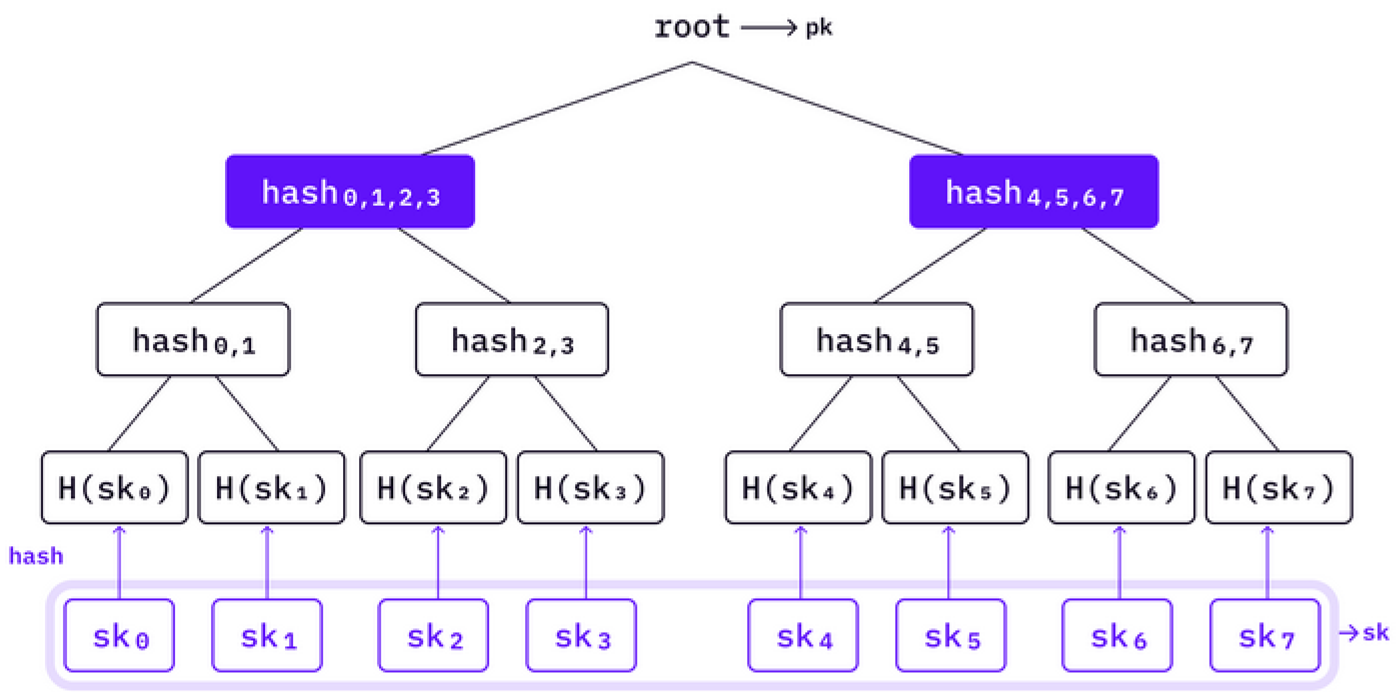
HORS - HORST

Algorithm 3 HORS Algorithm

procedure KEY GENERATION(t)
  Generate the secret key SK at random, $SK = (sk_0, sk_1, \ldots, sk_{t-1})$
  Compute the public key $PK = pk_0, pk_1, \ldots, pk_{t-1} = f(sk_0), f(sk_1), \ldots, f(sk_{t-1})$
  Output $(SK,\ PK)$
end procedure

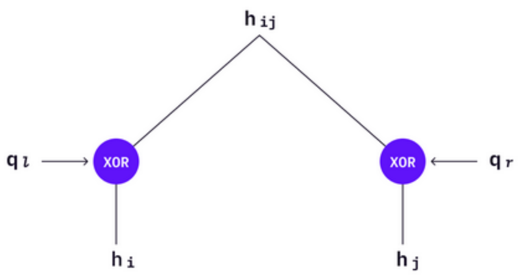
procedure SIGNING(m, κ, SK,k)
  Compute $h = H_k(m)$, $h = h_0||h_1||\ldots||h_{\kappa-1}$.
  $ORS_\kappa(m) = \{h_0, h_1, \ldots, h_{\kappa-1}\}$.
  $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{\kappa-1}) = (sk_{h_0}, sk_{h_1}, \ldots, sk_{h_{\kappa-1}})$
  Output (σ)
end procedure

procedure VERIFICATION(m, κ, σ, PK,k)
  Compute $h = H_k(m)$, $h = h_0||h_1||\ldots||h_{\kappa-1}$
  $ORS_\kappa(m) = \{h_0, h_1, \ldots, h_{\kappa-1}\}$
  for $0 \leq i \leq \kappa - 1$ **do**
    if $f(\sigma_i) = pk_{h_i}$ **then**
      $out = 1$
    else
      $out = 0$
      break
    end if
  end for
  Output (out)
end procedure



root → pk
hash0,1,2,3
hash4,5,6,7
hash0,1
hash2,3
hash4,5
hash6,7
H(sk0)
H(sk1)
H(sk2)
H(sk3)
H(sk4)
H(sk5)
H(sk6)
H(sk7)
hash
sk0
sk1
sk2
sk3
sk4
sk5
sk6
sk7
→sk

Bitmasks are used in HORST

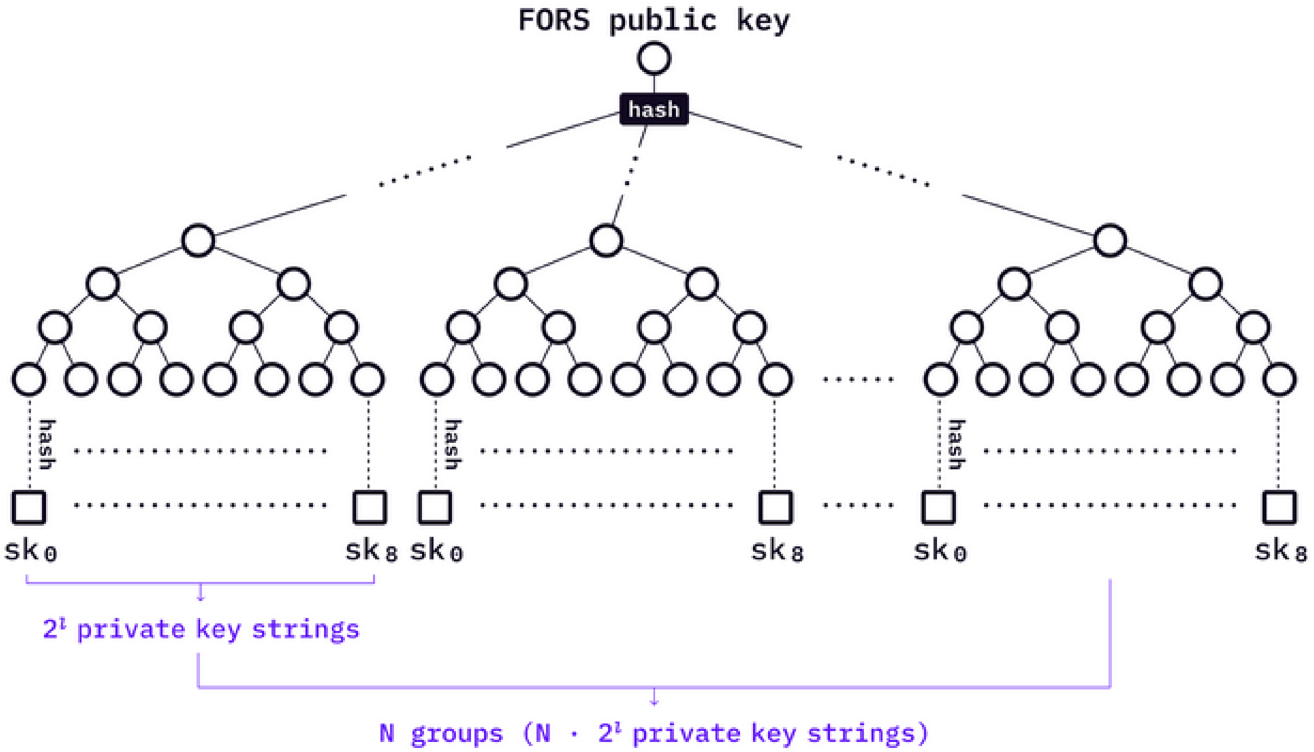


- To sign a message, HORST will select the corresponding private key based on the decimal value of each message chunk.
- Then an authentication path is generated according to the leaf
- For each message group, the signature is
- By doing this, the HORS tree root alone can be used as the public key

FORS

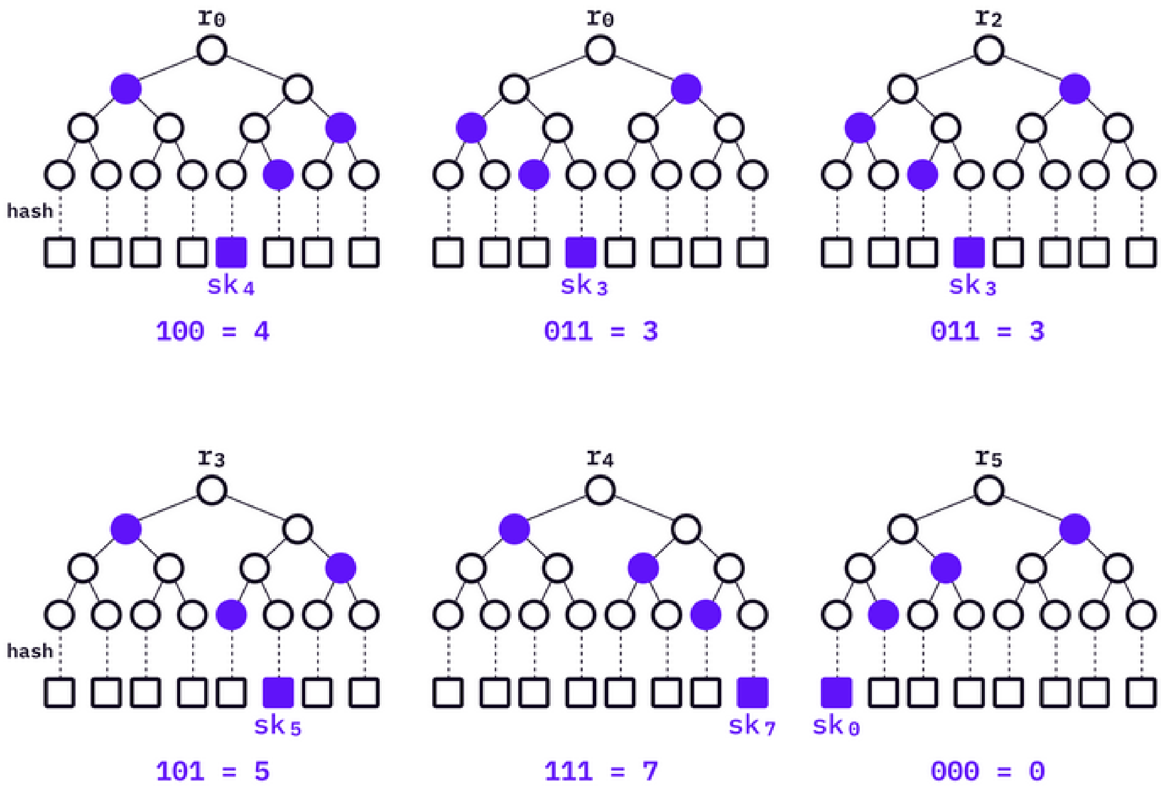
Forest of Random Subsets

FORS

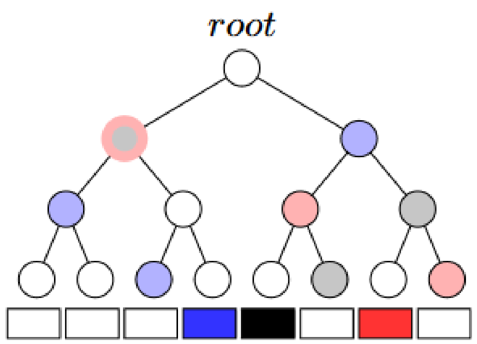


FORS public key
hash
hash
hash
hash
sk0
sk8
sk0
sk8
sk0
sk8
2l private key strings
N groups (N · 2l private key strings)

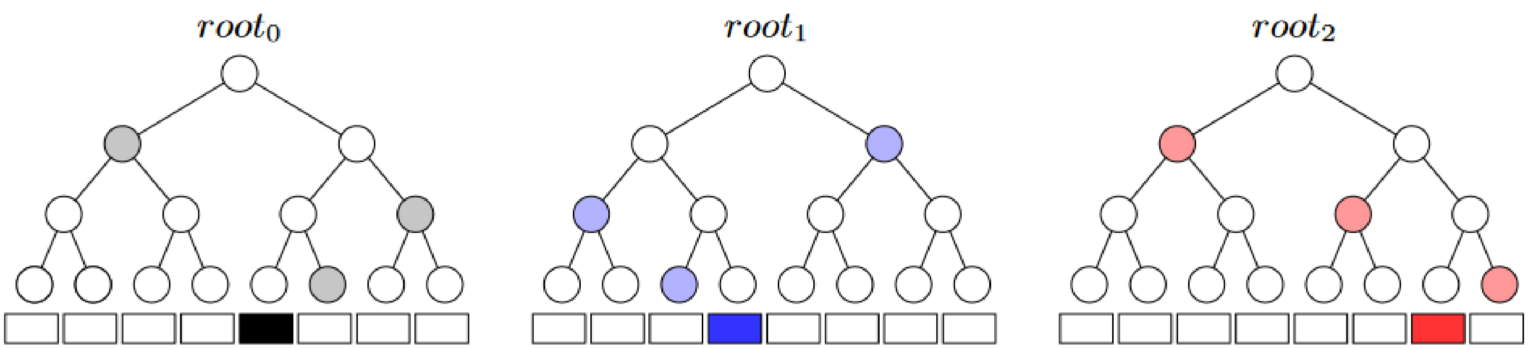
H(m)= 100 011 011 101 111 000



HORS and FORS signatures of the message 100 011 110 where $\kappa = 3$ and $t = 8$.



(a) HORS signature within a binary tree construction



(b) FORS signature within κ binary trees construction

SPHINCS+

Selected Algorithms: Digital Signature Algorithms

Algorithm	Algorithm Information	Submitters	Comments
CRYSTALS-DILITHIUM	Zip File (11MB) IP Statements Website	Vadim Lyubashevsky Leo Ducas Eike Kiltz Tancrede Lepoint Peter Schwabe Gregor Seiler Damien Stehle Shi Bai	Submit Comment View Comments
FALCON	Zip File (4MB) IP Statements Website	Thomas Prest Pierre-Alain Fouque Jeffrey Hoffstein Paul Kirchner Vadim Lyubashevsky Thomas Pornin Thomas Ricosset Gregor Seiler William Whyte Zhenfei Zhang	Submit Comment View Comments
SPHINCS+	Zip File (230MB) IP Statements Website	Andreas Hulsing Daniel J. Bernstein Christoph Dobraunig Maria Eichlseder Scott Fluhrer Stefan-Lukas Gazdag Panos Kampanakis	Submit Comment View Comments

Why eXtended Merkle Signature Scheme (XMSS) not candidates in the NIST Post-Quantum Cryptography Standardization process?

That is because NIST specifically stated that stateful schemes were not allowed in the NIST postquantum competition, because they could not be implemented using the API that NIST has defined (which does not allow any state).

That would appear to be reasonable, as stateful hash based signature methods do need extra care to implement safely; NIST 800-208 outlines what NIST believes are reasonable precautions - those precautions are not needed for other algorithms, and hence NIST kept those separate

Tweakable Hash Functions:

To make each hash call independet we use tweakable hash functions

$$\mathbf{T}_\ell : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{\ell n} \to \mathbb{B}^n,$$ (defined as **T_1)**

$$\mathrm{md} \leftarrow \mathbf{T}_\ell(\mathbf{PK}.\mathbf{seed}, \mathbf{ADRS}, M)$$

$$\mathbf{F} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^n \to \mathbb{B}^n,$$

$$\mathbf{F} \stackrel{\text{def}}{=} \mathbf{T}_1$$

$$\mathbf{H} : \mathbb{B}^n \times \mathbb{B}^{32} \times \mathbb{B}^{2n} \to \mathbb{B}^n$$

$$\mathbf{H} \stackrel{\text{def}}{=} \mathbf{T}_2$$

PRF and Message Digest:

PRF for pseudorandom key generation:

$$\mathbf{PRF} : \mathbb{B}^n \times \mathbb{B}^{32} \rightarrow \mathbb{B}^n.$$

to generate randomness for the message compression:

$$\mathbf{PRF_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \rightarrow \mathbb{B}^n.$$

Keyed hash function that can process arbitrary length messages:

$$\mathbf{H_{msg}} : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B}^* \rightarrow \mathbb{B}^m.$$

Address Scheme:

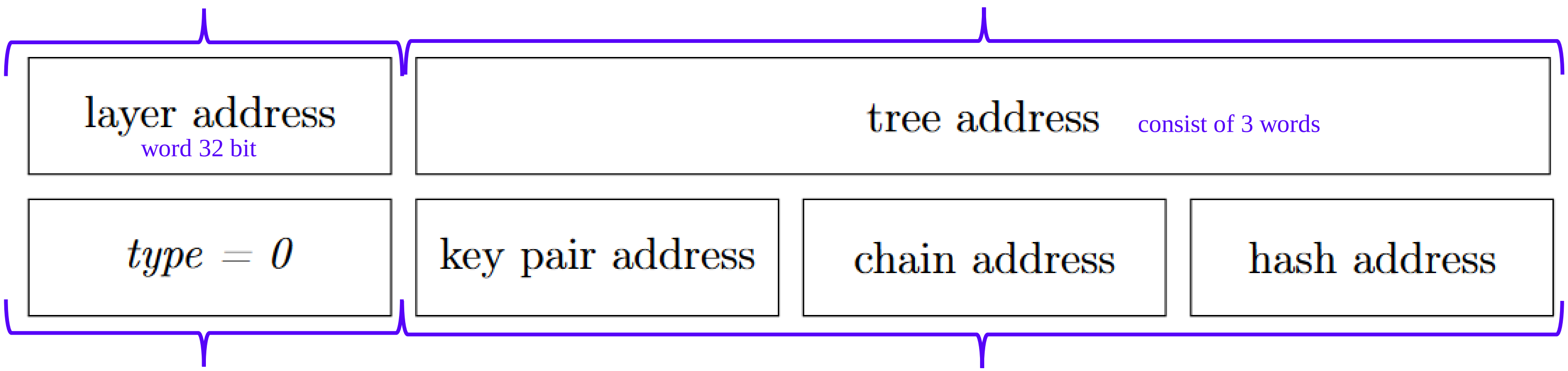
- ADRS is a 32-byte value
- All functions have to keep track of the current context, updating the addresses after each hash call.
- There are 5 different types of addresses

1. Hashes in WOTS+ scheme
2. Compression of the WOTS+ public key
3. Hashes within the main Merkle tree construction
4. Hashes in the Merkle tree in FORS
5. Compression of the tree roots of FORS

Address Scheme:

describes the height of a tree within the hypertree starting from height zero for trees on the bottom layer.

describes the position of a tree within a layer of a multi-tree starting with index zero for the leftmost tree.



defines the type of the address.
0 for a WOTS+ hash address
1 for the compression of the WOTS+ public key
2 for a hash tree address
3 for a FORS address
4 for the compression of FORS tree roots.

changes for each type

Address Scheme:

layer address	tree address		
type = *0*	key pair address	chain address	hash address

Figure 2: WOTS$^+$ hash address.

layer address	tree address	
type = *1*	key pair address	*padding* = *0*

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

layer address	tree address		
type = *2*	*padding* = *0*	tree height	tree index

Figure 4: hash tree address.

Address Scheme:

layer address	tree address		
type = *3*	key pair address	tree height	tree index

Figure 5: FORS tree address.

layer address	tree address	
type = *4*	key pair address	*padding* = *0*

Figure 6: FORS tree roots compression address.

layer address	tree address		
type = *5*	key pair address	chain address	hash address = 0

Figure 7: WOTS^+ key generation address.

Address Scheme:

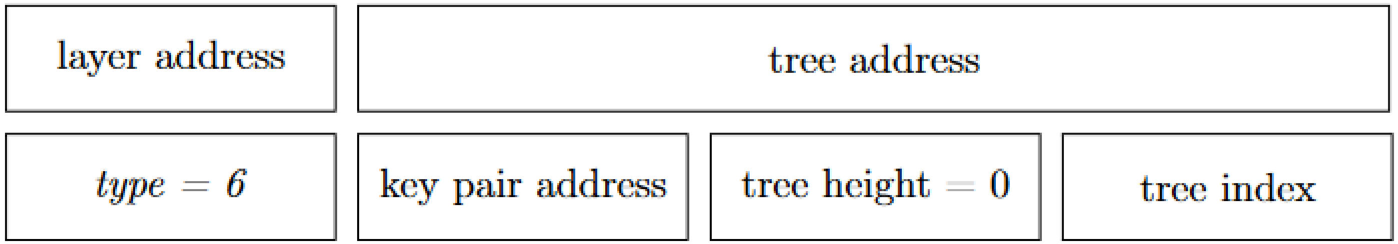


Figure 8: FORS key generation address.

we refer to them respectively using the constants

1. WOTS_HASH
2. WOTS_PK
3. TREE
4. FORS_TREE
5. FORS_ROOTS
6. WOTS_PRF
7. FORS_PRF

WOTS+

WOTS+ uses the parameters n and w; they both take positive integer values.

• n: the security parameter; it is the message length as well as the length of a private key, public key, or signature element in bytes.

• w: the Winternitz parameter; it is an element of the set {4, 16, 256}.

• len: the number of n-byte-string elements in a WOTS+ private key, public key, and signature

$$\text{len} = \text{len}_1 + \text{len}_2$$

$$\texttt{len}_1 = \left\lceil \frac{8n}{\log(w)} \right\rceil, \ \texttt{len}_2 = \left\lfloor \frac{\log\left(\texttt{len}_1(w-1)\right)}{\log(w)} \right\rfloor + 1$$

WOTS+

n determines

- The in-and output length of the tweakable hash function used for WOTS+
- The length of messages that can be processed by the WOTS+ signing algorithm.

w can be chosen from the set {4, 16, 256}

- Larger value of w results in shorter signatures but slower operations; it has no effect on security.
- Choices of w are limited since these values yield optimal trade-offs and easy implementation.

WOTS+

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

chain(X, i, s, PK.seed, ADRS) {
  if ( s == 0 ) {                                    2. when s=0, start hashing
    return X;
  }
  if ( (i + s) > (w - 1) ) {
    return NULL;
  }
  byte[n] tmp = chain(X, i, s - 1, PK.seed, ADRS);   1. Put X inside of S-1 chain

  ADRS.setHashAddress(i + s - 1);
  tmp = F(PK.seed, ADRS, tmp);                       3. iteratively take s-1 hash and return the final hash
  return tmp;
}
```

Algorithm 2: chain – Chaining function used in WOTS^{+}.

WOTS+ Private Keys

```
#Input: secret seed SK.seed, address ADRS
#Output: WOTS+ private key sk

wots_SKgen(SK.seed, ADRS) {
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
    skADRS.setChainAddress(i);
  skADRS.setHashAddress(0);
  sk[i] = PRF(SK.seed, skADRS);
 }
 return sk;
}
```

SkADRS changes the secret key for each chain

Algorithm 3: wots_SKgen – Generating a WOTS$^+$ private key.

WOTS+ Public Key Generation

```
#Input: secret seed SK.seed, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk

wots_PKgen(SK.seed, PK.seed, ADRS) {
  wotspkADRS = ADRS; // copy address to create OTS public key address
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
    skADRS.setChainAddress(i);
    skADRS.setHashAddress(0);
    sk[i] = PRF(SK.seed, skADRS);
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    tmp[i] = chain(sk[i], 0, w - 1, PK.seed, ADRS);
  }
  wotspkADRS.setType(WOTS_PK);
  wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_len(PK.seed, wotspkADRS, tmp);
  return pk;
}
```

Obtain sk from seed and address

Obtain tmp with taking w-1 times hash of sk

Take hash of tmp values and obtain pk

Algorithm 4: wots_PKgen – Generating a WOTS$^+$ public key.

WOTS+ Signature Generation

```
#Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
#Output: WOTS+ signature sig

wots_sign(M, SK.seed, PK.seed, ADRS) {
  csum = 0;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  if( (lg(w) % 8) != 0) {
    csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  }
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);   → Concatanates message and checksum

  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(WOTS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  for ( i = 0; i < len; i++ ) {
    skADRS.setChainAddress(i);
    skADRS.setHashAddress(0);
    sk = PRF(SK.seed, skADRS);   → Produces different sk by changing adress value
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sig[i] = chain(sk, 0, msg[i], PK.seed, ADRS);   → Produces signature with hashing sk msg[i] times
  }
  return sig;
}
```

Algorithm 5: `wots_sign` – Generating a WOTS+ signature on a message M.

WOTS+ Signature Generation

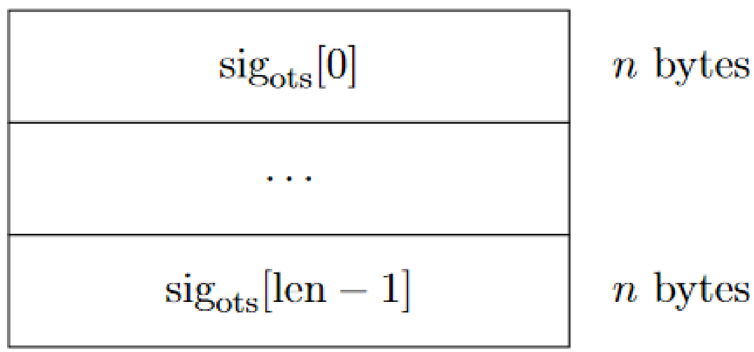


Figure 9: WOTS$^+$ Signature data format.

WOTS+ Public Key Computation from message and signature

```
#Input: Message M, WOTS+ signature sig, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk_sig derived from sig

wots_pkFromSig(sig, M, PK.seed, ADRS) {
  csum = 0;
  wotspkADRS = ADRS;

  // convert message to base w
  msg = base_w(M, w, len_1);

  // compute checksum
  for ( i = 0; i < len_1; i++ ) {
    csum = csum + w - 1 - msg[i];
  }

  // convert csum to base w
  csum = csum << ( 8 - ( ( len_2 * lg(w) ) % 8 ));
  len_2_bytes = ceil( ( len_2 * lg(w) ) / 8 );
  msg = msg || base_w(toByte(csum, len_2_bytes), w, len_2);
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    tmp[i] = chain(sig[i], msg[i], w - 1 - msg[i], PK.seed, ADRS);
  }

  wotspkADRS.setType(WOTS_PK);
  wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk_sig = T_len(PK.seed, wotspkADRS, tmp);
  return pk_sig;
}
```

Hash the signature w-1-msg[i] times

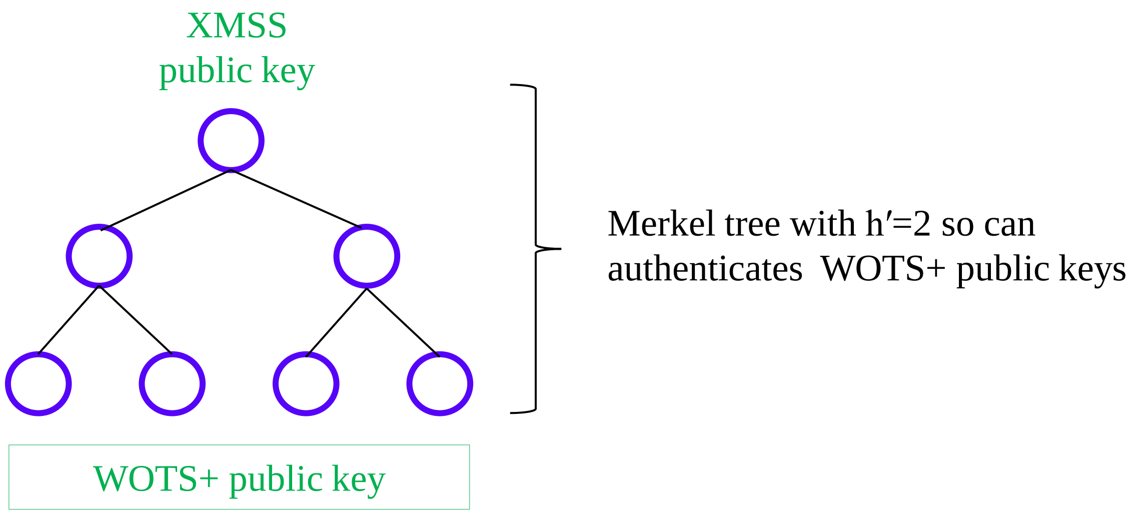
Take final hash and obtain pk

Algorithm 6: `wots_pkFromSig` – Computing a WOTS+ public key from a message and its signature.

XMSS

XMSS has the following parameters:

- h' : the height (number of levels - 1) of the tree.
- n : the length in bytes of messages as well as of each node.
- w : the Winternitz parameter as defined for WOTS+



XMSS

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack

treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       ADRS.setType(WOTS_HASH);
       ADRS.setKeyPairAddress(s + i);
       node = wots_PKgen(SK.seed, PK.seed, ADRS);
       ADRS.setType(TREE);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

→ Must have even number of leafs

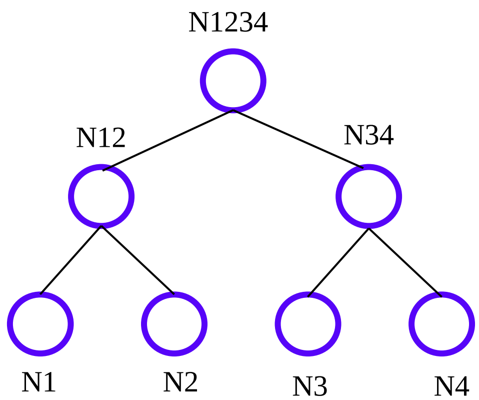
Algorithm 7: treehash – The TreeHash algorithm.

i= 0 Stack={N1}

i= 1 Stack={N1,N12}

i= 2 Stack={N1,N12,N3}

i= 3 Stack={N1,N3,N12,N1234}




```
for ( i = 0; i < 2^z; i++ ) {
  ADRS.setType(WOTS_HASH);
  ADRS.setKeyPairAddress(s + i);
  node = wots_PKgen(SK.seed, PK.seed, ADRS);
  ADRS.setType(TREE);
  ADRS.setTreeHeight(1);
  ADRS.setTreeIndex(s + i);
  while ( Top node on Stack has same height as node ) {
     ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
     node = H(PK.seed, ADRS, (Stack.pop() || node));
     ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
  }
  Stack.push(node);
}
return Stack.pop();
```

Add node to the end of stack

Return to the end of stack and delete it from stack

XMSS

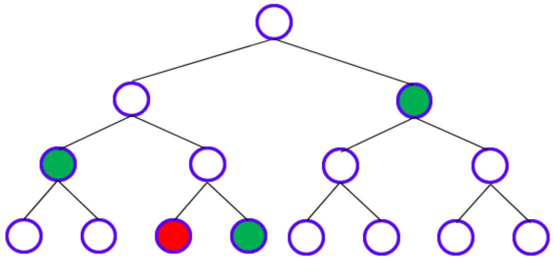
```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: XMSS public key PK

xmss_PKgen(SK.seed, PK.seed, ADRS) {
     pk = treehash(SK.seed, 0, h', PK.seed, ADRS)
     return pk;
}
```

Algorithm 8: xmss_PKgen – Generating an XMSS public key.

XMSS

sig ($\mathtt{len} \cdot n$ bytes)	Signature
AUTH [0] (n bytes)	Authentication node in layer 0
...	
AUTH [h-1] (n bytes)	

Figure 10: XMSS Signature



XMSS

```
# Input: n-byte message M, secret seed SK.seed, index idx, public seed PK.seed,
    address ADRS
# Output: XMSS signature SIG_XMSS = (sig || AUTH)

xmss_sign(M, SK.seed, idx, PK.seed, ADRS)
     // build authentication path
     for ( j = 0; j < h'; j++ ) {
       k = floor(idx / (2^j)) XOR 1;
       AUTH[j] = treehash(SK.seed, k * 2^j, j, PK.seed, ADRS);
     }

     ADRS.setType(WOTS_HASH);
     ADRS.setKeyPairAddress(idx);
     sig = wots_sign(M, SK.seed, PK.seed, ADRS);
     SIG_XMSS = sig || AUTH;
     return SIG_XMSS;
}
```

Algorithm 9: `xmss_sign` – Generating an XMSS signature.

TÜBİTAK BİLGEM

XMSS

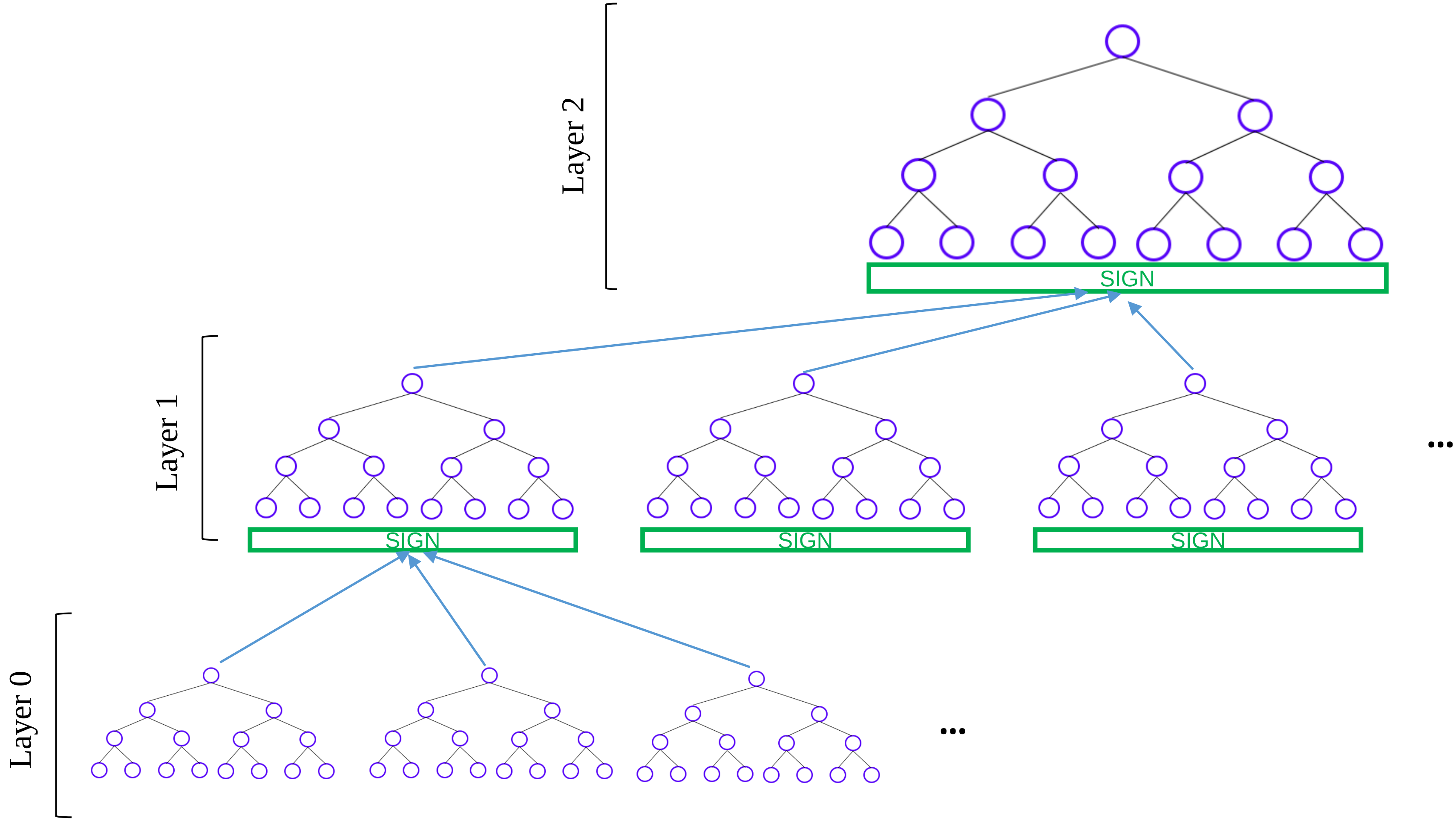
```
# Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte message M,
    public seed PK.seed, address ADRS
# Output: n-byte root value node[0]

xmss_pkFromSig(idx, SIG_XMSS, M, PK.seed, ADRS){

    // compute WOTS+ pk from WOTS+ sig
    ADRS.setType(WOTS_HASH);
    ADRS.setKeyPairAddress(idx);
    sig = SIG_XMSS.getWOTSSig();
    AUTH = SIG_XMSS.getXMSSAUTH();
    node[0] = wots_pkFromSig(sig, M, PK.seed, ADRS);

    // compute root from WOTS+ pk and AUTH
    ADRS.setType(TREE);
    ADRS.setTreeIndex(idx);
    for ( k = 0; k < h'; k++ ) {
      ADRS.setTreeHeight(k+1);
      if ( (floor(idx / (2^k)) % 2) == 0 ) {
        ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
        node[1] = H(PK.seed, ADRS, (node[0] || AUTH[k]));
      } else {
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
        node[1] = H(PK.seed, ADRS, (AUTH[k] || node[0]));
      }
      node[0] = node[1];
    }
    return node[0];
}
```

Changes according to position of node[0] in tree

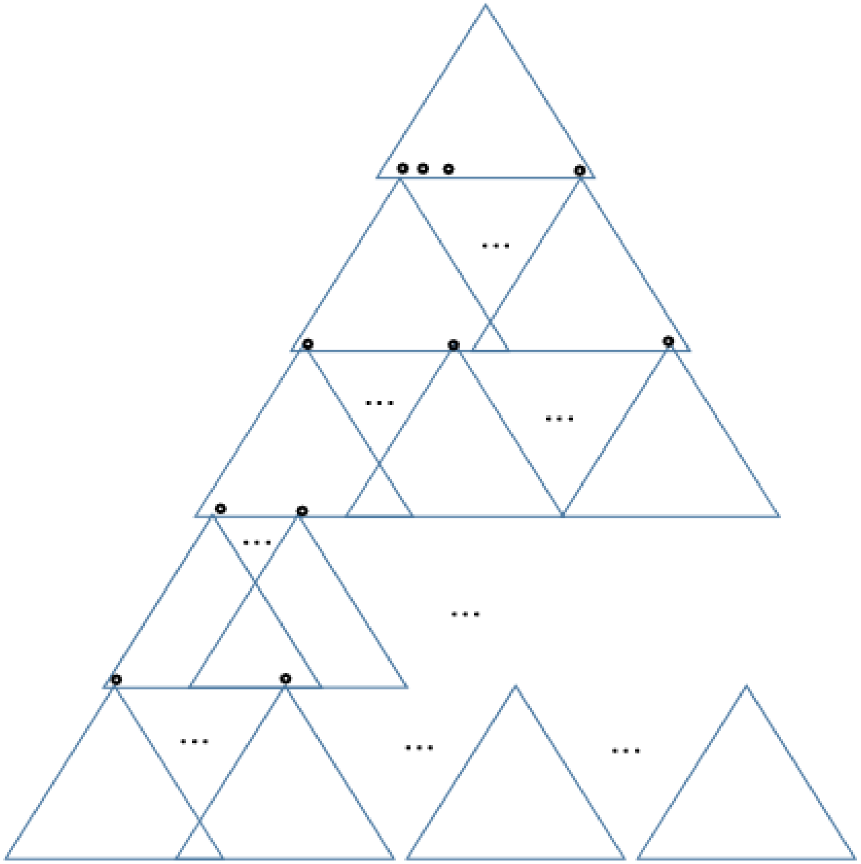
Algorithm 10: `xmss_pkFromSig` – Computing an XMSS public key from an XMSS signature.

The Hypertree

- A HT is a tree of several layers of XMSS trees
- All XMSS trees in HT have equal length
- Consider a HT of total height h that has d layers of XMSS trees of height $h' = h/d$.
- Then layer $d - 1$ contains one XMSS tree, layer $d - 2$ contains $2^{h'}$ XMSS trees, and so on. Finally, layer 0 contains $2^{h-h'}$ XMSS trees



The Hypertree of height 9 with 3 layers consist of XMSS trees with height 3



The Hypertree

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

Figure 11: HT signature

The Hypertree

```
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT

ht_PKgen(SK.seed, PK.seed){
      ADRS = toByte(0, 32);  →  32 byte consist of 0
      ADRS.setLayerAddress(d-1);
      ADRS.setTreeAddress(0);
      root = xmss_PKgen(SK.seed, PK.seed, ADRS);
      return root;
}
```

Algorithm 11: `ht_PKgen` – Generating an HT public key.

The Hypertree

```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index
    idx_tree, leaf index idx_leaf
# Output: HT signature SIG_HT

ht_sign(M, SK.seed, PK.seed, idx_tree, idx_leaf) {
     // init
     ADRS = toByte(0, 32);

     // sign
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     SIG_tmp = xmss_sign(M, SK.seed, idx_leaf, PK.seed, ADRS);
     SIG_HT = SIG_HT || SIG_tmp;
     root = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx_tree;
        idx_tree = (h - (j + 1) * (h / d)) most significant bits of idx_tree;
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idx_tree);
        SIG_tmp = xmss_sign(root, SK.seed, idx_leaf, PK.seed, ADRS);
        SIG_HT = SIG_HT || SIG_tmp;
        if ( j < d - 1 ) {
           root = xmss_pkFromSig(idx_leaf, SIG_tmp, root, PK.seed, ADRS);
        }
     }
     return SIG_HT;
}
```

Sign the message with the lower XMSS tree

Continue signing in the hypertree up to layer d-1

Return all signatures

Algorithm 12: ht_sign – Generating an HT signature

TÜBİTAK BİLGEM

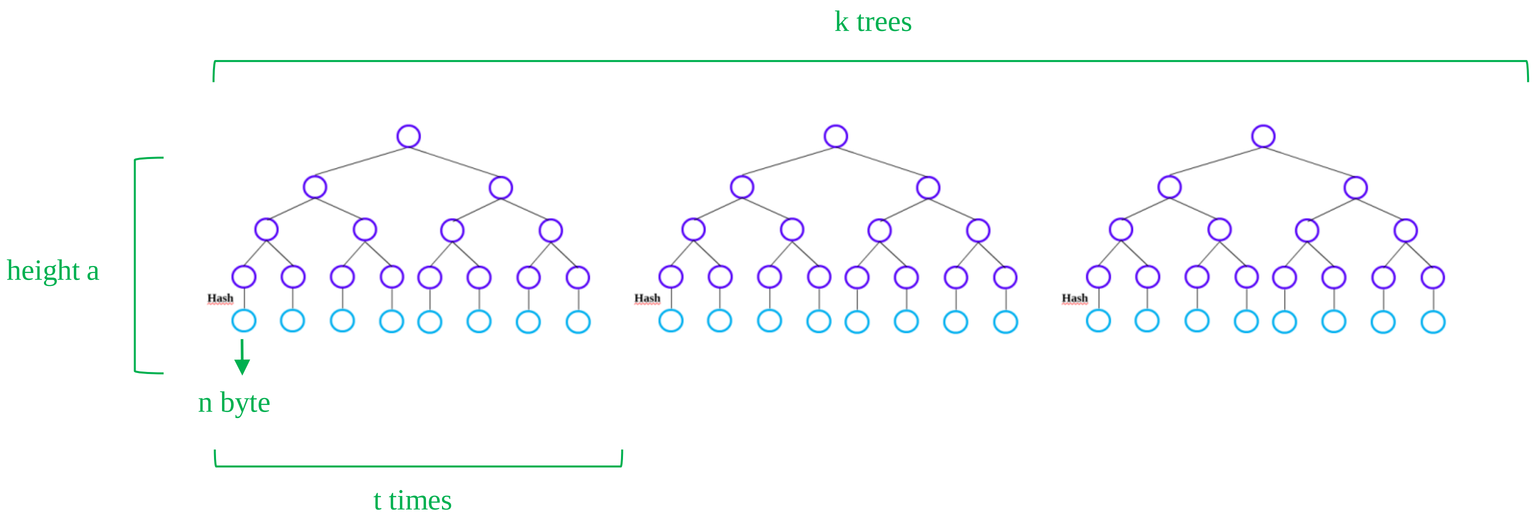
The Hypertree

```
# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree,
    leaf index idx_leaf, HT public key PK_HT.
# Output: Boolean

ht_verify(M, SIG_HT, PK.seed, idx_tree, idx_leaf, PK_HT){
     // init
     ADRS = toByte(0, 32);

     // verify
     SIG_tmp = SIG_HT.getXMSSSignature(0);  ⟶ Get signature of layer 0 XMSS
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     node = xmss_pkFromSig(idx_leaf, SIG_tmp, M, PK.seed, ADRS);  ⟶ Compute pk of layer 0 XMSS
     for ( j = 1; j < d; j++ ) {
        idx_leaf = (h / d) least significant bits of idx_tree;
        idx_tree = (h - (j + 1) * h / d) most significant bits of idx_tree;
        SIG_tmp = SIG_HT.getXMSSSignature(j);  ⟶ Get signature of layer j XMSS
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idx_tree);
        node = xmss_pkFromSig(idx_leaf, SIG_tmp, node, PK.seed, ADRS);  ⟶ Compute pk of layer j XMSS
     }
     if ( node == PK_HT ) {  ⟶ If final node is equal to PK_HT, then signature is valid
       return true;
     } else {
       return false;
     }
}
```

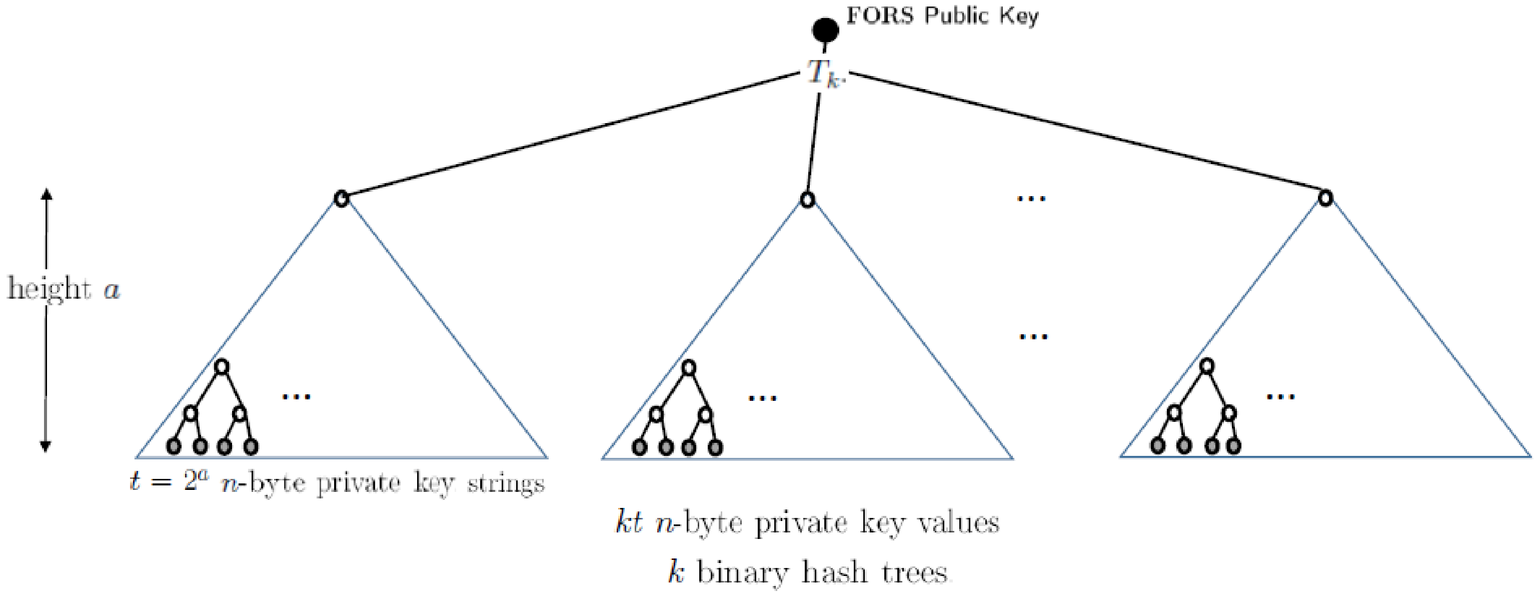
Algorithm 13: `ht_verify` – Verifying a HT signature $\mathbf{SIG}_{\mathbf{HT}}$ on a message M using a HT public key $\mathbf{PK}_{\mathbf{HT}}$

FORS

- Uses parameters k and $t = 2^a$
- Signs strings of length ka bits
- Privite key consist kt many random n byte strings
- Public key n byte value computed as a hash of root node of k hash trees



FORS


FORS Public Key
T_k
...
height a
$t = 2^a$ n-byte private key strings
kt n-byte private key values
k binary hash trees

FORS

```
#Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
#Output: FORS private key sk

fors_SKgen(SK.seed, ADRS, idx) {
  skADRS = ADRS; // copy address to create key generation address
  skADRS.setType(FORS_PRF);
  skADRS.setKeyPairAddress(ADRS.getKeyPairAddress());

  skADRS.setTreeHeight(0);
  skADRS.setTreeIndex(idx);
  sk = PRF(SK.seed, skADRS);  → Obtain sk from SK.seed

  return sk;  → Returns just 1 sk not array of sk's
}
```

Algorithm 14: fors_SKgen – Computing a FORS private key value.

FORS

Almost same to treehash in XMSS

```
# Input: Secret seed SK.seed, start index s, target node height z, public seed
    PK.seed, address ADRS
# Output: n-byte root node - top node on Stack

fors_treehash(SK.seed, s, z, PK.seed, ADRS) {
     if( s % (1 << z) != 0 ) return -1;
     for ( i = 0; i < 2^z; i++ ) {
       sk = fors_SKgen(SK.seed, ADRS, s+i)
       node = F(PK.seed, ADRS, sk);
       ADRS.setTreeHeight(1);
       ADRS.setTreeIndex(s + i);
       while ( Top node on Stack has same height as node ) {
          ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
          node = H(PK.seed, ADRS, (Stack.pop() || node));
          ADRS.setTreeHeight(ADRS.getTreeHeight() + 1);
       }
       Stack.push(node);
     }
     return Stack.pop();
}
```

Algorithm 15: The `fors_treehash` algorithm.

FORS

```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK

fors_PKgen(SK.seed, PK.seed, ADRS) {
     forspkADRS = ADRS; // copy address to create FTS public key address

     for(i = 0; i < k; i++){
         root[i] = fors_treehash(SK.seed, i*t, a, PK.seed, ADRS);
     }
     forspkADRS.setType(FORS_ROOTS);
     forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
     pk = T_k(PK.seed, forspkADRS, root);
     return pk;
}
```

Compute top node of each tree

Take hash of the roots and obtain pk

Algorithm 16: fors_PKgen – Generate a FORS public key.

FORS

```
#Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
#Output: FORS signature SIG_FORS

fors_sign(M, SK.seed, PK.seed, ADRS) {
  // compute signature elements
  for(i = 0; i < k; i++){
    // get next index
    unsigned int idx = bits i*log(t) to (i+1)*log(t) - 1 of M;

    // pick private key element
    SIG_FORS = SIG_FORS || fors_SKgen(SK.seed, ADRS, i*t + idx) ;

    // compute auth path
    for ( j = 0; j < a; j++ ) {
      s = floor(idx / (2^j)) XOR 1;
      AUTH[j] = fors_treehash(SK.seed, i * t + s * 2^j, j, PK.seed, ADRS);
    }
    SIG_FORS = SIG_FORS || AUTH;
  }
  return SIG_FORS;
}
```

Take corresponding sk of message chunk

Algorithm 17: `fors_sign` – Generating a FORS signature on string M.

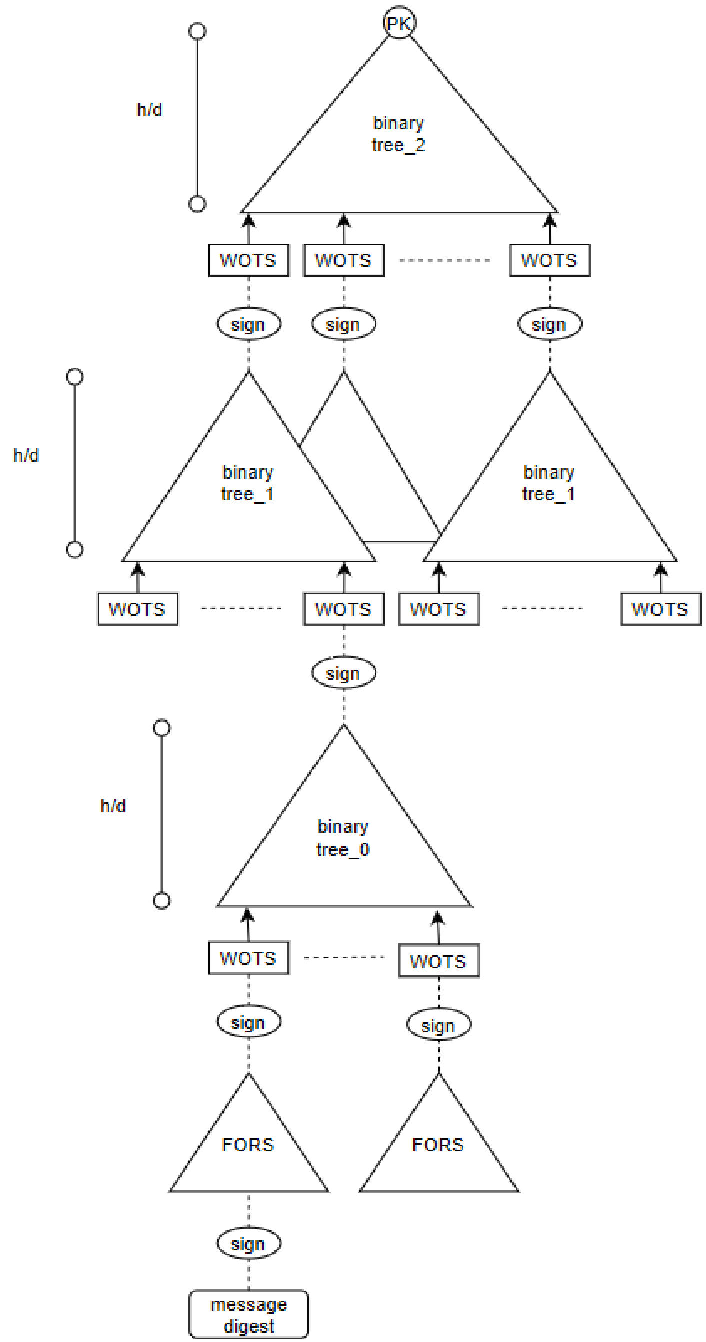
FORS

Private key value (tree 0) (n bytes)
AUTH (tree 0) ($\log t \cdot n$ bytes)
...
Private key value (tree $k-1$) (n bytes)
AUTH (tree $k-1$) ($\log t \cdot n$ bytes)

Figure 13: FORS signature

```
# Input: FORS signature SIG_FORS, (k lg t)-bit string M, public seed PK.seed,
   address ADRS
# Output: FORS public key

fors_pkFromSig(SIG_FORS, M, PK.seed, ADRS){

  // compute roots
  for(i = 0; i < k; i++){          Do for all trees
    // get next index
    unsigned int idx = bits i*log(t) to (i+1)*log(t) - 1 of M;

    // compute leaf
    sk = SIG_FORS.getSK(i);                      Get sk
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    node[0] = F(PK.seed, ADRS, sk);          Take hash of sk and enter the tree

    // compute root from leaf and AUTH
    auth = SIG_FORS.getAUTH(i);
    ADRS.setTreeIndex(i*t + idx);
    for ( j = 0; j < a; j++ ) {
      ADRS.setTreeHeight(j+1);
      if ( (floor(idx / (2^j)) % 2) == 0 ) {
        ADRS.setTreeIndex(ADRS.getTreeIndex() / 2);
        node[1] = H(PK.seed, ADRS, (node[0] || auth[j]));
      } else {                                                  Hash sk with corresponding layer auth
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2);
        node[1] = H(PK.seed, ADRS, (auth[j] || node[0]));
      }
      node[0] = node[1];          Give result to the node[0] and cont. with next auth.
    }
    root[i] = node[0];          Add root of i'th tree to the array
  }

  forspkADRS = ADRS; // copy address to create FTS public key address
  forspkADRS.setType(FORS_ROOTS);
  forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
  pk = T_k(PK.seed, forspkADRS, root);          Take hash of the root values
  return pk;
}
```

Algorithm 18: `fors_pkFromSig` – Compute a FORS public key from a FORS signature.


PK
h/d
binary tree_2
WOTS
WOTS
WOTS
sign
sign
sign
h/d
binary tree_1
binary tree_1
WOTS
WOTS
WOTS
WOTS
sign
h/d
binary tree_0
WOTS
WOTS
sign
sign
FORS
FORS
sign
message digest

PARAMETERS

n : the security parameter in bytes.

w : the Winternitz parameter

h : the height of the hypertree

d : the number of layers in the hypertree

k : the number of trees in FORS

t : the number of leaves of a FORS tree

```
# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)

spx_keygen( ){
     SK.seed = sec_rand(n);
     SK.prf = sec_rand(n);
     PK.seed = sec_rand(n);
     PK.root = ht_PKgen(SK.seed, PK.seed);
     return ( (SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root) );
}
```

Algorithm 19: spx_keygen – Generate a SPHINCS$^+$ key pair.

SPHINCS+

Used in generation randomization value for randomized message hash

SK.`seed` (n bytes)
SK.`prf` (n bytes)
PK.`seed` (n bytes)
PK.`root` (n bytes)

SPHINCS+ secret key

Used in generation WOTS+ and FORS privete keys.

PK.`seed` (n bytes)
PK.`root` (n bytes)

SPHINCS+ public key

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG

spx_sign(M, SK){
    // init
    ADRS = toByte(0, 32);  ──→ Give 0 to the adress

    // generate randomizer
    opt = PK.seed;
    if(RANDOMIZE){
      opt = rand(n);
    }                        Gives option so that signiture of same message will be different in each sign
    R = PRF_msg(SK.prf, opt, M);  ──→ Generate random number R
    SIG = SIG || R;

    // compute message digest and index
    digest = H_msg(R, PK.seed, PK.root, M);
    tmp_md = first floor((ka +7)/ 8) bytes of digest;
    tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
    tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;     split digest into 3 parts

    md = first ka bits of tmp_md;
    idx_tree = first h - h/d bits of tmp_idx_tree;
    idx_leaf = first h/d bits of tmp_idx_leaf;
```

```
    // FORS sign
    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idx_tree);
    ADRS.setType(FORS_TREE);
    ADRS.setKeyPairAddress(idx_leaf);

    SIG_FORS = fors_sign(md, SK.seed, PK.seed, ADRS);
    SIG = SIG || SIG_FORS;

    // get FORS public key
    PK_FORS = fors_pkFromSig(SIG_FORS, md, PK.seed, ADRS);

    // sign FORS public key with HT
    ADRS.setType(TREE);
    SIG_HT = ht_sign(PK_FORS, SK.seed, PK.seed, idx_tree, idx_leaf);
    SIG = SIG || SIG_HT;

    return SIG;
}
```

Algorithm 20: `spx_sign` – Generating a SPHINCS$^+$ signature

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

SPHINCS+ signature

SPHINCS+ signature verification:

1. recomputing message digest and index,
2. computing a candidate FORS public key
3. verifying the HT signature on that public key.

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean

spx_verify(M, SIG, PK){
     // init
     ADRS = toByte(0, 32);
     R = SIG.getR();
     SIG_FORS = SIG.getSIG_FORS();          Split signature
     SIG_HT = SIG.getSIG_HT();


     // compute message digest and index
     digest = H_msg(R, PK.seed, PK.root, M);
     tmp_md = first floor((ka +7)/ 8) bytes of digest;
     tmp_idx_tree = next floor((h - h/d +7)/ 8) bytes of digest;
     tmp_idx_leaf = next floor((h/d +7)/ 8) bytes of digest;

     md = first ka bits of tmp_md;
     idx_tree = first h - h/d bits of tmp_idx_tree;
     idx_leaf = first h/d bits of tmp_idx_leaf;

     // compute FORS public key
     ADRS.setLayerAddress(0);
     ADRS.setTreeAddress(idx_tree);
     ADRS.setType(FORS_TREE);
     ADRS.setKeyPairAddress(idx_leaf);
     PK_FORS = fors_pkFromSig(SIG_FORS, md, PK.seed, ADRS);

     // verify HT signature
     ADRS.setType(TREE);
     return ht_verify(PK_FORS, SIG_HT, PK.seed, idx_tree, idx_leaf, PK.root);
}
```

Algorithm 21: `spx_verify` – Verify a SPHINCS$^+$ signature **SIG** on a message M using a SPHINCS$^+$ public key **PK**

NIST security level 1		Size(bytes)		Relative time	
		Public key	Signature	Verification	Signing
Non PQ	NIST p-256 ECDSA	64	64	1(base)	1(base)
	RSA-2048	256	256	0.2	25
NIST selected algortihms	Dilithium2	1320	2420	0.3	2.5
	Falcon512	897	666	0.3	5
	SPHINC+-128s har.	32	7856	1.7	3000
	SPHINC+-128f har.	32	17088	4	200
Others,	XMSS-SHAKE_20_128 (can sign 1000000 messages)	32	900	2	10

https://blog.cloudflare.com/sizing-up-post-quantum-signatures/

Table 1: Overview of the number of function calls we require for each operation. We omit the single calls to $\mathbf{H}_{\mathbf{msg}}$, $\mathbf{PRF}_{\mathbf{msg}}$, and $\mathbf{T}_k$ for signing and single calls to $\mathbf{H}_{\mathbf{msg}}$ and $\mathbf{T}_k$ for verification as they are negligible when estimating speed.

	$\mathbf{F}$	$\mathbf{H}$	$\mathbf{PRF}$	$T_{\mathtt{len}}$
Key Generation	$2^{h/d}w\mathtt{len}$	$2^{h/d}-1$	$2^{h/d}\mathtt{len}$	$2^{h/d}$
Signing	$kt+d(2^{h/d})w\mathtt{len}$	$k(t-1)+d(2^{h/d}-1)$	$kt+d(2^{h/d})\mathtt{len}$	$d2^{h/d}$
Verification	$k+dw\mathtt{len}$	$k\log t+h$	–	d

Table 2: Key and signature sizes

	SK	PK	Sig
Size	$4n$	$2n$	$(h+k(\log t+1)+d\cdot len+1)n$

small fast

	n	h	d	$\log(t)$	k	w	bitsec	sec level	sig bytes
SPHINCS$^+$-128s	16	63	7	12	14	16	133	**1**	7 856
SPHINCS$^+$-128f	16	66	22	6	33	16	128	**1**	17 088
SPHINCS$^+$-192s	24	63	7	14	17	16	193	**3**	16 224
SPHINCS$^+$-192f	24	66	22	8	33	16	194	**3**	35 664
SPHINCS$^+$-256s	32	64	8	14	22	16	255	**5**	29 792
SPHINCS$^+$-256f	32	68	17	9	35	16	255	**5**	49 856