Karlsruher Institut für Technologie

SPHINCS$^+$ – from stateful building blocks to stateless signatures

Seminar: Post-Quantum Cryptography (WS23/24) | Supervisor: Roland Gröll

Patrick Spiesberger | uvwut@student.kit.edu

Today's Learning Goals

- Understand the SPHINCS approach
- Few-time signatures and stateless signatures - Why are they so good?
- Understand Merkle- and Hyper-trees in detail
- Recognise the small but important difference between SPHINCS and SPHINCS$^+$
- Understand the security features of SPHINCS$^+$ and transfer them to SPHINCS

Motivation | NIST PQ Competition

- In 2016: NIST initiated a competition to standardize a quantum-resistant cryptographic algorithm

- Only a “few” requirements:
  - stateless
  - at least 2^{64} securely signings with one key pair

  ⇒ ensure that signature are “drop-in replacements” for existing signatures

- Alternative candidate to CRYSTALS-Dilithium in NIST PQ competition

The SPHINCS Approach | HORS

- Last Week: One-time signatures → Today: Few-time signatures (FTS)

  ⇒ With each usage, a certain amount of information is disclosed, reducing the security of the key

  ⇒ FTS-Algorithm: Hash to Obtain Random Subset (HORS)

The SPHINCS Approach | HORS

- Last Week: One-time signatures → Today: Few-time signatures (FTS)

  ⇒ With each usage, a certain amount of information is disclosed, reducing the security of the key

  ⇒ FTS-Algorithm: Hash to Obtain Random Subset (HORS)

- HORS divides a hashed message into N chunks of l length

[1]

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORS

- HORS divides a hashed message into N chunks of l length

[1]

- $sk = (sk_0, sk_1, \ldots sk_{t-1})$ (PRG)

  (Hashing) ⇒ $pk = (pk_0, pk_1, \ldots, pk_{t-1})$

- compute signature with permutation of *i'th* chunk

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORS

- HORS divides a hashed message into N chunks of l length

[1]

- $sk = (sk_0, sk_1, \ldots sk_{t-1})$ (PRG)

  (Hashing) ⇒ $pk = (pk_0, pk_1, \ldots, pk_{t-1})$
- compute signature with permutation of *i'th* chunk

- H(m) splitted into 64 groups
- 2^4 private keys $(sk_1, \ldots, sk_{16})$
- Example: $\sigma_1 = sk_{m_1} = sk_5$

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORS

- $sk = (sk_0, sk_1, \ldots sk_{t-1})$ (PRG)

  (Hashing) $\Rightarrow pk = (pk_0, pk_1, \ldots, pk_{t-1})$

- compute signature with permutation of *i'th* chunk

Signing

Verification

- Hash each signature chunk one time
- Check if the calculated chunk is elements of the public key
- All verifications succeeded? ⇒ signature is valid

The SPHINCS Approach | HORS - Summary [2]

- Very fast, simple and short signatures
- As FTS scheme, reusing the keys is “okay” ⇒ But why?
- Very large public and private keys
  ⇒ but easy to calculate (using PRG and hashing)

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

Requirements of a few-times signature [2]

- Question: when can we spoof the signature of a hash?

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

Requirements of a few-times signature [2]

- Question: when can we spoof the signature of a hash?
- Answer: in case that every chunk has been hashed before

- First signature: preimage attack on public keys (… negligible)
- After many signatures: find a message M where hash(M) = $(x_0, x_1, \ldots, x_{k-1})$ are known from previous signatures
  ⇒ Learning goal: randomize hashes!

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

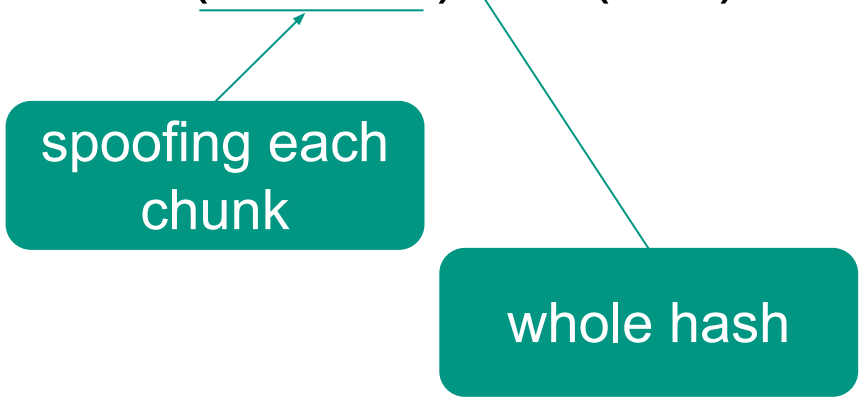
HORS as few-time signature scheme [2]

- Example: 4 messages already signed with HORS keys *(N = 16, l = 16)*

⇒ 16 x 4 = 64 secret values are known (ignoring overlaps)

- Spoofing a 16-chunk signature for a random message

  ⇒ $(64 / 2^{16})^{16} = (2^{-10})^{16} = 2^{-160}$



[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

HORS as few-time signature scheme [2]

- Example: 4 messages already signed with HORS keys *(N = 16, l = 16)*

⇒ 16 x 4 = 64 secret values are known (ignoring overlaps)

- Spoofing a 16-bit signature for a random message

⇒ $(64 / 2^{16})^{16} = (2^{-10})^{16} = 2^{-160}$

spoofing each chunk

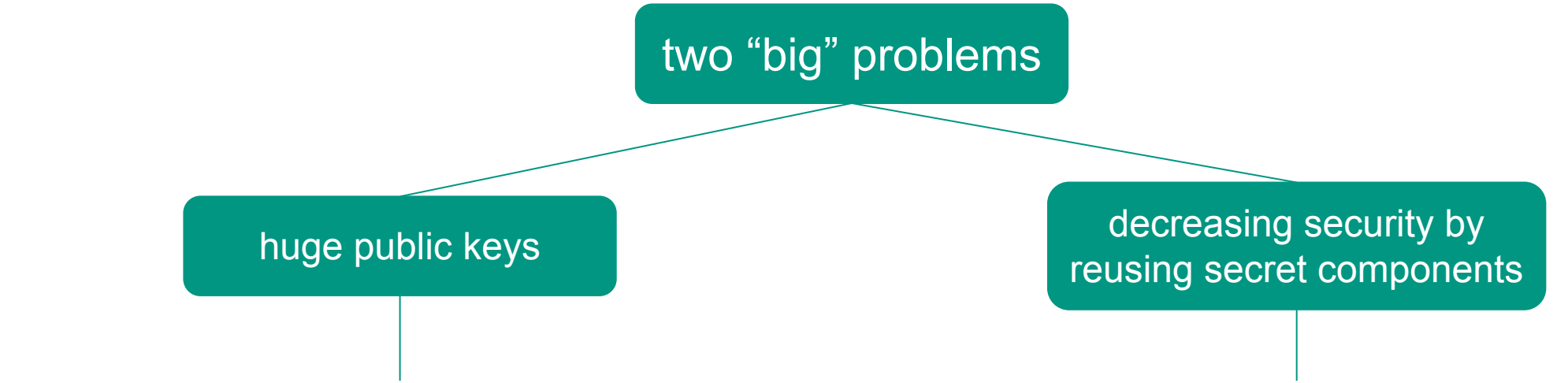
whole hash

⇒ signing 4 (... or 40) messages with same key is not really critical

… sounds good, doesn't it?

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

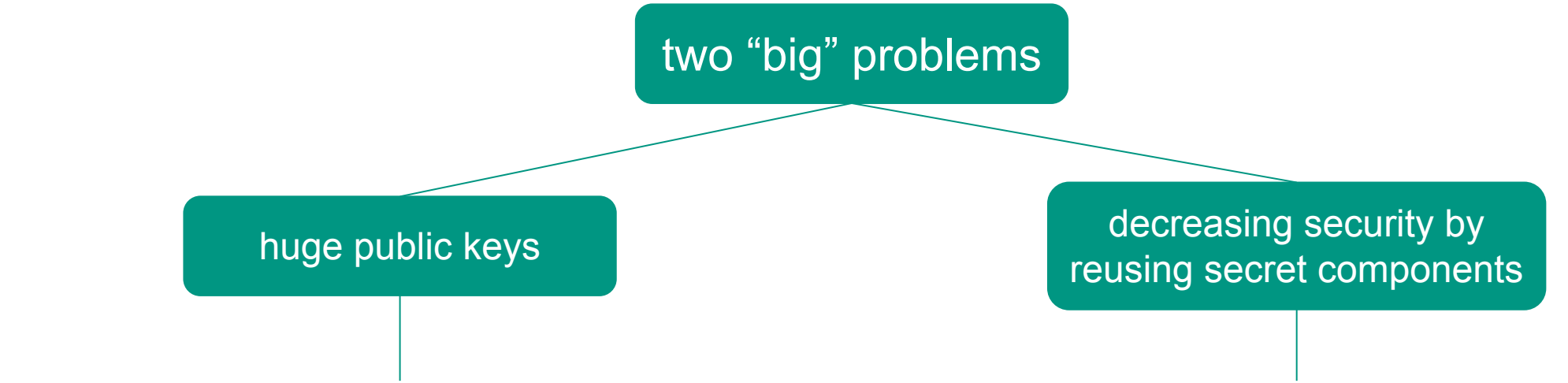
HORS as few-time signature scheme [2]



- Solution?

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

HORS as few-time signature scheme [2]



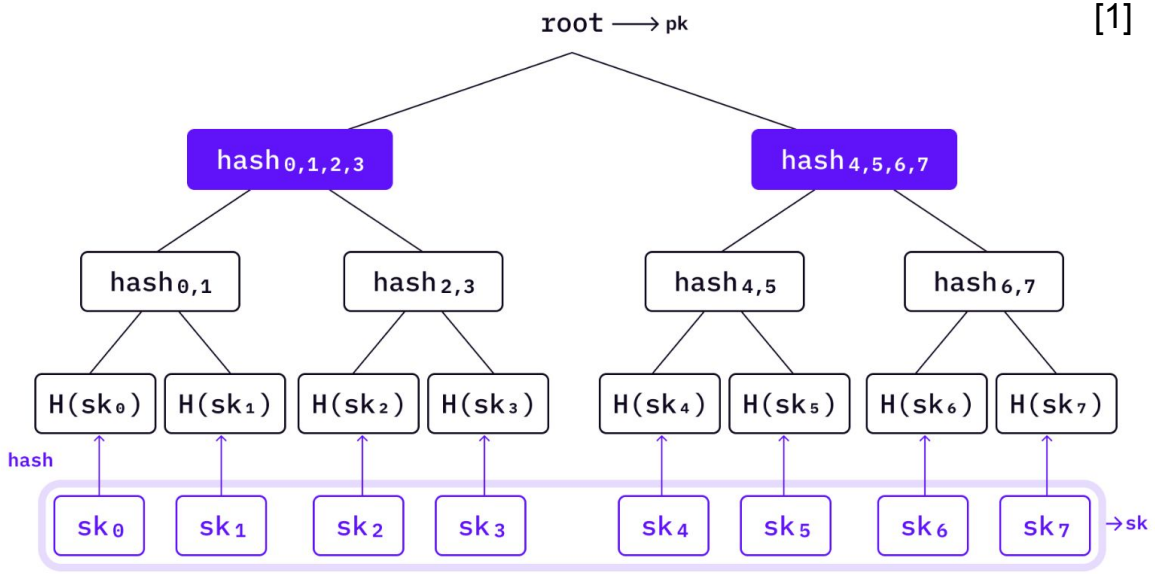
- Solution? Already seen in this slide! ⇒ HORSTrees
  (... and later FORS)

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | HORS → HORST

- HORS public keys are very long ⇒ Reduce size with “Merkle-trees”

[1]

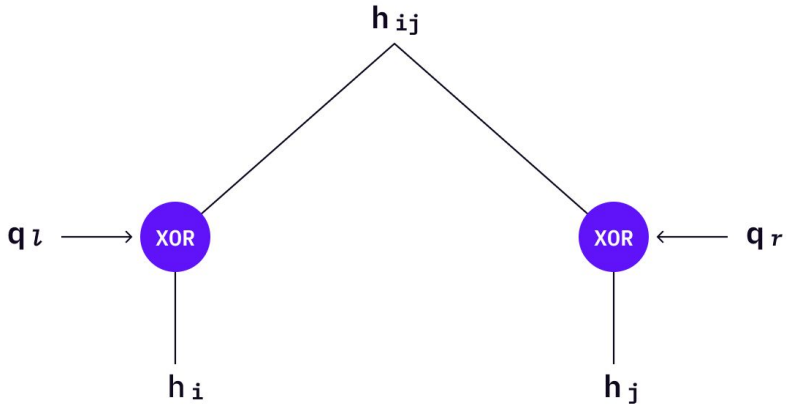


- Signature is more expensive, but only one hash value as public key

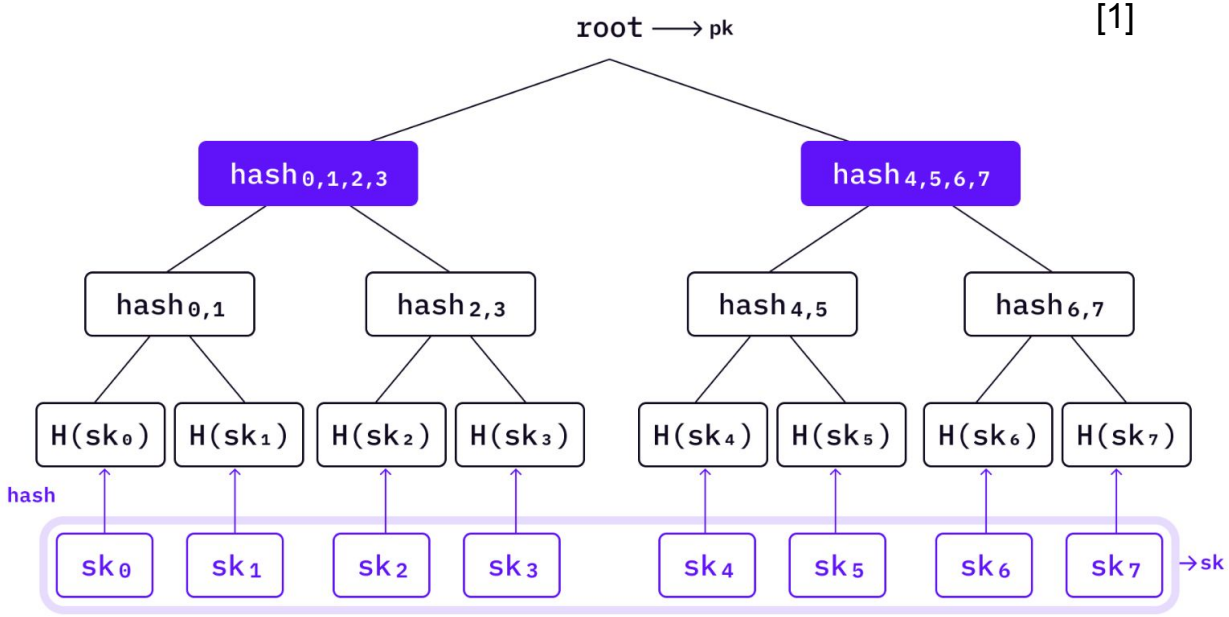
[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORS → HORST

- HORS public keys are very long ⇒ Reduce size with “Merkle-trees”
- Use of bitmasks q_l and q_r in every tree-level
  ⇒ *2h* bitmasks in total



[1]



[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

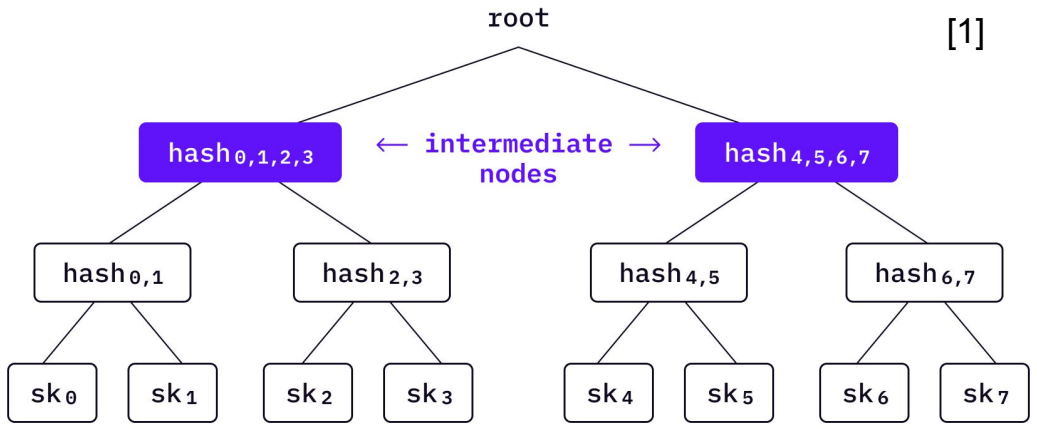
The SPHINCS Approach | HORST

- Signing messages: Select private key according to the value of each chunk
- Signature per chunk $\sigma_i = (sk_{m_i}, auth_{m_i}), 0 \leq i \leq N-1$

  $\Rightarrow$ calculated by leaf $authRoot = (N_0, N_1, \ldots, N_{2^x-1})$
- Root of tree (or intermediate nodes) can be used as public key
  $\Rightarrow$ reduced key size significantly
- Signature $\sigma = (\sigma_i, authRoot)(0 \leq i \leq N-1)$

The SPHINCS Approach | HORST

- Root of tree (**or intermediate nodes**) can be used as public key
- Example: $Hash_{0123}$ and $Hash_{4567}$ can be used for authentication too
- Why is it useful?

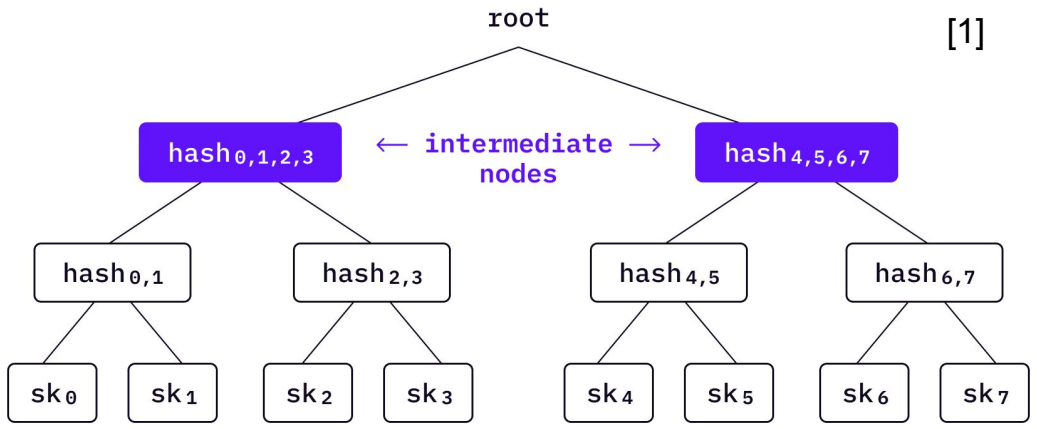


[1]

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORST

- Root of tree (**or intermediate nodes**) can be used as public key
- Example: $Hash_{0123}$ and $Hash_{4567}$ can be used for authentication too
- Why is it useful?
  ⇒ Tradeoff between size of *auth* and *authRoot*



[1]

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | HORST

- Verifying a signature:
  1. compute the hash with sk_{m_i} and $auth_{m_i}$
  2. check if $\sigma_i \in authRoot$ for each index

- Learning Goal: Understand why HORST is a very nice and simple FTS scheme improvement of HORS

The SPHINCS Approach | Stateless Signatures

- Until now: stateful signatures (LMS, XMSS, …)
- Today: Stateless hash-based signature
  - signer chooses keys randomly
  - no internal states
  - does not matter which keys have been used

But how ?

The SPHINCS Approach | Stateless Signatures [2]

Key Generation

1. Build an multilevel scheme with “Hyper-trees” of possible keys

2. Assign a small part of the tree to the signer ⇒ limit signer’s possible signatures

3. Make sure that the signer will not reuse a key (probabilistic)

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | Stateless Signatures [2]

Key Generation

1. Build an multilevel scheme with “Hyper-trees” of possible keys
2. Assign a small part of the tree to the signer ⇒ limit signer’s possible signatures
3. Make sure that the signer will not reuse a key (probabilistic)

Signing

1. Select random path of the Hyper-tree
2. Compute the multilevel signature
   ⇒ last key used for signing message
   ⇒ other keys uses to sign the roots of the merkle trees

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | Stateless Signatures

- WOTS+ (last lecture) combined with HORST?
  ⇒ a stateless hash-based signature algorithm

any problems ?

The SPHINCS Approach | Stateless Signatures

- WOTS+ (last lecture) combined with HORST?
  ⇒ a stateless hash-based signature algorithm

any problems ?

- [Recap] Neither WOTS+ nor HORST are stateless

  ⇒ pre generating keys to compute Merkle-Tree root

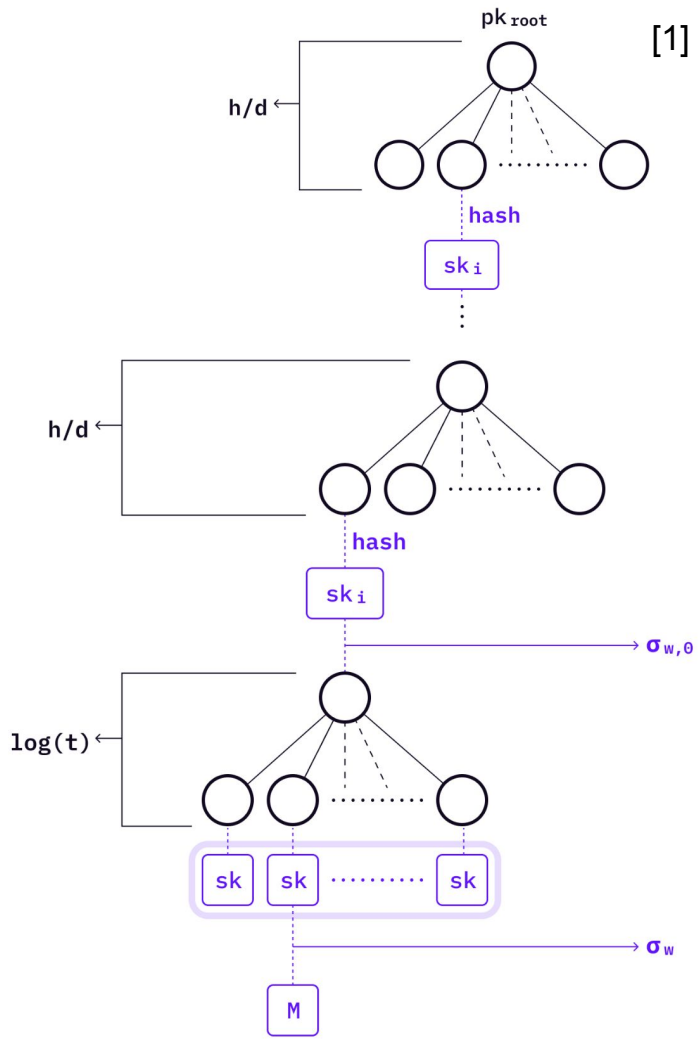
- Looking at SPHINCS (... nearly)
  - Hyper-trees
  - Random key path addressing scheme

The SPHINCS Approach | Hyper-trees

- WOTS+ (last lecture) combined with HORST?
  ⇒ a stateless hash-based signature algorithm
  - Hyper-trees
  - Random key path addressing scheme

- Hyper-tree: “tree of trees” with different layers

Winternitz parameter:	w
Hyper-tree height:	h
#Layers in HT:	d
#Leaves in HORST:	t
HT subtree size:	h/d



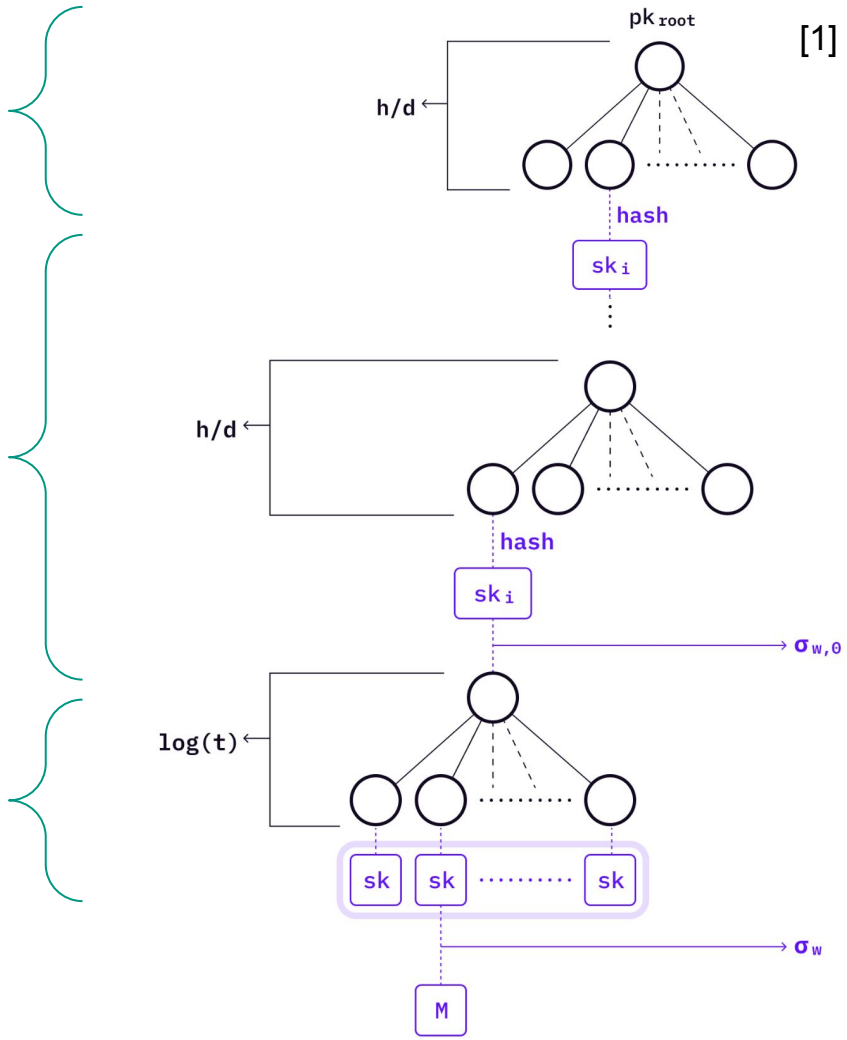
[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | Hyper-trees

L-Tree: Successful L-tree verification validates the entire signature

WOTS+ key pairs:
→ each leaf: public key string of WOTS+
→ corresponding private keys used to sign root of the next level tress

HORS trees: private keys to sign messages



[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

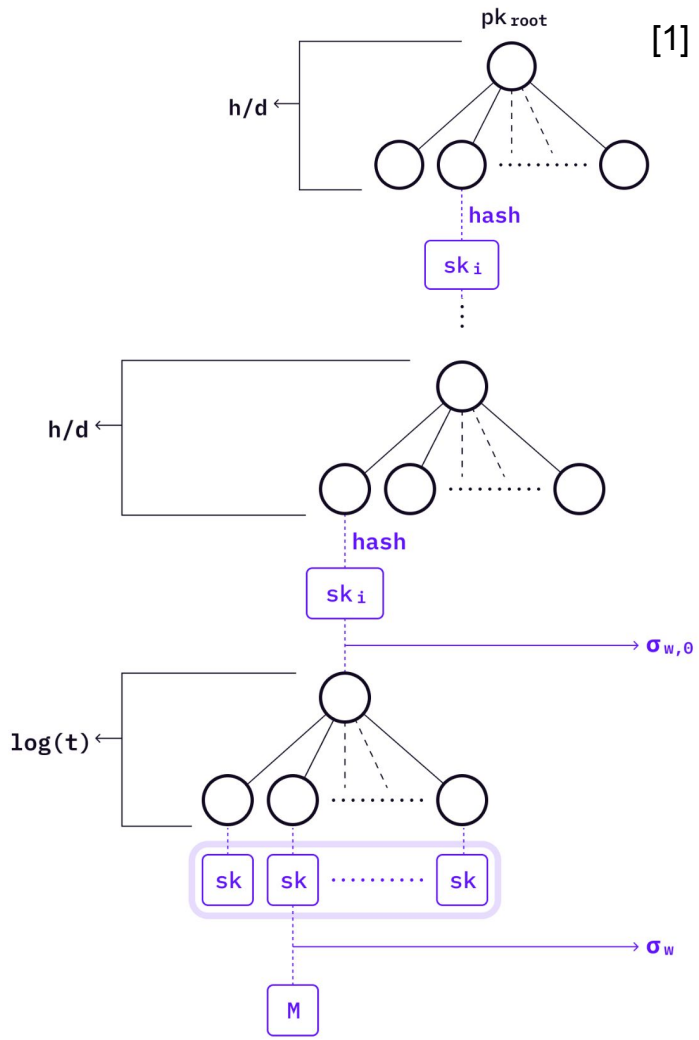
The SPHINCS Approach | Hyper-trees

L-Tree: Successful L-tree verification validates the entire signature

WOTS+ key pairs:
→ each leaf: public key string
→ corresponding private keys used to sign root of the next level tress

But why this effort?

HORS trees: private keys to sign messages



[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

The SPHINCS Approach | Hyper-trees [2]

Karlsruher Institut für Technologie

Single-level tree

- every key we use must be generated up front

⇒ memory* and computationally expensive

*1TB ≈ 2^{40} Byte

Multi-level tree

- only one key (for first tree) must be generated

⇒ 2^{60} required keys? No problem!

- Analogy: certificate chain

⇒ don't generate keys until they are required

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | Stateless Signatures [2]

Karlsruher Institut für Technologie

Key Generation

1. Build an multilevel scheme with “Hyper-trees” of possible keys
2. Assign a small part of the tree to the signer ⇒ limit signer’s possible signatures
3. Make sure that the signer will not reuse a key (probabilistic)

Signing

1. Select random path of the Hyper-tree
2. Compute the multilevel signature
   ⇒ last key used for signing message
   ⇒ other keys uses to sign the roots of the merkle trees

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | Key Generation

generated by PRG function (n-bits each)

$$SK = (SK_1, SK_2, Q)$$
$$PK = (PK_{root}, Q)$$

SK_1 used for generating HORST and WOTS+ private keys
SK_2 used for randomizing the message hash
Q bitmasks - as seen in HORST

The SPHINCS Approach | Key Generation

$$SK = (SK_1, SK_2, Q)$$
$$PK = (PK_{root}, Q)$$

1) Calculate addresses of top level tree leafs $A = (d-1||0||i)\ (i \in 2^{h/d} - 1)$
2) Generate Seed $S_A \longleftarrow F(A, SK_1)$ and create *PKs* of WOTS+
3) Compute PK_{root} as WOTS+ public Key

The SPHINCS Approach | Stateless Signatures [2]

Karlsruher Institut für Technologie

Key Generation

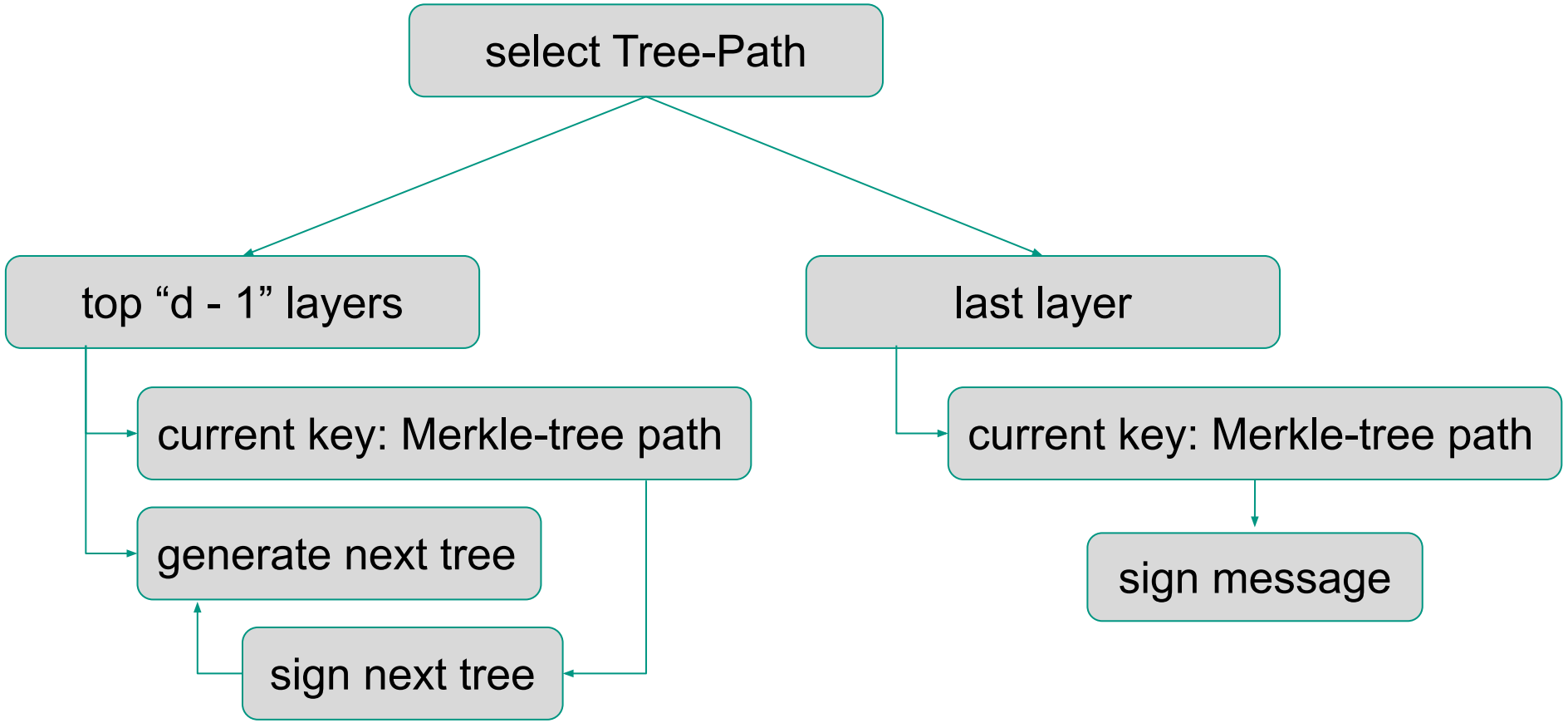
1. Build an multilevel scheme with “Hyper-trees” of possible keys
2. Assign a small part of the tree to the signer ⇒ limit signer’s possible signatures
3. Make sure that the signer will not reuse a key (probabilistic)

Signing

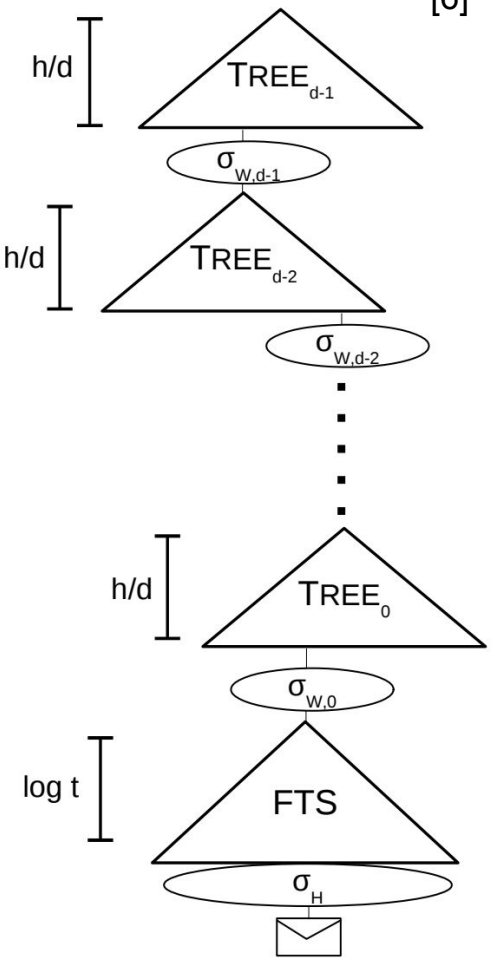
1. Select random path of the Hyper-tree
2. Compute the multilevel signature
   ⇒ last key used for signing message
   ⇒ other keys uses to sign the roots of the merkle trees

[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)

The SPHINCS Approach | Signing [2]

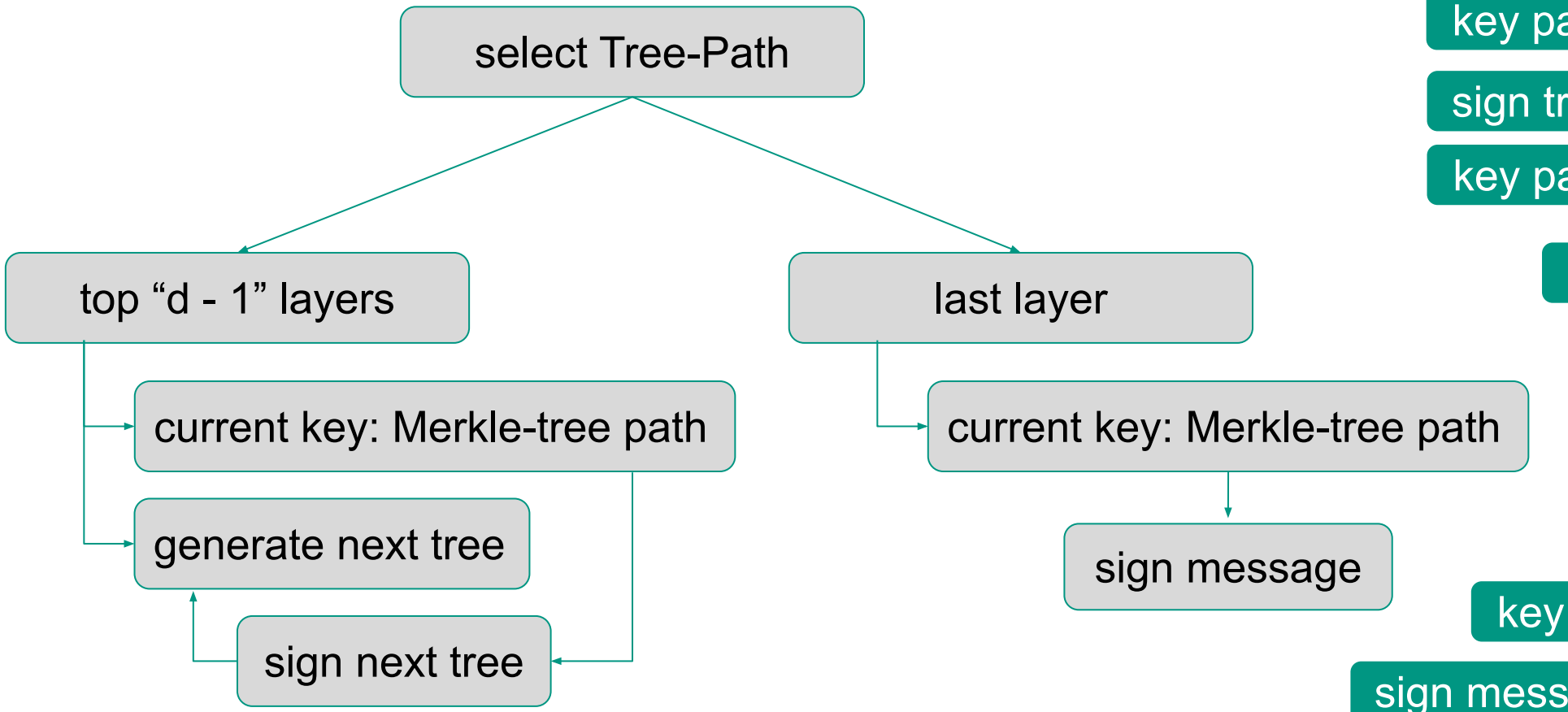


[6]

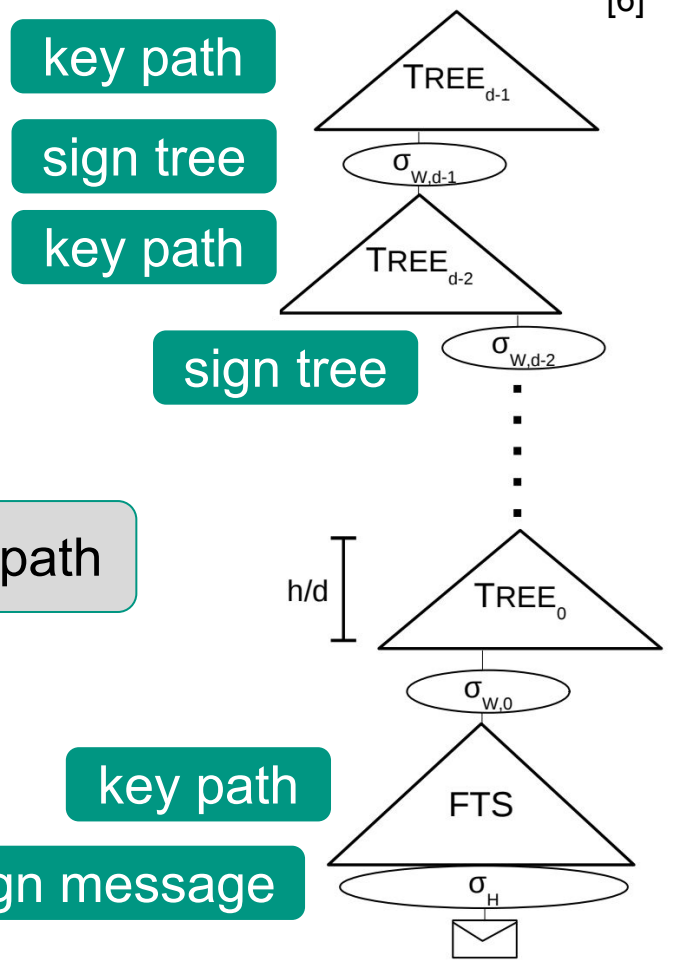


[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)
[6] Daniel J. Bernstein et al. “SPHINCS: practical stateless hash-based signatures”

The SPHINCS Approach | Signing [2]



[6]

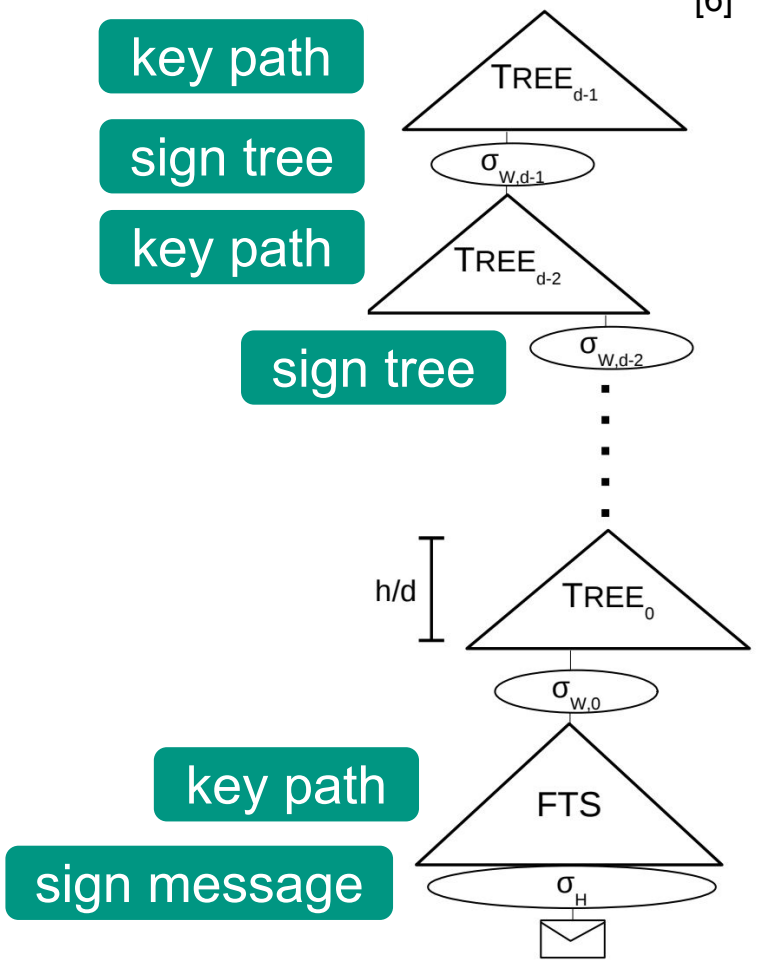


[2] COSIC seminar "Introduction to Hash Based Signatures" (John Kelsey, KU Leuven & NIST)
[6] Daniel J. Bernstein et al. “SPHINCS: practical stateless hash-based signatures”

The SPHINCS Approach | Verification [1]

- verify the last HORST signature
- verify “d - 1” WOTS+ signatures
- verify the root node

[6]



[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/
[6] Daniel J. Bernstein et al. “SPHINCS: practical stateless hash-based signatures”

SPHINCS$^+$ | Tweakable Hash Functions [5]

- Problem: Using one hash function all the time
- Solution? Tweakable Hash Functions!

- Use a family of similar hash functions ⇒ Tweak decides which hash function to use
- In SPHINCS$^+$ (ADRS): Enables independent hash function calls for varied key pairs and positions

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”

From SPHINCS to SPHINCS$^+$ [3]

1. Multi-Target Attack Protection

- Different hash functions for each call with unique keys and bitmasks
- Keys and bitmasks generated pseudorandomly from an address and public seed
- Introduction of tweakable hash functions with public seed and address for abstraction

[3] https://huelsing.net/wordpress/?p=558

From SPHINCS to SPHINCS$^+$ [3]

1. Multi-Target Attack Protection

- Different hash functions for each call with unique keys and bitmasks
- Keys and bitmasks generated pseudorandomly from an address and public seed
- Introduction of tweakable hash functions with public seed and address for abstraction

2. Tree-Less WOTS+ Public Key Compression

- Last nodes of WOTS+ chains compressed using a single tweakable hash function call
- Address and public seed used to key the call and generate a bitmask

[3] https://huelsing.net/wordpress/?p=558

From SPHINCS to SPHINCS+ [3]

Karlsruher Institut für Technologie

1. Multi-Target Attack Protection

- Different hash functions for each call with unique keys and bitmasks
- Keys and bitmasks generated pseudorandomly from an address and public seed
- Introduction of tweakable hash functions with public seed and address for abstraction

2. Tree-Less WOTS+ Public Key Compression

- Last nodes of WOTS+ chains compressed using a single tweakable hash function call
- Address and public seed used to key the call and generate a bitmask

3. FORS (Forest Of Random Subsets)

- Replaced HORST with FORS
- Dedicated set of secret key values per index derived from the message
- Enables the use of smaller parameters, resulting in smaller signature size and improved speed

[3] https://huelsing.net/wordpress/?p=558

From SPHINCS to SPHINCS+ [3]

Karlsruher Institut für Technologie

1. Multi-Target Attack Protection

- Different hash functions for each call with unique keys and bitmasks
- Keys and bitmasks generated pseudorandomly from an address and public seed
- Introduction of tweakable hash functions with public seed and address for abstraction

2. Tree-Less WOTS+ Public Key Compression

- Last nodes of WOTS+ chains compressed using a single tweakable hash function call
- Address and public seed used to key the call and generate a bitmask

3. FORS (Forest Of Random Subsets)

- Replaced HORST with FORS
- Dedicated set of secret key values per index derived from the message
- Enables the use of smaller parameters, resulting in smaller signature size and improved speed

4. Verifiable Index Selection

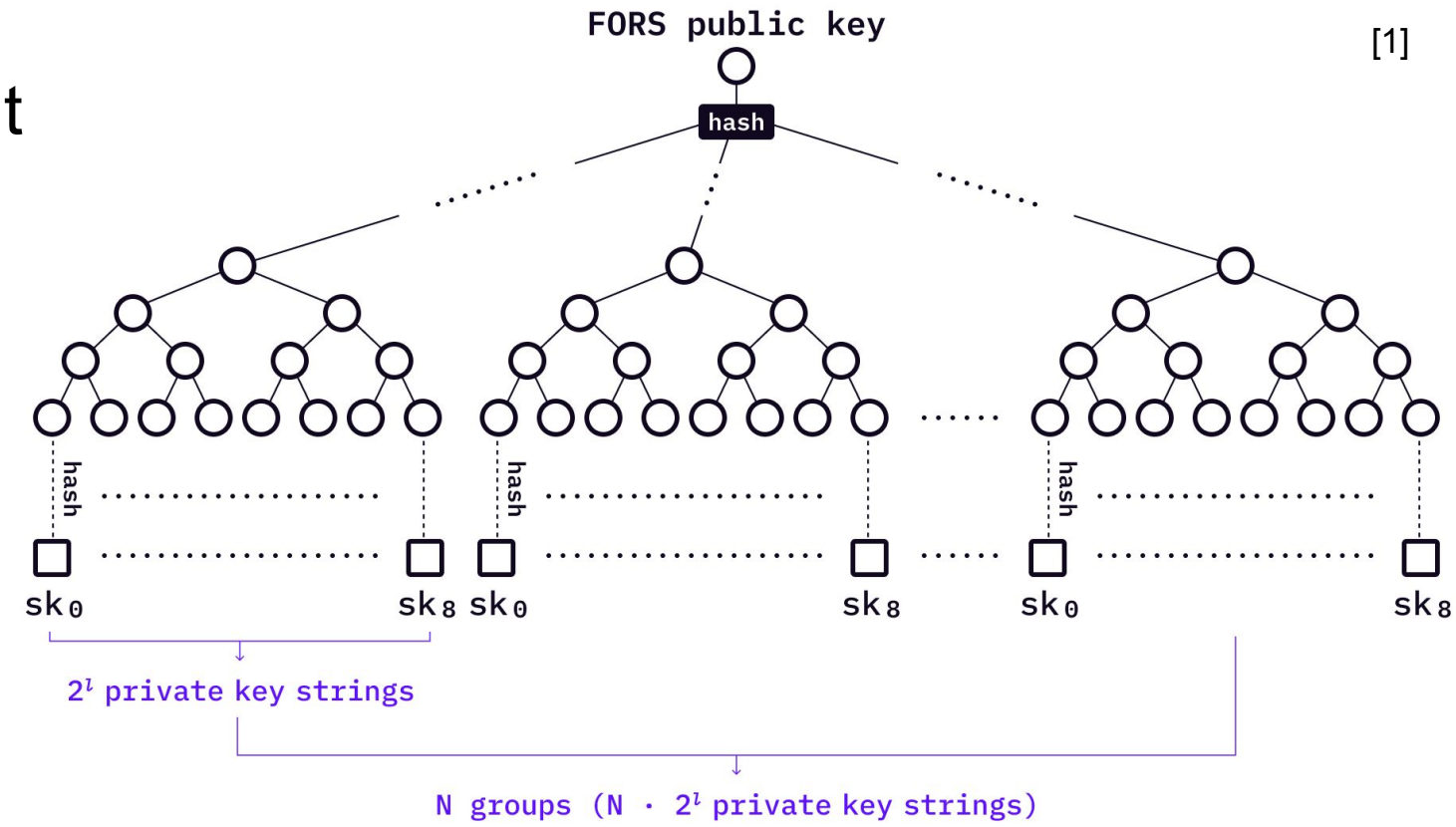
- computes the index together with the message digest, linking each message to a specific FORS instance
- Adversaries cannot use one hash computation to target all HORST instances.
- The index can be omitted in the SPHINCS+ signature

[3] https://huelsing.net/wordpress/?p=558

SPHINCS$^+$ | Forest of Random Subsets (FORS)

- Improvement of HORST
- Idea: same as HORST, but own root for each tree

  ⇒ FORS public key is the tweakable hash of the concatenation of all the roots

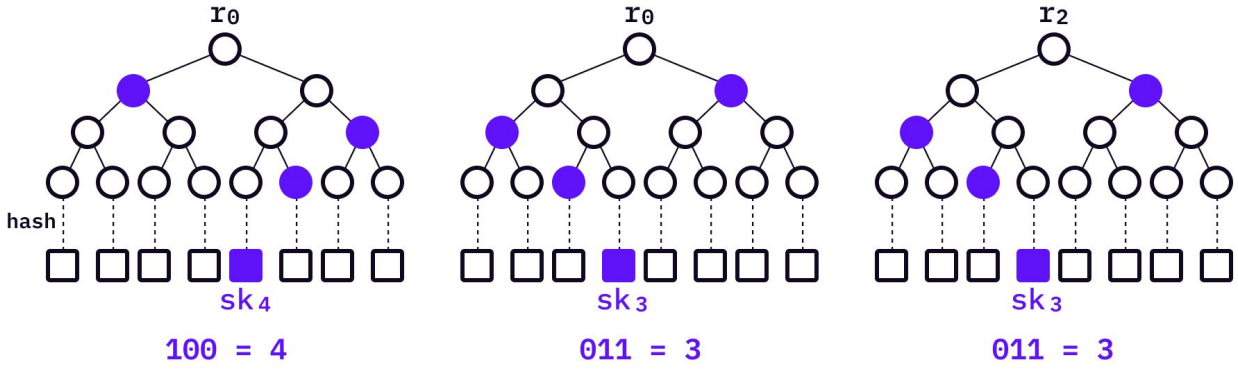


[1]

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

SPHINCS⁺ | Forest of Random Subsets (FORS)

- Let *H(m) = 100 011 011 …*



[1]

- We got this signature:

$$\sigma = (sk_4^1, auth_4^1, sk_3^2, auth_3^2, sk_3^3, auth_3^3, sk_5^4, auth_5^4, sk_7^5, auth_7^5, sk_0^6, auth_0^6)$$

- Verification:
  - compute root of every sk_i with $auth_i$
  - hash concatenation of all roots
    ⇒ if equals to pk_{FORS}, signature is valid

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

SPHINCS$^+$ | Signature Generation

- You remember SPHINCS? ... It is very similar

generated by PRG function (n-bits each)

$$SK = (SK.seed, SK.prf)$$
$$PK = (PK_{root}, PK.seed)$$

SK.seed used for generating FORS and WOTS+ private keys
SK.prf used for generating message digest

SPHINCS$^+$ | Signature Generation

- You remember SPHINCS? ... It is very similar

$$SK = (SK.seed, SK.prf)$$
$$PK = (PK_{root}, PK.seed)$$

root node of hyper-tree

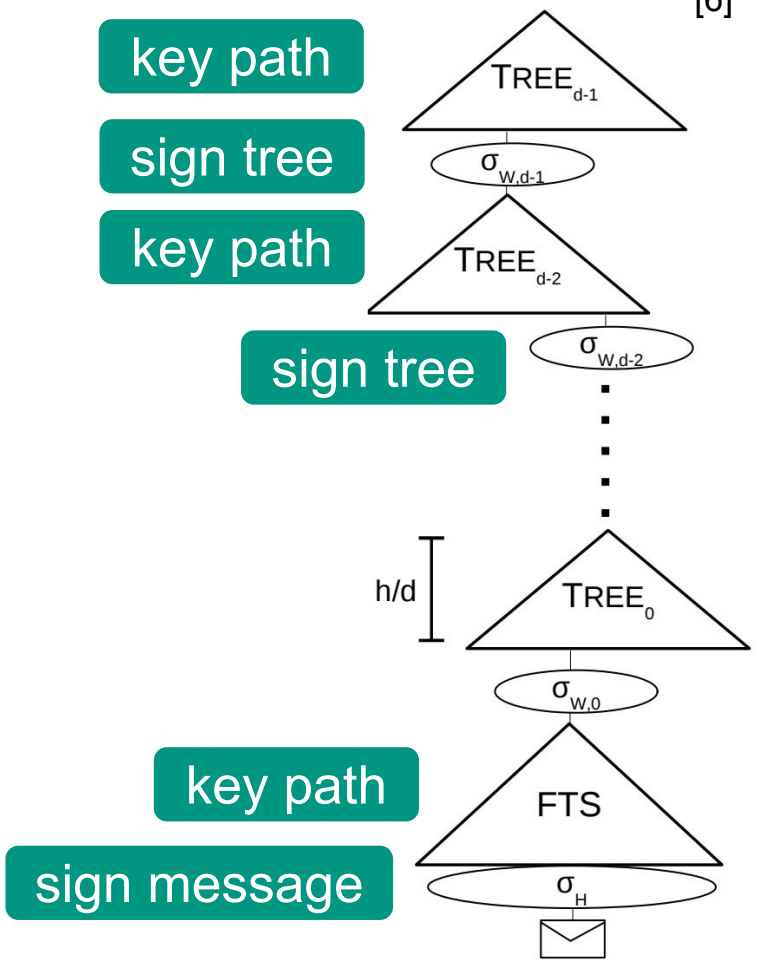
random seed
(e.g. used in THF)

- $\sigma_{SPHINCS+} = (MD, R, \sigma_F, \sigma_{w,0}, auth_{A_0}, \sigma_{w,1}, auth_{A_1}, \ldots, \sigma_{w,d-1}, auth_{A_d-1})$

[RECAP] SPHINCS Verification[1]

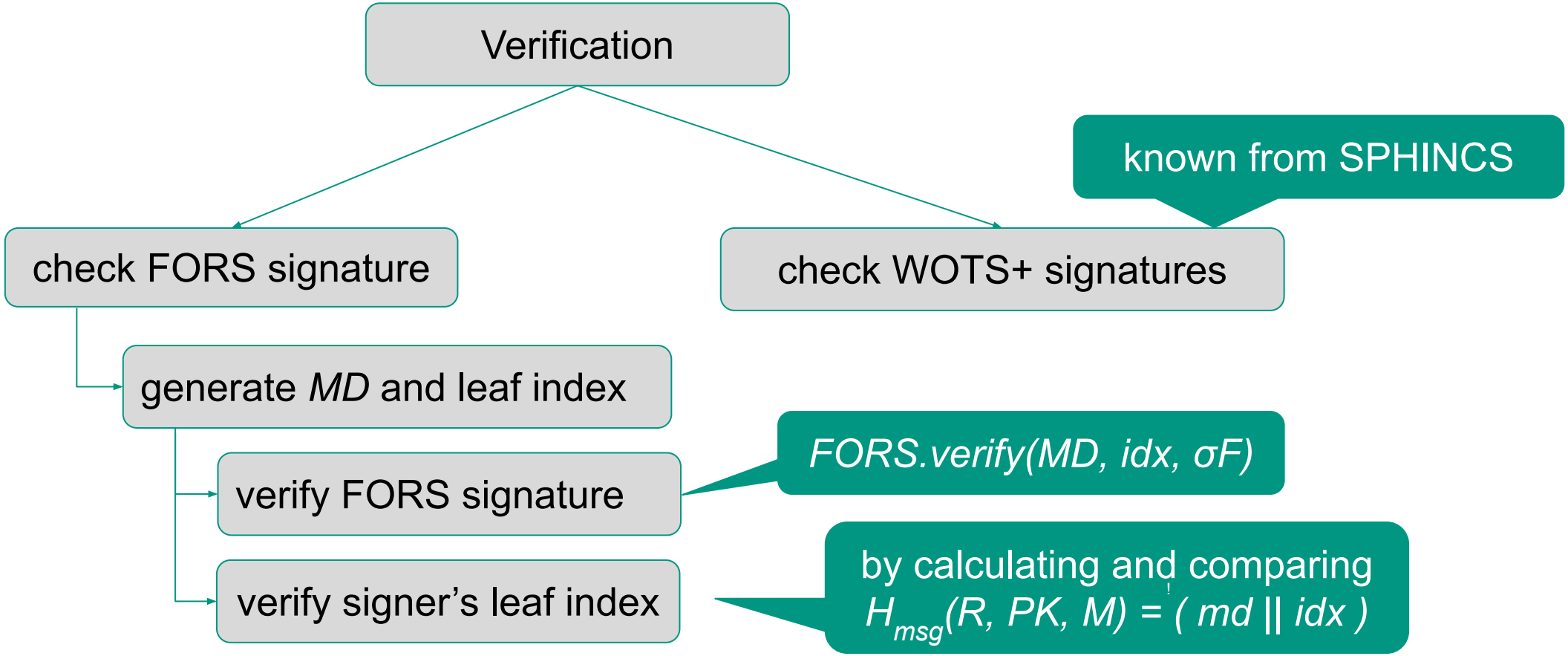
- verify the last HORST signature
- verify “d - 1” WOTS+ signatures
- verify the root node

[6]



[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/
[6] Daniel J. Bernstein et al. “SPHINCS: practical stateless hash-based signatures”

SPHINCS$^+$ | Signature Verification [1, 5]



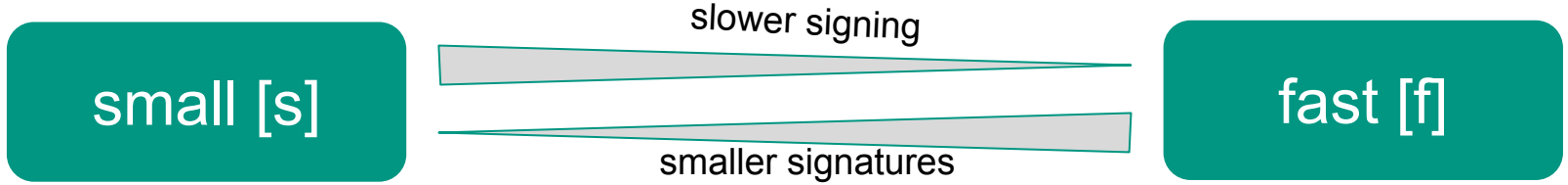
[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/
[5] Daniel J. Bernstein et al. "The SPHINCS$^+$ Signature Framework"

SPHINCS$^+$ | Fine-Tuning

- Different hashing algorithms: SHA-256, SHAKE256, Haraka [4]
- Different security parameter
- Many performance parameters
  - Winternitz parameter
  - height and level of the hyper-tree
  - amount of trees of secret values in FORS
  - amount of leaves per tree

[4] https://sphincs.org

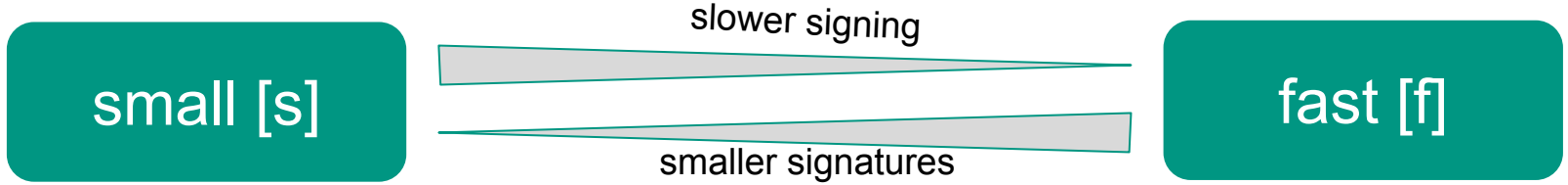
SPHINCS$^+$ | Signature size [5]



Scheme	sec	Cycles keypair	sign	verify	Bytes sig	pk	sk
SPHINCS$^+$-SHA-256-128s-simple	L1	49 078 104	835 272 076	2 348 916	8 080	32	64
SPHINCS$^+$-SHA-256-128s-robust	L1	94 988 100	1 624 566 118	4 700 588	8 080	32	64
SPHINCS$^+$-SHA-256-128f-simple	L1	1 602 368	51 805 308	5 676 578	16 976	32	64
SPHINCS$^+$-SHA-256-128f-robust	L1	2 978 018	96 974 576	11 401 188	16 976	32	64
SPHINCS$^+$-SHA-256-192s-simple	L3	69 860 954	1 737 629 602	3 662 790	17 064	48	96
SPHINCS$^+$-SHA-256-192s-robust	L3	134 664 612	3 024 929 742	7 784 118	17 064	48	96
SPHINCS$^+$-SHA-256-192f-simple	L3	2 116 010	66 380 214	9 611 814	35 664	48	96
SPHINCS$^+$-SHA-256-192f-robust	L3	4 390 738	133 192 018	19 219 918	35 664	48	96
SPHINCS$^+$-SHA-256-256s-simple	L5	85 946 882	1 121 074 298	4 903 926	29 792	64	128
SPHINCS$^+$-SHA-256-256s-robust	L5	350 260 762	4 064 645 574	13 790 402	29 792	64	128
SPHINCS$^+$-SHA-256-256f-simple	L5	5 298 662	133 374 038	9 408 596	49 216	64	128
SPHINCS$^+$-SHA-256-256f-robust	L5	21 672 826	495 051 104	26 825 462	49 216	64	128

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”

SPHINCS$^+$ | Signing performance [5]



Scheme		Cycles			Bytes		
	sec	keypair	sign	verify	sig	pk	sk
SPHINCS$^+$-SHA-256-128s-simple	L1	49 078 104	835 272 076	2 348 916	8 080	32	64
SPHINCS$^+$-SHA-256-128s-robust	L1	94 988 100	1 624 566 118	4 700 588	8 080	32	64
SPHINCS$^+$-SHA-256-128f-simple	L1	1 602 368	51 805 308	5 676 578	16 976	32	64
SPHINCS$^+$-SHA-256-128f-robust	L1	2 978 018	96 974 576	11 401 188	16 976	32	64
SPHINCS$^+$-SHA-256-192s-simple	L3	69 860 954	1 737 629 602	3 662 790	17 064	48	96
SPHINCS$^+$-SHA-256-192s-robust	L3	134 664 612	3 024 929 742	7 784 118	17 064	48	96
SPHINCS$^+$-SHA-256-192f-simple	L3	2 116 010	66 380 214	9 611 814	35 664	48	96
SPHINCS$^+$-SHA-256-192f-robust	L3	4 390 738	133 192 018	19 219 918	35 664	48	96
SPHINCS$^+$-SHA-256-256s-simple	L5	85 946 882	1 121 074 298	4 903 926	29 792	64	128
SPHINCS$^+$-SHA-256-256s-robust	L5	350 260 762	4 064 645 574	13 790 402	29 792	64	128
SPHINCS$^+$-SHA-256-256f-simple	L5	5 298 662	133 374 038	9 408 596	49 216	64	128
SPHINCS$^+$-SHA-256-256f-robust	L5	21 672 826	495 051 104	26 825 462	49 216	64	128

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”

SPHINCS$^+$ | Comparison

		Size (bytes)		Relative time	
		Public key	**Signature**	**Verification**	**Signing**
Non PQ	**NIST P-256**	64	64	1 (baseline)	1 (baseline)
	RSA-2048	256	256	0.2	25
NIST finalists	**Dilithium2**	1,320	2,420	0.3	2.5
	Falcon512	897	666	0.3	5 *
	Rainbow I	157,800	66	0.1	2.4
	Rainbow I CZ	58,800	66	12	2.4
NIST alternates*	**SPHINCS$^+$-128ss har.**	32	7,856	1.7	3,000
	SPHINCS$^+$-128fs har.	32	17,088	4	200
	Picnic-L1-full	34	32,061	21	60
	GeMMS128	352,190	33	0.4	5,000
Others	**SQISign**	64	204	500	60,000
	XMSS-SHAKE_20_128 *	32	900	2	10 *

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

SPHINCS⁺ | Security

- Post-Quantum Existential Unforgeability under Chosen-Message Attack
- SPHINCS⁺ fulfils PQ-EU-CMA if …

$$\begin{aligned}&\mathrm{InSec}^{PQ\text{-}EU\text{-}CMA}\left(\mathrm{SPHINCS}^{+};\xi,q_s\right) \\ &\quad\le \mathrm{InSec}^{PQ\text{-}PRF}\left(\mathbf{PRF};\xi,q_1\right)+\mathrm{InSec}^{PQ\text{-}PRF}\left(\mathbf{PRF_{msg}};\xi,q_s\right) \\ &\quad+\mathrm{InSec}^{PQ\text{-}ITSR}\left(\mathbf{H_{msg}};\xi,q_s\right)+\mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}\left(\mathbf{Th};\xi,q_2\right) \\ &\quad+3\cdot\mathrm{InSec}^{PQ\text{-}SM\text{-}TCR}\left(\mathbf{F};\xi,q_3\right)+\mathrm{InSec}^{PQ\text{-}SM\text{-}DSPR}\left(\mathbf{F};\xi,q_3\right),\end{aligned}$$

[5]

where $q_1 < 2^{h+1}(kt + \mathtt{len})$, $q_2 < 2^{h+2}(w \cdot \mathtt{len} + 2kt)$, *and* $q_3 < 2^{h+1}(kt + w \cdot \mathtt{len})$.

A has access to a quantum computer

negligible? scheme is PQ-EU-CMA secure

[8]

1. Gen(1^n) → (sk, pk)
2. A receives pk
3. A creates a message m
4. A gets s ← Sign(sk, m)
5. A repeats steps 3 and 4 as needed
6. A outputs (m^*, s^*) ($m^* \neq m$)
7. A wins if Ver(pk, m^*, s^*) = 1

[5] Daniel J. Bernstein et al. “The SPHINCS⁺ Signature Framework”
[8] https://www.cs.tau.ac.il/~canetti/f08-materials/scribe8.pdf

SPHINCS$^+$ | Security

- Post-Quantum Existential Unforgeability under Chosen-Message Attack
- SPHINCS$^+$ fulfils PQ-EU-CMA if …

no worries

A has access to a quantum computer

negligible? scheme is PQ-EU-CMA secure

[8]

1. Gen(1^n) → (sk, pk)
2. A receives pk
3. A creates a message m
4. A gets s ← Sign(sk, m)
5. A repeats steps 3 and 4 as needed
6. A outputs (m^*, s^*) ($m^* \neq m$)
7. A wins if Ver(pk, m^*, s^*) = 1

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”
[8] https://www.cs.tau.ac.il/~canetti/f08-materials/scribe8.pdf

SPHINCS$^+$ | PQ-EU-CMA [5]

- *TH* is post-quantum single-function multi-target-collision resistant for distinct tweaks
  - difficult to find different inputs with same output
  - function is resistant to collisions when distinct tweaks are used

- *F* is post-quantum single-function multi-target decisional second-preimage resistant for distinct tweaks
  - hard to find input that produces the same output as a given first input

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”

SPHINCS$^+$ | PQ-EU-CMA [5]

- PRF and PRF_{msg} are post-quantum pseudorandom function families
  - functions that are not distinguishable from real random functions
  - … not even by quantum computers
- H_{msg} is post-quantum interleaved target subset resilient
  - remains secure even if certain parts of the input or target are interleaved

proofs mainly by reduction

[5] Daniel J. Bernstein et al. “The SPHINCS$^+$ Signature Framework”

Recap | Today in a nutshell

- FTS (HORS, HORST, FORS): key reuse is not a security issue
- Stateless signature schemes: no need to remember which keys were used
- SPHINCS: stateless signature scheme built from stateful building blocks

HORST | WOTS+ | Hyper-Trees

- SPHINCS$^+$: Like SPHINCS, only with small but important changes

FORS | Tweakable Hash Functions | Message Digest

Backup | WOTS$^+$ private key

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

wots_SKgen(SK.seed, ADRS) {
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk[i] = PRF(SK.seed, ADRS);
  }
  return sk;
}
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | WOTS$^+$ public key

```
#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
  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk = PRF(SK.seed, ADRS);
    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;
}
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | WOTS$^+$ Signing

```
#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);

  for ( i = 0; i < len; i++ ) {
    ADRS.setChainAddress(i);
    ADRS.setHashAddress(0);
    sk = PRF(SK.seed, ADRS);
    sig[i] = chain(sk, 0, msg[i], PK.seed, ADRS);
  }
  return sig;
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | Hyper-trees Key Generation

```
# 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);
      ADRS.setLayerAddress(d-1);
      ADRS.setTreeAddress(0);
      root = xmss_PKgen(SK.seed, PK.seed, ADRS);
      return root;
}
```

```
# 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;
}
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | Hyper-trees Signing

```
# 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;
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | FORS Key Generation

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

fors_SKgen(SK.seed, ADRS, idx) {
  ADRS.setTreeHeight(0);
  ADRS.setTreeIndex(idx);
  sk = PRF(SK.seed, ADRS);
  return sk;
}
```

private key

```
# 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;
}
```

public key

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | FORS Signing

```
#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
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(i*t + idx);
    SIG_FORS = SIG_FORS || PRF(SK.seed, ADRS);

    // 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;
}
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | SPHINCS$^+$ Key Generation

```
# 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) );
}
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | SPHINCS$^+$ Signing

```
# 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);

    // generate randomizer
    opt = toByte(0, n);
    if(RANDOMIZE){
      opt = rand(n);
    }
    R = PRF_msg(SK.prf, opt, M);
    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;

    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, M, 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;
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | SPHINCS+ Verification

```
# 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();
    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);
```

[9] https://sphincs.org/data/sphincs+-round3-specification.pdf

Backup | SPHINCS signing in detail

- Generate two random n-bits R_1 and R_2 by $F(M, SK_2)$
  - compute the message digest $D \longleftarrow H(R_1, M)$
  - compute HORST address $i \longleftarrow Chop(R_2, h)$ and $Address_H = (d||i(0,(d-1)h/d)||i((d-1)h/d, h/d))$
- Generate HORST key pair and HOTST signature
  - generate HORST key pair seed by $Seed_H \longleftarrow F(Address_H, SK_1)$
  - generate HORST signature and public key by $(\sigma_H, pk_H) \longleftarrow HORST.sign(D, Seed_H, Q_H)$

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

Backup | SPHINCS signing in detail

- Generate all WOTS+ signatures along the SPHINCS path
  - compute all addresses of WOTS+ in the path $Address_{w,j} = (j||i(0,(d-1-j)h/d)||i((d-1-j)h/d),h/d))$ where j is the level and $j \in [0, d-1]$
  - compute all the seeds $Seed_{w,j} = F(Address_{w,j}, SK_1)$
  - generate WOTS+ signature $\sigma_{w,j}$ by $WOTS+.sign(pk_{w,j-1}, Seed_{w,j}, Q_{WOTS+})$ where $pk_{w,j-1}$ is the root of the tree of $j-1$ level. Also we need to generate the authentication path $auth_{A_j}$ of corresponding WOTS+ public key.

The SPHINCS signature is

$$\sigma_{SPHINCS} = (i, R_1, \sigma_H, \sigma_{w,0}, auth_{A_0}, \sigma_{w,1}, auth_{A_1}, \ldots, \sigma_{w,d-1}, auth_{A_d-1})$$

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/

Backup | SPHINCS verification in detail

- check HORST signature:
  - compute message digest $D \longleftarrow H(R_1, M)$
  - verify σ_H: $HORST.verify(D, Q_{HORST}, \sigma_H)$, reject if the result doesn't equal auth path root of HORST.
- check all WOTS+ signatures:
  - check $\sigma_{w,0}$ by $WOTS+.verify(pk_H, \sigma_{w,0}, Q_{HORST})$;
  - check the $\sigma_{w,i}$ by $WOTS+.verify(pk_{w,i}, \sigma_{w,i}, Q_{WOTS+})$ where $i \in [1, d-1]$
  - reject if any one of the WOTS+ signatures cannot be validated.
- On hyper-tree level $d-1$, the verifier gets the root of the hyper-tree. If the $root == PK_{root}$, the $\sigma_{SPHINCS}$ is validated, otherwise reject.

[1] https://research.dorahacks.io/2022/12/16/hash-based-post-quantum-signatures-2/