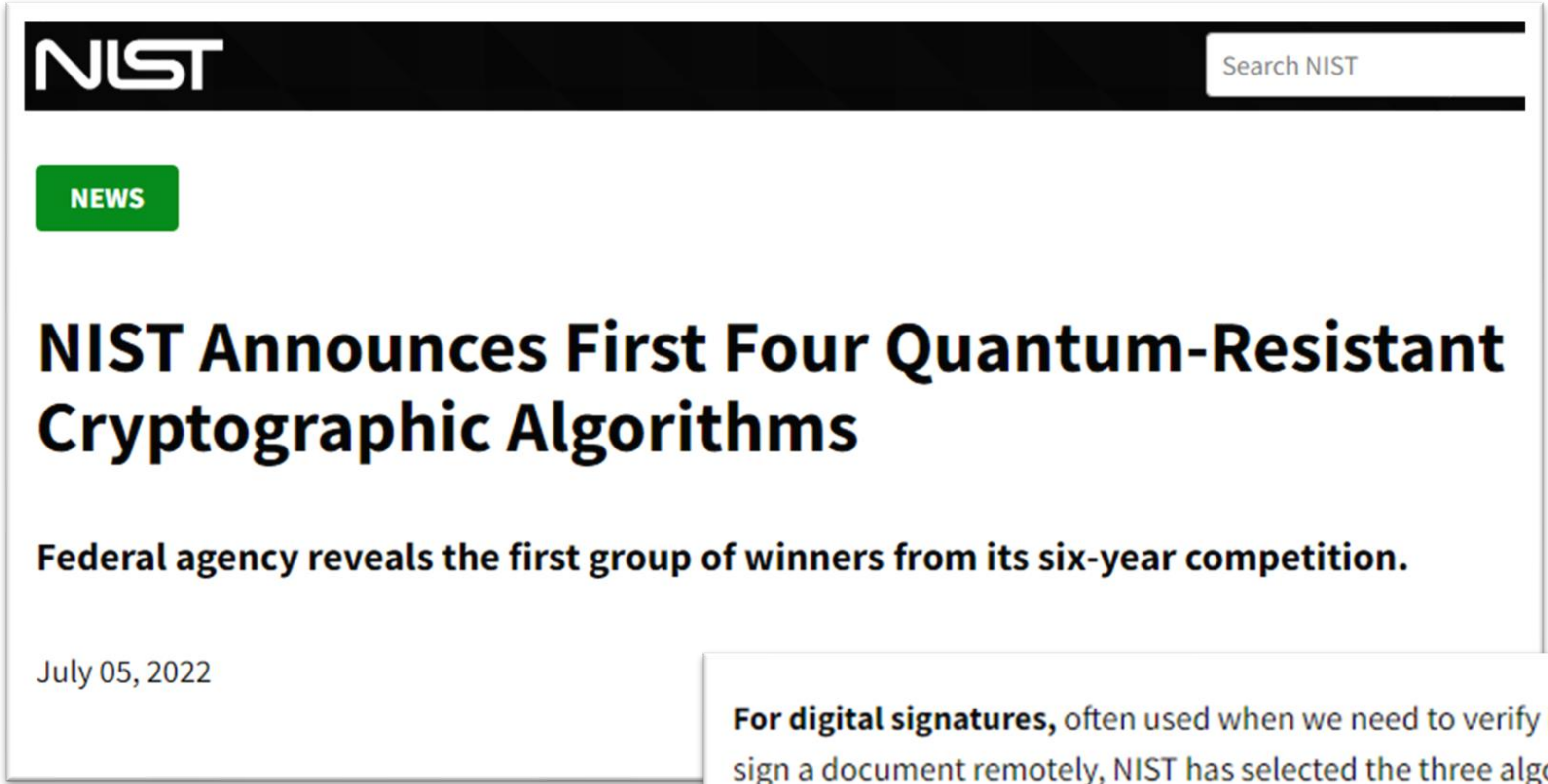
SPHINCS+

JP Aumasson, Taurus SA

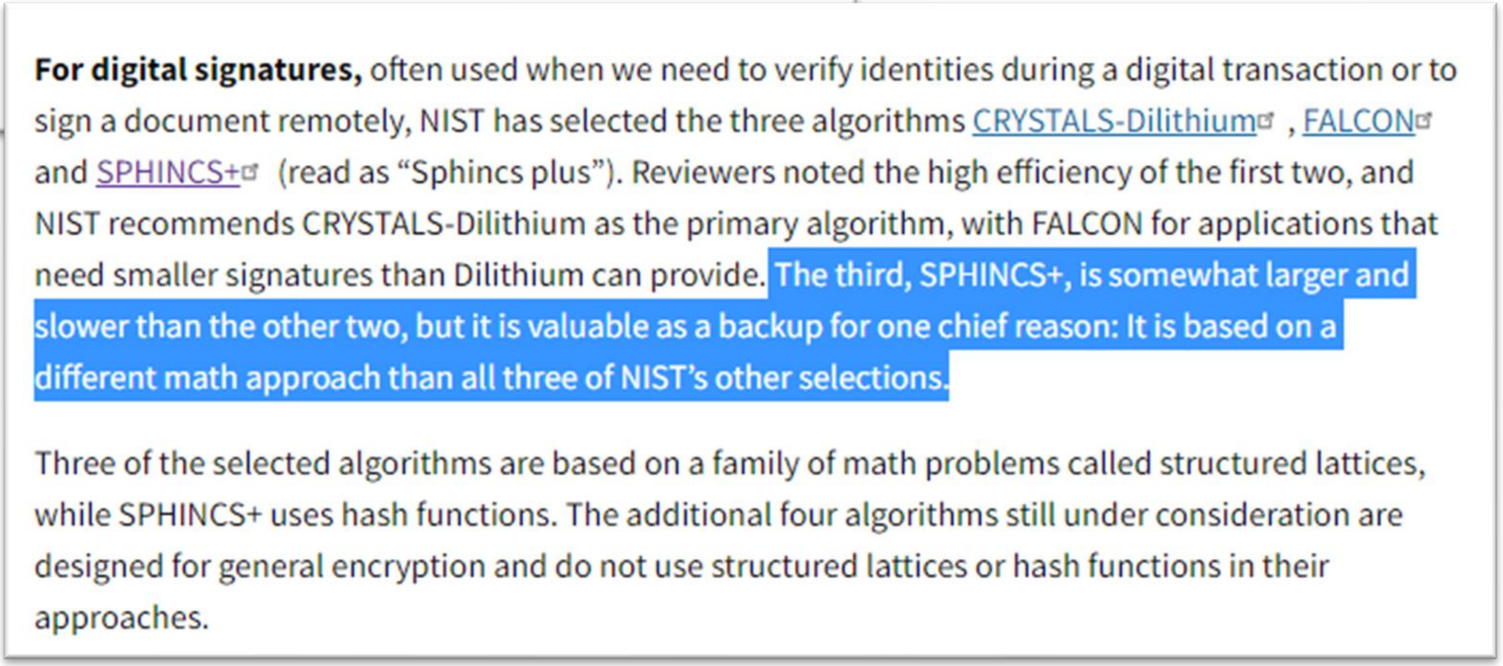
Introduction



NIST Announces First Four Quantum-Resistant Cryptographic Algorithms

Federal agency reveals the first group of winners from its six-year competition.

July 05, 2022



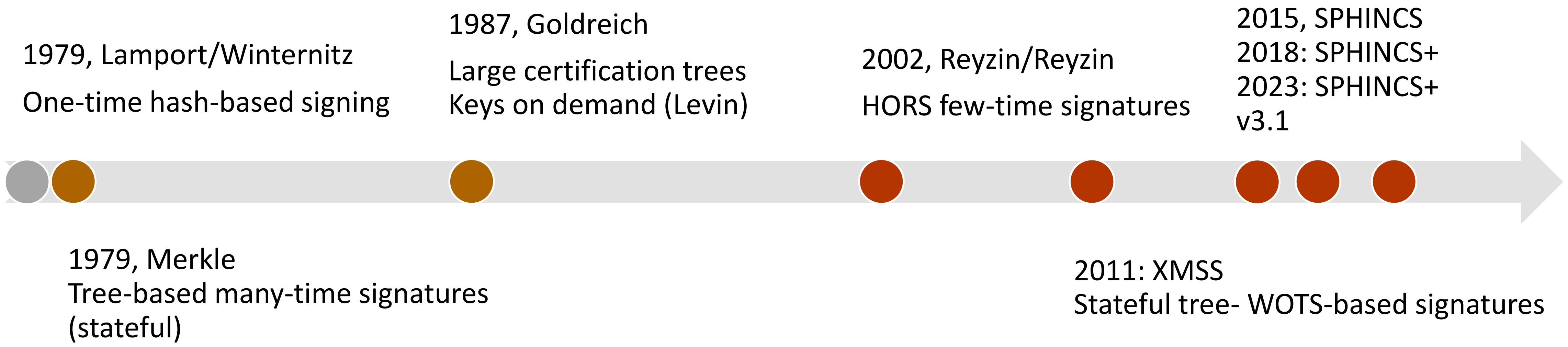
For digital signatures, often used when we need to verify identities during a digital transaction or to sign a document remotely, NIST has selected the three algorithms CRYSTALS-Dilithium , FALCON and SPHINCS+ (read as "Sphincs plus"). Reviewers noted the high efficiency of the first two, and NIST recommends CRYSTALS-Dilithium as the primary algorithm, with FALCON for applications that need smaller signatures than Dilithium can provide. The third, SPHINCS+, is somewhat larger and slower than the other two, but it is valuable as a backup for one chief reason: It is based on a different math approach than all three of NIST's other selections.

Three of the selected algorithms are based on a family of math problems called structured lattices, while SPHINCS+ uses hash functions. The additional four algorithms still under consideration are designed for general encryption and do not use structured lattices or hash functions in their approaches.

Hash functions

- The simplest and most reliable crypto primitive
- No mathematical **structure** or NP-hardness reduction
- NIST submissions must support **FIPS primitives** (SHA-2/3, SHAKE)
- Can NOT be used to build public-key **encryption** / KEMs
- **Quantum** resistance: black-box algorithms against…
  - (Second) preimage: theoretical $2^{n/2}$ bound + overhead…
  - Collision resistance: mostly unaffected

SPHINCS+ genealogy



1979, Lamport/Winternitz

One-time hash-based signing

1979, Merkle
Tree-based many-time signatures
(stateful)

1987, Goldreich

Large certification trees
Keys on demand (Levin)

2002, Reyzin/Reyzin

HORS few-time signatures

2011: XMSS
Stateful tree- WOTS-based signatures

2015, SPHINCS
2018: SPHINCS+
2023: SPHINCS+
v3.1

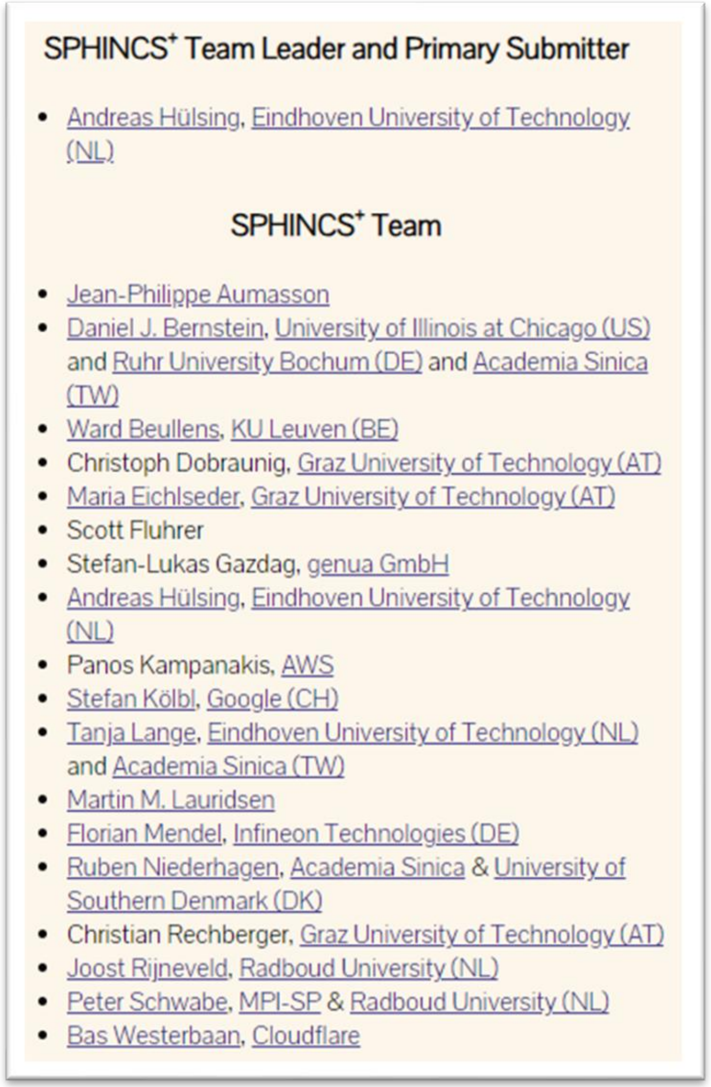
SPHINCS+ submission

https://sphincs.org/

Based on SPHINCS (2015)

Effort lead by Andreas Hülsing

(I was invited by the designers, after my submission Gravity-SPHINCS didn't make it to the 2nd round)



SPHINCS+ Team Leader and Primary Submitter

- Andreas Hülsing, Eindhoven University of Technology (NL)

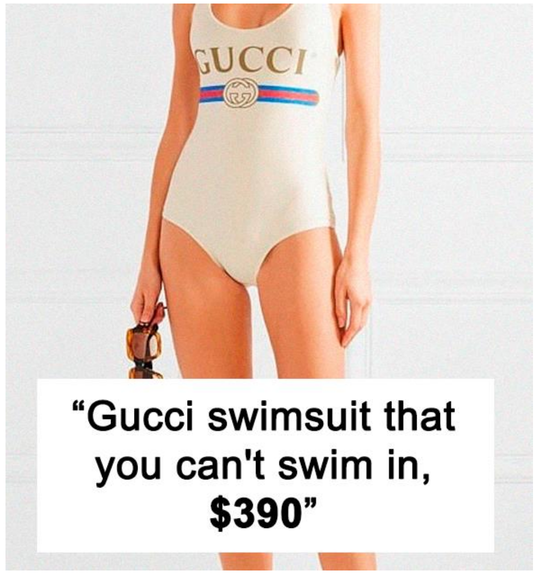
SPHINCS+ Team

- Jean-Philippe Aumasson
- Daniel J. Bernstein, University of Illinois at Chicago (US) and Ruhr University Bochum (DE) and Academia Sinica (TW)
- Ward Beullens, KU Leuven (BE)
- Christoph Dobraunig, Graz University of Technology (AT)
- Maria Eichlseder, Graz University of Technology (AT)
- Scott Fluhrer
- Stefan-Lukas Gazdag, genua GmbH
- Andreas Hülsing, Eindhoven University of Technology (NL)
- Panos Kampanakis, AWS
- Stefan Kölbl, Google (CH)
- Tanja Lange, Eindhoven University of Technology (NL) and Academia Sinica (TW)
- Martin M. Lauridsen
- Florian Mendel, Infineon Technologies (DE)
- Ruben Niederhagen, Academia Sinica & University of Southern Denmark (DK)
- Christian Rechberger, Graz University of Technology (AT)
- Joost Rijneveld, Radboud University (NL)
- Peter Schwabe, MPI-SP & Radboud University (NL)
- Bas Westerbaan, Cloudflare

Building blocks

Lamport one-time signatures (1979)

- Key generation:
  - Pick random strings K_0 and K_1 (your **private key**)
  - The public key is the two values **H**(K_0), **H**(K_1)
- To sign the bit 0, show K_0, to sign 1 show K_1
- To verify a sig **S** of i, check **H**(**S**) == **H**(K_i)

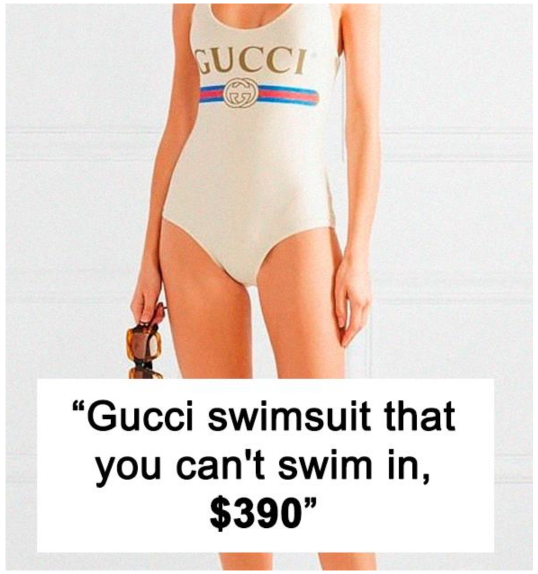


Lamport one-time signatures (1979)

- Key generation:
  - Pick random strings K_0 and K_1 (your **private key**)
  - The public key is the two values **H**(K_0), **H**(K_1)
- To sign the bit 0, show K_0, to sign 1 show K_1
- To verify a sig **S** of i, check **H**(**S**) == **H**(K_i)

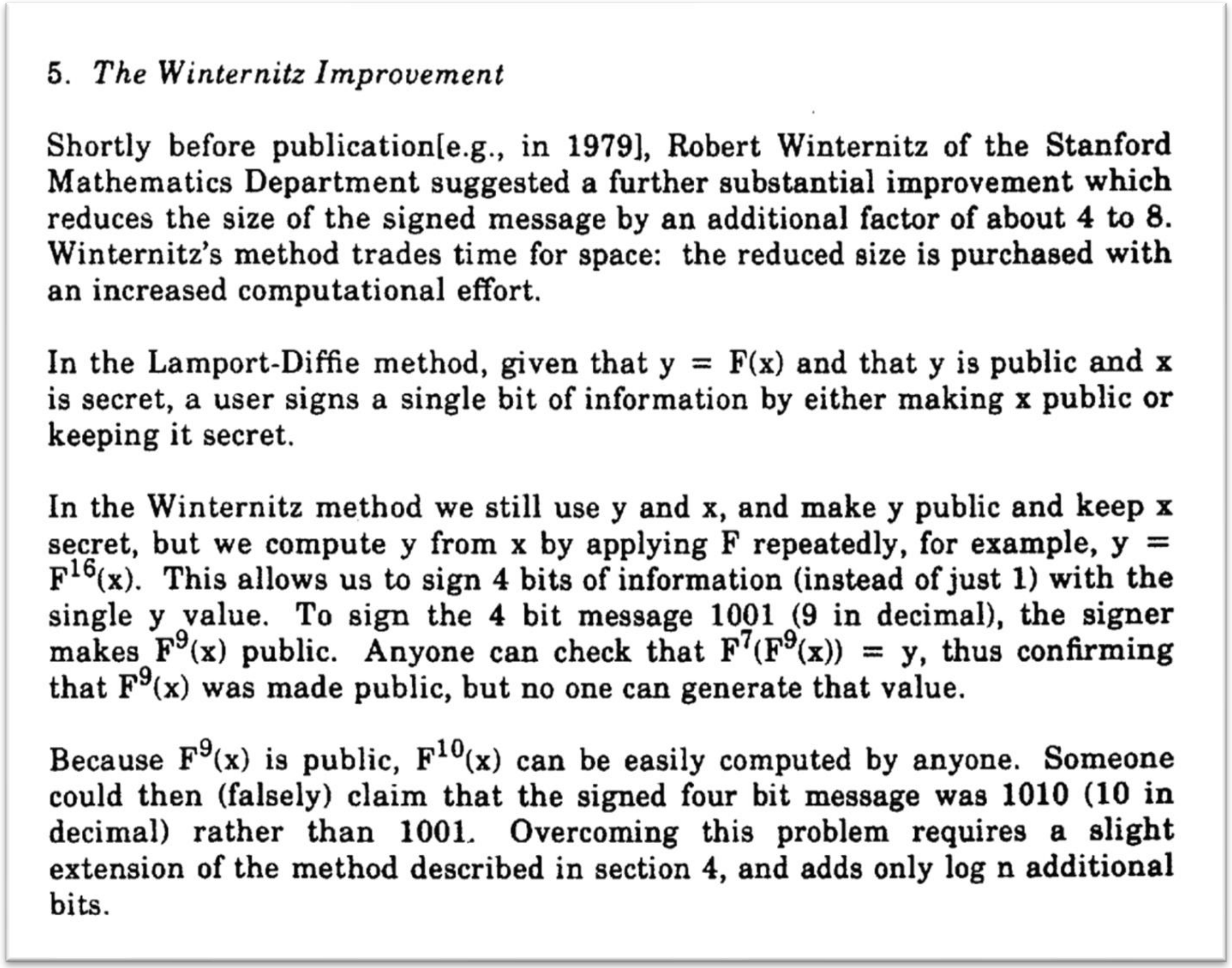
Problems:

- Needs as many keys as bits
- A key can be used **only once**



it's the most useless
and
the most expensive

Winternitz trick: sign more than a bit (1979)



5. *The Winternitz Improvement*

Shortly before publication[e.g., in 1979], Robert Winternitz of the Stanford Mathematics Department suggested a further substantial improvement which reduces the size of the signed message by an additional factor of about 4 to 8. Winternitz's method trades time for space: the reduced size is purchased with an increased computational effort.

In the Lamport-Diffie method, given that $y = F(x)$ and that y is public and x is secret, a user signs a single bit of information by either making x public or keeping it secret.

In the Winternitz method we still use y and x, and make y public and keep x secret, but we compute y from x by applying F repeatedly, for example, $y = F^{16}(x)$. This allows us to sign 4 bits of information (instead of just 1) with the single y value. To sign the 4 bit message 1001 (9 in decimal), the signer makes $F^9(x)$ public. Anyone can check that $F^7(F^9(x)) = y$, thus confirming that $F^9(x)$ was made public, but no one can generate that value.

Because $F^9(x)$ is public, $F^{10}(x)$ can be easily computed by anyone. Someone could then (falsely) claim that the signed four bit message was 1010 (10 in decimal) rather than 1001. Overcoming this problem requires a slight extension of the method described in section 4, and adds only log n additional bits.

Winternitz trick: sign more than a bit (1979)

- Key generation:
  - Pick a random string **K** as **private key**
  - The public key is **H**(**H**(**H**(**H**(…. (**K**)…)) = $\mathbf{H}^{\mathbf{w}}$ (**K**)
- To sign a number **x** in [0 .. **w** – 1], compute **S** = $\mathbf{H}^{\mathbf{x}}$ (**K**)
- To verify a sig **S** of **x**, check that $\mathbf{H}^{\mathbf{w-x}}$ (**S**) = public key

Winternitz trick: sign more than a bit (1979)

- Key generation:
  - Pick a random string **K** as **private key**
  - The public key is **H**(**H**(**H**(**H**(…. (**K**)…)) = $\mathbf{H}^{\mathbf{w}}$ (**K**)
- To sign a number **x** in [0 .. **w** – 1], compute **S** = $\mathbf{H}^{\mathbf{x}}$ (**K**)
- To verify a sig **S** of **x**, check that $\mathbf{H}^{\mathbf{w-x}}$ (**S**) = public key

Problems:

- Need for **w** = 256 to sign a byte: slow, large (size of a hash)
- A key can be used **only once**
- Need for a **checksum** to avoid malleability
- Using the same **H**() offers suboptimal security

Winternitz trick: sign more than a bit (1979)

- Key generation:
  - Pick a random string **K** as **private key**
  - The public key is
- To sign a nu
- To verify a sig

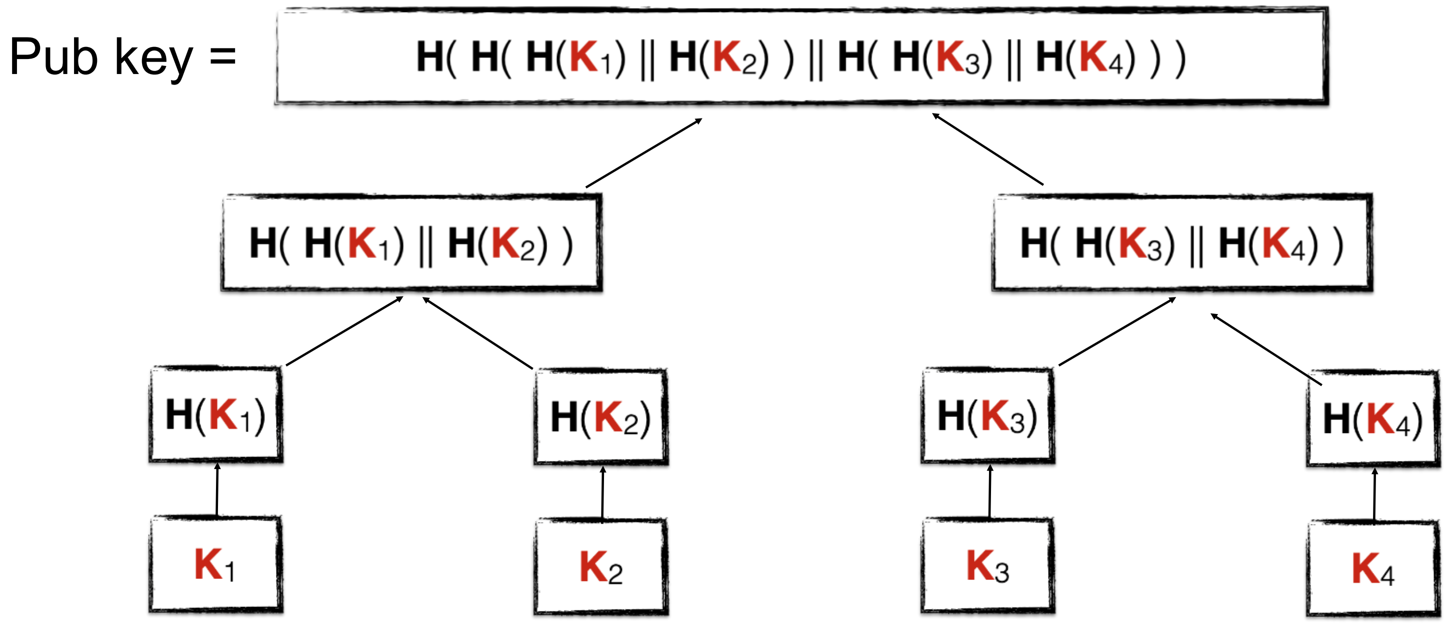
In SPHINCS+, **w** = 16
(4-bit blocks)

Problems:

- Need for **w** = 256 to sign a byte: slow, large (size of a hash)
- A key can be used **only once**
- Need for a **checksum** to avoid malleability
- Using the same **H**() offers suboptimal security

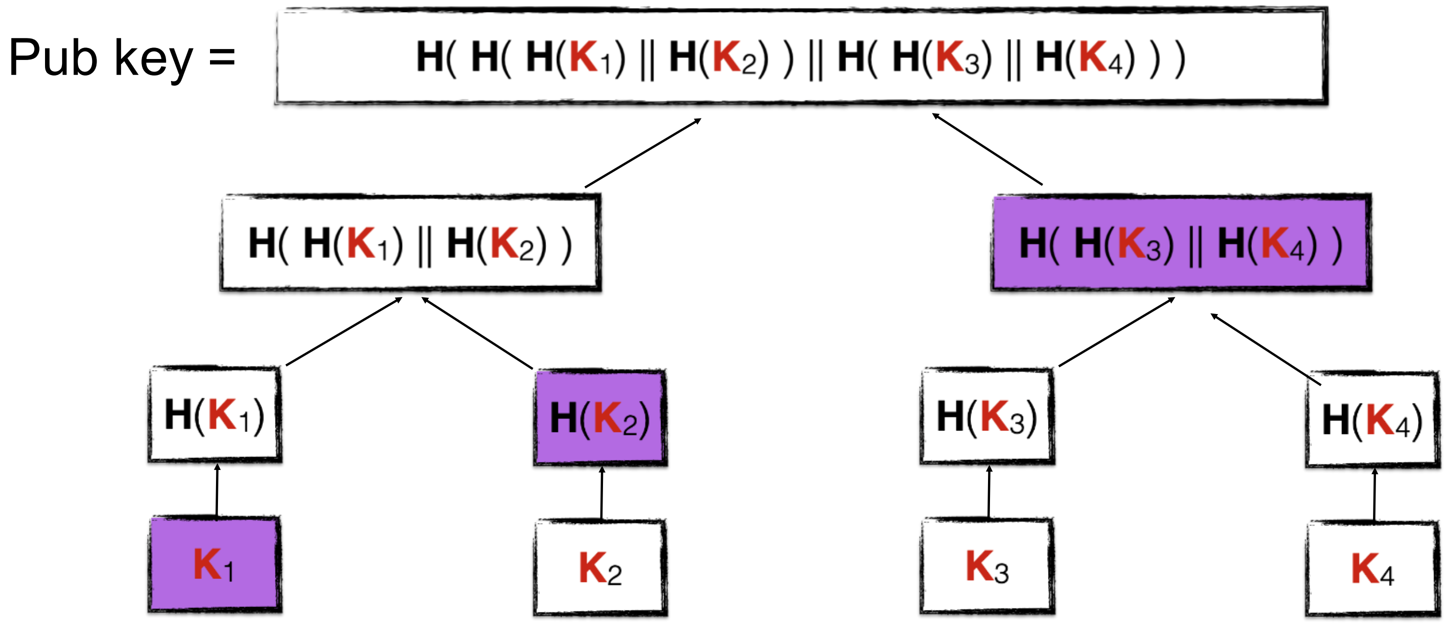
From one-time to many-time (1990)

Use a Merkle tree to “compress” many public keys into one



From one-time to many-time (1990)

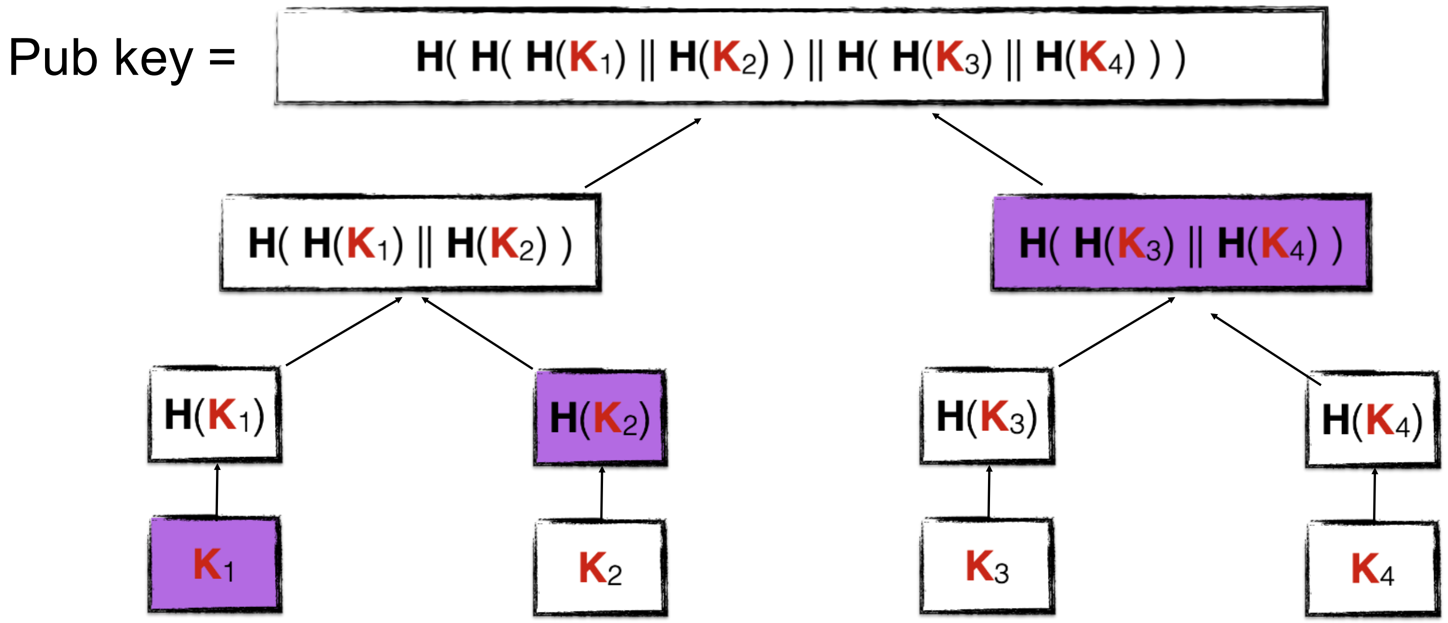
Sign using a key (leaf) and **provide its authentication path** to the root



Pub key =
$H(\,H(\,H(K_1) \,||\, H(K_2)\,) \,||\, H(\,H(K_3) \,||\, H(K_4)\,)\,)$
$H(\,H(K_1) \,||\, H(K_2)\,)$
$H(\,H(K_3) \,||\, H(K_4)\,)$
$H(K_1)$
$H(K_2)$
$H(K_3)$
$H(K_4)$
K_1
K_2
K_3
K_4

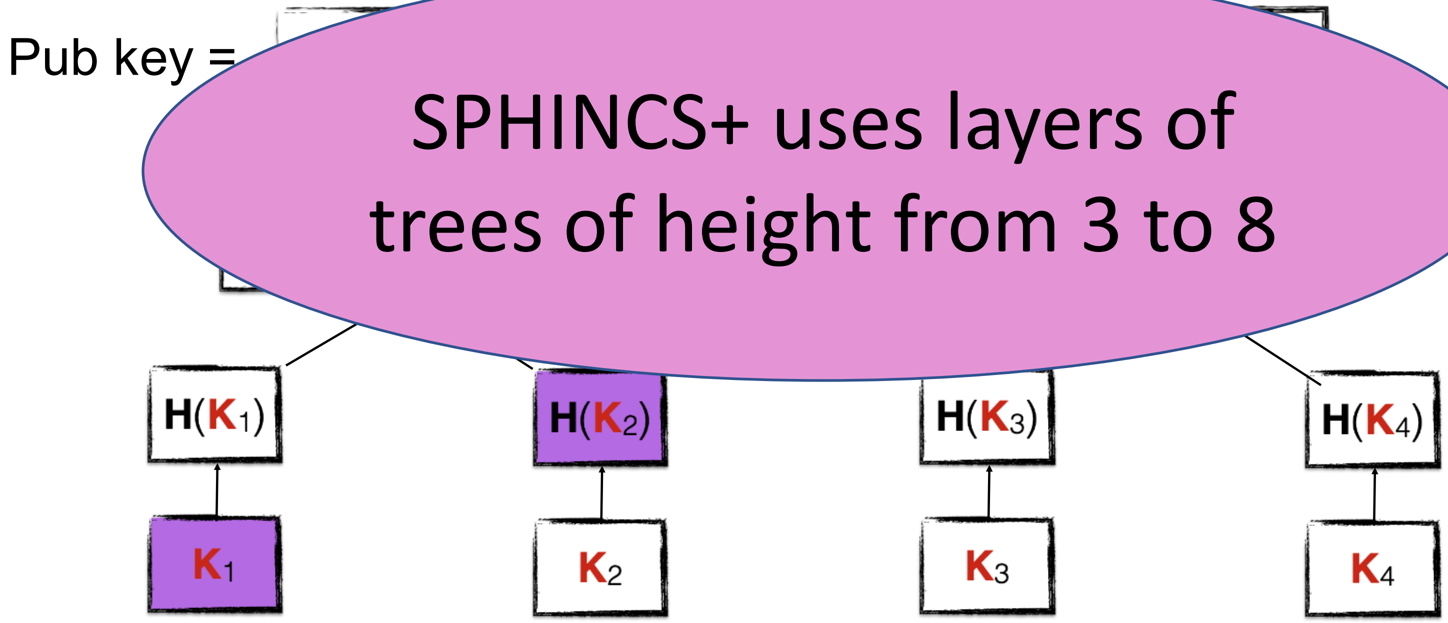
From one-time to many-time (1990)

Verification = recompute the **public key** (root of the tree)



From one-time to many-time (1990)

Verification = recompute the **public key** (root of the tree)



SPHINCS+ uses layers of trees of height from 3 to 8

HORS few-time signatures (2022)

Hash to Obtain a Random Subset

To sign **M**, use a *selection function S*: **M** → indexes

	1	2	3	4	5	…	*n*
Private keys	K_1	K_2	K_3	K_4	K_5	…	K_n
	↓	↓	↓	↓	↓		↓
Public keys	**H**(K_1)	**H**(K_2)	**H**(K_3)	**H**(K_4)	**H**(K_5)	…	**H**(K_n)

HORS few-time signatures (2022)

Hash to Obtain a Random Subset

To sign **M**, use a *selection function* S: **M** → indexes

For example, if S(**M**) = {1, 5} publish K_1 and K_5

	1	2	3	4	5	…	n
Private keys	K_1	K_2	K_3	K_4	K_5	…	K_n
	↓	↓	↓	↓	↓		↓
Public keys	**H**(K_1)	**H**(K_2)	**H**(K_3)	**H**(K_4)	**H**(K_5)	…	**H**(K_n)

HORS few-time signatures (2022)

Hash to Obtain a Random Subset

To sign **M**, use a *selection function S*: **M** → indexes

If too many messages are signed, all keys are revealed: insecure

	1	2	3	4	5	…	*n*
Private keys	K_1	K_2	K_3	K_4	K_5	…	K_n
	↓	↓	↓	↓	↓		↓
Public keys	$\mathbf{H}(K_1)$	$\mathbf{H}(K_2)$	$\mathbf{H}(K_3)$	$\mathbf{H}(K_4)$	$\mathbf{H}(K_5)$	…	$\mathbf{H}(K_n)$

HORS few-time signatures (2022)

Hash to Obtain a Random Subset

To sign **M**, use a *selection functi* indexes

If too many mes nsecure

SPHINCS+ uses trees built from the public keys, with 2^6 to 2^{14} values

	1					…	*n*
Private keys	K_1	K_2	K_3	K_4	K_5	…	K_n
	↓	↓	↓	↓	↓		↓
Public keys	$\mathbf{H}(K_1)$	$\mathbf{H}(K_2)$	$\mathbf{H}(K_3)$	$\mathbf{H}(K_4)$	$\mathbf{H}(K_5)$	…	$\mathbf{H}(K_n)$

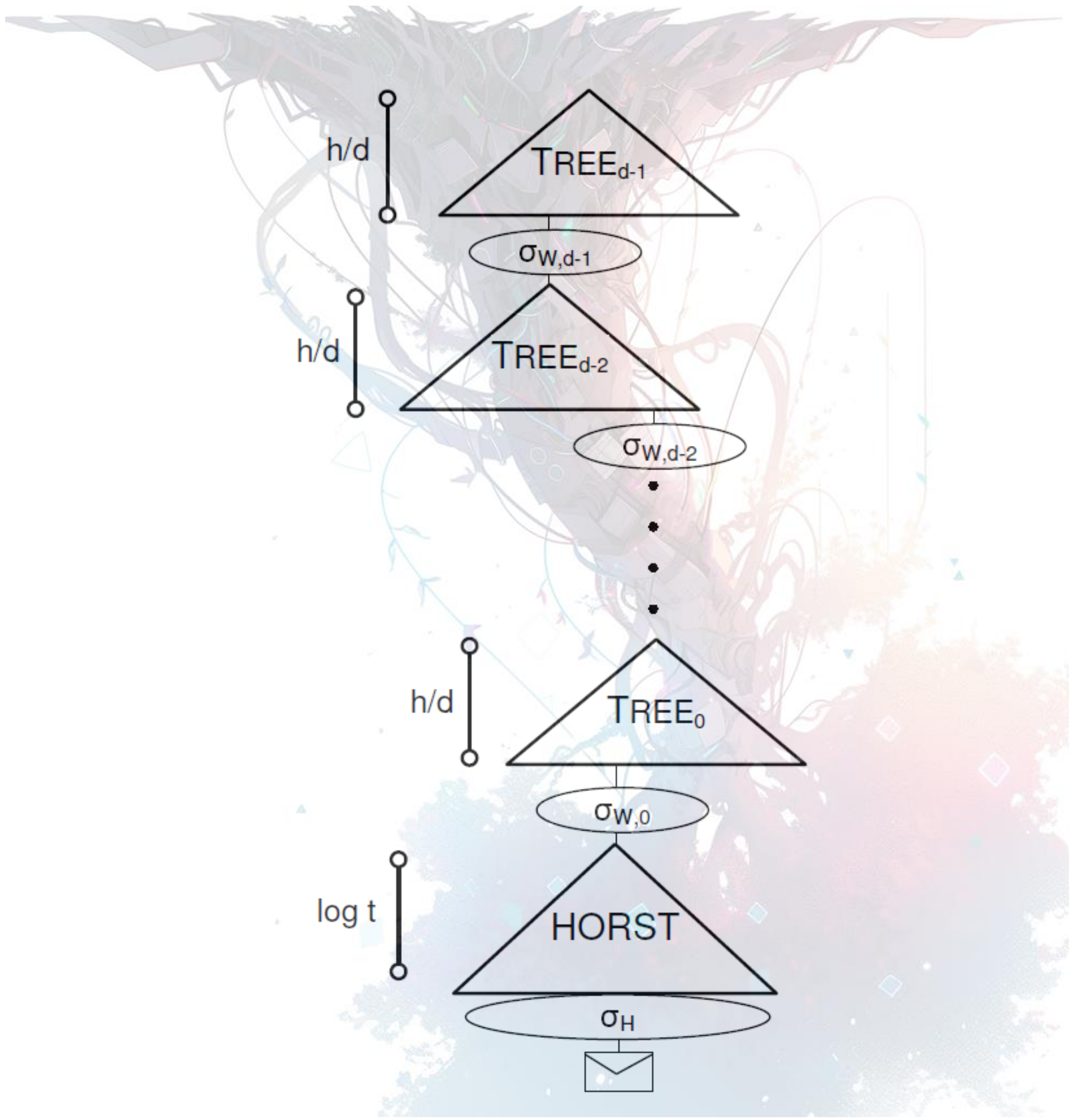
SPHINCS+ design

SPHINCS+ ideas

- Optimize all the previous constructions for security and efficiency
- Tree of trees (“**hypertree**”) where
  - Nodes are Winternitz/Merkle trees (optimized “WOTS+”)
  - Each leaf is a tree of HORS instances (“FORS”, forest of random subsets)
- The **private** key seeds DRBGs to generate
  - WOTS+ instances’ private keys
  - HORST instances’ private keys
- The **public** key includes the
  - Parameters of the tree (as a seed)
  - Root of the hypertree

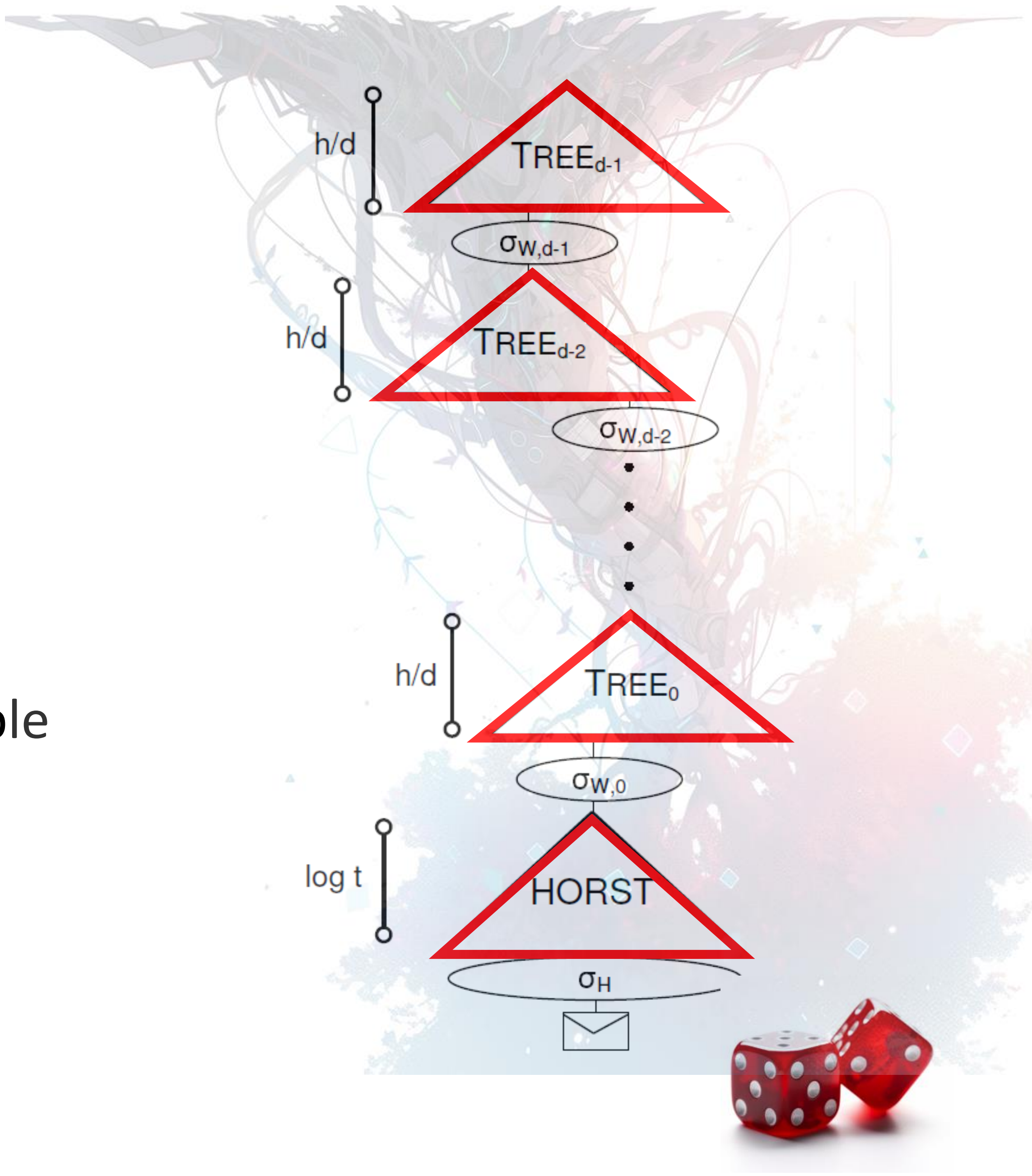
SPHINCS+ ideas

- Each tree's leaf signs the root of a tree underneath
- Messages are signed with a few-time signature (HORS)
- The actual verification only happens at the hypertree's root
- Internal parameters generated with a DRBG



SPHINCS+ signing

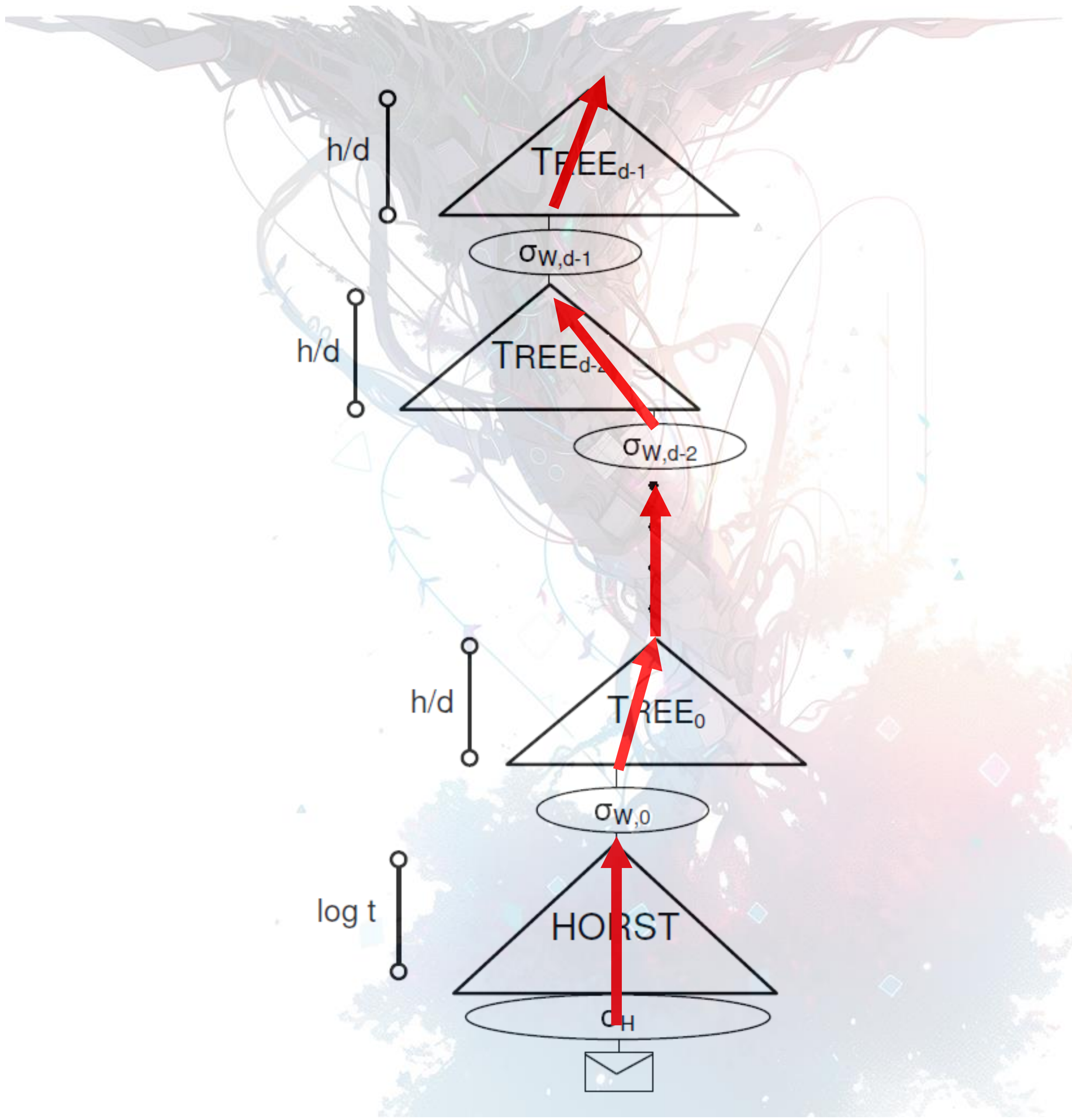
- Pick a random HORS instance
- Reconstruct TREEs to
  - “Sign” each tree’s root from a leaf
  - Compute the authentication path
- Signature consists of:
  - The **seed** used for pick unpredictable HORS signing leaf coordinate used (**H**(sk, m) + *optional randomness*)
  - The **HORS signature**
  - Each **TREE’s signature** (auth path)



SPHINCS+ verification

- Verify HORS instance signature
- "Connect" the HORS instance to the hypertree root (pubkey) by..
- Reconstructing TREE **roots** from authentication paths

No need to recompute tree, much faster than verification



SPHINCS+ crypto primitives

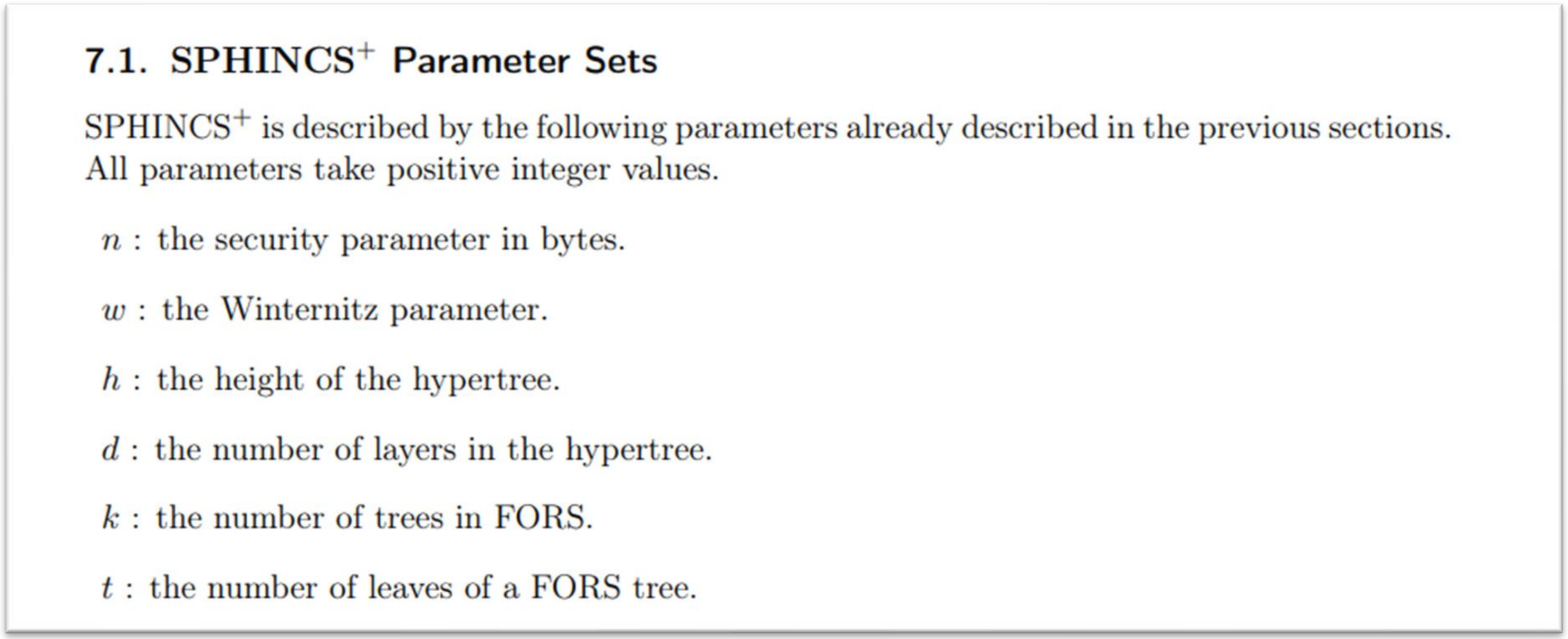
SPHINCS+ needs (tweakable) hashing, PRF, DRBG functionalities

3 options in the NIST submissions:

- Simplest with a keyable XOF: **SHAKE** proposed, as a FIPSable primitive
- **SHA-2** option: need HMAC and the MGF1 construction
- Non-FIPS option: sponge **Haraka**, faster for short input

SPHINCS+ instances

Parameters + choice of hash function + variant “robust” or “simple”



7.1. SPHINCS$^+$ Parameter Sets

SPHINCS$^+$ is described by the following parameters already described in the previous sections. All parameters take positive integer values.

n : the security parameter in bytes.

w : the Winternitz parameter.

h : the height of the hypertree.

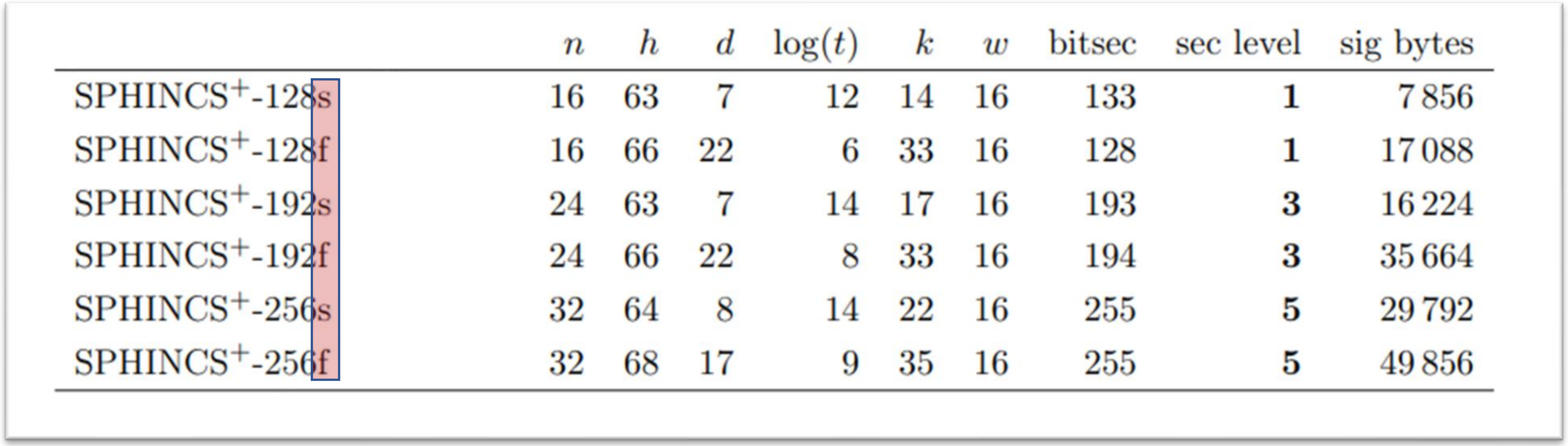
d : the number of layers in the hypertree.

k : the number of trees in FORS.

t : the number of leaves of a FORS tree.

SPHINCS+ instances

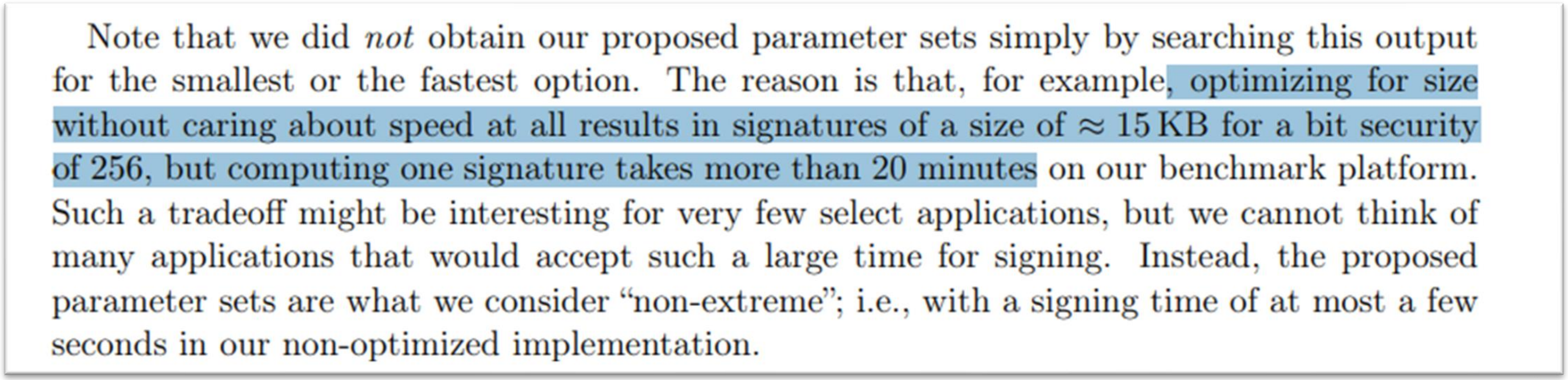
Trade-off speed / signature size (**s**mall & slow vs. **f**ast & large version)



	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

SPHINCS+ instances

Trade-off speed / signature size (**s**mall & slow vs. **f**ast & large version)

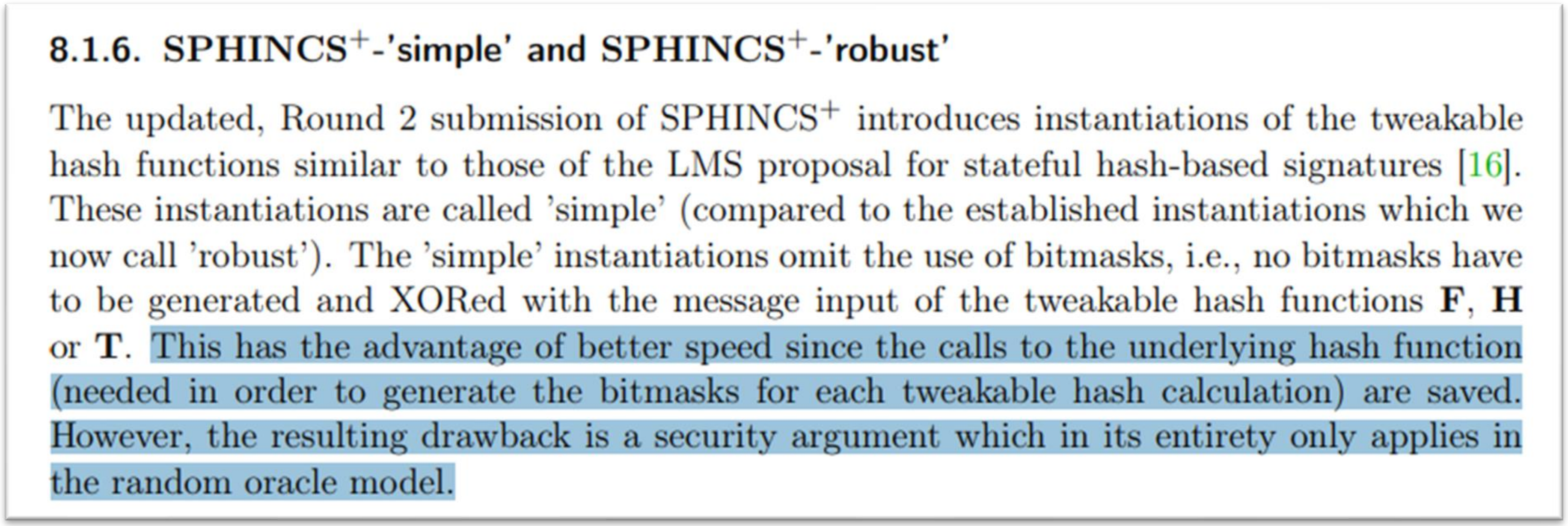


Note that we did *not* obtain our proposed parameter sets simply by searching this output for the smallest or the fastest option. The reason is that, for example, optimizing for size without caring about speed at all results in signatures of a size of $\approx 15\,\text{KB}$ for a bit security of 256, but computing one signature takes more than 20 minutes on our benchmark platform. Such a tradeoff might be interesting for very few select applications, but we cannot think of many applications that would accept such a large time for signing. Instead, the proposed parameter sets are what we consider "non-extreme"; i.e., with a signing time of at most a few seconds in our non-optimized implementation.

SPHINCS+ security

As secure as hash functions

- Game-based PQ-EU-CMA proof
- Requires multi-target second-preimage resistance
- “Collision-resilient”

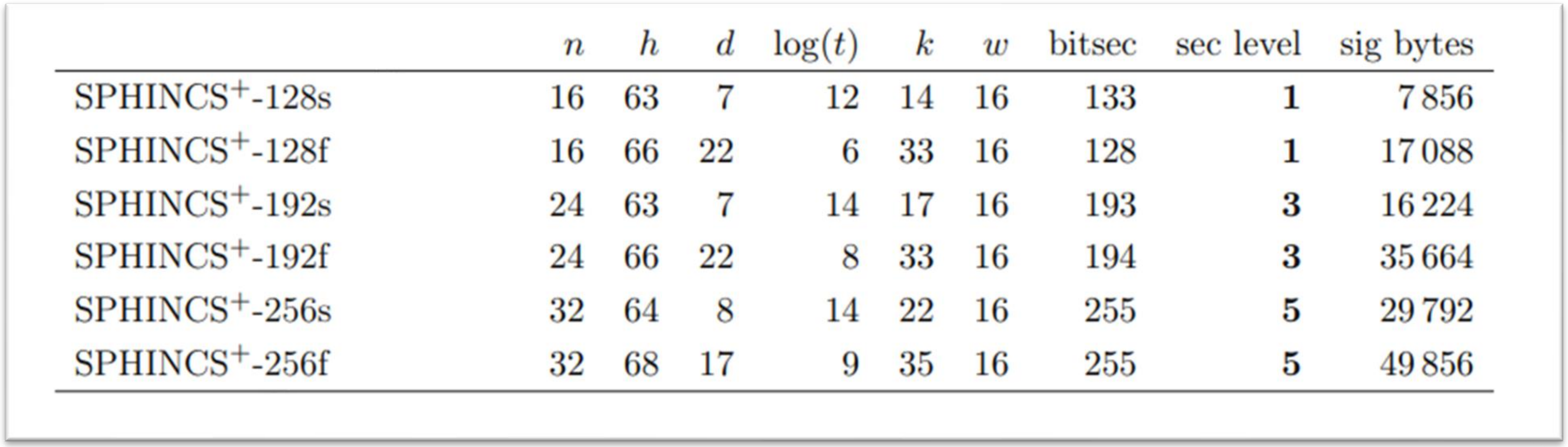


8.1.6. SPHINCS$^+$-'simple' and SPHINCS$^+$-'robust'

The updated, Round 2 submission of SPHINCS$^+$ introduces instantiations of the tweakable hash functions similar to those of the LMS proposal for stateful hash-based signatures [16]. These instantiations are called 'simple' (compared to the established instantiations which we now call 'robust'). The 'simple' instantiations omit the use of bitmasks, i.e., no bitmasks have to be generated and XORed with the message input of the tweakable hash functions **F**, **H** or **T**. This has the advantage of better speed since the calls to the underlying hash function (needed in order to generate the bitmasks for each tweakable hash calculation) are saved. However, the resulting drawback is a security argument which in its entirety only applies in the random oracle model.

Security levels

Depends mainly on the hash output size (from 128 to 256 bits)



	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

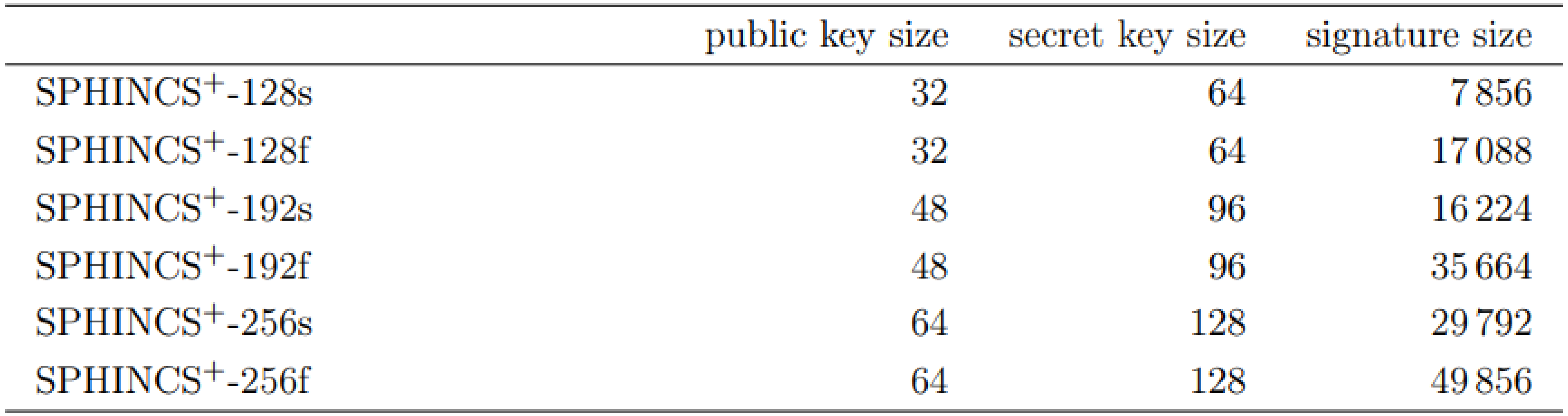
Software security

- Main risk: **incorrect/unsafe code**, owing to SPHINCS+’ complexity
- High assurance against **timing attacks**
- Like all cryptographic algorithms, may require protection against..
  - Fault attacks (laser, power glitches, etc.)
  - Side-channel attacks (EM, DPA, etc.)
- Implementations should include proper **testing**:
  - KATs from the reference code
  - Unit tests
  - Happy and sad paths
  - Arguments sanitization (type, size)

SPHINCS+ performance

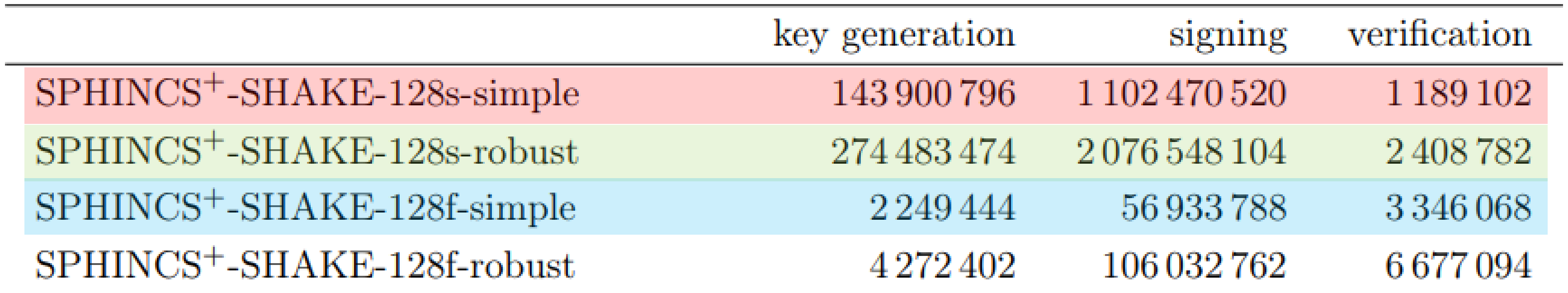
Signature size

Between 7 KiB and 49 KiB, while keys are small



	public key size	secret key size	signature size
SPHINCS$^+$-128s	32	64	7 856
SPHINCS$^+$-128f	32	64	17 088
SPHINCS$^+$-192s	48	96	16 224
SPHINCS$^+$-192f	48	96	35 664
SPHINCS$^+$-256s	64	128	29 792
SPHINCS$^+$-256f	64	128	49 856

Speed (3.1 GHz Haswell Xeon)



	key generation	signing	verification
SPHINCS$^+$-SHAKE-128s-simple	143 900 796	1 102 470 520	1 189 102
SPHINCS$^+$-SHAKE-128s-robust	274 483 474	2 076 548 104	2 408 782
SPHINCS$^+$-SHAKE-128f-simple	2 249 444	56 933 788	3 346 068
SPHINCS$^+$-SHAKE-128f-robust	4 272 402	106 032 762	6 677 094

- **Key gen**: 46, 88, 0.7, 1.3 milliseconds
- **Signing**: 355, 669, 18, 34 milliseconds
- **Verification**: 383, 777, 1079, 2153 microseconds

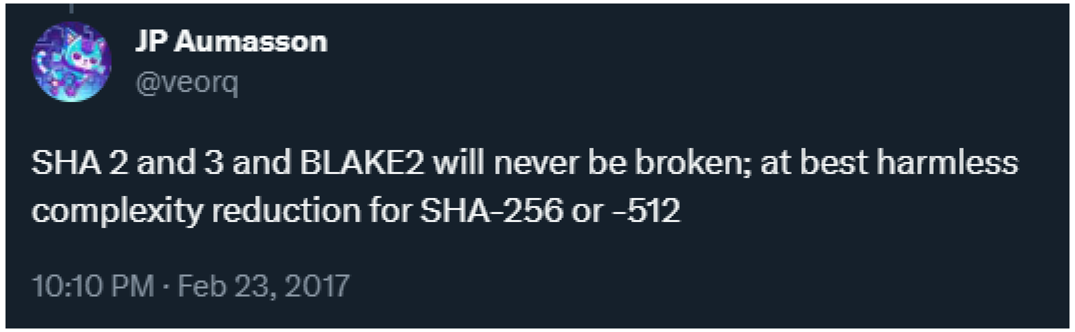
s instances (small sig & slow): Signing ≈ 1000× slow than verification

f instances (fast & large sig): Signing ≈ 15× slow than verification

Conclusion

Slow but reliable

The absence of a structure required for NP-hardness arguments makes SPHINCS+ safer than lattice- or code-based constructions



JP Aumasson
@veorq

SHA 2 and 3 and BLAKE2 will never be broken; at best harmless complexity reduction for SHA-256 or -512

10:10 PM · Feb 23, 2017

Depending on the use case, signatures' size is either a no-go or a non-issue

SPHINCS++

Many tricks and optimizations from XMSS and SPHINCS to SPHINCS+ v3.1

More optimizations possible, and more yet to be found

Challenges:

- Simplifying the constructions
- Simplifying the security arguments and underlying assumptions
- Further “compressing” signatures (more trees?)

Thank you ☺

Thanks to Andreas and Tanja for their feedback

Images AI-generated with Midjourney