MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

An Introduction to hash-based signatures

Peter Schwabe

December 13, 2021

NIST PQC candidates

So many NIST candidates and one thing they all have in common. . .

NIST PQC candidates

So many NIST candidates and one thing they all have in common. . . they all need a hash function.

NIST PQC candidates

So many NIST candidates and one thing they all have in common. . . they all need a hash function.

What can we do with ***just*** a hash function?

Hash-based signatures

- Hash functions map long strings to fixed-length strings
- Standard properties required from a cryptographic hash function:
  - **Collision resistance:** Hard two find two inputs that produce the same output
  - **Preimage resistance:** Given the output, it's hard to find the input
  - **2nd preimage resistance:** Given input and output, it's hard to find a second input, producing the same output

Hash-based signatures

- Hash functions map long strings to fixed-length strings
- Standard properties required from a cryptographic hash function:
  - **Collision resistance:** Hard two find two inputs that produce the same output
  - **Preimage resistance:** Given the output, it's hard to find the input
  - **2nd preimage resistance:** Given input and output, it's hard to find a second input, producing the same output
- Collision resistance is stronger assumption than (2nd) preimage resistance
- Ideally, don't want to rely on collision resistance

Signatures for 0-bit messages

Key generation

- Generate 256-bit random value r (secret key)
- Compute $p = h(r)$ (public key)

Signatures for 0-bit messages

Key generation

- Generate 256-bit random value r (secret key)
- Compute $p = h(r)$ (public key)

Signing

- Send $\sigma = r$

Signatures for 0-bit messages

Key generation

- Generate 256-bit random value r (secret key)
- Compute $p = h(r)$ (public key)

Signing

- Send $\sigma = r$

Verification

- Check that $h(r) = p$

Security of this scheme

- Clearly an attacker who can invert h can break the scheme
- Can we reduce from preimage-resistance to unforgeability?

Security of this scheme

- Clearly an attacker who can invert h can break the scheme
- Can we reduce from preimage-resistance to unforgeability?
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use oracle to compute x, s.t., $h(x) = y$
  - Idea: use public-key $\mathrm{pk} = y$, oracle will compute forgery x

Security of this scheme

- Clearly an attacker who can invert h can break the scheme
- Can we reduce from preimage-resistance to unforgeability?
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use oracle to compute x, s.t., $h(x) = y$
  - Idea: use public-key $\mathrm{pk} = y$, oracle will compute forgery x
  - $\ldots$ or will it?

Security of this scheme

- Clearly an attacker who can invert h can break the scheme
- Can we reduce from preimage-resistance to unforgeability?
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use oracle to compute x, s.t., $h(x) = y$
  - Idea: use public-key $\mathrm{pk} = y$, oracle will compute forgery x
  - $\ldots$ or will it?
- Problem: y is not an output of h
- What if $\mathcal{A}$ can distinguish legit pk from random?
- Need additional property of h: **undetectability**
- From now on assume that all our hash functions are undetectable

Signatures for 1-bit messages

Key generation

- Generate 256-bit random values $(r_0, r_1) = s$ (secret key)
- Compute $(h(r_0), h(r_1)) = (p_0, p_1) = p$ (public key)

Signatures for 1-bit messages

Key generation

- Generate 256-bit random values $(r_0, r_1) = s$ (secret key)
- Compute $(h(r_0), h(r_1)) = (p_0, p_1) = p$ (public key)

Signing

- Signature for message $b = 0$: $\sigma = r_0$
- Signature for message $b = 1$: $\sigma = r_1$

Signatures for 1-bit messages

Key generation

- Generate 256-bit random values $(r_0, r_1) = s$ (secret key)
- Compute $(h(r_0), h(r_1)) = (p_0, p_1) = p$ (public key)

Signing

- Signature for message $b = 0$: $\sigma = r_0$
- Signature for message $b = 1$: $\sigma = r_1$

Verification

Check that $h(\sigma) = p_b$

Security of this scheme

- Same idea as for 0-bit messages: reduce from preimage resistance

Security of this scheme

- Same idea as for 0-bit messages: reduce from preimage resistance
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use "public key" $(h(r_0), y)$ or $(y, h(r_1))$

Security of this scheme

- Same idea as for 0-bit messages: reduce from preimage resistance
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use "public key" $(h(r_0), y)$ or $(y, h(r_1))$
  - $\mathcal{A}$ asks for signature on either 0 or 1
  - If you can, answer with preimage, otherwise fail (abort)

Security of this scheme

- Same idea as for 0-bit messages: reduce from preimage resistance
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use "public key" $(h(r_0), y)$ or $(y, h(r_1))$
  - $\mathcal{A}$ asks for signature on either 0 or 1
  - If you can, answer with preimage, otherwise fail (abort)
  - Now $\mathcal{A}$ returns preimage, i.e., preimage of y

Security of this scheme

- Same idea as for 0-bit messages: reduce from preimage resistance
- Proof game:
  - Assume oracle $\mathcal{A}$ that computes forgery, given public key pk
  - Get input y, use "public key" $(h(r_0), y)$ or $(y, h(r_1))$
  - $\mathcal{A}$ asks for signature on either 0 or 1
  - If you can, answer with preimage, otherwise fail (abort)
  - Now $\mathcal{A}$ returns preimage, i.e., preimage of y
- Reduction only works with $1/2$ probability
- We get a **tightness loss** of $1/2$

One-time signatures for 256-bit messages

Key generation

- Generate 256-bit random values $s = (r_{0,0}, r_{0,1} \ldots, r_{255,0}, r_{255,1})$
- Compute $p = (h(r_{0,0}), h(r_{0,1}), \ldots, h(r_{255,0}), h(r_{255,1})) = (p_{0,0}, p_{0,1}, \ldots, p_{255,0}, p_{255,1})$

One-time signatures for 256-bit messages

Key generation

- Generate 256-bit random values $s = (r_{0,0}, r_{0,1} \dots, r_{255,0}, r_{255,1})$
- Compute $p = (h(r_{0,0}), h(r_{0,1}), \dots, h(r_{255,0}), h(r_{255,1})) = (p_{0,0}, p_{0,1}, \dots, p_{255,0}, p_{255,1})$

Signing

- Signature for message $(b_0, \dots, b_{255})$:
  $\sigma = (\sigma_0, \dots, \sigma_{255}) = (r_{0,b_0}, \dots, r_{255,b_{255}})$

One-time signatures for 256-bit messages

Key generation

- Generate 256-bit random values $s = (r_{0,0}, r_{0,1} \ldots, r_{255,0}, r_{255,1})$
- Compute $p = (h(r_{0,0}), h(r_{0,1}), \ldots, h(r_{255,0}), h(r_{255,1})) =$ $(p_{0,0}, p_{0,1}, \ldots, p_{255,0}, p_{255,1})$

Signing

- Signature for message $(b_0, \ldots, b_{255})$:
  $\sigma = (\sigma_0, \ldots, \sigma_{255}) = (r_{0,b_0,\ldots,}r_{255,b_{255}})$

Verification

- Check that $h(\sigma_0) = p_{0,b_0}$
- $\ldots$
- Check that $h(\sigma_{255}) = p_{255,b_{255}}$

Security of this scheme

- Same idea as before, replace one $p_{j,b}$ in the public key by challenge y
- Fail if signing needs the preimage of y
- In forgery, attacker has to flip at least one bit in m
- Chance of $1/256$ that attacker flips the bit with the challenge
- Overall tightness loss of $1/512$

Winternitz OTS (basic idea)

- Lamport signatures are rather large (8 KB)
- Can we tradeoff speed for size?
- Idea: use $h^w(r)$ intead of $h(r)$ ("hash chains")

Winternitz OTS (basic idea)

- Lamport signatures are rather large (8 KB)
- Can we tradeoff speed for size?
- Idea: use $h^w(r)$ intead of $h(r)$ ("hash chains")

Key generation

- Generate 256-bit random values $r_0, \ldots, r_{63}$ (secret key)
- Compute $(p_0, \ldots, p_{63}) = (h^{15}(r_0), \ldots, h^{15}(r_{63}))$ (public key)

Winternitz OTS (basic idea)

- Lamport signatures are rather large (8 KB)
- Can we tradeoff speed for size?
- Idea: use $h^w(r)$ intead of $h(r)$ ("hash chains")

Key generation

- Generate 256-bit random values $r_0, \ldots, r_{63}$ (secret key)
- Compute $(p_0, \ldots, p_{63}) = (h^{15}(r_0), \ldots, h^{15}(r_{63})$ (public key)

Signing

- Chop 256 bit message into 64 chunks of 4 bits $m = (m_0, \ldots, m_{63})$
- Compute $\sigma = (\sigma_0, \ldots, \sigma_{63}) = (h^{m_0}(r_0), \ldots, h^{m_{63}}(r_{63}))$

Winternitz OTS (basic idea)

- Lamport signatures are rather large (8 KB)
- Can we tradeoff speed for size?
- Idea: use $h^w(r)$ intead of $h(r)$ ("hash chains")

Key generation

- Generate 256-bit random values $r_0, \ldots, r_{63}$ (secret key)
- Compute $(p_0, \ldots, p_{63}) = (h^{15}(r_0), \ldots, h^{15}(r_{63})$ (public key)

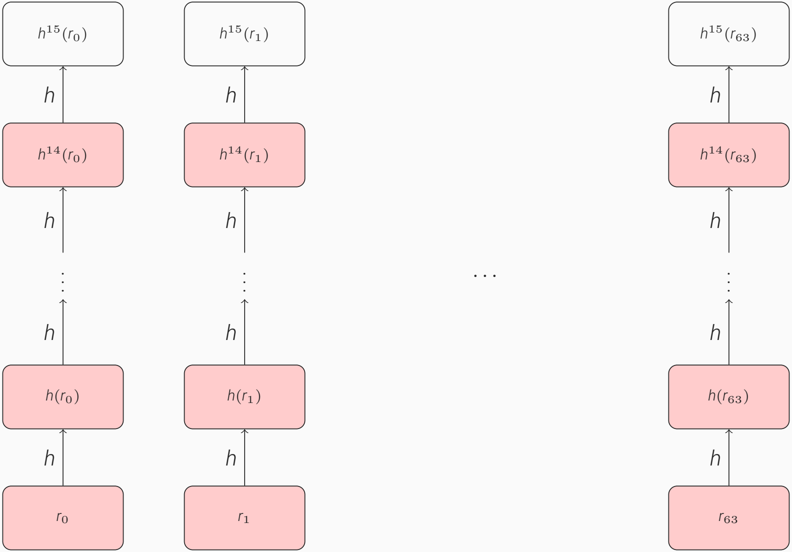
Signing

- Chop 256 bit message into 64 chunks of 4 bits $m = (m_0, \ldots, m_{63})$
- Compute $\sigma = (\sigma_0, \ldots, \sigma_{63}) = (h^{m_0}(r_0), \ldots, h^{m_{63}}(r_{63}))$

Verification

- Check that $p_0 = h^{15-m_0}(\sigma_0), \ldots, p_{63} = h^{15-m_{63}}(\sigma_{63})$

Winternitz OTS (basic idea, ctd.)



Winternitz OTS (making it secure)

- Once you signed, say, $m = (8, m_1, \ldots, m_{63})$, can easily forge signature on $m = (9, m_1, \ldots, m_{63})$
- Idea: introduce checksum, force attacker to "go down" some chain in exchange

Winternitz OTS (making it secure)

- Once you signed, say, $m = (8, m_1, \ldots, m_{63})$, can easily forge signature on $m = (9, m_1, \ldots, m_{63})$
- Idea: introduce checksum, force attacker to "go down" some chain in exchange
- Compute $c = 960 - \sum_{i=0}^{63} m_i \in \{0, \ldots, 960\}$
- Write c in radix 16, obtain c_0, c_1, c_2
- Compute hash chains for c_0, c_1, c_2 as well

Winternitz OTS (making it secure)

- Once you signed, say, $m = (8, m_1, \ldots, m_{63})$, can easily forge signature on $m = (9, m_1, \ldots, m_{63})$
- Idea: introduce checksum, force attacker to "go down" some chain in exchange
- Compute $c = 960 - \sum_{i=0}^{63} m_i \in \{0, \ldots, 960\}$
- Write c in radix 16, obtain c_0, c_1, c_2
- Compute hash chains for c_0, c_1, c_2 as well
- When increasing one of the m_i's, one of the c_i's decreases
- In total obtain 67 hash chains, signatures have 2144 bytes

WOTS notes

- The value $w = 16$ (15 hashes per chain) is tunable
- Can also use, e.g., 256 (chop message into bytes)

WOTS notes

- The value $w = 16$ (15 hashes per chain) is tunable
- Can also use, e.g., 256 (chop message into bytes)
- Lots of tradeoffs between speed and size
  - $w = 16$ yields ≈ 2.1 KB signatures
  - $w = 256$ yields ≈ 1.1 KB signatures
  - However, $w = 256$ makes signing and verification $\approx 8\times$ slower

WOTS notes

- The value $w = 16$ (15 hashes per chain) is tunable
- Can also use, e.g., 256 (chop message into bytes)
- Lots of tradeoffs between speed and size
    - $w = 16$ yields ≈ 2.1 KB signatures
    - $w = 256$ yields ≈ 1.1 KB signatures
    - However, $w = 256$ makes signing and verification $\approx 8\times$ slower
- Verification recovers (and compares) the full public key
- Can publish $h(\text{pk})$ instead of pk

From WOTS to WOTS$^+$

- An attacker who can compute preimages can go backwards in chains
- Problem: hard to prove that this is the only way to forge

From WOTS to WOTS$^+$

- An attacker who can compute preimages can go backwards in chains
- Problem: hard to prove that this is the only way to forge
- Proof needs to go the other way round
- Given forgery oracle, need to compute preimage for some given x
- Can again place preimage challenge anywhere inside the chains

From WOTS to WOTS$^+$

- An attacker who can compute preimages can go backwards in chains
- Problem: hard to prove that this is the only way to forge
- Proof needs to go the other way round
- Given forgery oracle, need to compute preimage for some given x
- Can again place preimage challenge anywhere inside the chains
- Problem: two ways for oracle to forge:
  - compute preimage (solve challenge)
  - find different chain that collides *further up*
- Forgery gives us either preimage *or collision*

From WOTS to WOTS$^+$

- An attacker who can compute preimages can go backwards in chains
- Problem: hard to prove that this is the only way to forge
- Proof needs to go the other way round
- Given forgery oracle, need to compute preimage for some given x
- Can again place preimage challenge anywhere inside the chains
- Problem: two ways for oracle to forge:
  - compute preimage (solve challenge)
  - find different chain that collides *further up*
- Forgery gives us either preimage *or collision*
- Idea (Hülsing, 2013): control one input in that collision, get 2nd preimage!

From WOTS to WOTS$^+$

- An attacker who can compute preimages can go backwards in chains
- Problem: hard to prove that this is the only way to forge
- Proof needs to go the other way round
- Given forgery oracle, need to compute preimage for some given x
- Can again place preimage challenge anywhere inside the chains
- Problem: two ways for oracle to forge:
  - compute preimage (solve challenge)
  - find different chain that collides *further up*
- Forgery gives us either preimage *or collision*
- Idea (Hülsing, 2013): control one input in that collision, get 2nd preimage!
- Replace $h(r)$ by $h(r \oplus b)$ for "bitmask" b
- Include bitmasks in public key
- Reduction can now *choose* inputs to hash function

How about the message hash?

- What if we want to sign messages longer than 256 bits?
- Simple answer: sign $h(m)$
- Requires collision-resistant hash-function h

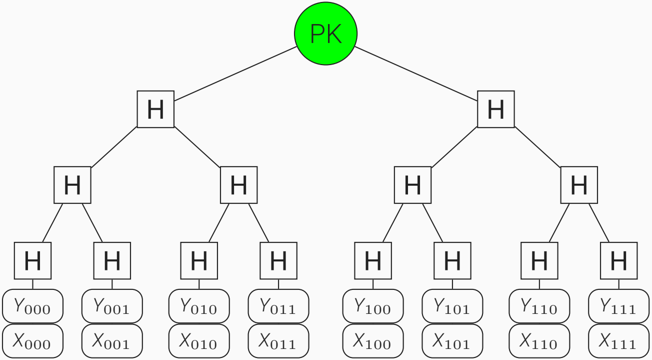
How about the message hash?

- What if we want to sign messages longer than 256 bits?
- Simple answer: sign $h(m)$
- Requires collision-resistant hash-function h
- Idea: randomize before feeding m into h
    - Pick random r
    - Compute $h(r \mid m)$
    - Send r as part of the signature

How about the message hash?

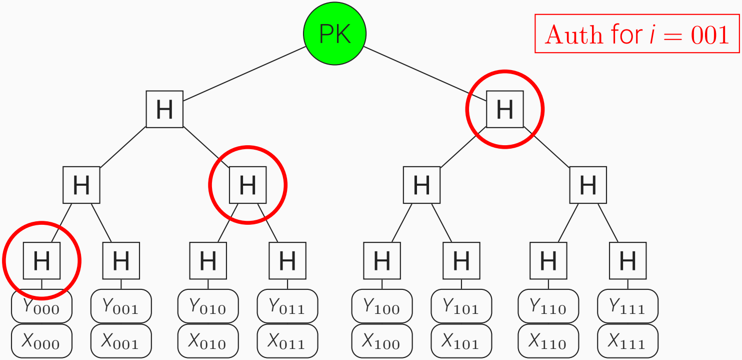
- What if we want to sign messages longer than 256 bits?
- Simple answer: sign $h(m)$
- Requires collision-resistant hash-function h
- Idea: randomize before feeding m into h
  - Pick random r
  - Compute $h(r \mid m)$
  - Send r as part of the signature
- Make deterministic: $r \leftarrow \mathrm{PRF}(s, m)$ for secret s
- Signature scheme is now **collision resilient**

Merkle Trees



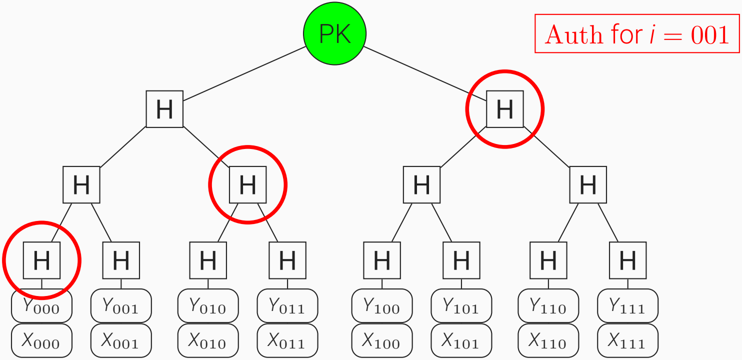
- Merkle, 1979: Leverage one-time signatures to multiple messages
- Binary hash tree on top of OTS public keys

Merkle Trees



- Merkle, 1979: Leverage one-time signatures to multiple messages
- Binary hash tree on top of OTS public keys

Merkle Trees



- Use OTS keys sequentially
- $\mathrm{SIG} = (i, \mathrm{sign}(M, X_i), Y_i, \mathrm{Auth})$
- Signer needs to remember current *index* ($\Rightarrow$ stateful scheme)

Merkle security

- Informally:
  - requires **EUF-CMA-secure** OTS
  - requires collision-resistant hash in the tree
- Can apply bitmask trick to get rid of collision-resistance assumption
- Merkle signatures are **stateful**

Keygen memory usage

- Keygen needs to compute the whole tree from leaves to root
- Naive implementation uses $\Theta(2^h)$ memory

Keygen memory usage

- Keygen needs to compute the whole tree from leaves to root
- Naive implementation uses $\Theta(2^h)$ memory
- Better approach, call TREEHASH for each leaf, left to right:

function TREEHASH(stack, leaf node N)
  while stack.peek() is on the same level as N **do**
    neighbor $\leftarrow$ stack.pop()
    $N \leftarrow H(neighbor || N)$
  end while
  stack.push(N)
end function

Keygen memory usage

- Keygen needs to compute the whole tree from leaves to root
- Naive implementation uses $\Theta(2^h)$ memory
- Better approach, call TREEHASH for each leaf, left to right:

  function TREEHASH(stack, leaf node N)
      while stack.peek() is on the same level as N **do**
          $neighbor \leftarrow$ stack.pop()
          $N \leftarrow H(neighbor || N)$
      end while
      stack.push(N)
  end function

- After going through all leaves, root will be on the top of the stack
- Memory requirement: $h + 1$ hashes

State size vs. signing speed

- KeyGen needs to compute the whole tree, but how about signing?

State size vs. signing speed

- KeyGen needs to compute the whole tree, but how about signing?
- Can simply remember the tree from KeyGen: large secret key

State size vs. signing speed

- KeyGen needs to compute the whole tree, but how about signing?
- Can simply remember the tree from KeyGen: large secret key
- Can recompute tree every time: very slow signing

State size vs. signing speed

- KeyGen needs to compute the whole tree, but how about signing?
- Can simply remember the tree from KeyGen: large secret key
- Can recompute tree every time: very slow signing
- Obvious tradeoff: remember last authentication path
- Most of the time can reuse most nodes

State size vs. signing speed

- KeyGen needs to compute the whole tree, but how about signing?
- Can simply remember the tree from KeyGen: large secret key
- Can recompute tree every time: very slow signing
- Obvious tradeoff: remember last authentication path
- Most of the time can reuse most nodes
- Signing speed now depends largely on index
- Idea: balance computations, store nodes required for future signatures
- Commonly used algorithm (again allowing tradeoffs): **BDS traversal** Buchmann, Dahmen, Schneider, 2008: *Merkle tree traversal revisited* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.420.4170&rep=rep1&type=pdf

Stateful signatures: downside

- Secret key changes with every signature
- Going back to previous secret key is security disaster

Stateful signatures: downside

- Secret key changes with every signature
- Going back to previous secret key is security disaster
- Huge problem in many contexts:
  - Backups
  - VM Snapshots
  - Load balancing
  - API is incompatible!

Stateful signatures: advantage

- Remember forward secrecy?: old ciphertexts remain secure after key compromise
- Signature **forward security**: old signatures remain valid after key compromise

Stateful signatures: advantage

- Remember forward secrecy?: old ciphertexts remain secure after key compromise
- Signature **forward security**: old signatures remain valid after key compromise
- Need “timestamp” baked into signature
- Secret key has to evolve to disable signing “in the past”

Stateful signatures: advantage

- Remember forward secrecy?: old ciphertexts remain secure after key compromise
- Signature **forward security**: old signatures remain valid after key compromise
- Need “timestamp” baked into signature
- Secret key has to evolve to disable signing “in the past”
- For Hash-based signatures:
  - generate OTS secret keys as $s_i = h(s_{i-1})$
  - store only next valid OTS secret key
  - Need to keep hashes of old public keys

Stateful signatures: advantage

- Remember forward secrecy?: old ciphertexts remain secure after key compromise
- Signature **forward security**: old signatures remain valid after key compromise
- Need “timestamp” baked into signature
- Secret key has to evolve to disable signing “in the past”
- For Hash-based signatures:
  - generate OTS secret keys as $s_i = h(s_{i-1})$
  - store only next valid OTS secret key
  - Need to keep hashes of old public keys
- After key compromise publish index of compromised key
- Signatures with lower index remain valid

Multi-tree constructions

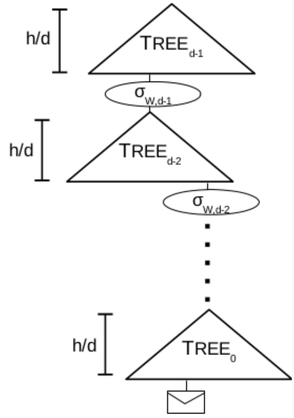
- Remember that KeyGen has to compute the whole tree
- Infeasible for very large trees

Multi-tree constructions

- Remember that KeyGen has to compute the whole tree
- Infeasible for very large trees
- Idea: generate all secret keys pseudo-randomly
- Use PRF on secret seed with position in the tree

Multi-tree constructions

- Remember that KeyGen has to compute the whole tree
- Infeasible for very large trees
- Idea: generate all secret keys pseudo-randomly
- Use PRF on secret seed with position in the tree
- Use hierarchy of trees, **connected via one-time signatures**
- Key generation computes only the top tree
- Many more size-speed tradeoffs



SPHINCS: stateless practical hash-based signatures (2015)

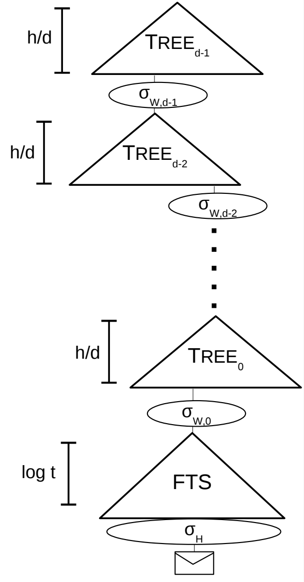
Daniel J. Bernstein
Daira Hopwood
Andreas Hülsing
Tanja Lange
Ruben Niederhagen
Louiza Papachristodoulou
Michael Schneider
Peter Schwabe
Zooko Wilcox-O'Hearn

SPHINCS: stateless practical hash-based incredibly nice cryptographic signatures (2015)

Daniel J. Bernstein
Daira Hopwood
Andreas Hülsing
Tanja Lange
Ruben Niederhagen
Louiza Papachristodoulou
Michael Schneider
Peter Schwabe
Zooko Wilcox-O'Hearn

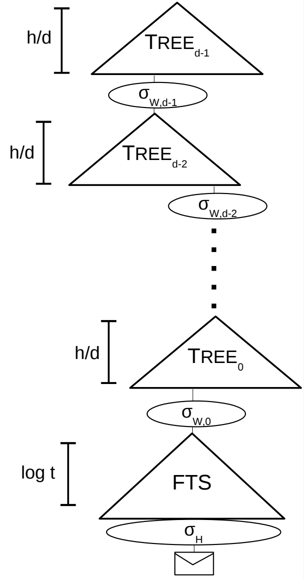
The SPHINCS approach

- Use a "hyper-tree" of total height h
- Parameter $d \geq 1$, such that $d \mid h$
- Each (Merkle) tree has height h/d
- (h/d)-ary certification tree



The SPHINCS approach

- Pick index (pseudo-)randomly
- Messages signed with *few-time* signature scheme
- Significantly reduce total tree height
- Require Pr[r-times Coll] · Pr[Forgery after r signatures] = negl(n)



The HORS few-time signature scheme

- Lamport signatures reveal half of the secret key with each signature

The HORS few-time signature scheme

- Lamport signatures reveal half of the secret key with each signature
- Idea in HORS:
  - use **much bigger** secret key
  - reveal only small portion
  - sign hash $g(m)$; attacker does not control output of g
  - attacker won't have *enough* secret-key to forge

The HORS few-time signature scheme

- Lamport signatures reveal half of the secret key with each signature
- Idea in HORS:
  - use **much bigger** secret key
  - reveal only small portion
  - sign hash $g(m)$; attacker does not control output of g
  - attacker won't have *enough* secret-key to forge
- Example parameters:
  - Generate sk $= (r_0, \ldots, r_{2^{16}})$
  - Compute public key $(h(r_0), \ldots, h(r_{2^{16}}))$

The HORS few-time signature scheme

- Lamport signatures reveal half of the secret key with each signature
- Idea in HORS:
  - use **much bigger** secret key
  - reveal only small portion
  - sign hash $g(m)$; attacker does not control output of g
  - attacker won't have *enough* secret-key to forge
- Example parameters:
  - Generate sk $= (r_0, \ldots, r_{2^{16}})$
  - Compute public key $(h(r_0), \ldots, h(r_{2^{16}}))$
  - Sign 512-bit hash $g(m) = (g_0, \ldots, g_{31})$
  - Each $g_i \in 0, \ldots, 2^{16}$

The HORS few-time signature scheme

- Lamport signatures reveal half of the secret key with each signature
- Idea in HORS:
  - use **much bigger** secret key
  - reveal only small portion
  - sign hash $g(m)$; attacker does not control output of g
  - attacker won't have *enough* secret-key to forge
- Example parameters:
  - Generate sk $= (r_0, \dots, r_{2^{16}})$
  - Compute public key $(h(r_0), \dots, h(r_{2^{16}}))$
  - Sign 512-bit hash $g(m) = (g_0, \dots, g_{31})$
  - Each $g_i \in 0, \dots, 2^{16}$
  - Signature is $(r_{g_0}, \dots, r_{g_{31}})$
  - Signature reveals 32 out of 65536 secret-key values
  - Even after, say, 5 signatures, attacker does not know enough secret key to forge with non-negligible probability

The HORST few-time signature scheme

- Problem with HORS: 2 MB public key
- public key becomes part of signature in complete construction!

The HORST few-time signature scheme

- Problem with HORS: 2 MB public key
- public key becomes part of signature in complete construction!
- Idea:
  - build hash-tree on top of public-key chunks
  - use root of tree as new public key (32 bytes)
  - include authentication paths in signature

The HORST few-time signature scheme

- Problem with HORS: 2 MB public key
- public key becomes part of signature in complete construction!
- Idea:
  - build hash-tree on top of public-key chunks
  - use root of tree as new public key (32 bytes)
  - include authentication paths in signature
- Signature size (naïve):

$$32 \cdot 32 + 32 \cdot 16 \cdot 32 = 17408 \text{ Bytes}$$

The HORST few-time signature scheme

- Problem with HORS: 2 MB public key
- public key becomes part of signature in complete construction!
- Idea:
  - build hash-tree on top of public-key chunks
  - use root of tree as new public key (32 bytes)
  - include authentication paths in signature
- Signature size (naïve):

$$32 \cdot 32 + 32 \cdot 16 \cdot 32 = 17408 \text{ Bytes}$$

- Signature size (somewhat optimized): 13312 Bytes

SPHINCS-256

- Designed for 128 bits of post-quantum security
- Support up to 2^{50} signatures
- 12 trees of height 5 each

SPHINCS-256

- Designed for 128 bits of post-quantum security
- Support up to 2^{50} signatures
- 12 trees of height 5 each
- $n = 256$ bit hashes in WOTS and HORST
- Winternitz paramter $w = 16$
- HORST with 2^{16} expanded-secret-key chunks (total: 2 MB)

SPHINCS-256

- Designed for 128 bits of post-quantum security
- Support up to 2^{50} signatures
- 12 trees of height 5 each
- $n = 256$ bit hashes in WOTS and HORST
- Winternitz paramter $w = 16$
- HORST with 2^{16} expanded-secret-key chunks (total: 2 MB)
- $m = 512$ bit message hash (BLAKE-512)
- ChaCha12 as PRG

Cost of SPHINCS-256 signing

- Three main components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key):
    $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H

Cost of SPHINCS-256 signing

- Three main components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation: $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key): $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H
- Full hash function would be overkill for F and H
- Construction in SPHINCS-256:
  - $F(M_1) = \text{Chop}_{256}(\pi(M_1||C))$
  - $H(M_1||M_2) = \text{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256})))$

Cost of SPHINCS-256 signing

- Three main components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation: $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key): $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H
- Full hash function would be overkill for F and H
- Construction in SPHINCS-256:
  - $F(M_1) = \text{Chop}_{256}(\pi(M_1||C))$
  - $H(M_1||M_2) = \text{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256})))$
- Use fast ChaCha12 permutation for π
- All building blocks (PRG, message hash, H, F) built from very similar permutations

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- ≈ 40 KB signature
- ≈ 1 KB public key (mainly bitmasks)
- ≈ 1 KB private key

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- ≈ 40 KB signature
- ≈ 1 KB public key (mainly bitmasks)
- ≈ 1 KB private key

High-speed implementation

- Target Intel Haswell with 256-bit AVX2 vector instructions
- Use $8\times$ parallel hashing, vectorize on high level
- ≈ 1.6 cycles/byte for H and F

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- ≈ 40 KB signature
- ≈ 1 KB public key (mainly bitmasks)
- ≈ 1 KB private key

High-speed implementation

- Target Intel Haswell with 256-bit AVX2 vector instructions
- Use $8\times$ parallel hashing, vectorize on high level
- ≈ 1.6 cycles/byte for H and F

SPHINCS-256 speed

- Signing: < 52 Mio. Haswell cycles (> 200 sigs/sec, 4 Core, 3GHz)
- Verification: < 1.5 Mio. Haswell cycles
- Keygen: < 3.3 Mio. Haswell cycles

From SPHINCS to SPHINCS$^+$, part I

- Remember tightness loss from many hash calls
- SPHINCS and SPHINCS$^+$ have *many* hash calls

From SPHINCS to SPHINCS$^+$, part I

- Remember tightness loss from many hash calls
- SPHINCS and SPHINCS$^+$ have *many* hash calls
- Think of it as attacker solving one out of many 2nd preimage challenges
- Trivial (pre-quantum) attack:
  - try all inputs of appropriate size
  - win if output matches **any of the challenges**

From SPHINCS to SPHINCS$^+$, part I

- Remember tightness loss from many hash calls
- SPHINCS and SPHINCS$^+$ have *many* hash calls
- Think of it as attacker solving one out of many 2nd preimage challenges
- Trivial (pre-quantum) attack:
  - try all inputs of appropriate size
  - win if output matches **any of the challenges**
- Idea: use different hash function for each call
- Use *address* in the tree to pick hash function

From SPHINCS to SPHINCS$^+$, part I

- Remember tightness loss from many hash calls
- SPHINCS and SPHINCS$^+$ have *many* hash calls
- Think of it as attacker solving one out of many 2nd preimage challenges
- Trivial (pre-quantum) attack:
  - try all inputs of appropriate size
  - win if output matches **any of the challenges**
- Idea: use different hash function for each call
- Use *address* in the tree to pick hash function
- Proposed in 2016 by Hülsing, Rijneveld, and Song
- First adopted in XMSS (see RFC 8391)

From SPHINCS to SPHINCS$^+$, part I

- Remember tightness loss from many hash calls
- SPHINCS and SPHINCS$^+$ have *many* hash calls
- Think of it as attacker solving one out of many 2nd preimage challenges
- Trivial (pre-quantum) attack:
  - try all inputs of appropriate size
  - win if output matches **any of the challenges**
- Idea: use different hash function for each call
- Use *address* in the tree to pick hash function
- Proposed in 2016 by Hülsing, Rijneveld, and Song
- First adopted in XMSS (see RFC 8391)
- Merge with random bitmasks into **tweakable hash function**
- NIST proposal: tweakable hash from SHA-256, SHAKE-256, or Haraka

From SPHINCS to SPHINCS$^+$, part II

- Verifiable index computation:
  - SPHINCS:
    - $(i, r) \leftarrow \mathrm{PRF}(s, m)$,
    - $d \leftarrow h(r, m)$
    - sign digest d with FTS
    - include i in signature

From SPHINCS to SPHINCS$^+$, part II

- Verifiable index computation:
  - SPHINCS:
    - $(i, r) \leftarrow \mathrm{PRF}(s, m)$,
    - $d \leftarrow h(r, m)$
    - sign digest d with FTS
    - include i in signature
  - SPHINCS$^+$:
    - $r \leftarrow \mathrm{PRF}(s, m)$
    - $(i, d) \leftarrow h(r, m)$,
    - sign digest d with FTS
    - include r in signature

From SPHINCS to SPHINCS$^+$, part II

- Verifiable index computation:
  - SPHINCS:
    - $(i, r) \leftarrow \mathrm{PRF}(s, m)$,
    - $d \leftarrow h(r, m)$
    - sign digest d with FTS
    - include i in signature
  - SPHINCS$^+$:
    - $r \leftarrow \mathrm{PRF}(s, m)$
    - $(i, d) \leftarrow h(r, m)$,
    - sign digest d with FTS
    - include r in signature
  - Verifier can check that d and i belong together
  - Attacker cannot pick d and i independently

From SPHINCS to SPHINCS^{+}, part II

- Verifiable index computation:
  - SPHINCS:
    - $(i, r) \leftarrow \mathrm{PRF}(s, m)$,
    - $d \leftarrow h(r, m)$
    - sign digest d with FTS
    - include i in signature
  - SPHINCS^{+}:
    - $r \leftarrow \mathrm{PRF}(s, m)$
    - $(i, d) \leftarrow h(r, m)$,
    - sign digest d with FTS
    - include r in signature
  - Verifier can check that d and i belong together
  - Attacker cannot pick d and i independently
- Additionally: Improvements to FTS (FORS)
- Use multiple smaller trees instead of one big tree
- Per signature, reveal one secret-key leaf per tree

Know more?

https://sphincs.org

Exercises

`https://cryptojedi.org/space2021-hashbased.tar.bz2`

1. Implement Lamport OTS using SHAKE-256 with 256-bit output.
   See file `lamport.c`
2. Implement Winternitz OTS with SHAKE-256 with 256-bit output, using $w = 16$ (chop message into 4-bit chunks).
   See file `winternitz.c`

Exercises

`https://cryptojedi.org/space2021-hashbased.tar.bz2`

1. Implement Lamport OTS using SHAKE-256 with 256-bit output.
   See file `lamport.c`
2. Implement Winternitz OTS with SHAKE-256 with 256-bit output, using $w = 16$ (chop message into 4-bit chunks).
   See file `winternitz.c`

More ideas (not today)

- Implement (stateful) Merkle signatures based on the Winternitz OTS from part 2, using tree height 10.
- Implement forward-secure version
- Implement configurable tradeoff between state size and speed (BDS traversal)