Cryptographic Engineering

An example of post-quantum crypto

Radboud University, Nijmegen, The Netherlands

Spring 2015

Crypto today

- Ephemeral ECDH on ≈ 256-bit curve to compute shared key
- Use EdDSA signatures for public-key authentication
- Use AES-128 for encryption
- Use HMAC-SHA256 for authentication

Crypto today

- Ephemeral ECDH on ≈ 256-bit curve to compute shared key
- Use EdDSA signatures for public-key authentication
- Use AES-128 for encryption
- Use HMAC-SHA256 for authentication

Various alternatives ...

- Traditional DH (in $\mathbb{Z}_p^*$)
- RSA signatures, DSA signatures

Crypto today

- Ephemeral ECDH on ≈ 256-bit curve to compute shared key
- Use EdDSA signatures for public-key authentication
- Use AES-128 for encryption
- Use HMAC-SHA256 for authentication

Various alternatives ...

- Traditional DH (in $\mathbb{Z}_p^*$)
- RSA signatures, DSA signatures
- Stream cipher, e.g., Salsa20
- Other authenticators, e.g., GHASH, Poly1305...

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers will

- break RSA (factoring),

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers will

- break RSA (factoring),
- break DH, DSA (discrete log),

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers will

- break RSA (factoring),
- break DH, DSA (discrete log),
- break ECC (ECDL),

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers will

- break RSA (factoring),
- break DH, DSA (discrete log),
- break ECC (ECDL),
- require doubling symmetric key sizes (e.g., use AES-256 instead of AES-128),

The end of crypto as we know it. . .

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers will

- break RSA (factoring),
- break DH, DSA (discrete log),
- break ECC (ECDL),
- require doubling symmetric key sizes (e.g., use AES-256 instead of AES-128),
- require doubling hash outputs to protect against preimage attacks.

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:
  - **Code-based crypto:** mainly encryption (e.g, McEliece)

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:
  - **Code-based crypto:** mainly encryption (e.g, McEliece)
  - **Lattice-based crypto:** encryption (e.g., NTRU) and signatures

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:
  - **Code-based crypto:** mainly encryption (e.g, McEliece)
  - **Lattice-based crypto:** encryption (e.g., NTRU) and signatures
  - **Multivariate crypto:** encryption and signatures

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:
  - **Code-based crypto:** mainly encryption (e.g, McEliece)
  - **Lattice-based crypto:** encryption (e.g., NTRU) and signatures
  - **Multivariate crypto:** encryption and signatures
  - **Hash-based signatures:** only signatures (e.g., XMSS)

Post-quantum crypto

- Asymmetric crypto that resists attacks by quantum computers
- Four main ideas for constructions:
  - **Code-based crypto:** mainly encryption (e.g, McEliece)
  - **Lattice-based crypto:** encryption (e.g., NTRU) and signatures
  - **Multivariate crypto:** encryption and signatures
  - **Hash-based signatures:** only signatures (e.g., XMSS)
- Less efficient (in time or space), than ECC
- For most of those: underlying problems not as well studied as, e.g., factoring or ECDLP
- Even less studied: attacks by quantum computers

PQCRYPTO

- EU project to make post-quantum cryptography practical
- 11 partners from academia and industry

PQCRYPTO

- EU project to make post-quantum cryptography practical
- 11 partners from academia and industry
- 3 technical work packages:
  - WP1: Post-quantum cryptography for small devices
  - WP2: Post-quantum cryptography for the Internet
  - WP3: Post-quantum cryptography for the cloud

PQCRYPTO

- EU project to make post-quantum cryptography practical
- 11 partners from academia and industry
- 3 technical work packages:
  - WP1: Post-quantum cryptography for small devices
  - WP2: Post-quantum cryptography for the Internet
  - WP3: Post-quantum cryptography for the cloud
- For more information, see `http://pqcrypto.eu/`

Hash-based signatures

- Security relies only on the security of cryptographic hash function

Hash-based signatures

- Security relies only on the security of cryptographic hash function
- Even if *one* hash function turns out to be insecure, can switch to another one

Hash-based signatures

- Security relies only on the security of cryptographic hash function
- Even if *one* hash function turns out to be insecure, can switch to another one
- If *all* hash functions are insecure, we're in bigger trouble anyway

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output
- **Key generation:**
  - Private key: (pseudo-)random $((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \ldots, (s_{255,0}, s_{255,1}))$, each $s_{i,j} \in \{0, 2^{256} - 1\}$
  - Public key: $((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), \ldots, (h(s_{255,0}), h(s_{255,1})))$

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output
- **Key generation:**
  - Private key: (pseudo-)random $((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \ldots, (s_{255,0}, s_{255,1}))$, each $s_{i,j} \in \{0, 2^{256} - 1\}$
  - Public key: $((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), \ldots, (h(s_{255,0}), h(s_{255,1})))$
- **Signing:**
  - Sign messages (hashes) of 256 bits $(m_0, \ldots, m_{255})$
  - Signature is $(s_{0,m_0}, s_{1,m_1}, s_{2,m_2}, \ldots, s_{255,m_{255}})$

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output
- **Key generation:**
  - Private key: (pseudo-)random $((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \ldots, (s_{255,0}, s_{255,1}))$, each $s_{i,j} \in \{0, 2^{256} - 1\}$
  - Public key: $((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), \ldots, (h(s_{255,0}), h(s_{255,1})))$
- **Signing:**
  - Sign messages (hashes) of 256 bits $(m_0, \ldots, m_{255})$
  - Signature is $(s_{0,m_0}, s_{1,m_1}, s_{2,m_2}, \ldots, s_{255,m_{255}})$
- **Verification:**
  - Compare hashes of signature components to elements of the public key

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output
- **Key generation:**
  - Private key: (pseudo-)random $((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \dots, (s_{255,0}, s_{255,1}))$, each $s_{i,j} \in \{0, 2^{256} - 1\}$
  - Public key: $((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), \dots, (h(s_{255,0}), h(s_{255,1})))$
- **Signing:**
  - Sign messages (hashes) of 256 bits $(m_0, \dots, m_{255})$
  - Signature is $(s_{0,m_0}, s_{1,m_1}, s_{2,m_2}, \dots, s_{255,m_{255}})$
- **Verification:**
  - Compare hashes of signature components to elements of the public key
- Secure only for a signature on *one* message

Lamport signatures

- One-time signature (OTS) scheme proposed by Lamport in 1979.
- Use cryptographic hash function h with 256-bit output
- **Key generation:**
  - Private key: (pseudo-)random $((s_{0,0}, s_{0,1}), (s_{1,0}, s_{1,1}), (s_{2,0}, s_{2,1}), \ldots, (s_{255,0}, s_{255,1}))$, each $s_{i,j} \in \{0, 2^{256} - 1\}$
  - Public key: $((h(s_{0,0}), h(s_{0,1})), (h(s_{1,0}), h(s_{1,1})), \ldots, (h(s_{255,0}), h(s_{255,1})))$
- **Signing:**
  - Sign messages (hashes) of 256 bits $(m_0, \ldots, m_{255})$
  - Signature is $(s_{0,m_0}, s_{1,m_1}, s_{2,m_2}, \ldots, s_{255,m_{255}})$
- **Verification:**
  - Compare hashes of signature components to elements of the public key
- Secure only for a signature on *one* message
- 16 KB private and public key, 8 KB signature

Merkle Trees

- Merkle, 1979: Leverage one-time signatures to multiple messages
- Idea: Put a binary hash tree on top of all public keys:
  - Leaves are hashes of public keys
  - All other nodes are hashes of their two child nodes

[picture on the blackboard]

Merkle Trees

- Merkle, 1979: Leverage one-time signatures to multiple messages
- Idea: Put a binary hash tree on top of all public keys:
  - Leaves are hashes of public keys
  - All other nodes are hashes of their two child nodes
- Maximal amount of messages to sign is fixed (number of leaves)

[picture on the blackboard]

Merkle Trees

- Merkle, 1979: Leverage one-time signatures to multiple messages
- Idea: Put a binary hash tree on top of all public keys:
  - Leaves are hashes of public keys
  - All other nodes are hashes of their two child nodes
- Maximal amount of messages to sign is fixed (number of leaves)
- Public key is the root node of the tree (256 bits)

[picture on the blackboard]

Merkle Trees

- Merkle, 1979: Leverage one-time signatures to multiple messages
- Idea: Put a binary hash tree on top of all public keys:
  - Leaves are hashes of public keys
  - All other nodes are hashes of their two child nodes
- Maximal amount of messages to sign is fixed (number of leaves)
- Public key is the root node of the tree (256 bits)
- Signature is the one-time signature plus *authentication path*

[picture on the blackboard]

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)
- Signature size: ≈ 25 KB
  - 8 KB Lamport Signature
  - 16 KB Lamport public key
  - $32 \cdot 32 = 1024$ bytes authentication path
  - 4 bytes for the index of the leaf node

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)
- Signature size: ≈ 25 KB
  - 8 KB Lamport Signature
  - 16 KB Lamport public key
  - $32 \cdot 32 = 1024$ bytes authentication path
  - 4 bytes for the index of the leaf node
- Practical…?

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)
- Signature size: ≈ 25 KB
  - 8 KB Lamport Signature
  - 16 KB Lamport public key
  - $32 \cdot 32 = 1024$ bytes authentication path
  - 4 bytes for the index of the leaf node
- Practical...?
  - Sizes and speeds are not too bad
  - Can even make signatures smaller (more later)

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)
- Signature size: ≈ 25 KB
  - 8 KB Lamport Signature
  - 16 KB Lamport public key
  - $32 \cdot 32 = 1024$ bytes authentication path
  - 4 bytes for the index of the leaf node
- Practical. . . ?
  - Sizes and speeds are not too bad
  - Can even make signatures smaller (more later)

A first analysis

- Let's fix 2^{32} signatures (≈ 4 Bio.)
- Key generation needs to compute the whole tree ($2^{33} - 1$ hashes)
- Signing remembers the previous authentication path
- Most of the time, need to compute only a few hashes for signing
- Public-key size: 32 bytes
- Secret-key: seed for the one-time-signature secret keys (e.g., 32 bytes)
- Signature size: ≈ 25 KB
  - 8 KB Lamport Signature
  - 16 KB Lamport public key
  - $32 \cdot 32 = 1024$ bytes authentication path
  - 4 bytes for the index of the leaf node
- Practical...?
  - Sizes and speeds are not too bad
  - Can even make signatures smaller (more later)
  - **We need to remember the state!**

The state

- Remembering the state means updating the secret key after each signing

The state

- Remembering the state means updating the secret key after each signing
- This is not compatible with
  - Backups
  - Keys shared across devices
  - Virtual-machine images
  - ...

The state

- Remembering the state means updating the secret key after each signing
- This is not compatible with
  - Backups
  - Keys shared across devices
  - Virtual-machine images
  - ...
- This is not even compatible with the *definition* of cryptographic signatures

ELIMINATE
THE STATE

Goldreich's approach

- Goldreich, 1986: stateless hash-based signatures
- Idea: Use binary tree as in Merkle, but
  - make the tree huge (e.g., height $h = 256$), such that one can pick leaves *at random*;
  - each node corresponds to an OTS key pair;
  - leaf nodes are used to sign messages;
  - non-leaf nodes are used to sign the hash of the public keys of the two child nodes.
- All OTS secret keys are generated from a seed

Analysis of Goldreich's approach

- Public key and secret are still small (e.g., 32 bytes)
- Key generation is fast (only generate root OTS key pair)

Analysis of Goldreich's approach

- Public key and secret are still small (e.g., 32 bytes)
- Key generation is fast (only generate root OTS key pair)
- Signing requires $2h = 512$ OTS key generations and $h = 256$ OTS signatures

Analysis of Goldreich's approach

- Public key and secret are still small (e.g., 32 bytes)
- Key generation is fast (only generate root OTS key pair)
- Signing requires $2h = 512$ OTS key generations and $h = 256$ OTS signatures
- Signature becomes very large, for example with Lamport OTS:
  - $256 \cdot 24\,$KB for Lamport signatures and public keys
  - $256 \cdot 32$bytes for authentication paths
  - 32 bytes for the index of the leaf node

Analysis of Goldreich's approach

- Public key and secret are still small (e.g., 32 bytes)
- Key generation is fast (only generate root OTS key pair)
- Signing requires $2h = 512$ OTS key generations and $h = 256$ OTS signatures
- Signature becomes very large, for example with Lamport OTS:
  - $256 \cdot 24$ KB for Lamport signatures and public keys
  - $256 \cdot 32$bytes for authentication paths
  - 32 bytes for the index of the leaf node
- Total size of 6 MB
- More efficient OTS helps, but still very large signatures

SPHINCS

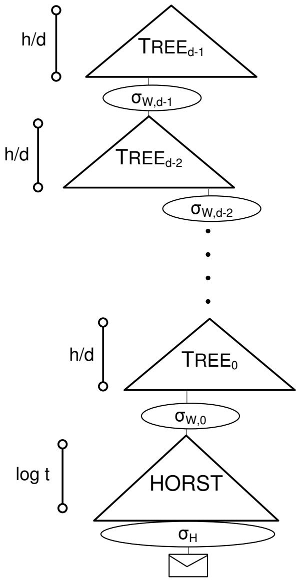
- Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, and Wilcox-O'Hearn, 2015:

  SPHINCS – Stateless, practical, hash-based, incredibly nice cryptographic signatures

SPHINCS

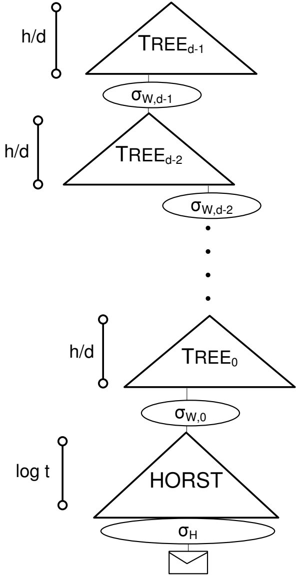
A high-level view on SPHINCS

- Use a "hyper-tree" of total height h
- Each tree has height h/d
- Inside the tree use Merkle approach
- Between trees use Goldreich approach



A high-level view on SPHINCS

- Use a "hyper-tree" of total height h
- Each tree has height h/d
- Inside the tree use Merkle approach
- Between trees use Goldreich approach
- Sign messages with a *few-time* signature scheme
- Significantly reduce total tree height



A zoom into SPHINCS

- We propose SPHINCS-256 for 128 bits of security
- In the following, only consider (slightly simplified) SPHINCS-256:
  - 12 trees of height 5 each
  - Use **WOTS** as one-time-signature scheme
  - Use **HORST** (HORS with tree) as few-time signature scheme
  - Fix $n = 256$ as bitlength of hashes in WOTS and HORST
  - Fix $m = 512$ as size of the message hash (BLAKE-512 hash function)
  - Use ChaCha12 as pseudorandom generator
- SPHINCS-256 really uses WOTS$^+$ instead of WOTS
- Some more modifications required for security proofs

Deterministic, collision-resilient, signing

- Typical setup for stateless hash-based signatures (e.g., Goldreich):
  - Obtain message M, compute $h(M)$
  - Sign $h(M)$ using random leaf from the tree

Deterministic, collision-resilient, signing

- Typical setup for stateless hash-based signatures (e.g., Goldreich):
  - Obtain message M, compute $h(M)$
  - Sign $h(M)$ using random leaf from the tree
- Two disadvantages of this approach:
  - Security requires collision resistance of H
  - Security depends on randomness generator

Deterministic, collision-resilient, signing

- Typical setup for stateless hash-based signatures (e.g., Goldreich):
  - Obtain message M, compute $h(M)$
  - Sign $h(M)$ using random leaf from the tree
- Two disadvantages of this approach:
  - Security requires collision resistance of H
  - Security depends on randomness generator
- Approach in SPHINCS:
  - Include long-term secret SK_2 in private key
  - Compute
    $= \text{BLAKE-512}(SK_2||M) = (R_1, R_2) \in \{0,1\}^{256} \times \{0,1\}^{256}$
  - Sign $D = \text{BLAKE-512}(R_1||M)$; include R_1 in the signature
  - Use last 60 bits of R_2 to select a leaf

Deterministic, collision-resilient, signing

- Typical setup for stateless hash-based signatures (e.g., Goldreich):
  - Obtain message M, compute $h(M)$
  - Sign $h(M)$ using random leaf from the tree
- Two disadvantages of this approach:
  - Security requires collision resistance of H
  - Security depends on randomness generator
- Approach in SPHINCS:
  - Include long-term secret SK_2 in private key
  - Compute
    $= \text{BLAKE-512}(SK_2||M) = (R_1, R_2) \in \{0,1\}^{256} \times \{0,1\}^{256}$
  - Sign $D = \text{BLAKE-512}(R_1||M)$; include R_1 in the signature
  - Use last 60 bits of R_2 to select a leaf
- Additional advantage of this deterministic signing: easier testing

Deterministic, collision-resilient, signing

- Typical setup for stateless hash-based signatures (e.g., Goldreich):
  - Obtain message M, compute $h(M)$
  - Sign $h(M)$ using random leaf from the tree
- Two disadvantages of this approach:
  - Security requires collision resistance of H
  - Security depends on randomness generator
- Approach in SPHINCS:
  - Include long-term secret SK_2 in private key
  - Compute
    $= \text{BLAKE-512}(SK_2||M) = (R_1, R_2) \in \{0,1\}^{256} \times \{0,1\}^{256}$
  - Sign $D = \text{BLAKE-512}(R_1||M)$; include R_1 in the signature
  - Use last 60 bits of R_2 to select a leaf
- Additional advantage of this deterministic signing: easier testing
- Similar trick in Ed25519 signatures (this is not specific to hash-based signatures!)

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \dots, sk_{t-1})$

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \ldots, sk_{t-1})$
- HORS public key: $H(sk_0), \ldots, H(sk_{t-1})$

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \ldots, sk_{t-1})$
- HORS public key: $H(sk_0), \ldots, H(sk_{t-1})$
- HORST public key: root of a Merkle tree on top of the HORS public key

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \ldots, sk_{t-1})$
- HORS public key: $H(sk_0), \ldots, H(sk_{t-1})$
- HORST public key: root of a Merkle tree on top of the HORS public key
- **Signing:**
  - Chop 512-bit message digest into k chunks $(m_0, \ldots, m_{k-1})$

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \ldots, sk_{t-1})$
- HORS public key: $H(sk_0), \ldots, H(sk_{t-1})$
- HORST public key: root of a Merkle tree on top of the HORS public key
- **Signing:**
  - Chop 512-bit message digest into k chunks $(m_0, \ldots, m_{k-1})$
  - Signature consists of k parts $(sk_{m_i}, \mathsf{Auth}_{m_i})$

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \dots, sk_{t-1})$
- HORS public key: $H(sk_0), \dots, H(sk_{t-1})$
- HORST public key: root of a Merkle tree on top of the HORS public key
- **Signing:**
  - Chop 512-bit message digest into k chunks $(m_0, \dots, m_{k-1})$
  - Signature consists of k parts $(sk_{m_i}, \mathrm{Auth}_{m_i})$
  - Auth_{m_i} is the authentication path in the Merkle tree

HORST

- Idea in SPHINCS: use a *few-time* signature scheme to sign the message digest
- HORST uses two parameters: $k = 32$ and $t = 2^{16}$
- Need that $k \cdot \log_2 t$ equals the length of the message hash
- HORS(T) secret key: t 256-bit pseudorandom values $(sk_0, \ldots, sk_{t-1})$
- HORS public key: $H(sk_0), \ldots, H(sk_{t-1})$
- HORST public key: root of a Merkle tree on top of the HORS public key
- **Signing:**
  - Chop 512-bit message digest into k chunks $(m_0, \ldots, m_{k-1})$
  - Signature consists of k parts $(sk_{m_i}, \mathsf{Auth}_{m_i})$
  - Auth_{m_i} is the authentication path in the Merkle tree
- Each signature reveals $k = 32$ out of 2^{16} secret-key pieces
- Can sign several times before an attacker has a good chance of having enough pieces

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream
- Going from the HORS secret key to the public key requires n-bit-to-n-bit hashing
- In our case: 256-bit-to-256-bit hashing F

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream
- Going from the HORS secret key to the public key requires n-bit-to-n-bit hashing
- In our case: 256-bit-to-256-bit hashing F
- Going from HORS public key to HORST public key needs $2n$-bit-to-n-bit hashing
- In our case: 512-bit-to-256-bit hashing H

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream
- Going from the HORS secret key to the public key requires n-bit-to-n-bit hashing
- In our case: 256-bit-to-256-bit hashing F
- Going from HORS public key to HORST public key needs $2n$-bit-to-n-bit hashing
- In our case: 512-bit-to-256-bit hashing H
- In total $2^{16} = 65536$ invocations of F
- In total $2^{16} - 1 = 65535$ invocations of H

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream
- Going from the HORS secret key to the public key requires n-bit-to-n-bit hashing
- In our case: 256-bit-to-256-bit hashing F
- Going from HORS public key to HORST public key needs $2n$-bit-to-n-bit hashing
- In our case: 512-bit-to-256-bit hashing H
- In total $2^{16} = 65536$ invocations of F
- In total $2^{16} - 1 = 65535$ invocations of H
- Note that F and H are much more special than a general cryptographic hash function (fixed input size!)

Analysis of HORST

- Secret-key expansion needs to generate 2MB of key stream
- Going from the HORS secret key to the public key requires n-bit-to-n-bit hashing
- In our case: 256-bit-to-256-bit hashing F
- Going from HORS public key to HORST public key needs $2n$-bit-to-n-bit hashing
- In our case: 512-bit-to-256-bit hashing H
- In total $2^{16} = 65536$ invocations of F
- In total $2^{16} - 1 = 65535$ invocations of H
- Note that F and H are much more special than a general cryptographic hash function (fixed input size!)
- Signing needs to compute 32 authentication paths
- Can compute the whole tree, extract required nodes
- Can also use more memory-friendly algorithm, extract nodes on the fly

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$
- Derive values $\ell_1 = \lceil (n/\log_2 w) \rceil = 64$ and
  $\ell_2 = \lfloor (\log_2 (\ell_1(w-1)))/\log_2 w \rfloor + 1 = 3$; set $\ell = \ell_1 + \ell_2$

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$
- Derive values $\ell_1 = \lceil (n/\log_2 w) \rceil = 64$ and
  $\ell_2 = \lfloor (\log_2 (\ell_1 (w-1)))/\log_2 w \rfloor + 1 = 3$; set $\ell = \ell_1 + \ell_2$
- **Secret key:** ℓ pseudorandom 256-bit values $(sk_0, \dots, sk_{\ell-1})$
- **Public key:** $(F^{w-1}(sk_0), \dots, F^{w-1}(sk_{\ell-1}))$

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$
- Derive values $\ell_1 = \lceil (n / \log_2 w) \rceil = 64$ and $\ell_2 = \lfloor (\log_2 (\ell_1 (w-1))) / \log_2 w \rfloor + 1 = 3$; set $\ell = \ell_1 + \ell_2$
- **Secret key:** ℓ pseudorandom 256-bit values $(sk_0, \ldots, sk_{\ell-1})$
- **Public key:** $(F^{w-1}(sk_0), \ldots, F^{w-1}(sk_{\ell-1}))$
- Signing of 256-bit message: chop into w-bit chunks $(m_0, \ldots, m_{\ell_1 - 1})$
- Compute $C = \sum_{i=0}^{\ell_1 - 1} (w - 1 - m_i)$, write as $(c_0, \ldots, c_{\ell_2 - 1})$
- **Signature:** $\sigma = (\sigma_0, \ldots, \sigma_{\ell-1}) = (F^{m_0}(sk_0), \ldots, F^{m_{\ell_1 - 1}}(sk_{\ell_1 - 1}), F^{c_0}(sk_{\ell_1}), \ldots, F^{c_{\ell_2 - 1}}(sk_{\ell - 1}))$

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$
- Derive values $\ell_1 = \lceil (n/\log_2 w) \rceil = 64$ and $\ell_2 = \lfloor (\log_2 (\ell_1(w-1)))/\log_2 w \rfloor + 1 = 3$; set $\ell = \ell_1 + \ell_2$
- **Secret key:** ℓ pseudorandom 256-bit values $(sk_0, \dots, sk_{\ell-1})$
- **Public key:** $(F^{w-1}(sk_0), \dots, F^{w-1}(sk_{\ell-1}))$
- Signing of 256-bit message: chop into w-bit chunks $(m_0, \dots, m_{\ell_1-1})$
- Compute $C = \sum_{i=0}^{\ell_1-1}(w - 1 - m_i)$, write as $(c_0, \dots, c_{\ell_2-1})$
- **Signature:** $\sigma = (\sigma_0, \dots, \sigma_{\ell-1}) = (F^{m_0}(sk_0), \dots, F^{m_{\ell_1-1}}(sk_{\ell_1-1}), F^{c_0}(sk_{\ell_1}), \dots, F^{c_{\ell_2-1}}(sk_{\ell-1}))$
- **Verification:** "Finish computing the hash chains", compare to public key

WOTS

- WOTS stands for *Winternitz one-time signatures*
- Uses *Winternitz parameter* w; for SPHINCS-256: $w = 16$
- Derive values $\ell_1 = \lceil (n / \log_2 w) \rceil = 64$ and
  $\ell_2 = \lfloor (\log_2 (\ell_1 (w-1))) / \log_2 w \rfloor + 1 = 3$; set $\ell = \ell_1 + \ell_2$
- **Secret key:** ℓ pseudorandom 256-bit values $(sk_0, \dots, sk_{\ell-1})$
- **Public key:** $(F^{w-1}(sk_0), \dots, F^{w-1}(sk_{\ell-1}))$
- Signing of 256-bit message: chop into w-bit chunks $(m_0, \dots, m_{\ell_1 - 1})$
- Compute $C = \sum_{i=0}^{\ell_1 - 1} (w - 1 - m_i)$, write as $(c_0, \dots, c_{\ell_2 - 1})$
- **Signature:** $\sigma = (\sigma_0, \dots, \sigma_{\ell-1}) =$
  $(F^{m_0}(sk_0), \dots, F^{m_{\ell_1 - 1}}(sk_{\ell_1 - 1}), F^{c_0}(sk_{\ell_1}), \dots, F^{c_{\ell_2 - 1}}(sk_{\ell-1}))$
- **Verification:** "Finish computing the hash chains", compare to public key
- Note: SPHINCS does not sign the hash of the public key, but the root of an L-tree on top of the WOTS public key
- An L-tree is a binary tree where nodes without siblings get promoted

Analysis of WOTS

- Crucial for SPHINCS performance: WOTS key generation
- $15 \cdot 67 = 1005$ invocations of F

Analysis of WOTS

- Crucial for SPHINCS performance: WOTS key generation
- $15 \cdot 67 = 1005$ invocations of F
- Computation of L-tree: 66 invocations of H

Analysis of WOTS

- Crucial for SPHINCS performance: WOTS key generation
- $15 \cdot 67 = 1005$ invocations of F
- Computation of L-tree: 66 invocations of H
- WOTS signature size: $32 \cdot 67 = 2144$ bytes

Hashing

- The performance of SPHINCS-256 is largely determined by
  - n-bit-to-n-bit hashing (F), and
  - $2n$-bit-to-n-bit hashing (H).
- Applying a full-fledged hash function would be overkill

Hashing

- The performance of SPHINCS-256 is largely determined by
  - n-bit-to-n-bit hashing (F), and
  - $2n$-bit-to-n-bit hashing (H).
- Applying a full-fledged hash function would be overkill
- Idea: use a fast permutation π, compute
  - $F(M_1) = \mathsf{Chop}(\pi(M_1||C), 256)$
  - $H(M_1||M_2) = \mathsf{Chop}(\pi(\pi(M_1||C) \oplus (M_2||0^p)), 256)$

Hashing

- The performance of SPHINCS-256 is largely determined by
  - n-bit-to-n-bit hashing (F), and
  - $2n$-bit-to-n-bit hashing (H).
- Applying a full-fledged hash function would be overkill
- Idea: use a fast permutation π, compute
  - $F(M_1) = \mathsf{Chop}(\pi(M_1||C), 256)$
  - $H(M_1||M_2) = \mathsf{Chop}(\pi(\pi(M_1||C) \oplus (M_2||0^p)), 256)$
- This is secure under certain assumptions about π

Hashing

- The performance of SPHINCS-256 is largely determined by
  - n-bit-to-n-bit hashing (F), and
  - $2n$-bit-to-n-bit hashing (H).
- Applying a full-fledged hash function would be overkill
- Idea: use a fast permutation π, compute
  - $F(M_1) = \mathsf{Chop}(\pi(M_1||C), 256)$
  - $H(M_1||M_2) = \mathsf{Chop}(\pi(\pi(M_1||C) \oplus (M_2||0^p)), 256)$
- This is secure under certain assumptions about π
- Speed is obiously largely determined by speed of π

The ChaCha permutation

- Consider b-bit permutation with c-bit capacity has $b - c$ bits input and $b - c$ bits output
- We need $(b - c) \geq 256$

The ChaCha permutation

- Consider b-bit permutation with c-bit capacity has $b - c$ bits input and $b - c$ bits output
- We need $(b - c) \geq 256$
- Keccak (SHA-3) permutation is extensively studied, but way too big ($b = 1600$, $c = 512$)
- Instead, use ChaCha12 permutation $b = 512, c = 256$

The ChaCha permutation

- Consider b-bit permutation with c-bit capacity has $b - c$ bits input and $b - c$ bits output
- We need $(b - c) \geq 256$
- Keccak (SHA-3) permutation is extensively studied, but way too big ($b = 1600$, $c = 512$)
- Instead, use ChaCha12 permutation $b = 512, c = 256$
- ChaCha is an improvement of Salsa, both proposed by Bernstein
- ChaCha12 uses 12 rounds to permute the 512-bit state
- Operations are on 32-bit words
- General structure is "add-rotate-xor" (ARX)

The ChaCha permutation

- Consider b-bit permutation with c-bit capacity has $b - c$ bits input and $b - c$ bits output
- We need $(b - c) \geq 256$
- Keccak (SHA-3) permutation is extensively studied, but way too big ($b = 1600$, $c = 512$)
- Instead, use ChaCha12 permutation $b = 512, c = 256$
- ChaCha is an improvement of Salsa, both proposed by Bernstein
- ChaCha12 uses 12 rounds to permute the 512-bit state
- Operations are on 32-bit words
- General structure is "add-rotate-xor" (ARX)
- The same permutation is used in Blake-512

SPHINCS-256 analysis

Overall computational cost of SPHINCS-256

- Two invocations of BLAKE-512 over the message together with short random

SPHINCS-256 analysis

Overall computational cost of SPHINCS-256

- Two invocations of BLAKE-512 over the message together with short random
- HORST signature:
  - Generation of $2\,$MB of random stream with ChaCha12 (65536 Chacha12 permutations)
  - 65536 invocations of F (65536 ChaCha12 permutations)
  - 65535 invocations of H (131070 ChaCha12 permutations)

SPHINCS-256 analysis

Overall computational cost of SPHINCS-256

- Two invocations of BLAKE-512 over the message together with short random
- HORST signature:
  - Generation of $2\,\text{MB}$ of random stream with ChaCha12 (65536 Chacha12 permutations)
  - 65536 invocations of F (65536 ChaCha12 permutations)
  - 65535 invocations of H (131070 ChaCha12 permutations)
- 12 WOTS authentication paths, each:
  - $32 \cdot 15 \cdot 67 = 32160$ invocations of F (32160 ChaCha12 perms.)
  - $32 \cdot 66 = 2112$ evaluations of H in the L-tree (4224 ChaCha12 perms.)
  - 31 evaluations of H for the binary hash tree (62 ChaCha12 perms.)

SPHINCS-256 analysis

Overall computational cost of SPHINCS-256

- Two invocations of BLAKE-512 over the message together with short random
- HORST signature:
  - Generation of $2\,\text{MB}$ of random stream with ChaCha12 (65536 Chacha12 permutations)
  - 65536 invocations of F (65536 ChaCha12 permutations)
  - 65535 invocations of H (131070 ChaCha12 permutations)
- 12 WOTS authentication paths, each:
  - $32 \cdot 15 \cdot 67 = 32160$ invocations of F (32160 ChaCha12 perms.)
  - $32 \cdot 66 = 2112$ evaluations of H in the L-tree (4224 ChaCha12 perms.)
  - 31 evaluations of H for the binary hash tree (62 ChaCha12 perms.)
- Total cost:
  $65536 + 65536 + 131070 + 12 \cdot (32160 + 4224 + 62) = 699494$ ChaCha12 permutations
- This ignores (neglible) cost for 12 WOTS signatures

Target architecture

- Intel Haswell processors featuring AVX2
- 16 vector registers of length 256 bits each
- Supports arithmetic on vector of integers
- Particularly interesting: arithmetic on 8×32-bit integers

Parallelizing ChaCha permutation

- Operations inside ChaCha permutation are 4-way parallel
- Most BLAKE implementations use this parallelism to vectorize

Parallelizing ChaCha permutation

- Operations inside ChaCha permutation are 4-way parallel
- Most BLAKE implementations use this parallelism to vectorize
- Could obviously also use this here, but:
  - We have 8-way parallel vectors in AVX2
  - Internal vectorization removes instruction-level parallelism
  - Needs frequent shuffling of vector entries

Parallelizing ChaCha permutation

- Operations inside ChaCha permutation are 4-way parallel
- Most BLAKE implementations use this parallelism to vectorize
- Could obviously also use this here, but:
  - We have 8-way parallel vectors in AVX2
  - Internal vectorization removes instruction-level parallelism
  - Needs frequent shuffling of vector entries
- Much better: vectorize 8 independent computations of F or H

Parallelizing ChaCha permutation

- Operations inside ChaCha permutation are 4-way parallel
- Most BLAKE implementations use this parallelism to vectorize
- Could obviously also use this here, but:
  - We have 8-way parallel vectors in AVX2
  - Internal vectorization removes instruction-level parallelism
  - Needs frequent shuffling of vector entries
- Much better: vectorize 8 independent computations of F or H
- This requires *interleaving* 32-bit words in memory

Parallelizing ChaCha permutation

- Operations inside ChaCha permutation are 4-way parallel
- Most BLAKE implementations use this parallelism to vectorize
- Could obviously also use this here, but:
  - We have 8-way parallel vectors in AVX2
  - Internal vectorization removes instruction-level parallelism
  - Needs frequent shuffling of vector entries
- Much better: vectorize 8 independent computations of F or H
- This requires *interleaving* 32-bit words in memory
- 8 way parallel computation of F: 420 Haswell cycles
- 8 way parallel computation of H: 836 Haswell cycles

Parallelizing WOTS

- WOTS key generation computes 67 independent hashing chains
- Could vectorize across those, but 67 is not divisible by 8

Parallelizing WOTS

- WOTS key generation computes 67 independent hashing chains
- Could vectorize across those, but 67 is not divisible by 8
- WOTS authentication-path computation computes 32 independent WOTS keys
- Efficiently vectorize those 32 independent key generations
- Again, this requires interleaving of 32-bit words

Parallelizing WOTS

- WOTS key generation computes 67 independent hashing chains
- Could vectorize across those, but 67 is not divisible by 8
- WOTS authentication-path computation computes 32 independent WOTS keys
- Efficiently vectorize those 32 independent key generations
- Again, this requires interleaving of 32-bit words
- Cost for WOTS signing is negligible; no need to vectorize

Parallelizing HORST

- Expanding the secret key: use fast vectorized ChaCha12 (by Andrew Moon)

Parallelizing HORST

- Expanding the secret key: use fast vectorized ChaCha12 (by Andrew Moon)
- Hashing from secret to HORS public key: 2^{16} parallel hashes
- Obvious how to vectorize, again, needs interleaving

Parallelizing HORST

- Expanding the secret key: use fast vectorized ChaCha12 (by Andrew Moon)
- Hashing from secret to HORS public key: 2^{16} parallel hashes
- Obvious how to vectorize, again, needs interleaving
- Consider the tree as 8 independent trees with "small tree on top"
- Vectorize across those 8 independent trees

Parallelizing HORST

- Expanding the secret key: use fast vectorized ChaCha12 (by Andrew Moon)
- Hashing from secret to HORS public key: 2^{16} parallel hashes
- Obvious how to vectorize, again, needs interleaving
- Consider the tree as 8 independent trees with "small tree on top"
- Vectorize across those 8 independent trees
- Again, this needs interleaving
- Can re-use the interleaving of the 2^{16} parallel hashes
- Could even consider the output of ChaCha12 as already interleaved (but: compatibility issues)

Parallelizing HORST

- Expanding the secret key: use fast vectorized ChaCha12 (by Andrew Moon)
- Hashing from secret to HORS public key: 2^{16} parallel hashes
- Obvious how to vectorize, again, needs interleaving
- Consider the tree as 8 independent trees with "small tree on top"
- Vectorize across those 8 independent trees
- Again, this needs interleaving
- Can re-use the interleaving of the 2^{16} parallel hashes
- Could even consider the output of ChaCha12 as already interleaved (but: compatibility issues)
- Handle the small tree on top non-vectorized (neglible)

Results

- SPHINCS-256 is slightly more complex (random bitmasks all over the place)
- Results for full SPHINCS-256 on Intel Haswell (Xeon E3-1275):
  - **Keygen:** $3\,237\,260$ cycles
  - **Signing:** $51\,636\,372$ cycles
  - **Verification:** $1\,451\,004$ cycles
- Sizes for SPHINCS-256:
  - **Public Key:** 1056 bytes
  - **Secret Key:** 1088 bytes
  - **Signature:** 41000 bytes
- For more details see http://sphincs.cr.yp.to