Implementing SPHINCS with restricted memory

Joost Rijneveld

Master Thesis in CS
Radboud University

May 2015

What?

- Quantum computers break classic public-key crypto

What?

- Quantum computers break classic public-key crypto
- SPHINCS: a post-quantum signature scheme
  - Implementation exists, uses several MBs of RAM

What?

- Quantum computers break classic public-key crypto
- SPHINCS: a post-quantum signature scheme
  - Implementation exists, uses several MBs of RAM
- ..make it use less memory!

What?

- Quantum computers break classic public-key crypto
- SPHINCS: a post-quantum signature scheme
  - Implementation exists, uses several MBs of RAM
- ..make it use less memory!

- This talk:
  - Relevant crypto context
  - SPHINCS
  - Implementation details

What?

- Quantum computers break classic public-key crypto
- SPHINCS: a post-quantum signature scheme
  - Implementation exists, uses several MBs of RAM
- ..make it use less memory!
- This talk:
  - Relevant crypto context
  - SPHINCS
  - Implementation details
- Not this talk:
  - Background on public key crypto / hashes in general
  - Other post-quantum crypto
  - Quantum computing / crypto

Cryptographic context

- SPHINCS[1]: Stateless, practical, **hash-based**, incredibly nice cryptographic signatures
- Hashes do not fall to Shor (but halved by Grover)
- Hash-based schemes: conservative choice post-quantum
  - Fundamental building block

[1] Daniel J. Bernstein, Diana Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe and Zooko Wilcox O'Hearn, 2015

Lamport signatures

- 'Classic example' of hash-based signatures

Lamport signatures

- ‘Classic example’ of hash-based signatures
- Private key: N pairs of random numbers

$$
\begin{array}{ccccccc}
\boxed{s_{0,0}} & \boxed{s_{1,0}} & \boxed{s_{2,0}} & \cdots & \boxed{s_{N-3,0}} & \boxed{s_{N-2,0}} & \boxed{s_{N-1,0}} \\
\boxed{s_{0,1}} & \boxed{s_{1,1}} & \boxed{s_{2,1}} & \cdots & \boxed{s_{N-3,1}} & \boxed{s_{N-2,1}} & \boxed{s_{N-1,1}}
\end{array}
$$

Lamport signatures

- 'Classic example' of hash-based signatures
- Private key: N pairs of random numbers

$$\begin{array}{ccccccc} s_{0,0} & s_{1,0} & s_{2,0} & \cdots & s_{N-3,0} & s_{N-2,0} & s_{N-1,0} \\ s_{0,1} & s_{1,1} & s_{2,1} & \cdots & s_{N-3,1} & s_{N-2,1} & s_{N-1,1} \end{array}$$

- Public key: hashes of these random numbers

$$\begin{array}{ccccccc} \mathrm{h}(s_{0,0}) & \mathrm{h}(s_{1,0}) & \mathrm{h}(s_{2,0}) & \cdots & \mathrm{h}(s_{N-3,0}) & \mathrm{h}(s_{N-2,0}) & \mathrm{h}(s_{N-1,0}) \\ \mathrm{h}(s_{0,1}) & \mathrm{h}(s_{1,1}) & \mathrm{h}(s_{2,1}) & \cdots & \mathrm{h}(s_{N-3,1}) & \mathrm{h}(s_{N-2,1}) & \mathrm{h}(s_{N-1,1}) \end{array}$$

Lamport signatures

- 'Classic example' of hash-based signatures
- Private key: N pairs of random numbers

$$s_{0,0} \quad s_{1,0} \quad s_{2,0} \quad \cdots \quad s_{N-3,0} \quad s_{N-2,0} \quad s_{N-1,0}$$
$$s_{0,1} \quad s_{1,1} \quad s_{2,1} \quad \cdots \quad s_{N-3,1} \quad s_{N-2,1} \quad s_{N-1,1}$$

- Public key: hashes of these random numbers

$$\mathrm{h}(s_{0,0}) \quad \mathrm{h}(s_{1,0}) \quad \mathrm{h}(s_{2,0}) \quad \cdots \quad \mathrm{h}(s_{N-3,0}) \quad \mathrm{h}(s_{N-2,0}) \quad \mathrm{h}(s_{N-1,0})$$
$$\mathrm{h}(s_{0,1}) \quad \mathrm{h}(s_{1,1}) \quad \mathrm{h}(s_{2,1}) \quad \cdots \quad \mathrm{h}(s_{N-3,1}) \quad \mathrm{h}(s_{N-2,1}) \quad \mathrm{h}(s_{N-1,1})$$

- Signature on N-bit value, e.g. $100\ldots110$

$$s_{0,1} \quad s_{1,0} \quad s_{2,0} \quad \cdots \quad s_{N-3,1} \quad s_{N-2,1} \quad s_{N-1,0}$$

- Verification: hash, compare to public key

Lamport signatures

- 'Classic example' of hash-based signatures
- Private key: N pairs of random numbers

$$\begin{array}{ccccccc} s_{0,0} & s_{1,0} & s_{2,0} & \cdots & s_{N-3,0} & s_{N-2,0} & s_{N-1,0} \\ s_{0,1} & s_{1,1} & s_{2,1} & \cdots & s_{N-3,1} & s_{N-2,1} & s_{N-1,1} \end{array}$$

- Public key: hashes of these random numbers

$$\begin{array}{ccccccc} \mathrm{h}(s_{0,0}) & \mathrm{h}(s_{1,0}) & \mathrm{h}(s_{2,0}) & \cdots & \mathrm{h}(s_{N-3,0}) & \mathrm{h}(s_{N-2,0}) & \mathrm{h}(s_{N-1,0}) \\ \mathrm{h}(s_{0,1}) & \mathrm{h}(s_{1,1}) & \mathrm{h}(s_{2,1}) & \cdots & \mathrm{h}(s_{N-3,1}) & \mathrm{h}(s_{N-2,1}) & \mathrm{h}(s_{N-1,1}) \end{array}$$

- Signature on N-bit value, e.g. $100\ldots110$

$$\begin{array}{ccccccc} s_{0,1} & s_{1,0} & s_{2,0} & \cdots & s_{N-3,1} & s_{N-2,1} & s_{N-1,0} \end{array}$$

- Verification: hash, compare to public key
- Can only do this **once**!

Merkle trees

- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys

Merkle trees

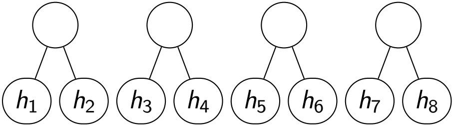
- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys
- Merkle: build ‘authentication tree’ on top

h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8

- Leaf $h_i = \mathrm{h}(\text{Public key } i)$

Merkle trees

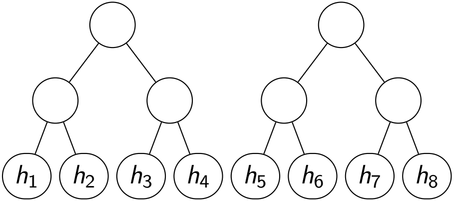
- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys
- Merkle: build 'authentication tree' on top



- Leaf h_i = h(Public key i)
- Parent = h(LeftChild $\|$ RightChild)

Merkle trees

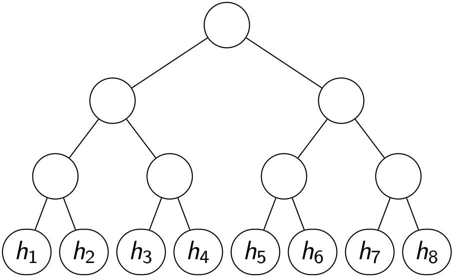
- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys
- Merkle: build ‘authentication tree’ on top



- Leaf $h_i = \mathrm{h}(\text{Public key } i)$
- Parent $= \mathrm{h}(\text{LeftChild} \parallel \text{RightChild})$

Merkle trees

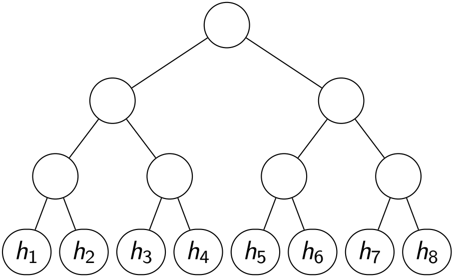
- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys
- Merkle: build 'authentication tree' on top



- Leaf h_i = h(Public key i)
- Parent = h(LeftChild $\|$ RightChild)

Merkle trees

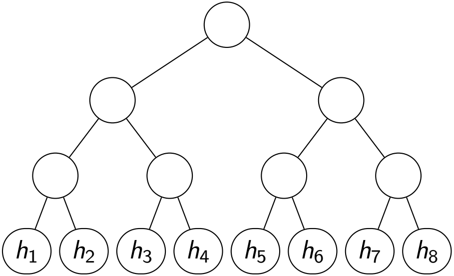
- One public key, multiple signatures?
  - OTS, so multiple signatures $\rightarrow$ multiple private keys
- Merkle: build 'authentication tree' on top



- Leaf h_i = h(Public key i)
- Parent = h(LeftChild $\|$ RightChild)
- New public key: root node

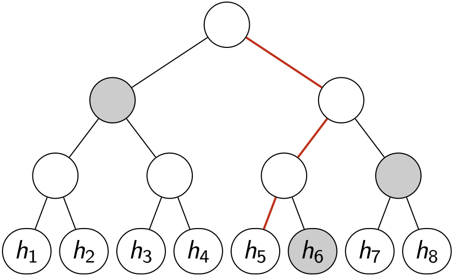
Merkle trees

- Signature must now include:
  - Lamport signature σ
  - Public key α
  - Position in the Merkle tree, e.g. 5
  - Nodes along the *authentication path*



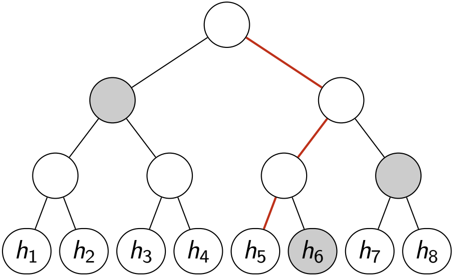
Merkle trees

- Signature must now include:
  - Lamport signature σ
  - Public key α
  - Position in the Merkle tree, e.g. 5
  - Nodes along the *authentication path*



Merkle trees

- Signature must now include:
  - Lamport signature σ
  - Public key α
  - Position in the Merkle tree, e.g. 5
  - Nodes along the *authentication path*



- Verification: reconstruct root node

Why not?

- Hashes (256-bit) survive post-quantum
- Signing is fast
- Keys are small
  - Private key generated from small seed
- Signatures are somewhat large..

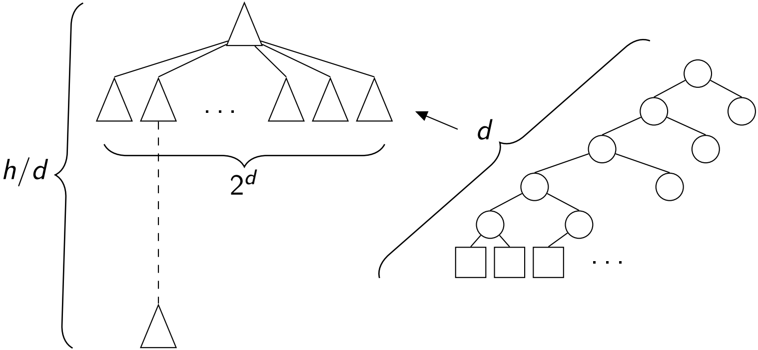
Why not?

- Hashes (256-bit) survive post-quantum
- Signing is fast
- Keys are small
  - Private key generated from small seed
- Signatures are somewhat large..

- Need to **remember** the last used index!
  - Terribly inconvenient

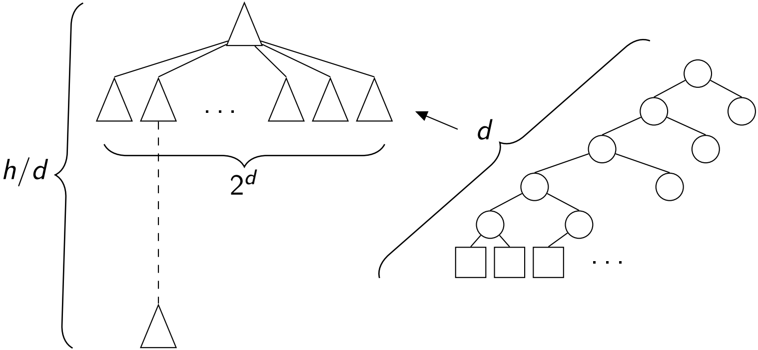
SPHINCS

- Large Merkle tree, height h
- Every d-th layer signs child node using an OTS
  - Effectively a hypertree of h/d Merkle trees
- Sign messages using leaf nodes



SPHINCS

- Large Merkle tree, height h
- Every d-th layer signs child node using an OTS
  - Effectively a hypertree of h/d Merkle trees
- Sign messages using leaf nodes
- No need to remember index: **stateless**



Stateless?

- *Large* tree $\rightarrow$ many leaf nodes $\rightarrow$ small chance of duplicates

Stateless?

- *Large* tree $\rightarrow$ many leaf nodes $\rightarrow$ small chance of duplicates
- Layers of OTS: no need to compute entire tree
- Layers of hashing: acceptable signature size

Stateless?

- *Large* tree $\rightarrow$ many leaf nodes $\rightarrow$ small chance of duplicates
- Layers of OTS: no need to compute entire tree
- Layers of hashing: acceptable signature size
- 'Few time signature scheme' (FTS) for leaf nodes
- Chance of a break becomes negligible

Key generation

- Generate random values SK_1 and SK_2
- Use SK_1: generate OTS keys of top sub-tree
- Compute root node (recall: the sub-tree is a Merkle tree)
  - PK: root node

Key generation

- Generate random values SK_1 and SK_2
- Use SK_1: generate OTS keys of top sub-tree
- Compute root node (recall: the sub-tree is a Merkle tree)
  - PK: root node

- In general: SK_1 generates OTS and FTS keys *deterministically*

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}
- Sign FTS key using OTS, produce σ_{OTS_1}

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}
- Sign FTS key using OTS, produce σ_{OTS_1}
- Compute authentication path through Merkle tree

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}
- Sign FTS key using OTS, produce σ_{OTS_1}
- Compute authentication path through Merkle tree
- Sign root node of subtree using next OTS, produce σ_{OTS_2}

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}
- Sign FTS key using OTS, produce σ_{OTS_1}
- Compute authentication path through Merkle tree
- Sign root node of subtree using next OTS, produce σ_{OTS_2}
- Repeat..

Signing

- Pick an FTS leaf node
  - But not randomly! $R = f(SK_2, M)$
  - Deterministic signatures
- Sign digest of M, produce σ_{FTS}
- Sign FTS key using OTS, produce σ_{OTS_1}
- Compute authentication path through Merkle tree
- Sign root node of subtree using next OTS, produce σ_{OTS_2}
- Repeat.. until root node
- Signature: $\Sigma = (R, \sigma_{FTS}, (\sigma_{OTS_1}, Auth_1), (\sigma_{OTS_2}, Auth_2), \ldots, (\sigma_{OTS_{h/d}}, Auth_{h/d}))$

SPHINCS-256

- 41KB signatures, 1KB keys
- 256-bit hash functions
  - 128-bit post-quantum security
- $h = 60, d = 5$: 12 layers of sub-trees
- 2^{60} leaf nodes

Building blocks

- OTS
- Hash functions
- Key expansion function
- FTS

Building blocks

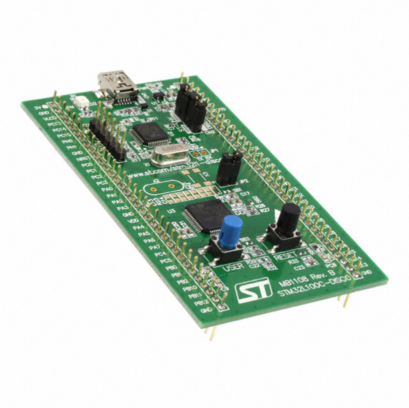
- OTS: *Winternitz OTS variant (WOTS+)*
- Hash functions: *BLAKE*, π_{ChaCha}
- Key expansion function: $ChaCha_{12}$
- FTS: *HORST*

Building blocks

- OTS: *Winternitz OTS variant (WOTS+)*
- Hash functions: *BLAKE*, π_{ChaCha}
- Key expansion function: $ChaCha_{12}$
- FTS: *HORST*
  - Contains 16-layer Merkle tree (so 2^{16} leafs)
  - Goal: 32 authentication paths, root node
  - Complete tree takes approx. 2MB RAM..

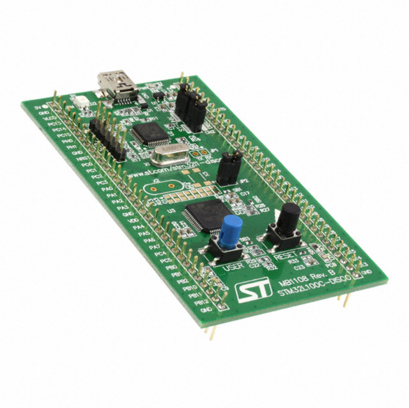
Platform and implementation

- STM32L100C board with Cortex M3
  - libopencm3 firmware
  - 32MHz, 32-bit architecture
  - **16KB RAM**



Platform and implementation

- STM32L100C board with Cortex M3
  - libopencm3 firmware
  - 32MHz, 32-bit architecture
  - **16KB RAM**
- Based on SPHINCS-256 for Haswell
  - Replaced `asm` with other implementations

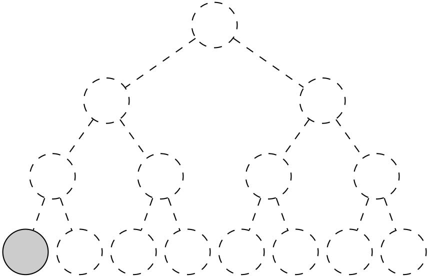


Treehash

- HORST tree is too large: 2MB!

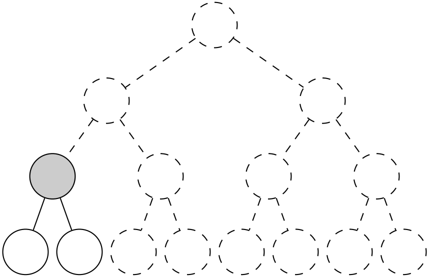
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



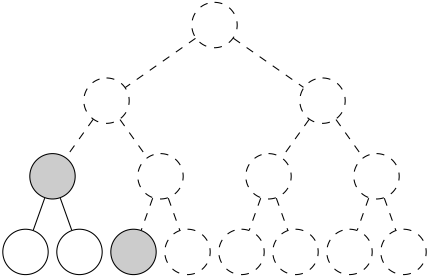
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



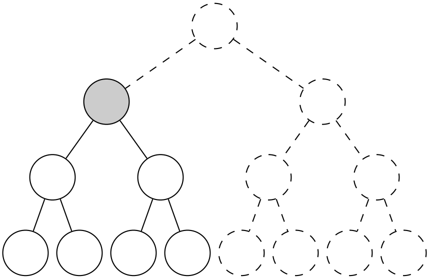
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



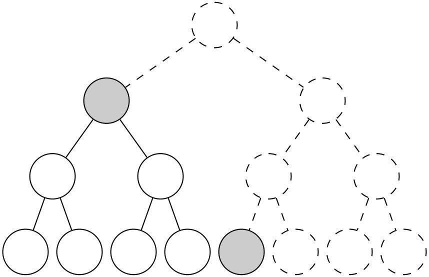
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



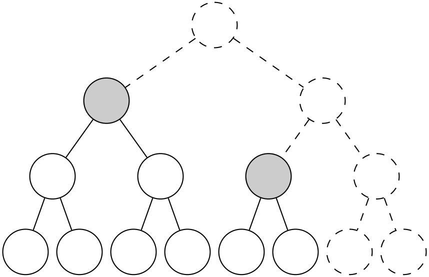
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



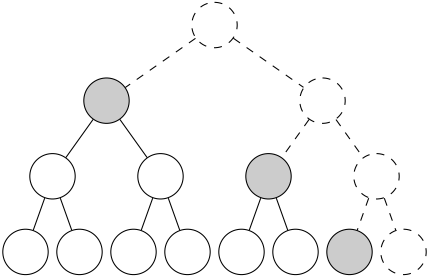
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



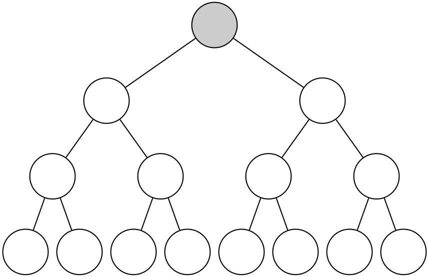
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



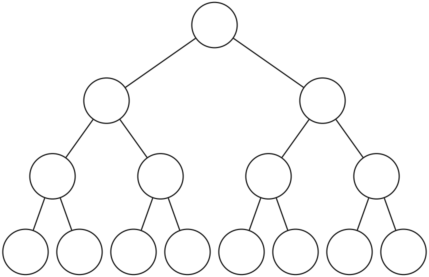
Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



Treehash

- HORST tree is too large: 2MB!
- Treehash: only remember relevant nodes
  - Maintain a stack: max. 16 nodes



Treehash considerations

- Identify relevant nodes

Treehash considerations

- Identify relevant nodes
- Identify relevant rounds

Treehash considerations

- Identify relevant nodes
- Identify relevant rounds
- Identify relevant nodes in rounds (bitmasks)

Treehash considerations

- Identify relevant nodes
- Identify relevant rounds
- Identify relevant nodes in rounds (bitmasks)
- Key observation: sort masks by round index
  - Simply maintain a pointer

Treehash considerations

- Identify relevant nodes
- Identify relevant rounds
- Identify relevant nodes in rounds (bitmasks)
- Key observation: sort masks by round index
  - Simply maintain a pointer
- Output in the appropriate order..

Streaming

- Cannot store signature $\rightarrow$ stream out immediately
- HORST not ordered properly!

Streaming

- Cannot store signature $\rightarrow$ stream out immediately
- HORST not ordered properly!
  - Tags (max. 832 bytes)
  - Re-arrange on the host

Streaming

- Cannot store signature $\rightarrow$ stream out immediately
- HORST not ordered properly!
  - Tags (max. 832 bytes)
  - Re-arrange on the host

- Cannot store expanded key material
- Interleave ChaCha12 and Treehash

Performance

- Works on 16KB RAM

Performance

- Works on 16KB RAM
- Signing: 1 681 333 801 cycles
- Key generation: 73 986 826 cycles
- Recall 32MHz, so roughly 52 seconds
- On 4-core Haswell:

  "[..] signs hundreds of messages per second."

Performance

- Works on 16KB RAM
- Signing: 1 681 333 801 cycles
- Key generation: 73 986 826 cycles
- Recall 32MHz, so roughly 52 seconds
- On 4-core Haswell:

  "[..] signs hundreds of messages per second."
- Hash-based? ChaCha cycles account for nearly 70%!

TODO

- Implement verification
- Implement ChaCha in ARMv7-M `asm`
- Operate on messages of arbitrary size
- Cache (partial) authentication paths

Conclusions

- SPHINCS could replace RSA / ECC / ... for signing
  - Stateless $\rightarrow$ drop-in replacement
  - Conservative security choice
- Feasible on limited platforms
  - Hard memory limit: ✓
  - Time efficiency: gradual optimisation

Lamport signatures

- 'Classic example' of hash-based signatures
- Private key: N pairs of random numbers

$$\begin{array}{ccccccc} s_{0,0} & s_{1,0} & s_{2,0} & \cdots & s_{N-3,0} & s_{N-2,0} & s_{N-1,0} \\ s_{0,1} & s_{1,1} & s_{2,1} & \cdots & s_{N-3,1} & s_{N-2,1} & s_{N-1,1} \end{array}$$

- Public key: hashes of these random numbers

$$\begin{array}{ccccccc} \mathrm{h}(s_{0,0}) & \mathrm{h}(s_{1,0}) & \mathrm{h}(s_{2,0}) & \cdots & \mathrm{h}(s_{N-3,0}) & \mathrm{h}(s_{N-2,0}) & \mathrm{h}(s_{N-1,0}) \\ \mathrm{h}(s_{0,1}) & \mathrm{h}(s_{1,1}) & \mathrm{h}(s_{2,1}) & \cdots & \mathrm{h}(s_{N-3,1}) & \mathrm{h}(s_{N-2,1}) & \mathrm{h}(s_{N-1,1}) \end{array}$$

- Signature on N-bit value, e.g. $100\ldots110$

$$\begin{array}{ccccccc} s_{0,1} & s_{1,0} & s_{2,0} & \cdots & s_{N-3,1} & s_{N-2,1} & s_{N-1,0} \end{array}$$

- Verification: hash, compare to public key
- Can only do this **once**!

The Winternitz improvement

- Trade time for signature and public key size
- Idea: sign groups of m bits, let $w = 2^m$

The Winternitz improvement

- Trade time for signature and public key size
- Idea: sign groups of m bits, let $w = 2^m$
- Private key: N/m random numbers

s_0 s_1 s_2 $\cdots$ $s_{N/m-3}$ $s_{N/m-2}$ $s_{N/m-1}$

The Winternitz improvement

- Trade time for signature and public key size
- Idea: sign groups of m bits, let $w = 2^m$
- Private key: N/m random numbers

$s_0 \quad s_1 \quad s_2 \quad \cdots \quad s_{N/m-3} \quad s_{N/m-2} \quad s_{N/m-1}$

- Public key: hash w times

$\mathrm{h}^w(s_0) \; \mathrm{h}^w(s_1) \; \mathrm{h}^w(s_2) \quad \cdots \quad \mathrm{h}^w(s_{N/m-3}) \; \mathrm{h}^w(s_{N/m-2}) \; \mathrm{h}^w(s_{N/m-1})$

The Winternitz improvement

- Trade time for signature and public key size
- Idea: sign groups of m bits, let $w = 2^m$
- Private key: N/m random numbers

$s_0 \quad s_1 \quad s_2 \quad \cdots \quad s_{N/m-3} \quad s_{N/m-2} \quad s_{N/m-1}$

- Public key: hash w times

$h^w(s_0) \; h^w(s_1) \; h^w(s_2) \quad \cdots \quad h^w(s_{N/m-3}) \; h^w(s_{N/m-2}) \; h^w(s_{N/m-1})$

- Signature on N-bit value, e.g. 1010 0110 0101 1100
  - For this example, assume $m = 4$, so $w = 16$

$h^{10}(s_0) \;\; h^6(s_1) \;\; h^5(s_2) \; h^{12}(s_3)$

The Winternitz improvement

- Trade time for signature and public key size
- Idea: sign groups of m bits, let $w = 2^m$
- Private key: N/m random numbers

s_0 s_1 s_2 $\cdots$ $s_{N/m-3}$ $s_{N/m-2}$ $s_{N/m-1}$

- Public key: hash w times

$h^w(s_0)$ $h^w(s_1)$ $h^w(s_2)$ $\cdots$ $h^w(s_{N/m-3})$ $h^w(s_{N/m-2})$ $h^w(s_{N/m-1})$

- Signature on N-bit value, e.g. 1010 0110 0101 1100
  - For this example, assume $m = 4$, so $w = 16$

$h^{10}(s_0)$ $h^6(s_1)$ $h^5(s_2)$ $h^{12}(s_3)$

- Verification: complete hashes to w, check with public key

HORST

- Few-time signature scheme, two parameters k, t, (e.g. $k = 32$, $t = 2^{16}$)
- Private key: t random numbers $s_0, s_1, \ldots, s_{t-1}$
- Public key: $h(s_0), h(s_1), \ldots, h(s_{t-1})$

HORST

- Few-time signature scheme, two parameters k, t, (e.g. $k = 32$, $t = 2^{16}$)
- Private key: t random numbers $s_0, s_1, \ldots, s_{t-1}$
- Public key: $h(s_0), h(s_1), \ldots, h(s_{t-1})$
  - Build a Merkle tree on top

HORST

- Few-time signature scheme, two parameters k, t, (e.g. $k = 32$, $t = 2^{16}$)
- Private key: t random numbers $s_0, s_1, \ldots, s_{t-1}$
- Public key: $h(s_0), h(s_1), \ldots, h(s_{t-1})$
  - Build a Merkle tree on top
- Signature on N-bit value (e.g. $N = 512$)
  - Split message (digest!) into k parts
  - Interpret message parts as integers $m_0, m_1, \ldots, m_{k-1}$
  - Reveal $s_{m_0}, s_{m_1}, \ldots, s_{m_{k-1}}$
  - Include authentication paths

HORST

- Few-time signature scheme, two parameters k, t, (e.g. $k = 32,\ t = 2^{16}$)
- Private key: t random numbers $s_0, s_1, \dots, s_{t-1}$
- Public key: $h(s_0), h(s_1), \dots, h(s_{t-1})$
  - Build a Merkle tree on top
- Signature on N-bit value (e.g. $N = 512$)
  - Split message (digest!) into k parts
  - Interpret message parts as integers $m_0, m_1, \dots, m_{k-1}$
  - Reveal $s_{m_0}, s_{m_1}, \dots, s_{m_{k-1}}$
  - Include authentication paths
- Very small chance of re-use