Improving Stateless Hash-Based Signatures

CT-RSA 2018

Jean-Philippe Aumasson\textsuperscript{1}, Guillaume Endignoux\textsuperscript{2}

Wednesday 18\textsuperscript{th} April, 2018

\textsuperscript{1}Kudelski Security

\textsuperscript{2}Work done while at Kudelski Security and EPFL
What are hash-based signatures?

- Good hash functions are hard to invert = preimage-resistance.
- We can use this property to create signature schemes\(^1\).

\(^1\)Whitfield Diffie and Martin E. Hellman. *New directions in cryptography.* 1976
Hash-based signatures

What are hash-based signatures?

- Good hash functions are hard to invert = preimage-resistance.
- We can use this property to create signature schemes\(^1\).

First step: scheme to sign 1-bit message.

- Key generation: commit to 2 secrets with \(H\)
- Sign bit \(b\): reveal \(\sigma = S_b\)
- Verify signature \(\sigma\): compare \(H(\sigma)\) with \(P_b\)

---

\(^1\)Whitfield Diffie and Martin E. Hellman. *New directions in cryptography*. 1976
Hash-based signatures

Second step: sign $n$-bit message $\Rightarrow n$ copies of the previous scheme.

Figure 1: Lamport signatures.
Hash-based signatures

**Second step:** sign $n$-bit message $\Rightarrow$ $n$ copies of the previous scheme.

$$
\begin{align*}
S_0,0 & \quad H & \quad \cdots & \quad H & \quad H & \quad S_1,0 \\
S_0,1 & \quad H & \quad \cdots & \quad H & \quad S_1,1 \\
S_1,0 & \quad H & \quad \cdots & \quad H & \quad S_1,1 \\
S_1,1 & \quad H & \quad \cdots & \quad H & \quad S_1,1 \\
& \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots \\
S_n,0 & \quad H & \quad \cdots & \quad H & \quad S_n,1 \\
S_n,1 & \quad H & \quad \cdots & \quad H & \quad S_n,1
\end{align*}
$$

*Figure 1:* Lamport signatures.

However, this is a **one-time** signature scheme.
Hash-based signatures

More constructions:

- **WOTS** (Winternitz one-time signatures) = compact version of the $n$-bit message scheme.
- **Merkle trees** = *stateful* multiple-time signatures.
- **HORS** = *stateless* few-time signatures.
- **HORST** = HORS with Merkle tree.
Hash-based signatures

SPHINCS = stateless many-time signatures (up to $2^{50}$ messages).

- Hyper-tree of WOTS signatures $\approx$ certificate chain
- Hyper-tree of height $H = 60$, divided in 12 layers of \{Merkle tree + WOTS\}

Sign message $M$:

- Select index $0 \leq i < 2^{60}$
- Sign $M$ with $i$-th HORST instance
- Chain of WOTS signatures.

Figure 2: SPHINCS.
Hash-based signatures in a nutshell:

- Post-quantum security well understood ⇒ Grover’s algorithm: preimage-search in $O(2^{n/2})$ instead of $O(2^n)$ for $n$-bit hash function.
- Signature size is quite large: 41 KB for SPHINCS (stateless), 8 KB for XMSS (stateful).
Contributions

We propose improvements to reduce signature size of SPHINCS:

- PRNG to obtain a random subset (PORS)
- Octopus: optimized multi-authentication in Merkle trees
- Secret key caching
- Non-masked hashing
PRNG to obtain a random subset
From HORS to PORS

Sign a message $M$ with HORS:

- Hash the message $H(M) = 28c5c...$
- Split the hash to obtain indices $\{2, 8, c, 5, c, \ldots\}$ and reveal values $S_2, S_8, \ldots$

$$i \rightarrow \text{SPHINCS leaf}$$

![Diagram showing SPHINCS tree with hash values and indices]

<table>
<thead>
<tr>
<th>Public key</th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
<th>$P_7$</th>
<th>$P_8$</th>
<th>$P_9$</th>
<th>$P_{10}$</th>
<th>$P_{11}$</th>
<th>$P_{12}$</th>
<th>$P_{13}$</th>
<th>$P_{14}$</th>
<th>$P_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
<td>$H$</td>
</tr>
<tr>
<td>Secret key</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_2$</td>
<td>$S_3$</td>
<td>$S_4$</td>
<td>$S_5$</td>
<td>$S_6$</td>
<td>$S_7$</td>
<td>$S_8$</td>
<td>$S_9$</td>
<td>$S_{10}$</td>
<td>$S_{11}$</td>
<td>$S_{12}$</td>
<td>$S_{13}$</td>
<td>$S_{14}$</td>
<td>$S_{15}$</td>
</tr>
</tbody>
</table>
Sign a message $M$ with HORS:

- Hash the message $H(M) = 28c5c...$
- Split the hash to obtain indices $\{2, 8, c, 5, c, \ldots\}$ and reveal values $S_2, S_8, \ldots$

Problems:

- Some indices may be the same $\Rightarrow$ fewer values revealed $\Rightarrow$ lower security...
- Attacker is free to choose the hyper-tree index $i$ $\Rightarrow$ larger attack surface.
From HORS to PORS

PORS = PRNG to obtain a random subset.

- Seed a PRNG from the message.
- Generate the hyper-tree index.
- Ignore duplicated indices.

Significant security improvement for the same parameters!
Advantages of PORS:

- Significant security improvement for the same parameters!
- Smaller hyper-tree than SPHINCS for same security level ⇒ Signatures are 4616 bytes smaller.
- Performance impact of PRNG vs. hash function is negligible ⇒ For SPHINCS, generate only 32 distinct values.
Octopus: multi-authentication in Merkle trees
Merkle tree of height $h = \text{compact way to authenticate any of } 2^h \text{ values.}$

- Small public value = root
- Small proofs of membership = $h$ authentication nodes
How to authenticate $k$ values?

- Use $k$ independent proofs $= kh$ nodes.
- This is suboptimal! Many redundant values...
How to authenticate $k$ values?

- Optimal solution: compute smallest set of authentication nodes.
How many bytes does it save?

- It depends on the shape of the “octopus”!
- Examples for $h = 4$ and $k = 4$: between 2 and 8 authentication nodes.
**Theorem**

Given a Merkle tree of height $h$ and $k$ leaves to authenticate, the minimal number of authentication nodes $n$ verifies:

$$h - \lceil \log_2 k \rceil \leq n \leq k(h - \lfloor \log_2 k \rfloor)$$

$\Rightarrow$ For $k > 1$, this is always better than the $kh$ nodes for $k$ independent proofs!
In the case of SPHINCS, \( k = 32 \) uniformly distributed leaves, tree of height \( h = 16 \).

In our paper, recurrence relation to compute average number of authentication nodes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of auth. nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent proofs</td>
<td>512</td>
</tr>
<tr>
<td>SPHINCS(^2)</td>
<td>384</td>
</tr>
<tr>
<td>Octopus (worst case)</td>
<td>352</td>
</tr>
<tr>
<td>Octopus (average)</td>
<td>324</td>
</tr>
</tbody>
</table>

\( \Rightarrow \) Octopus authentication saves **1909 bytes** for SPHINCS signatures on average.

\(^2\)SPHINCS has a basic optimization to avoid redundant nodes close to the root.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Octopus algorithm

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the paper, let’s see an example.
Conclusion
Take-aways

- Octopus + PORS = great improvement over HORST.
- These modifications are simple to understand \(\Rightarrow\) low risk of implementation bugs.
- More improvements in the paper.
Two open-source implementations:

- Reference C implementation, proposed for NIST pqcrypto standardization
  https://github.com/gravity-postquantum/gravity-sphincs
- Rust implementation with focus on clarity and testing
  https://github.com/gendx/gravity-rs
Conclusion

Thank you for your attention!
Secret key caching

WOTS signatures to “connect” Merkle trees are large ($\approx 2144$ bytes per WOTS).

Figure 3: SPHINCS.
⇒ We use a larger root Merkle tree, and cache more values in private key.

Figure 4: Secret key caching.
Non-masked hashing

- In SPHINCS, Merkle trees have a **XOR-and-hash** construction, to use a 2nd-preimage-resistant hash function $H$.
- Various masks, depending on location in hyper-tree; all stored in the public key.
- Post-quantum preimage search is faster with Grover’s algorithm $\Rightarrow$ We remove the masks and rely on collision-resistant $H$.

(a) Masked hashing in SPHINCS.

(b) Mask off.