Gravity-SPHINCS

First PQC Standardization Conference

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

Wednesday 11\textsuperscript{th} April, 2018

\textsuperscript{1}Kudelski Security

\textsuperscript{2}Work done while at Kudelski Security and EPFL
Introduction: SPHINCS
**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.
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).
Gravity-SPHINCS
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
Open-source implementations:

- **Reference C** implementation in the submission
- Optimized implementation for Intel (**AES-NI + SSE/AVX**)
  https://github.com/gravity-postquantum/gravity-sphincs
- **Rust** implementation with focus on clarity and testing
  https://github.com/gendx/gravity-rs
Some benchmarks on our optimized implementation

<table>
<thead>
<tr>
<th>Instance</th>
<th>S</th>
<th>M</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key generation</td>
<td>0.4 s</td>
<td>12 s</td>
<td>6 s</td>
</tr>
<tr>
<td>Sign</td>
<td>5 ms</td>
<td>7 ms</td>
<td>8 ms</td>
</tr>
<tr>
<td>Verify</td>
<td>0.04 ms</td>
<td>0.12 ms</td>
<td>0.16 ms</td>
</tr>
<tr>
<td>Signature size</td>
<td>≤ 12640</td>
<td>≤ 28929</td>
<td>≤ 35168</td>
</tr>
</tbody>
</table>

| Capacity       | 2^{10} | 2^{50} | 2^{64} |

---

1 Intel Core i5-6360U CPU @ 2.00 GHz
2 Size varies depending on the message and key
PRNG to obtain a random subset
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 $S_2, S_8, \ldots$
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 $S_2, S_8, \ldots$

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 $= k h$ nodes.
- This is suboptimal! Many redundant values...
How to authenticate $k$ values?

- Optimal solution: compute smallest set of authentication nodes.
Octopus

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\(^3\), 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(^4)</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>

⇒ Octopus authentication saves **1909 bytes** for SPHINCS signatures on average.

\(^3\)https://eprint.iacr.org/2017/933, to appear at CT-RSA

\(^4\)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 submission, let’s see an example.
Octopus algorithm

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

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

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

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

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

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

- Bottom-up algorithm to compute the optimal authentication nodes.
- Formal specification in the submission, let’s see an example.
Other optimizations
Secret key caching

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

Figure 2: SPHINCS.
Secret key caching

- We use a larger root Merkle tree, and cache more values in private key.
- Removing 3 levels = 6432 bytes saved!
- This cache can be regenerated from a small private seed (32 bytes).

![Figure 3: Secret key caching.](image)
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** $H$.

\[ m_i \]

(a) Masked hashing in SPHINCS.

(b) Mask off.
Conclusion
Take-aways

Hash-based signatures:

- well-understood security,
- fast signing, very fast verification.

What’s new in Gravity-SPHINCS?

- octopus + PORS = great improvement over HORST,
- secret-key caching = trade-off key generation time / signature size for a “powerful” signer,
- mask-less hashing = simpler scheme.
Thank you for your attention!