Hash-based signatures

Joost Rijneveld
joost@joostrijneveld.nl
Radboud University, Nijmegen, The Netherlands

Post-quantum cryptography

As it becomes more and more likely that practical, large-scale quantum computers will be built within the next several years or decades, cryptographers all over the world are trying to push for a transition to a new class of schemes and protocols: post-quantum cryptography.

Why?
Currently (or, rather, 'classically'), nearly all deployed asymmetric cryptography depends on the hardness of two mathematical problems: **integer factorization** and the **discrete logarithm problem**. This is what underlies public-key encryption, key exchange protocols and digital signatures that use RSA or elliptic curves. In 1994, **Shor's algorithm** was published. This algorithm, designed to be executed on a quantum computer, solves the aforementioned problems much more efficiently than is possible on a traditional computer, leading to secret key recovery.

Now what?
To counter this, cryptographic primitives are being designed that rely on different hard problems. Research focuses on several areas: lattices, error-correcting codes, multivariate quadratics, **hash functions** and super-singular isogenies — each with their own strengths and weaknesses.

This poster is aimed to provide an introduction to digital signatures based solely on the existence of a secure cryptographic hash function.

Singing individual bits – but only once

In 1979, Lamport described what is now known as 'Lamport one-time signatures' (**OTS**). Each key pair of $N \cdot k$ bits can be used to sign an N-bit message at a k-bit security level.

Private key: N pairs of random numbers;

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

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

Public key: hashes of these random numbers;

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

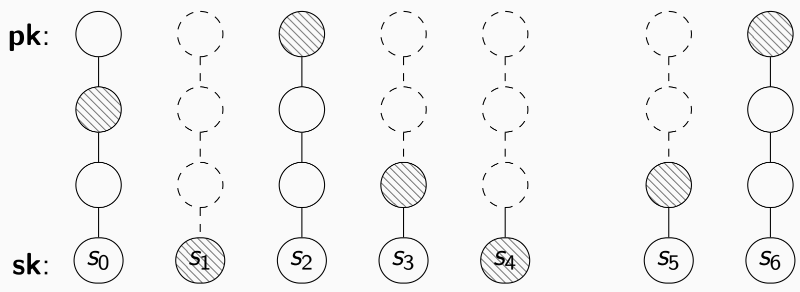
$h(s_{0,1})$ $h(s_{1,1})$ $h(s_{2,1})$ $\cdots$ $h(s_{N-3,1})$ $h(s_{N-2,1})$ $h(s_{N-1,1})$

Signature on N-bit value, e.g. 100...110:

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

Singing groups of bits

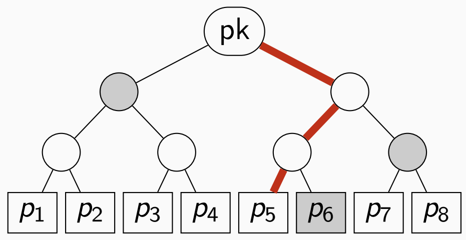
Also in 1979, Merkle described an improvement (**WOTS**, attributed to Winternitz) to sign groups of bits using hash chains, introducing a time/size trade-off. For example, let's sign 10 00 11 01 00 with trade-off $w = 4$. A checksum is needed to prevent forgeries: $\Sigma_{i=1}^{\ell_1}(w-1-m_i) = 7 = 01\ 11$.



The most common trade-off parameter $w = 16$ results in signatures of 2 KiB when signing a 256-bit value.

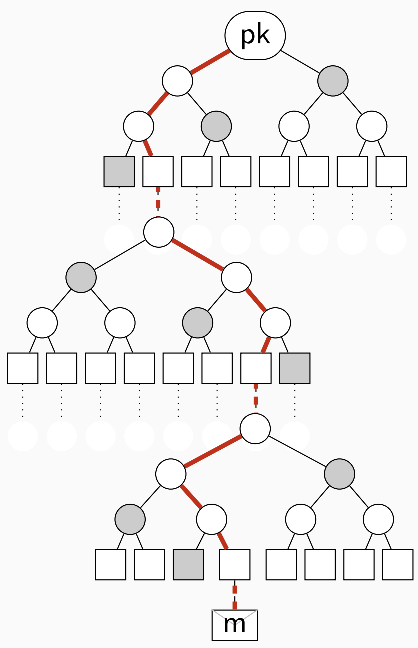
Merkle trees & XMSS

To be able to sign multiple messages with a single public key, many OTS key pairs can be authenticated together by placing them on the leaf nodes of a binary hash tree. A signature must now also include the **authentication path**, so that the verifier can reconstruct and compare the root node. Note that this makes the scheme **stateful**: the signer must remember never to re-use an OTS key pair attached to a leaf node to sign more than one message. Remembering more state can be used to speed up signature generation, using tree traversal algorithms. A concrete signature scheme using this Merkle tree construction is **XMSS**, recently described in RFC 8391.



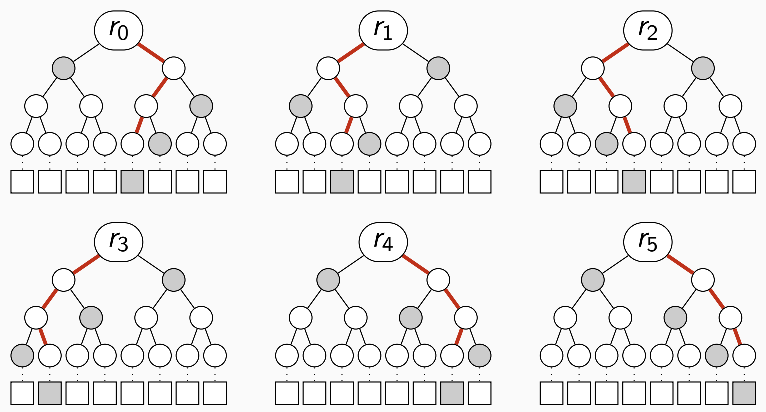
Building a hypertree

The number of messages that can be signed using a single key pair is directly related to the height of the tree. Using a large tree (e.g. 2^{60} leaf nodes) results in prohibitively slow key and signature generation. This is remedied by creating **certification trees**, signing the root node of one tree using one of the OTS key pair attached to a leaf node of a tree on the layer above it. This ensures one only has to generate a single tree per layer at a time, at the cost of a larger combined signature. This **hypertree** construction underlies the **XMSS**^{MT} scheme.



Eliminate the state & few-time signatures

Having to maintain a state is a major downside of the above construction. This is fundamentally incompatible with common signature APIs and makes practicalities such as key backups and signing across multiple machines much more involved. **SPHINCS** solves this by using a sufficiently large hypertree, such that one can safely pick a random leaf node instead. To reduce the required height, the SPHINCS framework uses a few-time signature scheme (e.g. **HORST** or **FORS**) that only degrades after signing several messages.



Forest of Random Subsets (FORS) splits the message into chunks, and reveals and authenticates secrets accordingly. In the above example, $m = 100\ 010\ 011\ 001\ 110\ 111$. The public key is $h(r_0, r_1, \ldots, r_5)$.