Intuitive Understanding of Quantum Computation and Post-Quantum Cryptography (Chapter 4)

Nguyen Thoi Minh Quan [*]

4 Hash-based signatures

I won't pretend that I can write better than Matthew Green's [1], Adam Langley's [2] excellent blog posts about hash-based signatures, so go there and read them :) Anyway, I'll briefly describe the hash-based signatures schemes based on Dan Boneh and Victor Shoup's book [3] to make this article self-contained.

Hash-based signature scheme is the safest signature scheme against quantum computers. All signature schemes use hash functions and hence they must rely on hash functions' security. All signature schemes, except hash-based signature schemes, must rely on additional security assumptions of other computational hard problems. Hash-based signature scheme, on the other hand, only depends on the security of hash functions. Furthermore, cryptographic hash functions are assumed to act like random oracles and as far as I know, quantum computers have limited success on breaking unstructured functions like random oracles and the best known quantum attack only has quadratic speedup compared to classical attacks. In terms of applied cryptography, I recommend you never deploy stateful hash-based signatures, instead use stateless hash-based signatures although the latter have lower performance. The chance that you screw up security of stateful hash-based signatures deployment is far higher than the chance of real general purpose quantum computers breaking your ECDSA signatures.

If you're familiar with how RSA or ECDSA signatures work and you look at hash-based signatures, you'll find the ideas behind hash-based signatures schemes pretty strange. They're elegant ideas, but they have nothing in common with our familiar RSA and ECDSA signatures. Let's review a few cryptographic concepts that are used as building blocks in hash-based signatures.

We call a function f a one-way function if given $y = f(x)$, it's hard to find x. In practice, standard cryptographic hash functions such as SHA256, SHA-3 are considered one-way functions.

A secure pseudo-random generator (PRG) is a deterministic algorithm G that given a short seed s, computes a long output r that is indistinguishable from random sequence. In practice, whenever we call $/dev/urandom$, we actually call a secure PRG function implemented in the operating system. When the operating system boots up, it only collects a small amount of random entropy (e.g. 256 bits) and uses it as seed to compute an arbitrary long stream of random sequence for us. If it's the 1st time you deal with PRG, you would feel that the deterministic property of G somewhat contradicts with the output's randomness property. We can intuitively explain this seemingly paradox as follows. The short seed is random and to the observer (attacker) of the output sequence, the seed is secret. Therefore, while the algorithm G is a deterministic process, as the starting point (seed) is a random secret that no one knows, the whole process to compute output looks like a random process.

A pseudo-random function (PRF) is a deterministic algorithm that takes 2 inputs: a secret key k and data x. The security property of PRF is that if the secret key k is chosen at random then the function $f(k, \cdot)$ acts like a random function. In practice, our familiar HMAC(k, x) is a PRF.

[*] https://www.linkedin.com/in/quan-nguyen-a3209817, https://scholar.google.com/citations?user=9uUqJ9IAAAAJ, https://github.com/cryptosubtlety, msuntmquan@gmail.com

We're ready to design hash-based signature schemes. The plan is to construct a one-time signature scheme that is safe to sign a single message. We then extend it to create a q-indexed signature scheme that can securely sign q messages. Finally, we'll build a stateless hash-based signature scheme that can sign practically arbitrary amounts of messages.

4.1 Lamport's one-time signature

To sign a 1-bit message $m \in \{0, 1\}$ using one-way function f such as SHA256, Lamport [4] invented the following algorithm. The private key sk is 2 large random numbers (e.g. 256-bit) (x_0, x_1). The public key pk is $(y_0, y_1) = (f(x_0), f(x_1))$. The signature σ of message m is $\sigma = S(sk, m) = x_m$. To verify the signature of m, the verifier checks whether $f(\sigma) \overset{?}{=} y_m$.

As f is a one-way function, given the public key $pk = (f(x_0), f(x_1))$, it's hard to compute the the private key $sk = (x_0, x_1)$, so why (x_0, x_1) must be both large and random? If $sk = (x_0, x_1)$ is small or predictable then the attacker can brute-force x, compute $f(x)$ and stop until it matches the public key $pk = (f(x_0), f(x_1))$.

It's simple to extend the algorithm to sign a v-bit message $m = m_1, \cdots, m_v$. The private key is $(x_{i,0}, x_{i,1}), i = \overline{1, v}$. The public key is $(y_{i,0}, y_{i,1}) = (f(x_{i,0}), f(x_{i,1})), i = \overline{1, v}$. The signature σ of message m is $(x_{1,m_1}, \cdots, x_{v,m_v})$.

Why is this algorithm called one-time signature? Let's take a look at an example. Let's say $v = 2$, the private key is $sk = (x_{1,0}, x_{2,0}, x_{1,1}, x_{2,1})$, the public key is $pk = (f(x_{1,0}), f(x_{2,0}), f(x_{1,1}), f(x_{2,1}))$. The signature of message $m = 00$ is $(x_{1,\mathbf{0}}, x_{2,\mathbf{0}})$. If we sign the 2nd message $m = 11$ with signature $(x_{1,\mathbf{1}}, x_{2,\mathbf{1}})$ then from the above two signatures, the attacker knows all 4 secret keys $(x_{1,0}, x_{2,0}, x_{1,1}, x_{2,1})$, i.e., the attacker can itself compute valid signatures of $m = 01$ and $m = 10$. Therefore, the above algorithm can only safely sign a single message.

4.2 q-indexed signature

Assuming that we have a one-time signature scheme with key pair (pk, sk) that can sign a single message $m \in M$. If we have q key pairs $(pk_1, sk_1), \cdots, (pk_q, sk_q)$ then we can sign q-messages of the form $(u, m_u) \in \{1, 2, \cdots q\} \times M$. Why? We use key sk_1 to sign the message $(1, m_1)$, key (pk_2, sk_2) to sign the message $(2, m_2)$, etc. In other words, the index u decides that we'll use the key sk_u to sign the message (u, m_u).

4.3 From q-index signatures to stateless many-time signatures

Let's say we have 2-index signature schemes with key sk_0 that can sign 2 messages of the form $(u, m_u), u = 1, 2$. How can we extend it to sign 4 messages? The trick is we use the key sk_0 to sign 2 public keys (pk', pk'') and we use the key (sk', pk') to sign 2 messages $(1, m_1'), (2, m_2')$ and the key (sk'', pk'') to sign 2 messages $(1, m_1''), (2, m_2'')$. Hence we can sign a total of $2^2 = 4$ messages. The structure looks like a Merkle tree where the root is the key sk_0 which signs the left child public key pk' and the right child public key pk''. These 2 intermediate nodes' keys (sk', pk') and (sk'', pk'') ,in turn, can sign leaf nodes $(1, m_1'), (2, m_2')$ and $(3, m_1''), (4, m_2'')$. Note that I purposely change the index of m'' from $\{1, 2\}$ to $\{3, 4\}$, it doesn't change the fact that it's a 2-indexed signature but the number $3, 4$ reflects the order of leaf nodes from left to right in the tree.

In theory, it's not difficult to extend the above algorithm to sign 2^d leaf nodes using a tree of depth d. However, there are 2 main problems that have to be solved. The 1st problem is we have to remember which leaf indices that have been used to sign the message. If we reuse the same leaf index to sign 2 different messages then the algorithm becomes insecure because 1 leaf index corresponds to a one-time signature. The 2nd problem is that it's infeasible to generate and store $2^d - 1$ root and intermediate keys.

To solve the 1st problem, we can choose large $d = 256$ and to sign a message m, we choose a random 256-bit leaf index. The chance of indices's collision is negligible, so we don't have to remember which indices have been used.

To solve the 2nd problem, we use pseudo-random generator (PRG) and pseudo-random function (PRF) to deterministically generate intermediate nodes's keys on the fly. Let's say the root key is $sk = (k, sk_0)$ where k is our secret seed. To sign a message m, we choose a random index leaf node $a = (a_1, \cdots, a_d)$. The index $a = (a_1, \cdots, a_d)$ means that in the path from root to the leaf, we've gone through the following intermediate nodes (a_1), (a_1, a_2), (a_1, a_2, a_3), $\cdots$, $(a_1, a_2, \cdots, a_d)$ where we go to the left if $a_k = 0$ and go to the right if $a_k = 1$. To compute the key at intermediate node $(a_1, \cdots, a_i)$, we computes a seed for that node $r_i = PRF(k, (a_1, \cdots, a_i))$ and uses the seed r_i to generate the key pairs $(pk_i, sk_i) = PRG(r_i)$ and we sign the generated public key using the key above it in the tree $S(sk_{i-1}, (a_i, pk_i))$. Finally, we use the key at depth $d - 1$ to sign the message m at depth d: $S(sk_{d-1}, (a_d, m)$.

References

[1] Matthew Green. Hash-based signatures: An illustrated primer. `https://blog.cryptographyengineering.com/2018/04/07/hash-based-signatures-an-illustrated-primer/`.

[2] Adam Langley. Hash based signatures. https://www.imperialviolet.org/2013/07/18/hashsig.html.

[3] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.*

[4] Leslie Lamport. Constructing digital signatures from a one-way function.