1 One-Time Signatures

We recall the definition of unforgeability under chosen-message attack for a signature scheme, which requires that for all nuppt $\mathcal{F}$,

$$\Pr_{(vk,sk)\leftarrow \mathsf{Gen}}\left[\mathcal{F}^{\mathsf{Sign}_{sk}(\cdot)}(vk) \text{ forges}\right] \leq \mathrm{negl}(n).$$

Constructing a scheme that satisfies such a strong security notion is very involved, so we start with an easier goal: unforgeability under a *one-time* chosen-message attack (uf-1cma). The definition is syntactically identical to the one above, but $\mathcal{F}$ is restricted to make (at most) one query to its signing oracle.

Here we describe *Lamport's one-time signature scheme* OTS for messages of length $\ell = \mathrm{poly}(n)$, which is based on a one-way function f:

- Gen: for $i \in [\ell]$, $b \in \{0,1\}$, choose independent $x^{i,b} \leftarrow \{0,1\}^n$ and let $y^{i,b} = f(x^{i,b})$. Output signing key $sk = \{x^{i,b}\}$ and verification key $vk = \{y^{i,b}\}$.

  Visually, we can represent the keys as tables:

$$sk = \begin{array}{|c|c|c|c|}\hline x^{1,0} & x^{2,0} & \cdots & x^{\ell,0} \\ \hline x^{1,1} & x^{2,1} & \cdots & x^{\ell,1} \\ \hline \end{array} \qquad vk = \begin{array}{|c|c|c|c|}\hline y^{1,0} & y^{2,0} & \cdots & y^{\ell,0} \\ \hline y^{1,1} & y^{2,1} & \cdots & y^{\ell,1} \\ \hline \end{array}$$

- $\mathsf{Sign}(sk, m \in \{0,1\}^\ell)$ reveals $\sigma = (x^{1,m_1}, x^{2,m_2}, \ldots, x^{\ell,m_\ell})$. That is, for each bit of the message it reveals one of the two preimages (either $x^{i,0}$ or $x^{i,1}$), as determined by the message bit.
- $\mathsf{Ver}(vk, m, \sigma)$ accepts if $y^{i,m_i} = f(\sigma_i)$ for all $i \in [\ell]$.

Correctness is by inspection.

Theorem 1.1. *Lamport's OTS is unforgable under a one-time chosen-message attack (uf-1cma-secure) assuming that f is a OWF.*

Proof. The idea is to give a reduction that uses a hypothetical forger $\mathcal{F}$ to the break the one-wayness of f. More precisely, design a simulator $\mathcal{S}$ which, given $y = f(x)$ for a uniformly random (but unknown) $x \leftarrow \{0,1\}^n$, uses $\mathcal{F}$ to find an $x' \in f^{-1}(y)$ with non-negligible advantage. The simulator will do this by "plugging" its given value of y into a random cell of the verification key vk (and constructing the rest of vk as usual), and "hoping" that it will be able to answer $\mathcal{F}$'s signing query while also extracting a preimage $x' \in f^{-1}(y)$ from the resulting forgery.

More formally, the simulator $\mathcal{S}^{\mathcal{F}}(y)$ works as follows. Given input y, it chooses $i^* \leftarrow [\ell]$ and $b^* \in \{0,1\}$. Let $y^{i^*,b^*} = y$. For all other indices, pick $x^{i,b} \leftarrow \{0,1\}^n$ and let $y^{i,b} = f(x^{i,b})$. Give $vk = \{y^{i,b}\}$ to $\mathcal{F}$. (We note that vk is properly distributed as in the real scheme, and that conditioned on the value of vk, the indices (i^*, b^*) are still uniformly random.)

Next, $\mathcal{F}$ is allowed at most one query to its signing oracle, which $\mathcal{S}$ must emulate. If $\mathcal{F}$ queries its signing oracle on a message $m \in \{0,1\}^\ell$, then $\mathcal{S}$ does the following:

- If $m_{i^*} = b^*$, then $\mathcal{S}$ quits (outputs $\perp$), as it does not know an inverse of $y^{i^*,b^*} = y$.
- Otherwise, $\mathcal{S}$ returns $\sigma = (x^{1,m_1}, \ldots, x^{\ell,m_\ell})$ as required.
- If $m'_{i^*} = b^*$ then output σ'_{i^*} else output $\perp$.

Finally, $\mathcal{F}$ outputs a forgery (m', σ'). If $m'_{i^*} = b^*$, then $\mathcal{S}$ outputs σ'_{i^*}; otherwise it quits (outputs $\perp$). Note that if $m'_{i^*} = b^*$ and σ'_{i^*} is a valid forgery for m', then $\sigma'_{i^*} \in f^{-1}(y^{i^*,b^*} = y)$, as desired.

We now analyze $\mathcal{S}^{\mathcal{F}}$. Clearly it is nuppt, hence $\mathbf{Adv}_f(\mathcal{S}^{\mathcal{F}})$ must be negligible. By the following claim, $\mathbf{Adv}_{\text{OTS}}(\mathcal{F})$ must therefore be negligible as well, as desired.

Claim 1.2. $\mathbf{Adv}_f(\mathcal{S}^{\mathcal{F}}) \geq \frac{1}{2\ell} \cdot \mathbf{Adv}_{OTS}(\mathcal{F})$.

We prove the claim: first, since $b^* \in \{0,1\}$ remains uniform (even given the vk), $\Pr[b^* \neq m_{i*}] = \frac{1}{2}$, so $\mathcal{S}$ is able to answer the signature query with probability $1/2$. Next, conditioned on a successful query phase, the index $i^* \in [\ell]$ is still uniformly random. Thus, for any $m' \neq m$ output by $\mathcal{F}$,

$$\Pr[m'_{i^*} = b^* \mid b^* \neq m_{i^*}] \geq \frac{1}{\ell},$$

because m' and m must differ in at least one position. Finally, conditioned on $m'_{i^*} = b^* \neq m_{i^*}$ (which occurs with probability at least $1/2\ell$ as argued above), the simulator outputs a valid preimage of y exactly when $\mathcal{F}$ outputs a valid forgery for m'. This completes the proof. □

Remark 1.3. Notice that the proof above showed *standard* unforgeability, not *strong* unforgeability. Is the scheme strongly unforgeable?

Question 1. Verify that this scheme is indeed only one-time. That is, show that if $\mathcal{F}$ is allowed to make two calls to the signing oracle, a forgery can be produced.

2 From One-Time to Many-Time

As you showed above, Lamport's scheme is only one-time. Still, we might imagine a way to obtain a many-time signature by choosing a "fresh" one-time signature key with each signed message, and authenticating it along with the message. However, this cannot work on its own, because the verification key vk in the one-time signature is much *longer* than the messages it can sign (about $2\ell n$ bits, for a length-preserving function f, versus ℓ bits).

We will now see a way to overcome this problem by "compressing" the fresh verification key using a special kind of hash function.

2.1 Collision-Resistant Hash Functions

We want a function that "compresses" its input, but for which it is hard to find two inputs that hash to the same output. This does not make much sense for a single *fixed* function, because a non-uniform adversary can simply have two colliding inputs "hard-wired" into it. But it does make sense for a function chosen at random from a family. We also want the function (i.e., its description) to be *public*, so that it can be run and checked by anyone. This means that the adversary should be given the description of the function, rather than just an oracle to it.

Definition 2.1. A family of hash functions $\{h_s \colon D \to R\}$ (where $|R| < |D|$) is *collision-resistant* if for all nuppt $\mathcal{A}$,

$$\Pr_{h \leftarrow \{h_s\}}[\mathcal{A}(s) \text{ outputs distinct } x, x' \text{ such that } h(x) = h(x')] \leq \text{negl}(n).$$

Example 2.2 (Based on discrete logarithm)**.** Consider the family $\{h_{p,g,y} \colon \mathbb{Z}_p^* \times \{0,1\} \to \mathbb{Z}_p^*\}$ for prime p, generator g of $\mathbb{Z}_p^*$, and $y \in \mathbb{Z}_p^*$:

$$h_{p,g,y}(x,b) = y^b g^x \bmod p.$$

A function from this family shrinks its input by 1 bit. To get additional compression, we can simply iterate the function by feeding its output (plus an additional bit) back to itself repeatedly. It is easy to check that a collision in the basic function can always be extracted from any collision in the iterated function (even if the colliding strings have different lengths).

Claim 2.3. *Under the discrete log assumption (i.e., that computing $\log_g(y)$ is hard), the family $\{h_{p,g,y}\}$ is a collision-resistant hash family.*

Proof. We will construct a reduction $\mathcal{S}(p,g,y)$ that uses a collision-finder for $\{h_{p,g,y}\}$ to compute $\log_g(y)$, i.e., finds the $z \in \mathbb{Z}_p^*$ such that $y = g^z \bmod p$. The reduction $\mathcal{S}$ simply asks for a collision $(x,b),(x',b')$ in $h_{p,g,y}$, and returns $x - x' \in \mathbb{Z}_p^*$. We claim that $\mathcal{S}$ succeeds whenever the collision is valid. To see why, observe that we cannot have $b = b'$, as $y^b g^x = y^b g^{x'} \in \mathbb{Z}_p^*$ implies $x = x' \in \mathbb{Z}_p^*$ by cancellation and the fact that exponentiation $x \mapsto g^x$ is a permutation on $\mathbb{Z}_p^*$. Therefore, $b \neq b'$; without loss of generality, assume $b = 0$ and $b' = 1$. Then $g^x = y g^{x'} \Rightarrow y = g^{x-x'}$, and $x - x' = \log_g(y)$, as desired. □

Example 2.4 (Based on subset-sum)**.** Let $N = 2^n$, and for a vector $a = (a_1, a_2, \ldots, a_{2n}) \in \mathbb{Z}_N$, define $h_a \colon \{0,1\}^{2n} \to \mathbb{Z}_N$ as $h_a(x) = \sum_i x_i a_i \bmod N$. Finding distinct $x, x' \in \{0,1\}^{2n}$ such that $h_a(x) = h_a(x')$ implies finding $z = x - x' \in \{0,\pm 1\}^{2n}$ such that $\sum_i a_i z_i = 0 \mod N$. This is widely believed to be hard for a *uniformly random* a. (Moreover, there is some strong and surprising evidence of hardness based on lattice problems, which we may return to later in the course.) Also note that the function already compresses its input by a factor of 2; moreover, the input length can be made even larger (say, $100n$) without making collisions significantly easier to find.

2.2 Improved OTS

Using a collision-resistant hash family $\{h_s : \{0,1\}^{\ell'} \to \{0,1\}^{\ell}\}$, we can now define a slightly modified OTS′ that uses OTS to sign messages of length $\ell' > \ell$:

- Gen: choose $h \leftarrow \{h_s\}$ and $(vk, sk) \leftarrow \mathsf{OTS.Gen}$. Output $vk' = (vk, h)$ and $sk' = (sk, h)$.
- $\mathsf{Sign}_{(sk,h)}(m \in \{0,1\}^{\ell'})$: output $\sigma \leftarrow \mathsf{OTS.Sign}_{sk}(h(m))$.
- $\mathsf{Ver}_{(vk,h)}(m, \sigma)$: output $\mathsf{OTS.Ver}_{vk}(h(m), \sigma)$.

Correctness is apparent. The unforgeability of this scheme (under one-time chosen-message attack) follows directly from the fact that to produce a valid forgery, $\mathcal{F}$ must either find a collision in h or break OTS, both of which are hard.

2.3 Chaining Signatures

We will now see how to use a CRHF and the concept of *chaining* one-time signatures to obtain a secure many-time signature scheme. The main idea is to generate a *fresh* key pair along with each signed message, and to authenticate it along with the message. The next message is signed with the key generated with the

prior message (and a new key pair is generated and authenticated along with it, etc.). Notice, however, that the signer must keep *state* to know which key to use and what to include in a given signature.

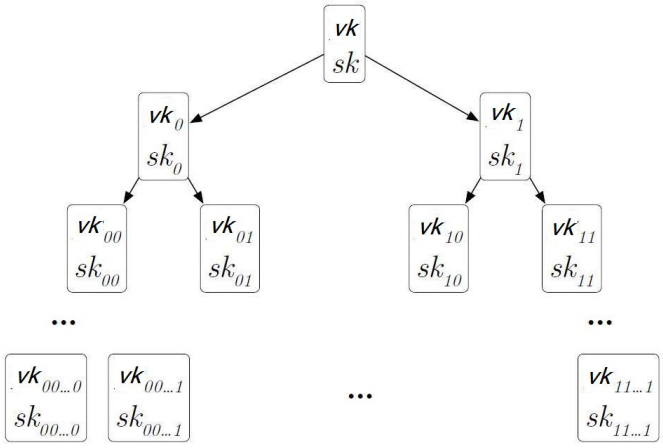
Assume we have a secure one-time signature scheme OTS, which we assume can sign messages that are as long as a verification key vk of OTS, plus ℓ. The full many-time signature scheme signs messages of length ℓ, and works as follows:

- Gen:
  - generate $(sk_1, vk_1) \leftarrow \text{OTS.}\mathsf{Gen}$
  - output $sk = sk_1$ and $vk = vk_1$
- $\mathsf{Sign}(sk_i, m_i)$ (for $i = 1, 2, \ldots$)
  - generate a fresh $(sk_{i+1}, vk_{i+1}) \leftarrow \text{OTS.}\mathsf{Gen}$
  - let $\sigma_i = \text{OTS.}\mathsf{Sign}_{sk_i}(m_i \| vk_{i+1})$
  - output $(vk_{i+1}, \sigma_i, m_i; \ldots; vk_2, \sigma_1, m_1; vk_1)$
    (The signature is the entire authentication chain; the message m_i itself appears in the chain.)
- $\mathsf{Ver}_{vk}(vk_{i+1}, \sigma_i, m_i; \ldots; vk_2, \sigma_1, m_1; vk_1)$
  - accept if $vk_1 = vk$ and $\text{OTS.}\mathsf{Ver}_{vk_j}(\sigma_j, m_j || vk_{j+1})$ accepts for all $j \in [i]$; otherwise, reject.

Notice that not only does the signer keep state, but the signature size increases *linearly* in the number of signatures ever generated by the signer. (The verification time also increases linearly, but at least the verifier does not need to keep any state.) It can be shown that this signature scheme is uf-cma-secure. The intuition is that a forgery must somewhere "break off" of the chain produced by the legitimate signer, and hence is a forgery against one of the vk_i, which is hard to produce due to the uf-1cma security of OTS. A formal proof is relatively routine.

2.4 Improvements

One way to improve the above many-message scheme is to authenticate *two* new verification keys instead of one at each step. This new construction builds, in a depth-first manner, a binary tree of depth n, where each node in the tree is associated with one key pair (sk, vk). Such a digital signature algorithm can perform up to 2^n signatures with signature size that depends only linearly in n.



At any point of time, the signer maintains a special node which it calls the *current node*. To sign a message m, if the current node is at a depth less than n in the tree, the signer generates and stores 2 more key-pairs and sets them as the children of the current node. It uses the signing key in the current node to sign the message together with the two new verification keys, and updates the current node as the left child. Otherwise, (i.e., if the current node is already at a depth n), the signer uses the current node to sign the message and updates the current node to be the successor in the preorder traversal of the tree.

Here the signer need only include a transcript of all the previously signed messages *down the path from the root to the current node*. In addition, the signer need only keep state containing the keys along the path to the current node, and their siblings.

Moreover, we can make the scheme entirely *stateless* by viewing the tree as *implicitly* fully determined by a PRF f_k. That is, in order to generate the key pair at a certain node, the signer computes f_k on a corresponding input, and uses the result as the random coins for OTS.Gen. Because f_k always returns the same output on the same input, the signer can consistently regenerate any desired node as needed. To sign a message, the signer then just signs relative to a randomly chosen leaf node. With high probability, no leaf is ever used more than once, so (intuitively, at least) the scheme is secure. A full proof is rather involved, but not especially difficult.

Answers

Question 1. Verify that this scheme is indeed only one-time. That is, show that if $\mathcal{F}$ is allowed to make two calls to the signing oracle, a forgery can be produced.

Answer. The key thing to notice is that when a message is signed, half the secret key is revealed. For example, if the oracle is queried on 0^ℓ, then the entire first row of the secret key is revealed. Querying 1^ℓ reveals the second row. As such, an attacker that is allowed two oracle queries can simply query 0^ℓ and 1^ℓ to reveal the entire secret key. From there, a valid signature can be produced for any message.