1 Introduction

In the previous lecture, we learned how to design “one-time signature schemes,” which are secure as long as the adversary only gets to have one message-signature pair. In this lecture, we cover two methods of extending one-time signature schemes to the full definition of security.

**Definition 1 (Security)** We model our attacker $A$ has a PPT that can request signatures for messages $m_i$ of its choice. Then, a signature scheme is “existentially unforgeable under an adaptive chosen message attack” if $A$ cannot produce a signature $\hat{\sigma}$ for any message $\hat{m}$ of her choice, with the exception of the signature pairs $(m_i, \sigma_i)$.

The Lamport OTS scheme described in the previous lecture does not have this full definition of security. In fact, if the attacker is allowed to receive even two messages, then she can break the scheme. In particular, the attacker can choose any message $m$ as the first message, and then choose its bitwise complement $\bar{m}$ as the second message. The two signatures will then reveal the entire signature key to the attacker, from which she can sign messages at will.

2 Naor-Yung scheme

2.1 Path-based scheme

In order to create a signature algorithm that satisfies the full definition of security, we chain together many Lamport OTS schemes, using each one only once.

**Algorithm 1 (Path-based scheme)** We create a signature scheme by starting with a Lamport OTS. Then, each time that the signer signs a message, she also creates a new Lamport OTS key pair.

**Key generation** $G(1^k)$ creates a Lamport OTS key pair $(VK, SK)$.

**Signature** To sign the $i$th message $m_i$, a random new Lamport OTS key pair $(VK_i, SK_i)$ is created. Then, the signer performs $\text{sign}_{SK_{i-1}}(m_i \circ VK_i)$, where the $\circ$ denotes concatenation. The signature returned is

$$\begin{cases} 
[m_1, & VK_1, \text{sign}_{SK}(m_1 \circ VK_1) \\
  m_2, & VK_2, \text{sign}_{SK}(m_2 \circ VK_2) \\
  \vdots \\
  m_i, & VK_{i-1}, \text{sign}_{SK_{i-1}}(m_i \circ VK_i) \\
  & VK_i
\end{cases}$$

Note that the signer saves all the previous $VK$’s and signatures, so in fact the only new things that the signer created for the signature of the $i$th message were $(VK_i, SK_i)$ and $\text{sign}_{SK_{i-1}}(m_i \circ VK_i)$.

**Verification** To verify the $i$th message, the verifier proceeds in a linked-list manner. First, she uses the publicly-known $VK$ to verify the signature of $m_1 \circ VK_1$. Then, she takes the $VK_1$ part of the message and uses it to verify the second signature. This process iterates, until finally she uses $VK_{i-1}$ to verify the signature of $m_i \circ VK_i$. Pictorially,

$$m_1, (VK, SK) \rightarrow m_2, (VK_1, SK_1) \rightarrow \cdots \rightarrow m_i, (VK_{i-1}, SK_{i-1})$$
There is one problem with the scheme described above, however. Namely, the verification key is as long as a signature key, so the above algorithm requires us to use the Lamport OTS scheme to sign a message longer than the signature key, which is not possible. Therefore, we use a collision-resistant hashing function \( h : \{0,1\}^* \rightarrow \{0,1\}^k \) first, and replace all instances of \( \text{sign}_{SK_i}(m_i \circ VK_i) \) by \( \text{sign}_{SK_{i-1}}(h(m_i \circ VK_i)) \). After the hash is in place, the algorithm is a secure signature algorithm.

**Theorem 1** The path-based algorithm is existentially unforgeable under a chosen message attack.

**Proof** Suppose that an adversary \( A \) obtains signature pairs of the type shown above and then makes a new signature with \( i \) components, which we will denote with asterisks. Note that \( A \) receives only a polynomial number of message-signature pairs, and each one is the result of generating random \((VK_j, SK_j)\) pairs, so the probability of any two messages using the same \((VK_j, SK_j)\) pair is negligible.

First, suppose that \( VK_{i-1}^* = VK_{i-1} \) for some signature that she was given. Because her signature must be different from the one that was given to her, it must be the case that \( m_i^* \neq m_i \). But, the fact that the attacker was able to produce a valid \( \text{sign}_{SK_{i-1}}(m_i, VK_i) \) means that she violated the one-time security of the Lamport scheme given by \((VK_{i-1}, SK_{i-1})\).

Second, suppose that \( VK_{i-1}^* \neq VK_{i-1} \) but \( VK_{i-2}^* = VK_{i-2} \). Then, the attacker got \( VK_{i-2} \) and one signature \( \text{sign}_{SK_{i-2}}(m_{i-1} \circ VK_{i-1}) \) and was able to produce a new signature \( \text{sign}_{SK_{i-2}}(m_{i-1} \circ VK_{i-1}) \). This violates the one-time security of the Lamport scheme given by \((VK_{i-1}, SK_{i-1})\).

Third, suppose that \( VK_{i-1}^* \neq VK_{i-1} \) and \( VK_{i-2}^* \neq VK_{i-2} \), but \( VK_{i-3}^* = VK_{i-3} \). We proceed inductively to solve this and all succeeding cases in the same manner as above.

So, the algorithm above is secure, but unfortunately it imposes harsh requirements on the signer’s and verifier’s computers. Let \( M \) denote the number of messages sent.

- The signature size of the each message includes all the previous messages and their signatures, so the signature size grows as \( O(M) \).
- The signature time is constant \( O(1) \), because to create a signature, the signer only needs to compute \( \text{sign}_{SK_{i-1}}(m_i \circ VK_i) \). The rest of the signature should be stored in its memory.
- The verification time grows as \( O(M) \) because the verifier must verify every previous message.
- The signer has to maintain \( O(M) \) state in memory, namely the entire signature of the previous message.

### 2.2 Tree-based scheme

We can improve the idea from above by using a tree of \((VK_i, SK_i)\) key pairs instead of a linked list. We start with \((VK, SK)\) like before, and to sign the first message \( m_1 \) we generate two child nodes \((VK_0, SK_0)\) and \((VK_1, SK_1)\), and sign \( \text{sign}_{SK}(m_1 \circ VK_0 \circ VK_1) \). Similarly, at each node of the tree, we sign a message by creating two child nodes and signing the message concatenated with the verification key of the child nodes. So, the tree is populated as follows.

```
  m_1,(VK,SK)
    /   \
 m_2,(VK_0,SK_0) m_3,(VK_1,SK_1)
     /      \
 m_4,(VK_00,SK_00) m_5,(VK_01,SK_01)
         /        \
 m_6,(VK_10,SK_10) m_7,(VK_11,SK_11)
```

2
Table 1: Complexities associated with three digital signature schemes

<table>
<thead>
<tr>
<th></th>
<th>Path-based scheme</th>
<th>Tree-based scheme</th>
<th>Naor-Yung scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature size</td>
<td>$O(M)$</td>
<td>$O(\log M)$</td>
<td>$O(\log M)$</td>
</tr>
<tr>
<td>Signing time</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log M)$</td>
</tr>
<tr>
<td>Verification time</td>
<td>$O(M)$</td>
<td>$O(\log M)$</td>
<td>$O(\log M)$</td>
</tr>
<tr>
<td>Signer's state</td>
<td>$O(M)$</td>
<td>$O(\log M)$</td>
<td>$O(\log M)$</td>
</tr>
</tbody>
</table>

The algorithm here is similar to the one before: to sign message $m_i$, we follow the list of messages that take us to $m_i$, and use that list in the manner shown in the previous section to construct a signature for $m_i$. Because the signature and verification process still works as before, the proof of security from the path-based method works here too.

The signature size is now $O(\log M)$ because the signature only has to record those messages, verification keys, and signatures for the appropriate messages along the path to get to the desired node. Similarly, the verification time is $O(\log M)$ now too. The signing time is still $O(1)$ because each signature has to create only one new node to the tree. Unfortunately, the state to maintain is still $O(M)$ because all nodes of the tree must be saved in memory.

### 2.3 Full Naor-Yung scheme

In this scheme, we imagine a tree-based scheme of height $n$, except that we only use the leaf nodes to hold messages; all of the interior nodes of the tree are only used to sign their children. Of course, such a scheme has the huge problem that we need to store $2^n$ signing keys in memory, which is not possible.

Recall that each Lamport signing key is a selection of $2^n$ random elements. Instead of creating all of these random elements and storing them in memory, we can instead create a seed $S$ and then use a pseudorandom function $PRF_S$ to generate the signing keys. In particular, we set $SK_0 = PRF_S(0)$, $SK_1 = PRF_S(1)$, $SK_{00} = PRF_S(00)$, and so on. Note that the signer never has to store the verification keys because they can be computed as $f$ applied to the appropriate signing key, where $f$ is the one-way function used to define the Lamport OTS scheme. As a result, now the signer only has to store $f$, $SK$, and the seed $S$, and then the entire tree can be computed from it.

Here is an example of our new tree layout, in the case where $n = 2$.

```
(VK, SK)  
    /   
(VK_0, SK_0)  (VK_1, SK_1)  
  /     /  
m_1, (VK_{00}, SK_{00}) m_2, (VK_{01}, SK_{01}) m_3, (VK_{10}, SK_{10}) m_4, (VK_{11}, SK_{11})
```

This scheme is tree-based so once again the signature size is $O(\log M)$. Similarly, verification time is still $O(\log M)$. On the other hand, now the tree is not stored in memory, so each signature requires re-creating the appropriate nodes of the tree using the pseudorandom function, so the signing time is $O(\log M)$. Finally, it would appear at first-glance that the signer maintains constant state. But in addition to keeping $f$, $SK$, and $S$, the signer must also remember the message number itself, so technically the signer maintains $O(\log M)$ state.

Table 1 contains the complexities of the four essential functions under each of the three models described above.

Theorem 2 The Naor-Yung scheme is existentially unforgeable under a chosen message attack.

Proof We demonstrate this in two steps.
1. Suppose that instead of using the pseudorandom function, we used truly random values to generate all the SK’s. Then, this scheme would be the tree-based scheme from the previous section, and the proof of security from Theorem 1 (modified slightly to use trees instead of linked lists) would hold.

2. Once the random values are replaced by a pseudorandom function, suppose that the security assumption fails, so there exists a PPT algorithm $A$ that can break security. Then, $A$ provides a statistical test for detecting pseudorandom functions from truly random ones, so $A$ violates the pseudorandomness property.

2.4 Universal one-way hash functions

The above scheme assumes the existence of one-way functions and of collision-resistant hash functions. But at the moment, it is an open question as to whether the one-way function assumption alone is sufficient to construct a collision-resistant hash function; instead, in practice more stringent assumptions are used like the hardness of the discrete logarithm problem. It would be nice if we could construct a digital signature algorithm that used only the one-way function assumption. We first recall the definition of a collision-resistant hash function.

**Definition 2 (Collision-resistant hash function)** A collection $H_n = \{h_i\}_{i \in \{0,1\}^n}$ of “collision-resistant hash functions” is a collection of functions $h_i : \{0,1\}^n \rightarrow \{0,1\}^{n-1}$ such that for any PPT algorithm $A$ that receives a function $h_i$ from the collection,

$$\Pr[A(h_i) = (x, y) \text{ such that } x \neq y \text{ but } h_i(x) = h_i(y)] < \text{neg}.$$ 

That is, it is hard for a PPT algorithm to find two elements that hash to the same value. We can create a slightly weaker notion of a hash function as follows.

**Definition 3 (Universal one-way hash function)** A collection $H_n = \{h_i\}_{i \in \{0,1\}^n}$ of “universal one-way hash functions” is a collection of functions $h_i : \{0,1\}^n \rightarrow \{0,1\}^{n-1}$ such that for any PPT algorithm $A$ that receives a function $h_i$ from the collection and a value $x$ in the domain of $h_i$,

$$\Pr[A(h_i, x) = y \text{ such that } x \neq y \text{ but } h_i(x) = h_i(y)] < \text{neg}.$$ 

This definition is slightly weaker because the PPT algorithm $A$ is given one input $x$ and does not have the freedom to choose both $x$ and $y$ at will.

Note that if we took any one-way permutation $f$, then by the definition of a permutation there would not exist any $x \neq y$ such that $f(x) = f(y)$. So this definition is non-trivial because it imposes the additional condition that the function must shrink the size of any input by 1 bit. However, the one-way permutation assumption is enough to create a universal one-way hash function.

**Theorem 3** If one-way permutations exist, then universal one-way hash functions exist.

**Proof** Let $f$ be a one-way permutation. Also, define the linear functions $g'_a : Z_p^* \rightarrow Z_p^*$ given by $g'_a(x) = ax$, which are certainly not hash functions because they are easily invertible. Additionally, define the function chop : $\{0,1\}^n \rightarrow \{0,1\}^{n-1}$ as the function that chops off the final bit of its input. Then, set $g_a(x) = \text{chop}(g'_a(x))$. Finally, we create the function $h_a(x) = g_a(f(x))$. We claim that $\{h_a(x)\}$ is a collection of universal one-way hash functions.

First, note that $h_a$ includes as its final step the chop function, which means it does reduce the length of its input as required. So, it remains to show that it satisfies the probability equation.

4
Suppose for the sake of contradiction that there exists a PPT adversary $A$ that given $(a, x_1)$ can output $x_2$ such that $h_a(x_1) = h_a(x_2)$. We will use this to construct a PPT $B$ that inputs $y$ and can output $f^{-1}(y)$.

The algorithm $B$ picks $\tilde{x}$ at random and computes $\tilde{y} = f(\tilde{x})$. We claim that there exists some $a$ such that $g_a(\tilde{y}) = y_10$ and $g_a(q_a) = y_11$, where $y_1$ is $n - 1$ bits long and the 0 or 1 denotes the final bit. Namely, it is the solution to $a(y - \tilde{y}) = y_10 - y_11$, which is linear so there does exist a solution $a$. With this value of $a$, we see that $g_a(y) = g_a(\tilde{y})$, so $h_a(f^{-1}(y)) = h_a(\tilde{x})$.

So, we give $(a, \tilde{x})$ to the algorithm $A$. If it succeeds, then it will output $f^{-1}(y)$ as desired. Therefore, the PPT $B$ can break the one-way permutation $f$, which is a contradiction so $A$ cannot exist.

We may replace the collision-resistant hash function in the Yang-Naor model above with a universal one-way hash function, and the proof of security in Theorem 1 really only used the universal one-way hash function property, not the collision-resistant property, so it still holds. Therefore, the entire digital signature algorithm can be implemented with just the one-way permutation assumption.

3 Random Oracle Model

Instead of using the Lamport OTS scheme and hash functions, we can develop a digital signature scheme using the concept of a random oracle. This scheme requires much harsher assumptions, but the resulting algorithm is much more efficient. As a result, it is the scheme commonly used today, with SHA1 or MD5 used as the random oracle function.

Algorithm 2 (Random oracle digital signature) The assumptions of the random oracle digital signature algorithm are the existence of families of trapdoor one-way functions and of random oracles.

Key generation $G(1^k)$ chooses a trapdoor one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ from a family of trapdoor functions and sets $VK = f$ and $SK = f^{-1}$.

Signature The signature of a message $m$ is $\text{sign}_{SK}(m) = f^{-1}(H(m))$.

Verification To verify message-signature pair $(m, \sigma)$, the verifier checks to see if $f(\sigma)$ equals $H(m)$.

Note that unlike with the Naor-Yung algorithms, each message can be signed without any knowledge of the previous signatures, so the signature length does not grow at all with the number of messages signed. At first glance, this algorithm appears to be like the failed Diffie-Hellman idea of digital signatures, but the presence of the random oracle function essentially “randomizes” the message $m$ before $f^{-1}$ is applied to it, foiling the attack that broke the Diffie-Hellman idea.

Theorem 4 The random oracle-based digital signature scheme is existentially unforgeable under a chosen message attack.

Proof Suppose for the sake of contradiction that there exists an algorithm $A$ that can break the scheme. That is, $A$ can choose messages $m_i$ and inquire about their hashes and signatures, and then it produces a valid message-signature pair $(m, \sigma)$. From this, we will construct a PPT $B$ that inputs $y$ and outputs $f^{-1}(y)$, breaking the one-way property of $f$.

Assume that $B$ knows in advance the maximum number of hashing/signature inquiries that $A$ will ask; it can do this for example by running $A$ many times and observing its behavior. Call the maximum number of queries $q$, which must be polynomial in $n$ because $A$ is a PPT algorithm. Also, for now consider the case where $A$ only asks hashing questions (we will deal with the signature questions later).

First, $B$ chooses at random $i^* \in \{1, \ldots, q\}$. The algorithm $A$, like any computer program, would not contain the random oracle but would rather make a function call to the random oracle to obtain hashes. But in this instance, $B$ contains $A$, so it has the power to give $A$ any values it likes for hashes. In our
case, we design $B$ to answer a hashing question to $m_\ast$ with $y$, and to answer the hashing question of any other $m_i$ with $f(x_i)$ for randomly chosen $x_i$.

Once $A$ returns its message-signature pair $(m, \sigma)$, there are three cases to consider.

**Case 1** If $m = m_\ast$, then the signature that $A$ returns equals $f^{-1}(H(m_\ast))$. But $B$ told $A$ that the hash of $m_\ast$ was $y$, so really the signature $\sigma = f^{-1}(y)$. Thus, the signature is the desired inverse to $y$.

**Case 2** If $m = m_i$ for some $i \neq i\ast$, then $A$ returns the signature $f^{-1}(H(m_i))$. But $B$ told $A$ that the hash of $m_i$ was $f(x_i)$, so the signature is $\sigma = x_i$. This yields no new information to $B$.

**Case 3** If $m$ is different from all of the $m_i$, then information-theoretically $A$ has no idea what the hash of $m$ is because it never asked for it. So, $A$ would have to compute $f^{-1}$ of a value that is totally random to it, which means that the probability of choosing a valid signature is $\frac{1}{2^n}$, which is negligible.

So, we may assume that if $A$ succeeds, then one of the first two cases occurred, in which case the probability of $A$ having chosen $m_\ast$ is $\frac{1}{2^n}$. So, given that $A$ succeeds (which occurs with non-negligible probability), $B$ can invert $y$ with probability $\frac{1}{2^n}$, which is non-negligible as desired.

Now, we must handle the case in which $A$ makes signature queries too. $B$ will handle the signature queries as follows: if $A$ asks for sign($m_\ast$), then $B$ will fail, and if $A$ asks for sign($m_i$) for $i \neq i\ast$, then $B$ will return the signature $x_i$, which is the correct signature given that $B$ told $A$ that the hash $H(m_i)$ equalled $f(x_i)$.

So, $B$ succeeds if $A$ does not ask for sign($m_\ast$) and then actually chooses $m = m_\ast$. These two events are independent, so the probability of both occurring is the product of the individual probabilities. $A$ asks at most $q$ questions, and the probability of it not asking for sign($m_\ast$) in any given question is $1 - \frac{1}{q}$, so the probability that $A$ never asks for sign($m_\ast$) is $(1 - \frac{1}{q})^q$. Also, we already know from above that the probability of $m = m_\ast$ equals $\frac{1}{q}$. So the probability that $B$ succeeds is

$$
\Pr[B \text{ succeeds}] = \Pr[A \text{ succeeds}] \times \Pr[A \text{ does not ask for sign($m_\ast$})] \times \Pr[A \text{ chooses } m = m_\ast]
$$

$$
= \Pr[A \text{ succeeds}] \times \left(1 - \frac{1}{q}\right)^q \times \frac{1}{q}
$$

$$
\approx \Pr[A \text{ succeeds}] \times \frac{1}{eq}
$$

because $q$ must be an increasing function of $n$, and $\lim_{q \to \infty} (1 - \frac{1}{q})^q = \frac{1}{e} \approx 0.37$. Finally, $e$ is a constant, $q$ is polynomial in $n$, and $A$ succeeds with non-negligible probability, so the entire probability of $B$ successfully breaking the one-way function $f$ is non-negligible, as desired. \qed