9 Digital Signatures: Definition and First Constructions. Hashing.

9.1 Definition

First note that encryption provides no guarantee that a message is authentic. For example, if a message is encrypted with the one-time pad, the adversary can flip any of its bits by simply flipping the corresponding bit of the ciphertext. In any public-key encryption scheme, there is no way to tell the source of the message, because the key is public. Furthermore, the message can be modified in transit by the adversary in every encryption scheme we studied so far.

Thus, encryption ensures secrecy, but can't help you figure out who the message came from or what it was really meant to say. We need digital signatures for that.

A digital signature scheme is a triple of probabilistic polynomial-time algorithms (Gen, Sig, Ver). The key generation algorithm Gen outputs (PK, SK) when given 1^k as input. The signing algorithm Sig takes SK and m as input, and outputs a signature σ. The verification algorithm Ver takes PK, m, σ as input and outputs 1 or 0 (or true/false, valid/invalid, etc.). We require that signatures produced by Sig verify as correct by Ver: if $(\mathrm{PK}, \mathrm{SK}) \leftarrow \mathrm{Gen}(1^k)$, then for all m, $\mathrm{Ver}(\mathrm{PK}, m, \mathrm{Sig}(\mathrm{SK}, m)) = 1$ (perhaps with probability $1 - \eta(k)$). We may also restrict the message space to some set M, and instead saying "for all m," say "for all $m \in M$."

The above description says nothing about security. Indeed, to define security, one has to try a few examples to better understand the notion. Here is an example: suppose my Gen generates an RSA pair $\mathrm{PK} = (n, e), \mathrm{SK} = (n, d)$; to sign m, let $\sigma = m^d \bmod n$, and to verify (m, σ), check if $m = \sigma^e \bmod n$. We identified a few problems with this in class: if you have signatures σ_1 on m_1 and σ_2 on m_2, you can compute $\sigma_1\sigma_2 \bmod n$ to obtain a signature on $m_1 m_2 \bmod n$. Also, without observing any signatures at all, you can pick a random $\sigma \in \mathbb{Z}_n^*$ and compute $m = \sigma^e \bmod n$ to get a valid pair (m, σ). Of course, m may not be a meaningful message, but it's difficult to know what will be "meaningful" for a particular application.

The definition we want is that the adversary be unable to come up with a signature on a new message (*any* message of the adversary's choice—so called "existential forgery"), even after observing signatures on other messages of its choice (so-called "adaptive chosen-message attack"—adaptive because the choice of the next message may depend on the signatures received for previous messages). In other words, the adversary $E^?$ is a probabilistic polynomial-time oracle machine (we use the superscript "?" to denote the fact that the machine has access to an oracle; when running the machine E with a specific oracle f, we write E^f, and sometimes denote by $\cdot$ the placeholder for an input to f given by E, e.g., $E^{f(\cdot)}$). Consider the following experiment of running the E with the oracle for the signing function:

`exp-forge`(k)
1. $(\mathrm{PK}, \mathrm{SK}) \leftarrow \mathrm{Gen}(1^k)$
2. $(m, \sigma) \leftarrow E^{\mathrm{Sig}_{\mathrm{SK}}(\cdot)}(1^k, \mathrm{PK})$
3. If m was not queried by E to its oracle and $\mathrm{Ver}_{\mathrm{PK}}(m, \sigma) = 1$, output 1. Else output 0.

Definition 1. A signature scheme Gen, Sig, Ver is existentially unforgeable under an adaptive chosen-message attack if for all probabilistic polynomial time $E^?$ there exists a negligible function η such that $\Pr[\texttt{exp-forge}(k) \to 1] \leq \eta(k)$ (the probability is taken over cointosses of Gen, E and the signing oracle $\mathrm{Sig}_{\mathrm{SK}}(\cdot)$).

The above definition is due to Goldwasser, Micali and Rivest [GMR88] (first appeared in 1984).

The Paradox Before 1984, it was believed by many to be impossible to achieve, for roughly the following reasons. Suppose you are trying to prove security of a signature scheme. The way you usually do it is as follows: if you have $E^?$ that can break the scheme, then you somehow use it to break your security assumption. Thus, you give $E^?$ some public key, it forges a signature, and the forgery helps you do something like invert RSA or break Discrete Logarithm. But in the process, $E^?$ will want oracle access to the signing oracle. So you will need to answer the queries of $E^?$ by making signatures on messages of adversary's choice. So you will need to know the secret key. But if you already know the secret key, how can $E^?$'s forgery help you solve a hard problem—after all, it seems like you could have made that forgery yourself, anyway? This reasoning turned out to be flawed, as we will see below.

9.2 Lamport's One-Time Signatures

Before building full-fledged signature schemes, let's consider one-time signatures. One-time signatures have the same definition, except that they allow you to sign only one message; in other words, the adversary $E^?$ is allowed only one oracle query (rather than polynomially many, as in the definition above). No security is guaranteed if two signatures are available to the adversary.

Let's restrict ourselves further to just single-bit messages for now. Here's is the signature scheme: let f be a one-way function (the same scheme will work with collections of one-way functions; the notation gets messier, which is why here we focus on a single f). Then for a secret key, choose two k-bit values x^0 and x^1, and let the public key be $y^0 = f(x_0)$ and $y^1 = f(x^1)$. To sign $m = 0$, output $\sigma = x^0$. To sign $m = 1$, output $\sigma = x^1$. To verify (m, σ), check if $f(\sigma) = y^m$.

Claim 1. The above one-time signature scheme for one-bit messages is secure.

Proof. Indeed, suppose it's not. Then out of the adversary $E^?$, we will build an inverter A for the one-way function. On input y, A flips a random coin to get b. Suppose $b = 0$. Then A picks x^0 at random, lets $y^0 = f(x^0)$, and runs $E^?$ on the public key (y^0, y). At some point $E^?$ may ask for a single signing query on some message m_1, for $m_1 = 0$ or $m_1 = 1$. If $m_1 = 0$, A returns x^0 in response to the query; else A aborts. If A did not abort and $E^?$ then outputs a forgery, it will have to be for $m = 1$ (because $E^?$ is required to forge on a new message), so A will learn σ such that $f(\sigma) = y$ and thus will invert f. Similar reasoning works for $b = 1$.

Note that if $E^?$ succeeds with probability ϵ then A will succeed with probability $\epsilon/2$, because the view of $E^?$ does not depend on A's choice of b—hence, the choice will be "lucky" (i.e., will match the query asked by $E^?$) half the time. □

It is easy to extend this scheme for l-bit messages: select $2l$ values $x_1^0, x_2^0, \ldots, x_l^0, x_1^1, x_2^1, \ldots, x_l^1$ for the secret key, and apply f to each to get $y_1^0, y_2^0, \ldots, y_l^0, y_1^1, y_2^1, \ldots, y_l^1$ for the public key. To sign message $m = m_0 m_1 \ldots m_l$, where m_i is the i-th bit, let $\sigma = (x_1^{m_1}, x_2^{m_2}, \ldots, x_l^{m_l})$. To verify $(m, \sigma = (z_1, z_2, \ldots, z_l)$, check if for each i, $1 \le i \le l$, $f(z_i) = y_i^{m_i}$. This scheme is due to Lamport [Lam79], apparently discovered by him as early as 1975.

Claim 2. The above one-time signature scheme for l-bit messages is secure.

Proof. The proof is the same as for the previous claim, except that A has to guess not only the bit b, but also the message position i. A then generates the secret key by selecting $x_1^0, x_2^0, \ldots, x_l^0, x_1^1, x_2^1, \ldots, x_l^1$, computes the public key $y_1^0, y_2^0, \ldots, y_l^0, y_1^1, y_2^1, \ldots, y_l^1$, and substitutes its input y in place of y_i^{1-b}. A will succeed in finding an inverse for y if it can answer the query of $E^?$ (i.e., if the i-th bit of the query message is b), and if the i-th bit of the forgery message is $1 - b$. To compute the probability of A's success, note that the query message and the forgery message have to differ in at least one bit position, which is position i with probability at least $1/l$. Thus, A has probability at least $1/l$ of guessing i correctly, and then probability $1/2$ of guessing b correctly. So success probability of A is at least $\epsilon/(2l)$. □

Note that the above bound on the success probability of A is "tight" in the following sense. If there is an algorithm that inverts f with probability δ, then we can build an adversary $E^?$ that succeeds in forging a signature with probability roughly $2l\delta$: namely, $E^?$ will try to invert f for at least one of the $2l$ public key values. If it succeeds, say, for value y_i^b, it will then query the some message whose i-th bit is $1 - b$, and then be able to forge a signature on a new message, which is the same as the query message except in the i-th bit.

Thus, it is meaningful to say that Lamport's signature scheme is $2l$ times less secure than the underlying one-way function.

What happened to the paradox? How was it that we were able to prove security of this scheme? After all, A needed to produce a signature and yet was able to invert a one-way function from a forgery! The point is that A produced a signature using one part of the key, while gained knowledge if the forgery used another part of key. The paradox failed to account for the idea that A may not know the entire secret key, and thus be able to forge signatures some messages and not others, and thus gain knowledge with some probability less than 1.

9.3 Collision-Resistant Hashing

Lamport's signatures can sign l-bit messages if the public key has $2l$ elements in it. We'd like to be able to sign arbitrary-length messages regardless of key length. In order to do so, we introduce the concept of *collision-resistant hashing*.

Definition 2. Let I be an index set, and for each $i \in I$, let $H_i : D_i \to R_i$ be a function. Then $\{H_i\}_{i \in I}$ is a collection of collision resistant hash functions if:

1. there exists a probabilistic polynomial-time algorithm Gen that on input 1^k outputs $i \in I$;
2. $|D_i| > |R_i|$, i.e., the function H_i actually does reduce its domain;
3. given i and $x \in D_i$, $H_i(x)$ is efficiently computable;
4. for all probabilistic polynomial-time C, there exists a negligible function η such that for all k, if $i \in I$ is chosen by $\mathrm{Gen}(1^k)$, then $\Pr[C(1^k, i) \to (x_1, x_2)$ such that $x_1 \neq x_2 \wedge H_i(x_1) = H_i(x_2)] \leq \eta(k)$, where the probability is over random choices made by Gen and C.

We will construct an example based on the discrete logarithm assumption. Let

$$I = \{(p, g, h) | p = 2q + 1 \text{ and } p, q \text{ are prime}, g, h \text{ are generators of } QR_p\}.$$

For $(p, g, h) \in I$ and $a \in \{1, 2, \ldots, q\}, b \in \{1, 2, \ldots, q\}$, define $H_{(p,g,h)}(a, b) = g^a h^b \bmod p$. The domain and range are $D_{p,g,h} = \{1, 2, \ldots, q\} \times \{1, 2, \ldots, q\}$, and $R_{p,g,h} = QR_p$ (note that we can make the range $R_{p,g,h} = \{1, 2, \ldots, q\}$ by simply outputting $p - (g^a h^b \bmod p)$ if $g^a h^b \bmod p > q$; this works because this is an efficiently computable bijection between QR_p and $\{1, 2, \ldots, q\}$, because exactly one of $x, p - x$ is in QR_p for $x \in \mathbb{Z}_p^*$ for safe p).

Assuming that safe primes can be efficiently generated (for condition 1 in the definition), and that discrete logarithm modulo a safe prime is hard, this turns out to be collision resistant, because of the following claim.

Claim 3. Suppose Gen, on input 1^k, generates a random k-bit safe prime p and two generators g, h of QR_p. Suppose C can find collisions with probability $\epsilon(k)$. Then there exists an algorithm that, given a random k-bit safe prime p and $g, h \in QR_p$, $g \neq 1$, finds $\log_g h$ modulo p with probability ϵ.

Proof. Given $p, g \neq 1, h$, first note that g is a generator of QR_p, because QR_p has prime order, so everything except 1 is a generator. To find $\log_g h$, do the following: if $h = 1$, output 0. Else h is a generator of QR_p, so run C on $1^{|p|}, p, g, h$. If C finds a collision, then it outputs $(a_1, b_1) \neq (a_2, b_2)$, such that $g^{a_1} h^{b_1} \equiv g^{a_2} h^{b_2} \pmod{p}$. Hence $g^{a_1 - a_2} \equiv h^{b_2 - b_1} \pmod{p}$. Note that exponents work modulo q, because the order of g and h is q. Note also that if $b_2 \neq b_1$, then $b_2 - b_1$ is relatively prime to q, because q is prime and

$|b_2 - b_1| < q$. Hence, if $b_2 \neq b_1$, we can compute an integer $c = (b_2 - b_1)^{-1} \bmod q$, and then $h = g^{(a_1 - a_2)c}$, so $\log_g h = (a_1 - a_2)c \bmod q$. The case of $b_1 = b_2$ is impossible, because then $g^{a_1 - a_2} \equiv 1$, so $a_1 = a_2$, but we assumed that C outputs two distinct pairs $(a_1, b_1), (a_2, b_2)$. □

The above collection of collision-resistant hash functions is not efficient enough for some practical applications. In practice, people often use ad hoc functions, such as MD5 [Riv92] or SHA-1 [NIS95], that are not actually families, but rather specific fixed functions (both take arbitrary length inputs; MD5 outputs 128 bits; SHA-1 outputs 160 bits). They are based not on number theory, but rather on sophisticated bit manipulations.

Note that for any fixed function, there always exists a collision (as long as the domain is larger than the range), and hence there always is a polynomial-time C that simply outputs that collision (even if no one knows what C is). So these don't quite fit our definition (though one can think of them as being chosen at random from some plausible family by their designers). However, it was until very recently believed that for both of these functions, to find a collision (i.e., to find such C) is quite difficult. One can view these functions as pulled out at random from some family by their designers.

Recent results put the security of these specific functions in doubt. In particular, using sophisticated cryptanalysis techniques, it is now possible to find collisions for MD5 [WFLY04] and for SHA-0 [BC04, Jou] (a precursor to SHA-1). No collisions for SHA-1 have been reported so far.

Finally, it must be noted that a hash function with k-bit outputs can be broken in roughly $2^{k/2}$ steps (this is known as the "birthday paradox"). Indeed, if you hash 2 random messages, the likelihood they collide is roughly 2^k. If you hash t random messages, that gives you $t(t-1)/2$ pairs, so the likelihood at least one pair collides is roughly $t^2/2^{k+1}$. This is actually an upper bound on the probability of collision. A lower bound is obtained by the following derivation: probability p of non-collision is, by simple counting,

$$
\begin{aligned}
p &= (1 - 1/2^k)(1 - 2/2^k)\cdots(1 - (t-1)/2^k) && (1)\\
&\leq \mathrm{e}^{(-1-2-\cdots-(t-1))/2^k} && (2)\\
&= \mathrm{e}^{t(t-1)/2^{k+1}} && (3)\\
&\leq 1 - t(t-1)/2^{k+1} + (t(t-1)/2^{k+1})^2/2\,. && (4)
\end{aligned}
$$

Line (2) and Line (4) follow from the Taylor series expansion of e^{-x} which gives $(1-x) \leq \mathrm{e}^{-x} \leq 1 - x + x^2/2$. Thus the probability of collision is at least $A - A^2/2$, where $A = t(t-1)/2^{k+1}$, and is not more than $t^2/2^{k+1}$.

So if $t = 2^{k/2}$, the likelihood of collision is quite good. Thus, a collision for a hash function with 128-bit output can be found by brute force in 2^{64} evaluations, and a collision for a hash function with 160-bit output in 2^{80} evaluations. 2^{80} operations seem infeasible today; 2^{64} are feasible with a good deal of time and money. (Note that the aforementioned attacks on MD5 and SHA-0 are much more sophisticated and much faster than this brute-force approach).

9.4 Signatures for Arbitrary-Length Messages

Recall that Lamport's signatures can sign only messages shorter than the public key (l-bit messages, whereas the public key has $2l$ long values). The following construction, using collision-resistant hashing, allows one to sign long messages with any signature scheme (if the original signature scheme was one-time, like Lamport's, then the resulting signature scheme will be one-time, as well).

Suppose I have a secure signature scheme $(\mathrm{Gen}', \mathrm{Sig}', \mathrm{Ver}')$ that is able to sign messages only of limited length, say, of length $l(k)$, where k is the security parameter. Suppose I also have a collision-resistant hash function family $\{H_i\}_{i \in I}$ with generation algorithm GenH, such that if $i \leftarrow \mathrm{GenH}(1^k)$, then H_i takes strings of some length $s(k)$ (or perhaps of unbounded length) and maps them to strings of size $l(k)$. Consider now the following signature scheme $(\mathrm{Gen}, \mathrm{Sig}, \mathrm{Ver})$:

1. Key generation algorithm $\text{Gen}(1^k)$: to generate a key, generate (PK', SK') using $\text{Gen}'(1^k)$; generate hash index i using $\text{GenH}(1^k)$; output (PK', i) as the public key and (SK', i) as the secret key.
2. Signing algorithm Sig: to sign a message m of size $s(k)$, first hash it to get $h = H_i(m)$, then sign the hash value using $\sigma = \text{Sig}'(\text{SK}', h)$, and output σ.
3. Verifying algorithm Ver: to verify a signature σ on a message m, first hash the message to get $h = H_i(m)$, then verify the signature on the hash value using $\text{Ver}'(\text{PK}', \sigma, h)$.

This scheme can sign much longer messages than the original scheme, because, we can have s be much greater than k (in fact, as Merkle trees below demonstrate, collision-resistant hash functions can be composed with themselves to work on longer inputs).

The proof of security is quite simple. Given a forger F for the the new scheme, we will construct a collision-finder C for $\{H_i\}$ and an adversary E that breaks the original signature scheme. One of the two will work with probability that is not negligible, which leads to a contradiction. Filling in the details is good homework exercise.

9.5 Merkle Trees and Merkle Signatures

9.5.1 Merkle Trees

A Merkle tree [Mer89] (the paper was written in 1979, but not published until 10 years later) is a way to "commit" to n messages with a single hash value in such a way that revealing any particular message requires revelation of only $\log n$ hash values. Here is how it works: place n values at the leaves of a complete binary tree (assume n is a power of 2 for ease of explanation). Compute the value at each nonleaf node in the tree as the hash of the values of its two children. The value at the root is the "commitment" to the values at the leaves. To reveal the value at leaf i, publish the value at leaf i as well as the value at the sibling of i and the values at siblings of each ancestor of i (the so-called "authenticating path" for i). One can verify that i was correctly revealed by simply computing hashes up the tree (first hash the value at i with its sibling, then the result with its own sibling, and so on) and checking that the ultimate hash value matches the root. Note that it is impossible to reveal two different values at a particular leaf without finding a hash function collision—hence, if the hash function is secure, Merkle tree root binds its publisher to the leaf values.

Merkle trees have multiple applications outside of signature design. For instance, if a creator of a dataset does not have enough capacity to answer queries on the data, it can outsource that capacity to an unsecured responder: put the data in a Merkle tree, publish the root, and ship the tree to (possibly more than one) responder. The responder needs not be trusted: it simply needs to provide authenticating paths to demonstrate that its responses to the queries are correct. Thus, for example, you can get guaranteed authentic stock quotes without having to ask the stock exchange, as long as the exchange publishes a trustworthy (e.g., digitally signed) Merkle tree root.

9.5.2 Merkle Signatures

We now use Merkle trees to extend one-time signatures to multi-time signatures. The idea is to generate n one-time public keys, and place them in a Merkle tree. The root of the Merkle tree becomes the public key of the signature scheme. The details follow.

Consider a one-time signature scheme $(\text{Gen}', \text{Sig}', \text{Ver}')$ and a collision-resistant function collection $\{H_i\}_{i \in I}$ with generation function GenH. Out of these, Merkle builds a signature scheme $(\text{Gen}, \text{Sig}, \text{Ver})$ to sign n messages as follows:

1. Key generation algorithm $\text{Gen}(1^k)$: run Gen' n times to get n key pairs $(\text{PK}_j, \text{SK}_j)$ for the one-time scheme. Run $\text{GenH}(1^k)$ to obtain select an index i (i.e., a hash function H_i) out of the collection. Merkle hash the one-time public keys to get the Merkle root r: in other words, compute

$s_1^1 = H(\mathrm{PK}_1, \mathrm{PK}_2), s_2^1 = H(\mathrm{PK}_3, \mathrm{PK}_4), \ldots, s_{n/2}^1 = H(\mathrm{PK}_{n-1}, \mathrm{PK}_n)$; repeat this process on the s^1 values to get $s_1^2 = H(s_1^1, s_2^1), s_2^2 = H(s_3^1, s_4^1), \ldots, s_{n/4}^2 = H(s_{n/2-1}^1, s_{n/2}^1)$; repeat again ($\log n$ times over all) to get the single root r. Output r as the public key and keep the hash index i, the n pairs $(\mathrm{PK}_j, \mathrm{SK}_j)$ and the Merkle tree as the secret key.

2. Signing algorithm Sig: to sign the j-th message m_j, sign it using the j-th one-time key: let $\sigma_j = \mathrm{Sig}'(\mathrm{SK}_j, m_j)$. Output σ_j, PK_j, and the authenticating path of PK_j that leads to r (the authenticating path consists of the siblings of the path from PK_j to the root r).

3. Verifying algorithm Ver: to verify the signature, first use Ver$'$ to check that the signature is correct with respect to PK_j, then use the authenticating path for PK_j to check that it leads to r (i.e., hash PK_j and its sibling to get the value at the parent of PK_j; hash that and its sibling to get the grandparent, and so on, until the root value, which should match r). If both checks hold, output 1; else output 0.

To prove that $(\mathrm{Gen}, \mathrm{Sig}, \mathrm{Ver})$ is a secure signature scheme, we will prove that if it is not secure, then either $\{H_i\}_{i \in I}$ is not a collision resistant family, or $(\mathrm{Gen}', \mathrm{Sig}', \mathrm{Ver}')$ is not a secure one-time signature scheme.

Recall that a hash function is not collision resistant if there exists an algorithm C that, on input $(1^k, i)$, finds a collision with probability that is not negligible in k: two pairs (x_1, x_2), (y_1, y_2) such that $H_i(x_1, x_2) = H_i(y_1, y_2)$.

Recall also that a one-time signature scheme is not secure if there exists an algorithm E that, on input PK_O for the one-time signature scheme, asks a single query (a signature σ for a message m), and then outputs a new message m' and signature σ' such that $\mathrm{Ver}'(\mathrm{PK}_O, \sigma', m') = 1$ with not negligible probability.

So suppose F is a forger for $(\mathrm{Gen}, \mathrm{Sig}, \mathrm{Ver})$. F gets a public key (Merkle root) A as input, asks a bunch of queries m_j and receives signatures σ_j, and then ultimately outputs a new message m'' and signature σ'' such that $\mathrm{Ver}(\sigma'', m'') = 1$ with not negligible probability. I will build both C and E out of F, and then argue that either one of those two will work.

Recall that C gets $(1^k, i)$ as input. To build C out of F, do as follows:

1. Perform the key generation algorithm Gen, as above, except don't generate another hash key, but rather use i given in the input. Obtain the public key r and the secret key $(\mathrm{PK}_j, \mathrm{SK}_j)$ and the Merkle tree.

2. Give r as input to F, and answer F's signing queries using the signing algorithm Sig (since you know all the secret keys).

3. Now F outputs σ'', m''. Of course, σ'' has to contain a public key for the one-time signature scheme, and its authenticating path to r. If that public key is not one of $(\mathrm{PK}_1, \mathrm{PK}_2, \ldots, \mathrm{PK}_n)$, then the authenticating path to r must result in a hash function collision somewhere in the Merkle tree. Output that collision.

Recall that E gets a public key for the one-time signature scheme, PK_O, as input, and is allowed one query to the signing oracle $\mathrm{Sig}(\mathrm{SK}_O, \cdot)$ for that scheme. To build E out of F, do as follows:

1. Guess a value ℓ between 1 and n at random.

2. Perform the key generation algorithm G, as above, except don't generate the ℓ-th public key but rather use PK_O given in the input in its place. Obtain the public key A and almost all of the secret keys $(\mathrm{PK}_j, \mathrm{SK}_j)$ (except SK_ℓ) and the Merkle tree.

3. Give A as input to F, and answer F's signing queries using the signing algorithm S, except for the ℓ-th signing query: for the ℓ-th signing query, use the one-time signing oracle $\mathrm{Sig}(\mathrm{SK}_O, \cdot)$ to which E is allowed one-time access.

4. Now F outputs σ'', m''. Of course, σ'' has to contain a public key for the one-time signature scheme, and the signature with σ' respect to that public key. If that public key happens to be one of $(\mathrm{PK}_1, \mathrm{PK}_2, \ldots, \mathrm{PK}_n)$, and, in fact, happens to be the ℓ-th one, then the one-time signature σ' is a valid signature with respect to $\mathrm{PK}_\ell = \mathrm{PK}_O$. Output m'', σ'.

All that's left is to argue that either C or E (or both) have not negligible probability of success. Well, if F succeeds with probability ϵ, then either with probability at least $\epsilon/2$ it forges a signature using a new public key, or with probability at least $\epsilon/2$ it forges a signature using a public key from the tree. In the first case, C succeeds with probability at least $\epsilon/2$, and in the second case, E succeeds with probability at least $\epsilon/2n$.

9.6 Naor-Yung Signatures

Naor-Yung signatures [NY89] are similar to Merkle signatures, but the idea is to build the tree top-down instead of bottom-up, in order to avoid having to know the bound on the number of signatures in advance, and having to have long key generation. Instead of hashing two children to get the parent, the parent signs the public keys of the two children. The tree can be built as deep and as wide as needed. To avoid having to memorize the secrets key of all the children, one can generate these using a pseudorandom generator, or, better yet, using a pseudorandom function (to be discussed in later lectures). Working out the details is a good exercise.

9.7 Stateful vs. Stateless Signatures

Merkle's and Naor-Yung signature schemes have a serious drawback: they require the signer to keep some state from one signature to the next. That is, to sign, the signer needs not only the secret key, but also information about what happened so far. If that information is incorrect, the signature scheme can be broken (because, for example, the signer might sign two messages with the same one-time key). If you look carefully at our definition of signatures, keeping state is not allowed: the input to the signer is the secret key and the message to be signed, but nothing else. Aside from a syntactic problem that stateful signatures do not formally satisfy the definition, there is an implementation problem: when implementing the signer in, for example, a computer, a smart card or a cellular phone, one needs to provide not only for secure storage of the secret key (which is fixed once at key generation), but also of the state (which changes with each signature). Thus, one needs to provide for secure read/write memory that survives power loss and potential physical attacks aimed at modifying it (as opposed to read-only memory for the secret key in the case of stateless schemes).

The first provably secure stateless signature scheme was a modification of [GMR88] (which itself was the first provably secure *stateful* signature scheme) due to Goldreich [Gol86]. We will not discuss it here. Naor-Yung signatures can also be modified to be stateless using similar techniques (the main idea is to build a tree of depth k, and go to a random leaf each time—the probability that you will go the same leaf twice is very low). More recent examples (also not discussed here) are to due Gennaro, Halevi and Rabin [GHR99], and Cramer and Shoup [CS00]; perhaps the most efficient one is due to Fischlin [Fis02].

References

[BC04] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In Matt Franklin, editor, *Advances in Cryptology—CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*. Springer-Verlag, 15–19 August 2004. See `http://eprint.iacr.org/2004/146` for an updated version.

[CS00] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[Fis02] Marc Fischlin. The cramer-shoup strong-rsa signature scheme revisited. Technical Report 2002/017, Cryptology e-print archive, `http://eprint.iacr.org`, 2002.

[GHR99] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In Jacques Stern, editor, *Advances in Cryptology—EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer-Verlag, 2–6 May 1999.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[Gol86] Oded Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In Andrew M. Odlyzko, editor, *Advances in Cryptology—CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110. Springer-Verlag, 1987, 11–15 August 1986.

[Jou] Antoine Joux. Collisions in SHA-0. Presented at the rump session of CRYPTO 2004.

[Lam79] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, October 1979.

[Mer89] Ralph C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1990, 20–24 August 1989.

[NIS95] FIPS publication 180-1: Secure hash standard, April 1995. Available from `http://csrc.nist.gov/fips/`.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 33–43, Seattle, Washington, 15–17 May 1989.

[Riv92] Ronald L. Rivest. *IETF RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992. Available from `http://www.ietf.org/rfc/rfc1321.txt`.

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Technical Report 2004/199, Cryptology e-print archive, `http://eprint.iacr.org`, 2004.