

## Chapter 5

# Authentication

### 5.1 Message Authentication

Suppose Bob receives a message addressed from Alice. How does Bob ensure that the message *received* is the same as the message *sent* by Alice? For example, if the message was actually sent by Alice, how does Bob ensure that the message was not tampered with by any malicious intermediary?

In day-to-day life, we use signatures or other physical methods to solve the forementioned problem. Historically, governments have used elaborate and hard-to-replicate seals, watermarks, special papers, holograms, etc. to address this problem. In particular, these techniques help ensure that only, say, the physical currency issued by the government is accepted as money. All of these techniques rely on the physical difficulty of “forging” an official “signature.”

In this chapter, we will discuss digital methods which make it difficult to “forge” a “signature.” Just as with encryption, there are two different approaches to the problem based on whether private keys are allowed: *message authentication codes* and *digital signatures*. Message Authentication Codes (MACs) are used in the private key setting. Only people who know the secret key can check if a message is valid. Digital Signatures extend this idea to the public key setting. Anyone who knows the public key of Alice can verify a signature issued by Alice, and only those who know the secret key can issue signatures.

## 5.2 Message Authentication Codes

**Definition 91.1** (MAC).  $(\text{Gen}, \text{Tag}, \text{Ver})$  is a MAC over the message space  $\{\mathcal{M}\}_n$  if the following hold:

- $\text{Gen}$  is a p.p.t. algorithm that returns a key  $k \leftarrow \text{Gen}(1^n)$ .
- $\text{Tag}$  is a p.p.t. algorithm that on input key  $k$  and message  $m$  outputs a tag  $\sigma \leftarrow \text{Tag}_k(m)$ .
- $\text{Ver}$  is a deterministic polynomial-time algorithm that on input  $k, m$  and  $\sigma$  outputs “accept” or “reject”.
- For all  $n \in N$ , for all  $m \in \mathcal{M}_n$ ,

$$\Pr[k \leftarrow \text{Gen}(1^n) : \text{Ver}_k(m, \text{Tag}_k(m)) = \text{“accept”}] = 1$$

The above definition requires that verification algorithms always correctly “accepts” a valid signature.

The goal of an adversary is to forge a MAC. In this case, the adversary is said to forge a MAC if it is able to construct a tag  $\sigma'$  such that it is a valid signature for some message. We could consider many different adversaries with varying powers depending on whether the adversary has access to signed messages; whether the adversary has access to a signing oracle; and whether the adversary can pick the message to be forged. The strongest adversary is the one who has oracle access to  $\text{Tag}$  and is allowed to forge any chosen message.

**Definition 92.1** (Security of a MAC). A message authentication code  $(\text{Gen}, \text{Tag}, \text{Ver})$  is *secure* if for all non-uniform p.p.t. adversaries  $A$ , there exists a negligible function  $\epsilon(n)$  such that for all  $n$ ,

$$\Pr[k \leftarrow \text{Gen}(1^n); m, \sigma \leftarrow A^{\text{Tag}_k(\cdot)}(1^n) : \\ A \text{ did not query } m \wedge \text{Ver}_k(m, \sigma) = \text{“accept”}] \leq \epsilon(n)$$

We now show a construction of a MAC using pseudorandom functions.

---

### CONSTRUCTION 92.1: MAC SCHEME

---

Let  $F = \{f_s\}$  be a family of pseudorandom functions such that  $f_s : \{0, 1\}^{|s|} \rightarrow \{0, 1\}^{|s|}$ .

$$\text{Gen}(1^n): k \leftarrow \{0, 1\}^n$$

$$\text{Tag}_k(m): \text{Output } f_k(m)$$

$$\text{Ver}_k(m, \sigma): \text{Output "accept" if and only if } f_k(m) = \sigma.$$

**Theorem 93.1.** *If there exists a pseudorandom function, then the above scheme is a Message Authentication Code over the message space  $\{0, 1\}_{n \in N}$ .*

*Proof.* (Sketch) Consider the above scheme when a random function  $RF$  is used instead of the pseudorandom function  $F$ . In this case,  $A$  succeeds with a probability at most  $2^{-n}$ , since  $A$  only wins if  $A$  is able to guess the  $n$  bit random string which is the output of  $RF_k(m)$  for some new message  $m$ . From the security property of a pseudorandom function, there is no non uniform ppt distinguisher which can distinguish the output of  $F$  and  $RF$  with a non negligible probability. Hence, we conclude that  $(\text{Gen}, \text{Tag}, \text{Ver})$  is secure.  $\square$

### 5.3 Digital Signature Schemes

With message authentication codes, both the signer and verifier need to share a secret key. In contrast, digital signatures mirror real-life signatures in that anyone who knows Alice (but not necessarily her secrets) can verify a signature generated by Alice. Moreover, digital signatures possess the property of *non-repudiability*, i.e., if Alice signs a message and sends it to Bob, then Bob can prove to a third party (who also knows Alice) the validity of the signature. Hence, digital signatures can be used as certificates in a public key infrastructure.

**Definition 93.1** (Digital Signatures).  $(\text{Gen}, \text{Sign}, \text{Ver})$  is a *digital signature scheme* over the message space  $\{M_n\}_{n \in N}$  if

- $\text{Gen}(1^n)$  is a PPT which on input  $n$  outputs a public key  $pk$  and a secret key  $sk$ :  $pk, sk \leftarrow \text{Gen}(1^n)$ .
- $\text{Sign}$  is a PPT which on input a secret key  $sk$  and message  $m$  outputs a signature  $\sigma$ :  $\sigma \leftarrow \text{Sign}_{sk}(m)$ .
- $\text{Ver}$  is a deterministic p.t. algorithm which on input a public key  $pk$ , a message  $m$  and a signature  $\sigma$  returns either “accept” or “reject”.

- For all  $n \in N$ , for all  $m \in \mathcal{M}_n$ ,

$$\Pr[pk, sk \leftarrow \text{Gen}(1^n) : \text{Ver}_{pk}(m, \text{Sign}_{sk}(m)) = \text{“accept”}] = 1$$

The security of a digital signature can be defined in terms very similar to the security of a MAC. The adversary can make a polynomial number of queries to a signing oracle. It is not considered a forgery if the adversary  $A$  produces a signature on a message  $m$  on which it has queried the signing oracle. Note that by definition of a public key infrastructure, the adversary has free oracle access to the verification algorithm  $\text{Ver}_{pk}$ .

**Definition 94.1.** (Security of Digital Signatures).  $(\text{Gen}, \text{Sign}, \text{Ver})$  is secure if for all non-uniform p.p.t. adversaries  $A$ , there exists a negligible function  $\epsilon(n)$  such that  $\forall n \in N$ ,

$$\Pr[pk, sk \leftarrow \text{Gen}(1^n); m, \sigma \leftarrow A^{\text{Sign}_{sk}(\cdot)}(1^n) : A \text{ did not query } m \wedge \text{Ver}_{pk}(m, \sigma) = \text{“accept”}] \leq \epsilon(n)$$

In contrast, a digital signature scheme is said to be *one-time secure* if Definition 5.3 is satisfied under the constraint that the adversary  $A$  is only allowed to query the signing oracle *once*. In general, however, we need a digital signature scheme to be many-message secure. The construction of the one-time secure scheme, however, gives insight into the more general construction.

## 5.4 A One-Time Digital Signature Scheme for $\{0, 1\}^n$

To produce a many-message secure digital signature scheme, we first describe a digital signature scheme and prove that it is one-time secure for  $n$ -bit messages. We then extend the scheme to handle arbitrarily long messages. Finally, we take that scheme and show how to make it many-message secure.

Our one-time secure digital signature scheme is a triple  $(\text{Gen}, \text{Sign}, \text{Ver})$ .  $\text{Gen}$  produces a secret key consisting of  $2n$  random elements and a public key consisting of the image of the same  $2n$  elements under a one-way function  $f$ .

---

### CONSTRUCTION 94.1: ONE-TIME DIGITAL SIGNATURE SCHEME

---

$\text{Gen}(1^n)$ : For  $i = 1$  to  $n$ , and  $b = 0, 1$ , pick  $x_b^i \leftarrow U_n$ . Output the keys:

$$\text{sk} = \begin{pmatrix} x_0^1 & x_0^2 & \cdots & x_0^n \\ x_1^1 & x_1^2 & \cdots & x_1^n \end{pmatrix}$$

$$\text{pk} = \begin{pmatrix} f(x_0^1) & f(x_0^2) & \cdots & f(x_0^n) \\ f(x_1^1) & f(x_1^2) & \cdots & f(x_1^n) \end{pmatrix}$$

$\text{Sign}_{sk}(m)$ : For  $i = 1$  to  $n$ ,  $\sigma_i \leftarrow x_{m_i}^i$ . Output  $\sigma = (\sigma_1, \dots, \sigma_n)$ .

$\text{Ver}_{pk}(\sigma, m)$ : Output accept if and only if  $f(\sigma_i) = f(x_{m_i}^i)$  for all  $i \in [1, n]$ .

For example, to sign the message  $m = 010$ ,  $\text{Sign}_{sk}(m)$  returns  $x_0^1, x_1^2, x_0^3$ . From these definitions, it is immediately clear that (Gen, Sign, Ver) is a digital signature scheme. However, this signature scheme is not many-message secure because after two signature queries (on say, the message  $0 \dots 0$  and  $1 \dots 1$ ), it is possible to forge a signature on any message.

Nonetheless, the scheme is one-time secure. The intuition behind the proof is as follows. If after one signature query on message  $m$ , if  $A$  produces a pair  $m', \sigma'$  that satisfies  $\text{Ver}_{sk}(m', \sigma') = \text{accept}$  and  $m \neq m'$ , then  $A$  must be able to invert  $f$  on a new point. Thus  $A$  has broken the one-way function  $f$ .

**Theorem 95.1.** *If  $f$  is a one-way function, then (Gen, Sign, Ver) is one-time secure.*

*Proof.* By contradiction. Suppose  $f$  is a one-way function, and suppose we are given an adversary  $A$  that succeeds with probability  $\epsilon(n)$  in breaking the one-time signature scheme. We construct a new adversary  $B$  that inverts  $f$  with probability  $\frac{\epsilon(n)}{\text{poly}(n)}$ .

$B$  is required to invert a one-way function  $f$ , so it is given a string  $y$  and access to  $f$ , and needs to find  $f^{-1}(y)$ . The intuition behind the construction of  $B$  is that  $A$  on a given instance of (Gen, Sign, Ver) will produce at least one value in its output that is the inverse of  $f(x_j^i)$  for some  $x_j^i$  not known to  $A$ . Thus, if  $B$  creates an instance of (Gen, Sign, Ver) and replaces one of the  $f(x_j^i)$  with  $y$ , then there is some non-negligible probability that  $A$  will succeed in inverting it, thereby inverting the one-way function.

Let  $m$  and  $m'$  be the two messages chosen by  $A$  ( $m$  is  $A$ 's request to the signing oracle, and  $m'$  is in  $A$ 's output). If  $m$  and  $m'$  were always going to differ in a given position, then it would be easy to decide where to put  $y$ . Instead,  $B$  generates an instance of (Gen, Sign, Ver) using  $f$  and replaces one of the values in  $\text{pk}$  with  $y$ . With some probability,  $A$  will choose a pair  $m, m'$  that differ in the position  $B$  chose for  $y$ .  $B$  proceeds as follows:

- Pick a random  $i \in \{1, \dots, n\}$  and  $c \in \{0, 1\}$
- Generate  $\text{pk}, \text{sk}$  using  $f$  and replace  $f(x_c^i)$  with  $y$
- Internally run  $m', \sigma' \leftarrow A(\text{pk}, 1^n)$

- $A$  may make a query  $m$  to the signing oracle.  $B$  answers this query if  $m_i$  is  $1 - c$ , and otherwise aborts (since  $B$  does not know the inverse of  $y$ )
- if  $m'_i = c$ , output  $\sigma'_i$ , and otherwise output  $\perp$

To find the probability that  $B$  is successful, first consider the probability that  $B$  aborts while running  $A$  internally; this can only occur if  $A$ 's query  $m$  contains  $c$  in the  $i$ th bit, so the probability is  $\frac{1}{2}$ . This probability follows because  $B$ 's choice of  $c$  is independent of  $A$ 's choice of  $m$  ( $A$  cannot determine where  $B$  put  $y$ , since all the elements of  $\text{pk}$ , including  $y$ , are the result of applications of  $f$  to a random value). The probability that  $B$  chose a bit that differs between  $m$  and  $m'$  is greater than  $\frac{1}{n}$  (since there must be at least one such bit), and  $A$  succeeds with probability  $\epsilon$ .

Thus  $B$  returns  $f^{-1}(y) = \sigma'_i$  and succeeds with probability greater than  $\frac{\epsilon}{2n}$ . The security of  $f$  implies that  $\epsilon(n)$  must be negligible, which implies that  $(\text{Gen}, \text{Sign}, \text{Ver})$  is one-time secure.  $\square$

Now, we would like to sign longer messages with the same length key. To do so, we will need a new tool: collision-resistant hash functions.

## 5.5 Collision-Resistant Hash Functions

Intuitively, a hash function is a function  $h(x) = y$  such that the representation of  $y$  is smaller than the representation of  $x$ , so  $h$  compresses  $x$ . The output of hash function  $h$  on a value  $x$  is often called the *hash* of  $x$ . Hash functions have a number of useful applications in data structures. For example, the Java programming language provides a built-in method that maps any string to a number in  $[0, 2^{32})$ . The following simple program computes the hash for a given string.

```
public class Hash {

    public void main(String args[]) {
        System.out.println( args[0].hashCode() );
    }
}
```

By inspecting the Java library, one can see that when run on a string  $s$ , the `hashCode` function computes and returns the value

$$T = \sum_i s[i] \cdot 31^{n-i}$$

where  $n$  is the length of the string and  $s[i]$  is the  $i$ th character of  $s$ . This function has a number of positive qualities: it is easy to compute, and it is  $n$ -wise independent on strings of length  $n$ . Thus, when used to store strings in a hash table, it performs very well.

For a hash function to be cryptographically useful, however, we require that it be hard to find two elements  $x$  and  $x'$  such that  $h(x) = h(x')$ . Such a pair is called a *collision*, and hash functions for which it is hard to find collisions are said to satisfy *collision resistance* or are said to be *collision-resistant*. Before we formalize collision resistance, we should note why it is useful: rather than signing a message  $m$ , we will sign the hash of  $m$ . Then even if an adversary  $A$  can find another signature  $\sigma$  on some bit string  $y$ ,  $A$  will not be able to find any  $x$  such that  $h(x) = y$ , so  $A$  will not be able to find a message that has signature  $\sigma$ . Further, given the signature of some message  $m$ ,  $A$  will not be able to find an  $m'$  that has  $h(m) = h(m')$  (if  $A$  could find such an  $m'$ , then  $m$  and  $m'$  would have the same signature).

With this in mind, it is easy to see that the Java hash function does not work well as a cryptographic hash function. For example, it is very easy to change the last two digits of a string to make a collision. (This is because the contribution of the last two symbols to the output is  $31 * s[n-1] + s[n]$ . One can easily find two pairs of symbols which contribute the same value here, and therefore when pre-pended with the same prefix, result in the same hash.)

### A Family of Collision-Resistant Hash Functions

It is not possible to guarantee collision resistance against a non-uniform adversary for a single hash function  $h$ : since  $h$  compresses its input, there certainly exist two inputs  $x$  and  $x'$  that comprise a collision. Thus, a non-uniform adversary can have  $x$  and  $x'$  hard-wired into their circuits. To get around this issue, we must introduce a family of collision-resistant hash functions.

**Definition 97.1.** A set of functions  $H = \{h_i : D_i \rightarrow R_i\}_{i \in I}$  is a *family of collision-resistant hash functions* (CRH) if:

- (ease of sampling) Gen runs in PPT:  $\text{Gen}(1^n) \in I$
- (compression)  $|R_i| < |D_i|$
- (ease of evaluation) Given  $x, i \in I$ , the computation of  $h_i(x)$  can be done in PPT

- (collision resistance)  $\forall$  non-uniform PPT  $A$ .  $\exists$  negligible  $\epsilon$  such that  $\forall n \in \mathbb{N}$ .

$$\Pr[i \leftarrow \text{Gen}(1^n); x, x' \leftarrow A(1^n, i) : h_i(x) = h_i(x') \wedge x \neq x'] \leq \epsilon(n)$$

Note that compression is a relatively weak property and does not even guarantee that the output is compressed by one bit. In practice, we often require that  $|h(x)| < \frac{|x|}{2}$ . Also note that if  $h$  is collision-resistant, then  $h$  is one-way.<sup>1</sup>

### Attacks on CRHFs

Collision-resistance is a stronger property than one-wayness, so finding an attack on a collision-resistant hash functions is easier than finding an attack on a one-way function. We now consider some possible attacks.

**Enumeration.** If  $|D_i| = 2^d$ ,  $|R_i| = 2^n$ , and  $x, x'$  are chosen at random, what is the probability of a collision between  $h(x)$  and  $h(x')$ ?

In order to analyze this situation, we must count the number of ways that a collision can occur. Let  $p_y$  be the probability that  $h$  maps a element from the domain into  $y \in R_i$ . The probability of a collision at  $y$  is therefore  $p_y^2$ . Since a collision can occur at either  $y_1$  or  $y_2$ , etc., the probability of a collision can be written as

$$\Pr[\text{collision}] = \sum_{y \in R_i} p_y^2$$

Since  $\sum_{y \in R_i} p_y = 1$ , by the Cauchy-Schwartz Theorem ??, we have that

$$\sum_{y \in R_i} p_y^2 > \frac{1}{|R_i|}$$

The probability that  $x$  and  $x'$  are not identical is  $\frac{1}{|D_i|}$ . Combining these two shows that the total probability of a collision is greater than  $\frac{1}{2^n} - \frac{1}{2^d}$ . In other words, enumeration requires searching most of the range to find a collision.

---

<sup>1</sup>The question of how to construct a CRH from a one-way permutation, however, is still open. There is a weaker kind of hash function: the universal one-way hash function (UOWF). A UOWF satisfies the property that it is hard to find a collision for a particular message; a UOWF can be constructed from a one-way permutation.

**Birthday attack.** Instead enumerating pairs of values, consider a set of random values  $x_1, \dots, x_t$ . Evaluate  $h$  on each  $x_i$  and look for a collision between any pair  $x_i$  and  $x_{i'}$ . By the linearity of expectations, the expected number of collisions is the number of pairs multiplied by the probability that a random pair collides. This probability is

$$\binom{t}{2} \left( \frac{1}{|R_i|} \right) \approx \frac{t^2}{|R_i|}$$

so  $O(\sqrt{|R_i|}) = O(2^{n/2})$  samples are needed to find a collision with good probability. In other words, the birthday attack only requires the attacker to do computation on the order of the square root of the size of the output space.<sup>2</sup> This attack is much more efficient than the best known attacks on one-way functions, since those attacks require enumeration.

Now, we would like to show that, given some standard cryptographic assumptions, we can produce a CRH that compresses by one bit. Given such a CRH, we can then construct a CRH that compresses more.<sup>3</sup>

---

CONSTRUCTION 99.1: COLLISION RESISTANT HASH FUNCTION

---

**Gen( $1^n$ ):** Outputs a triple  $(g, p, y)$  such that  $p$  is an  $n$ -bit prime,  $g$  is a generator for  $\mathbb{Z}_p^*$ , and  $y$  is a random element in  $\mathbb{Z}_p^*$ .

**$h_{p,g,y}(x, b)$ :** Given any  $n$ -bit string  $x$  and bit  $b$ ,

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

**Theorem 99.1.** *Under the Discrete Logarithm assumption, construction 5.5 is a collision-resistant hash function that compresses by 1 bit.*

*Proof.* Notice that both Gen and  $h$  are efficiently computable, and  $h$  compresses by one bit (since the input is in  $\mathbb{Z}_p^* \times \{0, 1\}$  and the output is in  $\mathbb{Z}_p^*$ ). We need to prove that if we could find a collision, then we could also find the discrete logarithm of  $y$ .

---

<sup>2</sup>This attack gets its name from the *birthday paradox*, which uses a similar analysis to show that with 23 randomly chosen people, the probability of two of them having the same birthday is greater than 50%.

<sup>3</sup>Suppose that  $h$  is a hash function that compresses by one bit. Note that the naïve algorithm that applies  $h$   $k$  times to an  $n + k$  bit string is not secure, although it compresses by more than 1 bit, because in this case  $m$  and  $h(m)$  both hash to the same value.

To do so, suppose that  $A$  finds a collision with non-negligible probability  $\epsilon$ . We construct a  $B$  that finds the discrete logarithm also with probability  $\epsilon$ .

Note first that if  $h_i(x, b) = h_i(x', b)$ , then  $y^b g^x \bmod p = y^b g^{x'} \bmod p$ , so  $g^x \bmod p = g^{x'} \bmod p$ , so  $x = x'$ .

So, for a collision to occur, one value of the second parameter of  $h$  must be 1 and the other value of the second parameter of  $h$  must be 0. That is, for any collision  $(x, b) = (x', b')$ , it holds that  $b \neq b'$ . Without loss of generality, assume that  $b = 0$ . Then,

$$g^x = y g^{x'} \bmod p$$

so

$$y = g^{x-x'} \bmod p$$

which allows  $B$  to find the discrete logarithm of  $y$ .  $B(p, g, y)$  calls  $A(p, g, y) \rightarrow (x, b), (x', b')$ . If  $b = 0$ , then  $B$  returns  $x - x'$ , and otherwise it returns  $x' - x$ .  $\square$

Thus we have constructed a CRH that compresses by one bit. Note further that this reduction is actually an algorithm for computing the discrete logarithm that is better than brute force: since the Birthday Attack on a CRH only requires searching  $2^{k/2}$  keys rather than  $2^k$ , the same attack works on the discrete logarithm by applying the above algorithm each time. Of course, there are much better (even deterministic) attacks on the discrete logarithm problem.<sup>4</sup>

### Multiple-bit Compression

Given a CRHF function that compresses by one bit, it is possible to construct a CRHF function that compresses by polynomially-many bits. The idea is to apply the simple one-bit function repeatedly.

## 5.6 One-Time Secure Digital Signature Schemes

We now use a family of Collision-Resistant Hash Functions (CRHFs) to construct a one-time signature scheme for messages in  $\{0, 1\}^*$ . Digital signature schemes that operate on the hash of a message are said to be in the *hash-and-sign* paradigm.

---

<sup>4</sup>Note that there is also a way to construct a CRH from the Factoring assumption:

$$h_{N,y}(x, b) = y^b x^2 \bmod N$$

Here, however, there is a trivial collision if we do not restrict the domain:  $x$  and  $-x$  map to the same value. For instance, we might take only the first half of the values in  $\mathbb{Z}_p^*$ .

**Theorem 100.1.** *If there exists a CRH from  $\{0, 1\}^* \rightarrow \{0, 1\}^n$  and there exists a one-way function (OWF), then there exists a one-time secure digital signature scheme for  $\{0, 1\}^*$ .*

We define a new one-time secure digital signature scheme  $(\text{Gen}', \text{Sign}', \text{Ver}')$  for  $\{0, 1\}^*$  by

---

CONSTRUCTION 101.1: ONE-TIME DIGITAL SIGNATURE FOR  $\{0, 1\}^*$

---

$\text{Gen}'(1^n)$ : Run the generator  $(pk, sk) \leftarrow \text{Gen}_{\text{Sig}}(1^n)$  and sampling function  $i \leftarrow \text{Gen}_{\text{CRH}}(1^n)$ . Output  $pk' = (pk, i)$  and  $sk' = (sk, i)$ .

$\text{Sign}'_{sk'}(m)$ : Sign the hash of message  $m$ : output  $\text{Sign}_{sk}(h_i(m))$ .

$\text{Ver}'_{pk'}(\sigma, m)$ : Verify  $\sigma$  on the hash of  $m$ : Output  $\text{Ver}_{pk}(h_i(m), \sigma)$

---

*Proof.* We will only provide a sketch of the proof here.

Let  $\{h_i\}_{i \in I}$  be a CRH with sampling function  $\text{Gen}_{\text{CRH}}(1^n)$ , and let  $(\text{Gen}_{\text{Sig}}, \text{Sign}, \text{Ver})$  be a one-time secure digital signature scheme for  $\{0, 1\}^n$  (as constructed in the previous sections.)

Now suppose that there is a PPT adversary  $A$  that breaks  $(\text{Gen}', \text{Sign}', \text{Ver}')$  with non-negligible probability  $\epsilon$  after only one oracle call  $m$  to  $\text{Sign}'$ . To break this digital signature scheme,  $A$  must output  $m' \neq m$  and  $\sigma'$  such that  $\text{Ver}'_{pk'}(m', \sigma') = \text{accept}$  (so  $\text{Ver}_{pk}(h_i(m'), \sigma') = \text{accept}$ ). There are only two possible cases:

1.  $h(m) = h(m')$ .

In this case,  $A$  found a collision  $(m, m')$  in  $h_i$ , which is known to be hard, since  $h_i$  is a member of a CRH.

2.  $A$  never made any oracle calls, or  $h(m) \neq h(m')$ .

Either way, in this case,  $A$  obtained a signature  $\sigma'$  using  $(\text{Gen}, \text{Sign}, \text{Ver})$  to a new message  $h(m')$ . But obtaining such a signature violates the assumption that  $(\text{Gen}, \text{Sign}, \text{Ver})$  is a one-time secure digital signature scheme.

To make this argument more formal, turn the two cases above into two adversaries  $B$  and  $C$ . Adversary  $B$  tries to invert a hash function from the CRH, and  $C$  tries to break the digital signature scheme.

$B(1^n, i)$  operates as follows to find a collision for  $h_i$ .

- Generate keys  $pk, sk \leftarrow \text{Gen}_{\text{Sig}}(1^n)$
- Call  $A$  to get  $m', \sigma' \leftarrow A^{\text{Sign}_{sk}(h_i(\cdot))}(1^n, (pk, i))$ .
- Output  $m, m'$  where  $m$  is the query made by  $A$  (if  $A$  made no query, then abort).

$C^{\text{Sign}_{sk}(\cdot)}(1^n, pk)$  operates as follows to break the one-time security of  $(\text{Gen}, \text{Sign}, \text{Ver})$ .

- Generate index  $i \leftarrow \text{Gen}_{\text{CRH}}(1^n)$
- Call  $A$  to get  $m', \sigma' \leftarrow A(1^n, (pk, i))$ 
  - When  $A$  calls  $\text{Sign}'_{(sk, i)}(m)$ , query signing oracle  $\text{Sign}_{sk}(h_i(m))$
- Output  $h_i(m'), \sigma'$ .

So, if  $A$  succeeds with non-negligible probability, then either  $B$  or  $C$  must succeed with non-negligible probability.  $\square$

## 5.7 Signing Many Messages

Now that we have extended one-time signatures on  $\{0, 1\}^n$  to operate on  $\{0, 1\}^*$ , we turn to increasing the number of messages that can be signed. The main idea is to generate new keys for each new message to be signed. Then we can still use our one-time secure digital signature scheme  $(\text{Gen}, \text{Sign}, \text{Ver})$ . The disadvantage is that the signer must keep state to know which key to use and what to include in a given signature.

We start with a pair  $(pk_0, sk_0) \leftarrow \text{Gen}(1^n)$ . To sign the first message  $m_1$ , we perform the following steps:

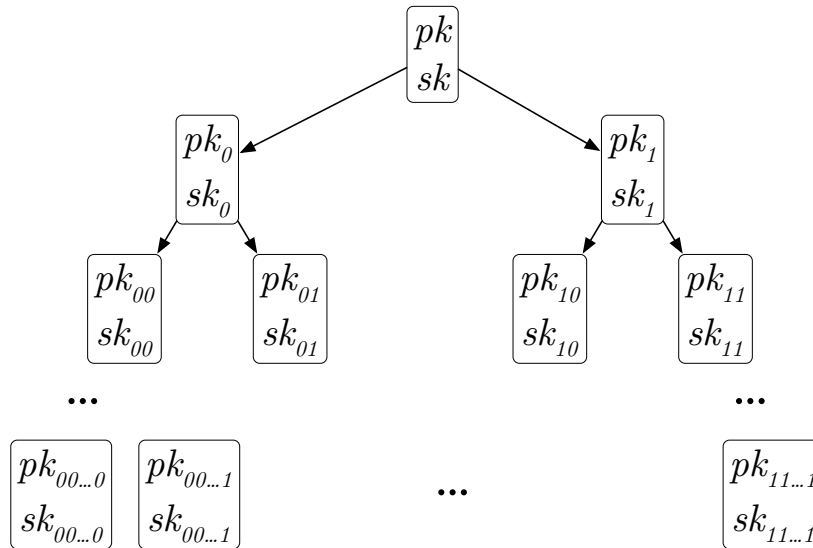
- Generate a new key pair for the next message:  $pk_1, sk_1 \leftarrow \text{Gen}(1^n)$
- Create signature  $\sigma_1 = \text{Sign}_{sk_0}(m_1 \parallel pk_1)$  on the concatenation of message  $m_1$  and new public key  $pk_1$ .
- Output  $\sigma'_1 = (1, \sigma_1, m_1, pk_1)$

Thus, each signature attests to the next public key. Similarly, to sign second message  $m_2$ , we generate  $pk_2, sk_2 \leftarrow \text{Gen}(1^n)$ , set  $\sigma_2 = \text{Sign}_{sk_1}(m_2 \parallel pk_2)$ , and output  $\sigma'_2 = (2, \sigma_2, \sigma'_1, m_2, pk_2)$ . Notice that we need to include  $\sigma'_1$  (the previous signature) to show that the previous public key is correct. These signatures satisfy many-message security, but the signer must keep state, and signature size grows linearly in the number of signatures ever performed by

the signer. Proving that this digital signature scheme is many-message secure is left as an exercise. We now focus on how to improve this basic idea by keeping the size of the signature constant.

### Improving the Construction

A simple way to improve this many-message secure digital signature scheme is to attest to two new key pairs instead of one at each step. This new construction builds a balanced binary tree of depth  $n$  of key pairs, where each node and leaf in the tree is associated with one public-private key pair  $pk, sk$ , and each non-leaf node public key is used to attest to its two child nodes. Each of the  $2^n$  leaf nodes can be used to attest to a message. Such a digital signature algorithm can perform up to  $2^n$  signatures with signature size  $n$  (the size follows because a signature using a particular key pair  $pk_i, sk_i$  must provide signatures attesting to each key pair on the path from  $pk_i, sk_i$  to the root). The tree looks as follows.



To sign the first message  $m$ , the signer generates and stores  $pk_0, sk_0, pk_{00}, sk_{00}, \dots, pk_{0^n}, sk_{0^n}$  along with all of their siblings in the tree. Then  $pk_0$  and  $pk_1$  are signed with  $sk$ , producing signature  $\sigma_0$ ,  $pk_{00}$  and  $pk_{01}$  are signed with  $sk_0$ , producing signature  $\sigma_1$ , and so on. Finally, the signer returns the signature

$$\sigma = (pk, \sigma_0, pk_0, \sigma_1, pk_{00}, \dots, \sigma_{n-1}, pk_{0^n}, \text{Sign}_{sk_{0^n}}(m))$$

as a signature for  $m$ . The verification function  $\text{Ver}$  then uses  $pk$  to check that  $\sigma_0$  attests for  $pk_0$ , uses  $pk_0$  to check that  $\sigma_1$  attests for  $pk_{00}$ , and so on up to  $pk_{0^n}$ , which is used to check that  $\text{Sign}_{sk_{0^n}}(m)$  is a correct signature for  $m$ .

For an arbitrary message, the next unused leaf node in the tree is chosen, and any needed signatures attesting to the path from that leaf to the root are generated (some of these signatures will have been generated previously). Then the leaf node key is used to sign the message in the same manner as above

Proving that this scheme is many-message secure is left as an exercise for the student. The key idea is that fact that  $(\text{Gen}, \text{Sign}, \text{Ver})$  is one-time secure, and each signature is only used once. Thus, forging a signature in this scheme requires creating a second signature.

For all its theoretical value, however, this many-message secure digital signature scheme still requires the signer to keep a significant amount of state. The state kept by the signer is

- The number of messages signed
  - To remove this requirement, we will assume that messages consist of at most  $n$  bits. Then, instead of using the leaf nodes as key pairs in increasing order, use the  $n$ -bit representation of  $m$  to decide which leaf to use. That is, use  $\text{pk}_m, \text{sk}_m$  to sign  $m$ .
- All previously generated keys
- All previously generated signatures (for the authentication paths to the root)

We can remove the requirement that the signer remembers the previous keys and previous signatures if we have a pseudo-random function to regenerate all of this information on demand. In particular, we generate a public key  $\text{pk}$  and secret key  $\text{sk}'$ . The secret key, in addition to containing the secret key  $\text{sk}$  corresponding to  $\text{pk}$ , also contains two seeds  $s_1$  and  $s_2$  for two pseudo-random functions  $f$  and  $g$ . We then generate  $\text{pk}_i$  and  $\text{sk}_i$  for node  $i$  by using  $f_{s_1}(i)$  as the randomness in the generation algorithm  $\text{Gen}(1^n)$ . Similarly, we generate any needed randomness for the signing algorithm on message  $m$  with  $g_{s_2}(m)$ . Then we can regenerate any path through the tree on demand without maintaining any of the tree as state at the signer.

## 5.8 Intuition for Constructing Efficient Digital Signature

Consider the following method for constructing a digital signature scheme from a trapdoor permutation: