7

Hash Functions

7.1 Signatures and Hash Functions

The reader might have noticed that the signature schemes described in Chapter 6 allow only "small" messages to be signed. For example, when using the **DSS**, a 160-bit message is signed with a 320-bit signature. In general, we will want to sign much longer messages. A legal document, for example, might be many megabytes in size.

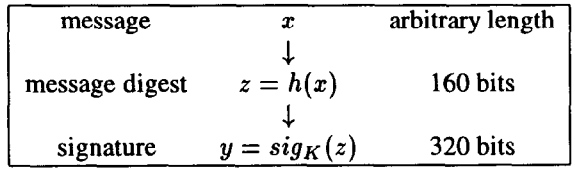
A naive attempt to solve this problem would be to break a long message into 160-bit chunks, and then to sign each chunk independently. This is analogous to encrypting a long string of plaintext by encrypting each plaintext character independently using the same key (e.g., ECB mode in the **DES**).

But there are several problems with this approach in creating digital signatures. First of all, for a long message, we will end up with an enormous signature (twice as long as the original message in the case of the **DSS**). Another disadvantage is that most "secure" signature schemes are slow since they typically use complicated arithmetic operations such as modular exponentiation. But an even more serious problem with this approach is that the various chunks of a signed message could be rearranged, or some of them removed, and the resulting message would still be verified. We need to protect the integrity of the entire message, and this cannot be accomplished by independently signing little pieces of it.

The solution to all of these problems is to use a very fast public *cryptographic hash function*, which will take a message of arbitrary length and produce a *message digest* of a specified size (160 bits if the **DSS** is to be used). The message digest will then be signed. For the **DSS**, the use of a hash function h is depicted diagramatically in Figure 7.1

When Bob wants to sign a message x, he first constructs the message digest $z = h(x)$, and then computes the signature $y = sig_K(z)$. He transmits the ordered pair (x, y) over the channel. Now the verification can be performed (by anyone) by first reconstructing the message digest $z = h(x)$ using the public hash function h, and then checking that $ver_K(z, y) = \text{true}$.

FIGURE 7.1
Signing a message digest



7.2 Collision-free Hash Functions

We have to be careful that the use of a hash function h does not weaken the security of the signature scheme, for it is the message digest that is signed, not the message. It will be necessary for h to satisfy certain properties in order to prevent various forgeries.

The most obvious type of attack is for an opponent, Oscar, to start with a valid signed message (x, y), where $y = sig_K(h(x))$. (The pair (x, y) could be any message previously signed by Bob.) Then he computes $z = h(x)$ and attempts to find $x' \neq x$ such that $h(x') = h(x)$. If Oscar can do this, (x', y) would be a valid signed message, i.e., a *forgery*. In order to prevent this type of attack, we require that h satisfy the following collision-free property:

DEFINITION 7.1 *A hash function h is **weakly collision-free** if, given a message x, it is computationally infeasible to find a message $x' \neq x$ such that $h(x') = h(x)$.*

Another possible attack is the following: Oscar first finds two messages $x \neq x'$ such that $h(x) = h(x')$. Oscar then gives x to Bob and persuades him to sign the message digest $h(x)$, obtaining y. Then (x', y) is a valid forgery.

This motivates a different collision-free property:

DEFINITION 7.2 *A hash function h is **strongly collision-free** if it is computationally infeasible to find messages x and x' such that $x' \neq x$ and $h(x') = h(x)$.*

Observe that strongly collision-free implies weakly collision-free.

Here is a third variety of attack. As we mentioned in Section 6.2, it is often possible with certain signature schemes to forge signatures on random message digests z. Suppose Oscar computes a signature on such a random z, and then he finds a message x such that $z = h(x)$. If he can do this, then (x, y) is a valid forgery. To prevent this attack, we desire that h satisfy the same one-way property that was mentioned previously in the context of public-key cryptosystems and the **Lamport Signature Scheme**:

FIGURE 7.2
Using an inversion algorithm A to find collisions for a hash function h

1. choose a random $x \in X$
2. compute $z = h(x)$
3. compute $x_1 = \mathbf{A}(z)$
4. **if** $x_1 \neq x$ **then**
   x_1 and x collide under h (success)
   else
   QUIT (failure).

DEFINITION 7.3 *A hash function h is* ***one-way*** *if, given a message digest z, it is computationally infeasible to find a message x such that $h(x) = z$.*

We are now going to prove that the strongly collision-free property implies the one-way property. This is done by proving the contrapositive statement. More specifically, we will prove that an arbitrary inversion algorithm for a hash function can be used as an oracle in a Las Vegas probabilistic algorithm that finds collisions.

This reduction can be accomplished with a fairly weak assumption on the relative sizes of the domain and range of the hash function. We will assume for the time being that the hash function $h : X \rightarrow Z$, where X and Z are finite sets and $|X| \geq 2|Z|$. This is a reasonable assumption: If we think of an element of X as being encoded as a bitstring of length $\log_2 |X|$ and an element of Z as being encoded as a bitstring of length $\log_2 |Z|$, then the message digest $z = h(x)$ is at least one bit shorter than the message x. (Eventually, we will be interested in the situation where the message domain X is infinite, since we want to be able to deal with messages of arbitrary length. Our argument also applies in this situation.)

We are assuming that we have an inversion algorithm for h. That is, we have an algorithm $\mathbf{A}$ which accepts as input a message digest $z \in Z$, and finds an element $\mathbf{A}(z) \in X$ such that $h(\mathbf{A}(z)) = z$.

We prove the following theorem.

THEOREM 7.1
Suppose $h : X \rightarrow Z$ is a hash function where $|X|$ and $|Z|$ are finite and $|X| \geq 2|Z|$. Suppose $\mathbf{A}$ is an inversion algorithm for h. Then there exists a probabilistic Las Vegas algorithm which finds a collision for h with probability at least $1/2$.

PROOF Consider the algorithm $\mathbf{B}$ presented in Figure 7.2. Clearly $\mathbf{B}$ is a probabilistic algorithm of the Las Vegas type, since it either finds a collision or

returns no answer. Thus our main task is to compute the probability of success. For any $x \in X$, define $x \sim x_1$ if $h(x) = h(x_1)$. It is easy to see that $\sim$ is an equivalence relation. Define

$$[x] = \{x_1 \in X : x \sim x_1\}.$$

Each equivalence class $[x]$ consists of the inverse image of an element of Z, so the number of equivalence classes is at most $|Z|$. Denote the set of equivalence classes by C.

Now, suppose x is the element of X chosen in step 1. For this x, there are $|[x]|$ possible x_1's that could be returned in step 3. $|[x]| - 1$ of these x_1's are different from x and thus lead to success in step 4. (Note that the algorithm **A** does not know the representative of the equivalence class $[x]$ that was chosen in step 1.) So, given a particular choice $x \in X$, the probability of success is $(|[x]| - 1)/|[x]|$.

The probability of success of the algorithm **B** is computed by averaging over all possible choices for x:

$$\begin{aligned}
p(\text{success}) &= \frac{1}{|X|} \sum_{x \in X} \frac{|[x]| - 1}{|[x]|} \\
&= \frac{1}{|X|} \sum_{c \in C} \sum_{x \in c} \frac{|c| - 1}{|c|} \\
&= \frac{1}{|X|} \sum_{c \in C} (|c| - 1) \\
&= \frac{1}{|X|} \left(\sum_{c \in C} |c| - \sum_{c \in C} 1 \right) \\
&\geq \frac{|X| - |Z|}{|X|} \\
&\geq \frac{|X| - |X|/2}{|X|} \\
&= \frac{1}{2}.
\end{aligned}$$

Hence we have constructed a Las Vegas algorithm with success probability at least $1/2$. ∎

Hence, it is sufficient that a hash function satisfy the strongly collision-free property, since it implies the other two properties. So in the remainder of this chapter we restrict our attention to strongly collision-free hash functions.

7.3 The Birthday Attack

In this section, we determine a necessary security condition for hash functions that depends only on the cardinality of the set Z (equivalently, on the size of the message digest). This necessary condition results from a simple method of finding collisions which is informally known as the *birthday attack*. This terminology arises from the so-called *birthday paradox*, which says that in a group of 23 random people, at least two will share a birthday with probability at least $1/2$. (Of course this is not a paradox, but it is probably counter-intuitive). The reason for the terminology "birthday attack" will become clear as we progress.

As before, let us suppose that $h : X \to Z$ is a hash function, X and Z are finite, and $|X| \geq 2|Z|$. Denote $|X| = m$ and $|Z| = n$. It is not hard to see that there are at least n collisions — the question is how to find them. A very naive approach is to choose k random distinct elements $x_1, \ldots, x_k \in X$, compute $z_i = h(x_i)$, $1 \leq i \leq k$, and then determine if a collision has taken place (by sorting the z_i's, for example).

This process is analogous to throwing k balls randomly into n bins and then checking to see if some bin contains at least two balls. (The k balls correspond to the k random x_i's, and the n bins correspond to the n possible elements of Z.)

We will compute a lower bound on the probability of finding a collision by this method. This lower bound will depend on k and n, but not on m. Since we are interested in a lower bound on the collision probability, we will make the assumption that $|h^{-1}(z)| \approx m/n$ for all $z \in Z$. (This is a reasonable assumption: if the inverse images are not approximately equal, then the probability of finding a collision will increase.)

Since the inverse images are all (roughly) the same size and the x_i's are chosen at random, the resulting z_i's can be thought of as random (not necessarily distinct) elements of Z. But it is a simple matter to compute the probability that k random elements $z_1, \ldots, z_k \in Z$ are distinct. Consider the z_i's in the order $z_1, \ldots, z_k$. The first choice z_1 is arbitrary; the probability that $z_2 \neq z_1$ is $1 - 1/n$; the probability that z_3 is distinct from z_1 and z_2 is $1 - 2/n$, etc.

Hence, we estimate the probability of no collisions to be

$$\left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right) \ldots \left(1 - \frac{k-1}{n}\right) = \prod_{i=1}^{k-1}\left(1 - \frac{i}{n}\right).$$

If x is a small real number, then $1 - x \approx e^{-x}$. This estimate is derived by taking the first two terms of the series expansion

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \ldots.$$

Then our estimated probability of no collisions is

$$\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right) \approx \prod_{i=1}^{k-1} e^{\frac{-i}{n}}$$
$$= e^{\frac{-k(k-1)}{n}}.$$

So we estimate the probability of at least one collision to be

$$1 - e^{\frac{-k(k-1)}{n}}.$$

If we denote this probability by ϵ, then we can solve for k as a function of n and ϵ:

$$e^{\frac{-k(k-1)}{n}} \approx 1-\epsilon$$
$$\frac{-k(k-1)}{n} \approx \ln(1-\epsilon)$$
$$k^2 - k \approx n \ln \frac{1}{1-\epsilon}.$$

If we ignore the term $-k$, then we estimate

$$k \approx \sqrt{n \ln \frac{1}{1-\epsilon}}.$$

If we take $\epsilon = .5$, then our estimate is

$$k \approx 1.17\sqrt{n}.$$

So this says that hashing just over $\sqrt{n}$ random elements of X yields a collision with a probability of 50%. Note that a different choice of ϵ leads to a different constant factor, but k will still be proportional to $\sqrt{n}$.

If X is the set of all human beings, Y is the set of 365 days in a non-leap year (i.e., excluding February 29), and $h(x)$ denotes the birthday of person x, then we are dealing with the birthday paradox. Taking $n = 365$ in our estimate, we get $k \approx 22.3$. Hence, as mentioned earlier, there will be at least one duplicated birthday among 23 random people with probability at least $1/2$.

This birthday attack imposes a lower bound on the sizes of message digests. A 40-bit message digest would be very insecure, since a collision could be found with probability $1/2$ with just over 2^{20} (about a million) random hashes. It is usually suggested that the minimum acceptable size of a message digest is 128 bits (the birthday attack will require over 2^{64} hashes in this case). The choice of a 160-bit message digest for use in the **DSS** was undoubtedly motivated by these considerations.

FIGURE 7.3
Chaum-van Heijst-Pfitzmann Hash Function

Suppose p is a large prime and $q = (p-1)/2$ is also prime. Let α and β be two primitive elements of $\mathbb{Z}_p$. The value $\log_\alpha \beta$ is not public, and we assume that it is computationally infeasible to compute its value.
The hash function

$$h : \{0, \ldots, q-1\} \times \{0, \ldots, q-1\} \rightarrow \mathbb{Z}_p \backslash \{0\}$$

is defined as follows:

$$h(x_1, x_2) = \alpha^{x_1} \beta^{x_2} \bmod p.$$

7.4 A Discrete Log Hash Function

In this section, we describe a hash function, due to Chaum, van Heijst, and Pfitzmann, that will be secure provided a particular discrete logarithm cannot be computed. This hash function is not fast enough to be of practical use, but it is conceptually simple and provides a nice example of a hash function that can be proved secure under a reasonable computational assumption. The **Chaum-van Heijst-Pfitzmann Hash Function** is presented in Figure 7.3. We now prove a theorem concerning the security of this hash function.

THEOREM 7.2
Given one collision for the **Chaum-van Heijst-Pfitzmann Hash Function** *h, the discrete logarithm* $\log_\alpha \beta$ *can be computed efficiently.*

PROOF Suppose we are given a collision

$$h(x_1, x_2) = h(x_3, x_4),$$

where $(x_1, x_2) \neq (x_3, x_4)$. So we have the following congruence:

$$\alpha^{x_1} \beta^{x_2} \equiv \alpha^{x_3} \beta^{x_4} \pmod{p},$$

or

$$\alpha^{x_1 - x_3} \equiv \beta^{x_4 - x_2} \pmod{p}.$$

Denote

$$d = \gcd(x_4 - x_2, p-1).$$

Since $p - 1 = 2q$ and q is prime, it must be the case that $d \in \{1, 2, q, p-1\}$. Hence, we have four possibilities for d, which we will consider in turn.

First, suppose that $d = 1$. Then let

$$y = (x_4 - x_2)^{-1} \bmod (p-1).$$

We have that

$$\begin{aligned}\beta &\equiv \beta^{(x_4-x_2)y} \pmod p \\ &\equiv \alpha^{(x_1-x_3)y} \pmod p,\end{aligned}$$

so we can compute the discrete logarithm $\log_\alpha \beta$ as follows:

$$\log_\alpha \beta = (x_1 - x_3)(x_4 - x_2)^{-1} \bmod (p-1).$$

Next, suppose that $d = 2$. Since $p - 1 = 2q$ where q is odd, we must have $\gcd(x_4 - x_2, q) = 1$. Let

$$y = (x_4 - x_2)^{-1} \bmod q.$$

Now

$$(x_4 - x_2)y = kq + 1$$

for some integer k, so we have

$$\begin{aligned}\beta^{(x_4-x_2)y} &\equiv \beta^{kq+1} \pmod p \\ &\equiv (-1)^k \beta \pmod p \\ &\equiv \pm\beta \pmod p,\end{aligned}$$

since

$$\beta^q \equiv -1 \pmod p.$$

So we have

$$\begin{aligned}\alpha^{(x_4-x_2)y} &\equiv \beta^{(x_1-x_3)y} \pmod p \\ &\equiv \pm\beta \pmod p.\end{aligned}$$

It follows that

$$\log_\alpha \beta = (x_1 - x_3)y \bmod (p-1)$$

or

$$\log_\alpha \beta = (x_1 - x_3)y + q \bmod (p-1).$$

We can easily test which of these two possibilities is the correct one. Hence, as in the case $d = 1$, we have calculated the discrete logarithm $\log_\alpha \beta$.

The next possibility is that $d = q$. But

$$0 \le x_1 \le q - 1$$

and

$$0 \leq x_3 \leq q-1,$$

so

$$-(q-1) \leq x_4 - x_2 \leq q-1.$$

So it is impossible that $\gcd(x_4 - x_2, p-1) = q$; in other words, this case does not arise.

The final possibility is that $d = p-1$. This happens only if $x_2 = x_4$. But then we have

$$\alpha^{x_1}\beta^{x_2} \equiv \alpha^{x_3}\beta^{x_2} \pmod{p},$$

so

$$\alpha^{x_1} \equiv \alpha^{x_3} \pmod{p},$$

and $x_1 = x_3$. Thus $(x_1, x_2) = (x_3, x_4)$, a contradiction. So this case is not possible, either.

Since we have considered all possible values for d, we conclude that the hash function h is strongly collision-free provided that it is infeasible to compute the discrete logarithm $\log_\alpha \beta$ in $\mathbb{Z}_p$. ∎

We illustrate the result of the above theorem with an example.

Example 7.1

Suppose $p = 12347$ (so $q = 6173$), $\alpha = 2$ and $\beta = 8461$. Suppose we are given the collision

$$\alpha^{5692}\beta^{144} \equiv \alpha^{212}\beta^{4214} \pmod{12347}.$$

Thus $x_1 = 5692$, $x_2 = 144$, $x_3 = 212$ and $x_4 = 4214$. Now, $\gcd(x_4 - x_2, p-1) = 2$, so we begin by computing

$$\begin{aligned} y &= (x_4 - x_2)^{-1} \bmod q \\ &= (4214 - 144)^{-1} \bmod 6173 \\ &= 4312. \end{aligned}$$

Next, we compute

$$\begin{aligned} y' &= (x_1 - x_3)y \bmod (p-1) \\ &= (5692 - 212)4312 \bmod 12346 \\ &= 11862. \end{aligned}$$

Now it is the case that $\log_\alpha \beta \in \{y', y' + q \bmod (p-1)\}$. Since

$$\alpha^{y'} \bmod p = 2^{11862} \bmod 12346 = 9998,$$

we conclude that

$$\begin{aligned}\log_\alpha \beta &= y' + q \bmod (p-1)\\ &= 11862 + 6173 \bmod 12346\\ &= 5689.\end{aligned}$$

As a check, we can verify that

$$2^{5689} \equiv 8461 \pmod{12347}.$$

Hence, we have determined $\log_\alpha \beta$. □

7.5 Extending Hash Functions

So far, we have considered hash functions with a finite domain. We now study how a strongly collision-free hash function with a finite domain can be extended to a strongly collision-free hash function with an infinite domain. This will enable us to sign messages of arbitrary length.

Suppose $h : (\mathbb{Z}_2)^m \to (\mathbb{Z}_2)^t$ is a strongly collision-free hash function, where $m \geq t + 1$. We will use h to construct a strongly collision-free hash function $h^* : X \to (\mathbb{Z}_2)^t$, where

$$X = \bigcup_{i=m}^{\infty} (\mathbb{Z}_2)^i.$$

We first consider the situation where $m \geq t + 2$.

We will think of elements of X as bit-strings. $|x|$ denotes the length of x (i.e., the number of bits in x), and $x \| y$ denotes the concatenation of the bit-strings x and y. Suppose $|x| = n > m$. We can express x as the concatenation

$$x = x_1 \| x_2 \| \ldots \| x_k,$$

where

$$|x_1| = |x_2| = \ldots = |x_{k-1}| = m - t - 1$$

and

$$|x_k| = m - t - 1 - d,$$

FIGURE 7.4
Extending a hash function h to h^* ($m \geq t + 2$)

1. **for** $i = 1$ **to** $k - 1$ **do**
   $y_i = x_i$
2. $y_k = x_k \,||\, 0^d$
3. let y_{k+1} be the binary representation of d
4. $g_1 = h(0^{t+1} \,||\, y_1)$
5. **for** $i = 1$ **to** k **do**
   $g_{i+1} = h(g_i \,||\, 1 \,||\, y_{i+1})$
6. $h^*(x) = g_{k+1}$.

where $0 \leq d \leq m - t - 2$. Hence, we have that

$$k = \left\lceil \frac{n}{m - t - 1} \right\rceil .$$

We define $h^*(x)$ by the algorithm presented in Figure 7.4.

Denote

$$y(x) = y_1 \,||\, y_2 \,||\, \cdots \,||\, y_{k+1}.$$

Observe that y_k is formed from x_k by padding on the right with d zeroes, so that all the blocks y_i ($1 \leq i \leq k$) are of length $m - t - 1$. Also, in step 3, y_{k+1} should be padded on the left with zeroes so that $|y_{k+1}| = m - t - 1$.

In order to hash x, we first construct $y(x)$, and then "process" the blocks $y_1, y_2, \ldots, y_{k+1}$ in a particular fashion. It is important that $y(x) \neq y(x')$ whenever $x \neq x'$. In fact, y_{k+1} is defined in such a way that the mapping $x \mapsto y(x)$ will be an injection.

The following theorem proves that h^* is secure provided that h is secure.

THEOREM 7.3

Suppose $h : (\mathbb{Z}_2)^m \rightarrow (\mathbb{Z}_2)^t$ is a strongly collision-free hash function, where $m \geq t + 2$. Then the function $h^* : \bigcup_{i=m}^{\infty} (\mathbb{Z}_2)^i \rightarrow (\mathbb{Z}_2)^t$, as constructed in Figure 7.4, is a strongly collision-free hash function.

PROOF Suppose that we can find $x \neq x'$ such that $h^*(x) = h^*(x')$. Given such a pair, we will show how we can find a collision for h in polynomial time. Since h is assumed to be strongly collision-free, we will obtain a contradiction, and thus h^* will be proved to be strongly collision-free.

Denote

$$y(x) = y_1 \,||\, y_2 \,||\, \cdots \,||\, y_{k+1}$$

and

$$y(x') = y'_1 \,||\, y'_2 \,||\, \cdots \,||\, y'_{\ell+1},$$

where x and x' are padded with d and d' 0's, respectively, in step 2. Denote the values computed in steps 4 and 5 by $g_1, \ldots, g_{k+1}$ and $g'_1, \ldots, g'_{\ell+1}$, respectively.

We identify two cases, depending on whether or not $|x| \equiv |x'| \pmod{m-t-1}$.

case 1: $|x| \not\equiv |x'| \pmod{m-t-1}$.

Here $d \neq d'$ and $y_{k+1} \neq y'_{\ell+1}$. We have

$$\begin{aligned} h(g_k \,||\, 1 \,||\, y_{k+1}) &= g_{k+1} \\ &= h^*(x) \\ &= h^*(x') \\ &= g'_{\ell+1} \\ &= h(g'_\ell \,||\, 1 \,||\, y'_{\ell+1}), \end{aligned}$$

which is a collision for h since $y_{k+1} \neq y'_{\ell+1}$.

case 2: $|x| \equiv |x'| \pmod{m-t-1}$.

It is convenient to split this into two subcases:

case 2a: $|x| = |x'|$.

Here we have $k = \ell$ and $y_{k+1} = y'_{k+1}$. We begin as in case 1:

$$\begin{aligned} h(g_k \,||\, 1 \,||\, y_{k+1}) &= g_{k+1} \\ &= h^*(x) \\ &= h^*(x') \\ &= g'_{k+1} \\ &= h(g'_k \,||\, 1 \,||\, y'_{k+1}). \end{aligned}$$

If $g_k \neq g'_k$, then we find a collision for h, so assume $g_k = g'_k$. Then we have

$$\begin{aligned} h(g_{k-1} \,||\, 1 \,||\, y_k) &= g_k \\ &= g'_k \\ &= h(g'_{k-1} \,||\, 1 \,||\, y'_k). \end{aligned}$$

Either we find a collision for h, or $g_{k-1} = g'_{k-1}$ and $y_k = y'_k$. Assuming we do not find a collision, we continuing working backwards, until finally

we obtain

$$\begin{aligned} h(0^{t+1} \,||\, y_1) &= g_1 \\ &= g'_1 \\ &= h(0^{t+1} \,||\, y'_1). \end{aligned}$$

If $y_1 \neq y'_1$, then we find a collision for h, so we assume $y_1 = y'_1$. But then $y_i = y'_i$ for $1 \leq i \leq k+1$, so $y(x) = y(x')$. But this implies $x = x'$ since the mapping $x \mapsto y(x)$ is an injection. Since we assumed $x \neq x'$, we have a contradiction.

case 2b: $|x| \neq |x'|$.

Without loss of generality, assume $|x'| > |x|$, so $\ell > k$. This case proceeds in a similar fashion as case 2a. Assuming we find no collisions for h, we eventually reach the situation where

$$\begin{aligned} h(0^{t+1} \,||\, y_1) &= g_1 \\ &= g'_{\ell-k+1} \\ &= h(g'_{\ell-k} \,||\, 1 \,||\, y'_{\ell-k+1}). \end{aligned}$$

But the $(t+1)$st bit of $0^{t+1} \,||\, y_1$ is a 0 and the $(t+1)$st bit of $g'_{\ell-k} \,||\, 1 \,||\, y'_{\ell-k+1}$ is a 1. So we find a collision for h.

Since we have considered all possible cases, we have the desired conclusion. ∎

The construction of Figure 7.4 can be used only when $m \geq t+2$. Let's now look at the situation where $m = t+1$. We need to use a different construction for h^*. As before, suppose $|x| = n > m$. We first encode x in a special way. This will be done using the function f defined as follows:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 01. \end{aligned}$$

The algorithm to construct $h^*(x)$ is presented in Figure 7.5.

The encoding $x \mapsto y = y(x)$, defined in step 1, satisfies two important properties:

1. If $x \neq x'$, then $y(x) \neq y(x')$ (i.e., $x \mapsto y(x)$ is an injection).
2. There do not exist two strings $x \neq x'$ and a string z such that $y(x) = z \,||\, y(x')$. (In other words, no encoding is a *postfix* of another encoding. This is easily seen because each string $y(x)$ begins with 11, and there do not exist two consecutive 1's in the remainder of the string.)

FIGURE 7.5
Extending a hash function h to h^* ($m = t + 1$)

1. let $y = y_1 y_2 \dots y_k = 11 \parallel f(x_1) \parallel f(x_2) \parallel \dots \parallel f(x_n)$
2. $g_1 = h(0^t \parallel y_1)$
3. **for** $i = 1$ **to** $k - 1$ **do**
   $g_{i+1} = h(g_i \parallel y_{i+1})$
4. $h^*(x) = g_k$.

THEOREM 7.4
Suppose $h : (\mathbb{Z}_2)^{t+1} \rightarrow (\mathbb{Z}_2)^t$ is a strongly collision-free hash function. Then the function $h^* : \bigcup_{i=t+1}^{\infty} (\mathbb{Z}_2)^i \rightarrow (\mathbb{Z}_2)^t$, as constructed in Figure 7.5, is a strongly collision-free hash function.

PROOF Suppose that we can find $x \neq x'$ such that $h^*(x) = h^*(x')$. Denote

$$y(x) = y_1 y_2 \dots y_k$$

and

$$y(x') = y'_1 y'_2 \dots y'_\ell.$$

We consider two cases.

case 1: $k = \ell$.

As in Theorem 7.3, either we find a collision for h, or we obtain $y = y'$. But this implies $x = x'$, a contradiction.

case 2: $k \neq \ell$.

Without loss of generality, assume $\ell > k$. This case proceeds in a similar fashion. Assuming we find no collisions for h, we have the following sequence of equalities:

$$\begin{aligned} y_k &= y'_\ell \\ y_{k-1} &= y'_{\ell-1} \\ &\vdots \quad \vdots \\ y_1 &= y'_{\ell-k+1}. \end{aligned}$$

But this contradicts the "postfix-free" property stated above.

We conclude that h^* is collision-free. ∎

We summarize the two constructions of in this section, and the number of applications of h needed to compute h^*, in the following theorem.

THEOREM 7.5
Suppose $h : (\mathbb{Z}_2)^m \to (\mathbb{Z}_2)^t$ is a strongly collision-free hash function, where $m \geq t+1$. Then there exists a strongly collision-free hash function

$$h^* : \bigcup_{i=m}^{\infty} (\mathbb{Z}_2)^i \to (\mathbb{Z}_2)^t.$$

The number of times h is computed in the evaluation of h^* is at most

$$\begin{cases} 1 + \left\lceil \frac{n}{m-t-1} \right\rceil & \text{if } m \geq t+2 \\ 2n+2 & \text{if } m = t+1, \end{cases}$$

where $|x| = n$.

7.6 Hash Functions From Cryptosystems

So far, the methods we have described lead to hash functions that are probably too slow to be useful in practice. Another approach is to use an existing private-key cryptosystem to construct a hash function. Let us suppose that $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ is a computationally secure cryptosystem. For convenience, let us assume also that $\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_2)^n$. Here we should have $n \geq 128$, say, in order to prevent birthday attacks. This precludes using **DES** (as does the fact that the key length of **DES** is different from the plaintext length).

Suppose we are given a bitstring

$$x = x_1 \| x_2 \| \ldots \| x_k,$$

where $x_i \in (\mathbb{Z}_2)^n$, $1 \leq i \leq k$. (If the number of bits in x is not a multiple of n, then it will be necessary to pad x in some way, such as was done in Section 7.5. For simplicity, we will ignore this now.)

The basic idea is to begin with a fixed "initial value" $g_0 = \text{IV}$, and then construct $g_1, \ldots, g_k$ in order by a rule of the form

$$g_i = f(x_i, g_{i-1}),$$

where f is a function that incorporates the encryption function of our cryptosystem. Finally, define the message digest $h(x) = g_k$.

Several hash functions of this type have been proposed, and many of them have been shown to be insecure (independent of whether or not the underlying

FIGURE 7.6
Constructing M in MD4

1. $d = 447 - (|x| \bmod 512)$
2. let ℓ denote the binary representation of $|x| \bmod 2^{64}$, $|\ell| = 64$
3. $M = x \,||\, 1 \,||\, 0^d \,||\, \ell$

cryptosystem is secure). However, four variations of this theme that appear to be secure are as follows:

$$g_i = e_{g_{i-1}}(x_i) \oplus x_i$$
$$g_i = e_{g_{i-1}}(x_i) \oplus x_i \oplus g_{i-1}$$
$$g_i = e_{g_{i-1}}(x_i \oplus g_{i-1}) \oplus x_i$$
$$g_i = e_{g_{i-1}}(x_i \oplus g_{i-1}) \oplus x_i \oplus g_{i-1}.$$

7.7 The MD4 Hash Function

The **MD4 Hash Function** was proposed in 1990 by Rivest, and a strengthened version, called **MD5**, was presented in 1991. The **Secure Hash Standard** (or **SHS**) is more complicated, but it is based on the same underlying methods. It was published in the Federal Register on January 31, 1992, and adopted as a standard on May 11, 1993. (A proposed revision was put forward on July 11, 1994, to correct a "technical flaw" in the **SHS**.) All of the above hash functions are very fast, so they are practical for signing very long messages.

In this section, we will describe **MD4** in detail, and discuss some of the modifications that are employed in **MD5** and the **SHS**.

Given a bitstring x, we will first produce an array

$$M = M[0]M[1]\ldots M[N-1],$$

where each $M[i]$ is a bitstring of length 32 and $N \equiv 0 \bmod 16$. We will call each $M[i]$ a *word*. M is constructed from x using the algorithm presented in Figure 7.6.

In the construction of M, we append a single 1 to x, then we concatenate enough 0's so that the length becomes congruent to 448 modulo 512, and finally we concatenate 64 bits that contain the binary representation of the (original)

length of x (reduced modulo 2^{64}, if necessary). The resulting string M has length divisible by 512. So when we break M up into 32-bit words, the resulting number of words, denoted by N, will be divisible by 16.

Now we proceed to construct a 128-bit message digest. A high-level description of the algorithm is presented in Figure 7.7. The message digest is constructed as the concatenation of the four words A, B, C and D, which we refer to as *registers*. The four registers are initialized in step 1. Now we process the array M 16 words at a time. In each iteration of the loop in step 2, we first take the "next" 16 words of M and store them in an array X (step 3). The values of the four registers are then stored (step 4). Then we perform three "rounds" of hashing. Each round consists of one operation on each of the 16 words in X (we will describe these operations in more detail shortly). The operations done in the three rounds produce new values in the four registers. Finally, the four registers are updated in step 8 by adding back the values that were stored in step 4. This addition is defined to be addition of positive integers, reduced modulo 2^{32}.

The three rounds in **MD4** are different (unlike **DES**, say, where the 16 rounds are identical). We first describe several different operations that are employed in these three rounds. In the following description, X and Y denote input words, and each operation produces a word as output. Here are the operations employed:

$X \wedge Y$	bitwise "and" of X and Y
$X \vee Y$	bitwise "or" of X and Y
$X \oplus Y$	bitwise "xor" of X and Y
$\neg X$	bitwise complement of X
$X + Y$	integer addition modulo 2^{32}
$X \ll s$	circular left shift of X by s positions ($0 \leq s \leq 31$)

Note that all of these operations are very fast, and the only arithmetic operation that is used is addition modulo 2^{32}. If **MD4** is actually implemented, it will be necessary to take into account the underlying architecture of the computer it is run on in order to perform addition correctly. Suppose $a_1a_2a_3a_4$ are the four bytes in a word. We think of each a_i as being an integer in the range $0, \ldots, 255$, represented in binary. In a *big-endian* architecture (such as a Sun SPARCstation), this word represents the integer

$$a_1 2^{24} + a_2 2^{16} + a_3 2^8 + a_4.$$

In a *little-endian* architecture (such as the Intel 80xxx line), this word represents the integer

$$a_4 2^{24} + a_3 2^{16} + a_2 2^8 + a_1.$$

MD4 assumes a little-endian architecture. It is important that the message digest is independent of the underlying architecture. So if we wish to run **MD4** on a big-endian computer, it will be necessary to perform the addition operation $X + Y$ as follows:

FIGURE 7.7
The MD4 hash function

1. $A = 67452301$ (hex)
   $B = efcdab89$ (hex)
   $C = 98badcfe$ (hex)
   $D = 10325476$ (hex)
2. **for** $i = 0$ **to** $N/16 - 1$ **do**
3. **for** $j = 0$ **to** 15 **do**
   $X[j] = M[16i + j]$
4. $AA = A$
   $BB = B$
   $CC = C$
   $DD = D$
5. **Round1**
6. **Round2**
7. **Round3**
8. $A = A + AA$
   $B = B + BB$
   $C = C + CC$
   $D = D + DD$

1. Interchange x_1 and x_4; x_2 and x_3; y_1 and y_4; and y_2 and y_3.

2. Compute $Z = X + Y \bmod 2^{32}$

3. Interchange z_1 and z_4; and z_2 and z_3.

Rounds 1, 2, and 3 of **MD4** respectively use three functions f, g and h. Each of f, g and h is a bitwise boolean function that takes two words as input and produces a word as output. They are defined as follows:

$$f(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$
$$g(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$h(X, Y, Z) = X \oplus Y \oplus Z.$$

The complete description of Rounds 1, 2 and 3 of **MD4** are presented in Figures 7.8–7.10.

FIGURE 7.8
Round 1 of MD4

1. $A = (A + f(B, C, D) + X[0]) \lll 3$
2. $D = (D + f(A, B, C) + X[1]) \lll 7$
3. $C = (C + f(D, A, B) + X[2]) \lll 11$
4. $B = (B + f(C, D, A) + X[3]) \lll 19$
5. $A = (A + f(B, C, D) + X[4]) \lll 3$
6. $D = (D + f(A, B, C) + X[5]) \lll 7$
7. $C = (C + f(D, A, B) + X[6]) \lll 11$
8. $B = (B + f(C, D, A) + X[7]) \lll 19$
9. $A = (A + f(B, C, D) + X[8]) \lll 3$
10. $D = (D + f(A, B, C) + X[9]) \lll 7$
11. $C = (C + f(D, A, B) + X[10]) \lll 11$
12. $B = (B + f(C, D, A) + X[11]) \lll 19$
13. $A = (A + f(B, C, D) + X[12]) \lll 3$
14. $D = (D + f(A, B, C) + X[13]) \lll 7$
15. $C = (C + f(D, A, B) + X[14]) \lll 11$
16. $B = (B + f(C, D, A) + X[15]) \lll 19$

MD4 was designed to be very fast, and indeed, software implementations on Sun SPARCstations attain speeds of 1.4 Mbytes/sec. On the other hand, it is difficult to say something concrete about the security of a hash function such as **MD4** since it is not "based" on a well-studied problem such as factoring or the **Discrete Log** problem. So, as is the case with **DES**, confidence in the security of the system can only be attained over time, as the system is studied and (one hopes) not found to be insecure.

Although **MD4** has not been broken, weakened versions that omit either the first or the third round can be broken without much difficulty. That is, it is easy to find collisions for these two-round versions of **MD4**. A strengthened version of **MD4**, called **MD5**, was proposed in 1991. **MD5** uses four rounds instead of three, and runs about 30% slower than **MD4** (about .9 Mbytes/sec on a SPARCstation).

The **Secure Hash Standard** is yet more complicated, and slower (about .2 Mbytes/sec on a SPARCstation). We will not give a complete description, but we will indicate a few of the modifications employed in the **SHS**.

FIGURE 7.9
Round 2 of MD4

1. $A = (A + g(B, C, D) + X[0] + 5A827999) \lll 3$
2. $D = (D + g(A, B, C) + X[4] + 5A827999) \lll 5$
3. $C = (C + g(D, A, B) + X[8] + 5A827999) \lll 9$
4. $B = (B + g(C, D, A) + X[12] + 5A827999) \lll 13$
5. $A = (A + g(B, C, D) + X[1] + 5A827999) \lll 3$
6. $D = (D + g(A, B, C) + X[5] + 5A827999) \lll 5$
7. $C = (C + g(D, A, B) + X[9] + 5A827999) \lll 9$
8. $B = (B + g(C, D, A) + X[13] + 5A827999) \lll 13$
9. $A = (A + g(B, C, D) + X[2] + 5A827999) \lll 3$
10. $D = (D + g(A, B, C) + X[6] + 5A827999) \lll 5$
11. $C = (C + g(D, A, B) + X[10] + 5A827999) \lll 9$
12. $B = (B + g(C, D, A) + X[14] + 5A827999) \lll 13$
13. $A = (A + g(B, C, D) + X[3] + 5A827999) \lll 3$
14. $D = (D + g(A, B, C) + X[7] + 5A827999) \lll 5$
15. $C = (C + g(D, A, B) + X[11] + 5A827999) \lll 9$
16. $B = (B + g(C, D, A) + X[15] + 5A827999) \lll 13$

1. **SHS** is designed to run on a big-endian architecture, rather than a little-endian architecture.
2. **SHS** produces a 5-register (160-bit) message digest.
3. **SHS** processes the message 16 words at a time, as does **MD4**. However, the 16 words are first "expanded" into 80 words. Then a sequence of 80 operations is performed, one on each word.

The following "expansion function" is used. Given the 16 words $X[0], \ldots, X[15]$, we compute 64 more words by the recurrence relation

$$X[j] = X[j-3] \oplus X[j-8] \oplus X[j-14] \oplus X[j-16], 16 \leq j \leq 79. \quad (7.1)$$

The result of Equation 7.1 is that each of the words $X[16], \ldots, X[79]$ is formed as the exclusive-or of a predetermined subset of the words $X[0], \ldots, X[15]$.

For example, we have

$$X[16] = X[0] \oplus X[2] \oplus X[8] \oplus X[13]$$

FIGURE 7.10
Round 3 of MD4

1. $A = (A + h(B, C, D) + X[0] + 6ED9EBA1) \lll 3$
2. $D = (D + h(A, B, C) + X[8] + 6ED9EBA1) \lll 9$
3. $C = (C + h(D, A, B) + X[4] + 6ED9EBA1) \lll 11$
4. $B = (B + h(C, D, A) + X[12] + 6ED9EBA1) \lll 15$
5. $A = (A + h(B, C, D) + X[2] + 6ED9EBA1) \lll 3$
6. $D = (D + h(A, B, C) + X[10] + 6ED9EBA1) \lll 9$
7. $C = (C + h(D, A, B) + X[6] + 6ED9EBA1) \lll 11$
8. $B = (B + h(C, D, A) + X[14] + 6ED9EBA1) \lll 15$
9. $A = (A + h(B, C, D) + X[1] + 6ED9EBA1) \lll 3$
10. $D = (D + h(A, B, C) + X[9] + 6ED9EBA1) \lll 9$
11. $C = (C + h(D, A, B) + X[5] + 6ED9EBA1) \lll 11$
12. $B = (B + h(C, D, A) + X[13] + 6ED9EBA1) \lll 15$
13. $A = (A + h(B, C, D) + X[3] + 6ED9EBA1) \lll 3$
14. $D = (D + h(A, B, C) + X[11] + 6ED9EBA1) \lll 9$
15. $C = (C + h(D, A, B) + X[7] + 6ED9EBA1) \lll 11$
16. $B = B + h(C, D, A) + X[15] + 6ED9EBA1) \lll 15$

$$X[17] = X[1] \oplus X[3] \oplus X[9] \oplus X[14]$$
$$X[18] = X[2] \oplus X[4] \oplus X[10] \oplus X[15]$$
$$X[19] = X[0] \oplus X[2] \oplus X[3] \oplus X[5] \oplus X[8] \oplus X[11] \oplus X[13]$$
$$\vdots$$
$$X[79] = X[1] \oplus X[4] \oplus X[5] \oplus X[8] \oplus X[9] \oplus X[12] \oplus X[13].$$

The proposed revision of the **SHS** concerns the expansion function. It is proposed that Equation 7.1 be replaced by the following:

$$X[j] = (X[j-3] \oplus X[j-8] \oplus X[j-14] \oplus X[j-16]) \lll 1, 16 \leq j \leq 79. \quad (7.2)$$

As before, the operation "$\lll 1$" means a circular left shift of one position.

7.8 Timestamping

One difficulty with signature schemes is that a signing algorithm may be compromised. For example, suppose that Oscar is able to determine Bob's secret exponent a in the **DSS**. Then, of course, Oscar can forge Bob's signature on any message he likes. But another (perhaps even more serious) problem is that the compromise of a signing algorithm calls in to question the authenticity of all messages signed by Bob, including those he signed before Oscar stole the signing algorithm.

Here is yet another undesirable situation that could arise: Suppose Bob signs a message and later wishes to disavow it. Bob might publish his signing algorithm and then claim that his signature on the message in question is a forgery.

The reason these types of events can occur is that there is no way to determine when a message was signed. This suggests that we consider ways of *timestamping* a (signed) message. A timestamp should provide proof that a message was signed at a particular time. Then, if Bob's signing algorithm is compromised, it would not invalidate any signatures he made previously. This is similar conceptually to the way credit cards work: if someone loses a credit card and notifies the bank that isssued it, it becomes invalid. But purchases made prior to the loss of the card are not affected.

In this section, we will describe a few methods of timestamping. First, we observe that Bob can produce a convincing timestamp on his own. First, Bob obtains some "current" publicly available information which could not have been predicted before it happened. For example, such information might consist of all the major league baseball scores from the previous day, or the values of all the stocks listed on the New York Stock Exchange. Denote this information by pub.

Now, suppose Bob wants to timestamp his signature on a message x. We assume that h is a publicly known hash function. Bob will proceed according to the algorithm presented in Figure 7.11. Here is how the scheme works: The presence of the information pub means that Bob could not have produced y before the date in question. And the fact that y is published in the next day's newspaper proves that Bob did not compute y after the date in question. So Bob's signature y is bounded within a period of one day. Also observe that Bob does not reveal the message x in this scheme since only z is published. If necessary, Bob can prove that x was the message he signed and timestamped simply by revealing it.

It is also straightforward to produce timestamps if there is a trusted timestamping service available (i.e., an electronic notary public). Bob can compute $z = h(x)$ and $y = sig_K(z)$ and then send (z, y) to the timestamping service, or TSS. The TSS will then append the date D and sign the triple (z, y, D). This works perfectly well provided that the signing algorithm of the TSS remains secure and provided that the TSS cannot be bribed to backdate timestamps. (Note also that this method establishes only that Bob signed a message before a certain time. If Bob also wanted to establish that he signed it after a certain date, he could incorporate some public information pub as in the previous method.)

FIGURE 7.11
Timestamping a signature on a message x

1. Bob computes $z = h(x)$
2. Bob computes $z' = h(z \parallel pub)$
3. Bob computes $y = sig_K(z')$
4. Bob publishes (z, pub, y) in the next day's newspaper.

FIGURE 7.12
Timestamping $(z_n, y_n, \mathrm{ID}_n)$

1. The TSS computes $L_n = (t_{n-1}, \mathrm{ID}_{n-1}, z_{n-1}, y_{n-1}, h(L_{n-1}))$
2. The TSS computes $C_n = (n, t_n, z_n, y_n, \mathrm{ID}_n, L_n)$
3. The TSS computes $s_n = sig_{\mathrm{TSS}}(h(C_n))$
4. The TSS sends $(C_n, s_n, \mathrm{ID}_{n+1})$ to ID_n.

If it is undesirable to trust the TSS unconditionally, the security can be increased by sequentially linking the messages that are timestamped. In such a scheme, Bob would send an ordered triple $(z, y, \mathrm{ID(Bob)})$ to the TSS. Here z is the message digest of the message x; y is Bob's signature on z; and ID(Bob) is Bob's identifying information. The TSS will be timestamping a sequence of triples of this form. Denote by $(z_n, y_n, \mathrm{ID}_n)$ the nth triple to be timestamped by the TSS, and let t_n denote the time at which the nth request is made.

The TSS will timestamp the nth triple using the algorithm in Figure 7.12. The quantity L_n is "linking information" that ties the nth request to the previous one. (L_0 will be taken to be some predetermined dummy information to get the process started.)

Now, if challenged, Bob can reveal his message x_n, and then y_n can be verified. Next, the signature s_n of the TSS can be verified. If desired, then ID_{n-1} or ID_{n+1} can be requested to produce their timestamps, $(C_{n-1}, s_{n-1}, \mathrm{ID}_n)$ and $(C_{n+1}, s_{n+1}, \mathrm{ID}_{n+2})$, respectively. The signatures of the TSS can be checked in these timestamps. Of course, this process can be continued as far as desired, backwards and/or forwards.

7.9 Notes and References

The discrete log hash function described in Section 7.4 is due to Chaum, van Heijst, and Pfitzmann [CvHP92]. A hash function that can be proved secure provided that a composite integer n cannot be factored is given by Gibson [GIB91] (see Exercise 7.4 for a description of this scheme).

The material on extending hash functions in Section 7.5 is based on Dåmgard [DA90]. Similar methods were discovered by Merkle [ME90].

For infomation concerning the construction of hash functions from private-key cryptosystems, see Preneel, Govaerts, and Vandewalle [PGV94].

The **MD4** hashing algorithm was presented in Rivest [RI91], and the **Secure Hash Standard** is described in [NBS93]. An attack against two of the three rounds of **MD4** is given by den Boer and Bossalaers [DBB92]. Other recently proposed hash functions include N**-hash** [MOI90] and **Snefru** [ME90A].

Timestamping is discussed in Haber and Stornetta [HS91] and Bayer, Haber, and Stornetta [BHS93].

A thorough survey of hashing techniques can be found in Preneel, Govaerts, and Vandewalle [PGV93].

Exercises

7.1 Suppose $h : X \to Y$ is a hash function. For any $y \in Y$, let

$$h^{-1}(y) = \{x : h(x) = y\}$$

and denote $s_y = |h^{-1}(y)|$. Define

$$N = |\{\{x_1, x_2\} : h(x_1) = h(x_2)\}|.$$

Note that N counts the number of unordered pairs in X that collide under h. Answer the following:

(a) Prove that

$$\sum_{y \in Y} s_y = |X|,$$

so the mean of the s_y's is

$$\overline{s} = \frac{|X|}{|Y|}.$$

(b) Prove that

$$N = \sum_{y \in Y} \binom{s_y}{2} = \frac{1}{2} \sum_{y \in Y} {s_y}^2 - \frac{|X|}{2}.$$

(c) Prove that

$$\sum_{y \in Y} (s_y - \overline{s})^2 = 2N + |X| - \frac{|X|^2}{|Y|}.$$

FIGURE 7.13
Hashing $4m$ bits to m bits

1. write $x \in (\mathbb{Z}_2)^{4m}$ as $x = x_1 \parallel x_2$, where $x_1, x_2 \in (\mathbb{Z}_2)^{2m}$
2. define $h_2(x) = h_1(h_1(x_1) \parallel h_1(x_2))$.

(d) Using the result proved in part (c), prove that

$$N \geq \frac{1}{2}\left(\frac{|X|^2}{|Y|} - |X|\right).$$

Further, show that equality is attained if and only if

$$s_y = \frac{|X|}{|Y|}$$

for every $y \in Y$.

7.2 As in Exercise 7.1, suppose $h : X \to Y$ is a hash function, and let

$$h^{-1}(y) = \{x : h(x) = y\}$$

for any $y \in Y$. Let ϵ denote the probability that $h(x_1) = h(x_2)$, where x_1 and x_2 are random (not necessarily distinct) elements of X. Prove that

$$\epsilon \geq \frac{1}{|Y|},$$

with equality if and only if

$$|h^{-1}(y)| = \frac{|X|}{|Y|}$$

for every $y \in Y$.

7.3 Suppose $p = 15083$, $\alpha = 154$ and $\beta = 2307$ in the **Chaum-van Heijst-Pfitzmann Hash Function**. Given the collision

$$\alpha^{7431}\beta^{5564} \equiv \alpha^{1459}\beta^{954} \pmod{p},$$

compute $\log_\alpha \beta$.

7.4 Suppose $n = pq$, where p and q are two (secret) distinct large primes such that $p = 2p_1 + 1$ and $q = 2q_1 + 1$, where p_1 and q_1 are prime. Suppose that α is an element of order $2p_1q_1$ in $\mathbb{Z}_n{}^*$ (this is the largest order of any element in $\mathbb{Z}_n{}^*$). Define a hash function $h : \{1, \ldots, n^2\} \to \mathbb{Z}_n{}^*$ by the rule $h(x) = \alpha^x \bmod n$.

Now, suppose that $n = 603241$ and $\alpha = 11$ are used to define a hash function h of this type. Suppose that we are given three collisions for h: $h(1294755) = h(80115359) = h(52738737)$. Use this information to factor n.

7.5 Suppose $h_1 : (\mathbb{Z}_2)^{2m} \to (\mathbb{Z}_2)^m$ is a strongly collision-free hash function.

(a) Define $h_2 : (\mathbb{Z}_2)^{4m} \to (\mathbb{Z}_2)^m$ as in Figure 7.13. Prove that h_2 is strongly collision-free.

(b) For an integer $i \geq 2$, define a hash function $h_i : (\mathbb{Z}_2)^{2^i m} \to (\mathbb{Z}_2)^m$ recursively from h_{i-1}, as indicated in Figure 7.14. Prove that h_i is strongly collision-free.

7.6 Using the (original) expansion function of the **SHS**, Equation 7.1, express each of $X[16], \ldots, X[79]$ in terms of $X[0], \ldots, X[15]$. Now, for each pair $X[i], X[j]$,

FIGURE 7.14
Hashing $2^i m$ bits to m bits

1. write $x \in (\mathbb{Z}_2)^{2^i m}$ as $x = x_1 \| x_2$, where $x_1, x_2 \in (\mathbb{Z}_2)^{2^{i-1} m}$
2. define $h_i(x) = h_1(h_{i-1}(x_1) \| h_{i-1}(x_2))$.

where $1 \leq i < j \leq 15$, use a computer program to determine λ_{ij}, which denotes the number of $X[k]$'s ($16 \leq k \leq 79$) such that $X[i]$ and $X[j]$ both occur in the expression for $X[k]$. What is the range of values λ_{ij}?