Chapter 8

Hash Functions

8.1 Hash Functions

A basic component of many cryptographic algorithms is what is known as a hash function. When a hash function satisfies certain non-invertibility properties, it can be used to make many algorithms more efficient. In the following, we discuss the basic properties of hash functions and attacks on them. We also briefly discuss the random oracle model, which is a method of analyzing the security of algorithms that use hash functions. Later, in Chapter 9, hash functions will be used in digital signature algorithms. They also play a role in security protocols in Chapter 10, and in several other situations.

A cryptographic hash function $h$ takes as input a message of arbitrary length and produces as output a message digest of fixed length, for example 160 bits as depicted in Figure 8.1. Certain properties should be satisfied:

1. Given a message $m$, the message digest $h(m)$ can be calculated very quickly.

2. Given a message digest $y$, it is computationally infeasible to find an $m$ with $h(m) = y$ (in other words, $h$ is a one-way, or preimage resistant, function).
3. It is computationally infeasible to find messages $m_1$ and $m_2$ with $h(m_1) = h(m_2)$ (in this case, the function $h$ is said to be strongly collision-free).

Note that since the set of possible messages is much larger than the set of possible message digests, there should always be many examples of messages $m_1$ and $m_2$ with $h(m_1) = h(m_2)$. The requirement (3) says that it should be hard to find examples. In particular, if Bob produces a message $m$ and its hash $h(m)$, Alice wants to be reasonably certain that Bob does not know another message $m'$ with $h(m') = h(m)$, even if both $m$ and $m'$ are allowed to be random strings of symbols.

Requirement (3) is the hardest one to satisfy. In fact, in 2004, Wang, Feng, Lai, and Yu found many examples of collisions for the popular hash functions MD4, MD5, HAVAL-128, and RIPEMD. This weakness is cause for concern for using these algorithms. In fact, the MD5 collisions have been used by Ondrej Mikle to create two different and meaningful documents with the same hash.

**Example.** Let $n$ be a large integer. Let $h(m) = m \pmod{n}$ be regarded as an integer between 0 and $n - 1$. This function clearly satisfies (1). However, (2) and (3) fail: Given $y$, let $m = y$. Then $h(m) = y$. So $h$ is not one-way. Similarly, choose any two values $m_1$ and $m_2$ that are congruent mod $n$. Then $h(m_1) = h(m_2)$, so $h$ is not strongly collision-free.

**Example.** The following example, sometimes called the discrete log hash function, is due to Chaum, van Heijst, and Pfitzmann. It satisfies (2) and (3) but is much too slow to be used in practice. However, it demonstrates the basic idea of a hash function.
First we select a large prime number $p$ such that $q = (p - 1)/2$ is also prime (see Exercise 9 in Chapter 9). We now choose two primitive roots $\alpha$ and $\beta$ for $p$. Since $\alpha$ is a primitive root, there exists $a$ such that $\alpha^a \equiv \beta \pmod{p}$. However, we assume that $a$ is not known (finding $a$, if not given it in advance, involves solving a discrete log problem, which we assume is hard).

The hash function $h$ will map integers mod $q^2$ to integers mod $p$. Therefore the message digest contains approximately half as many bits as the message. This is not as drastic a reduction in size as is usually required in practice, but it suffices for our purposes.

Write $m = x_0 + x_1 q$ with $0 \leq x_0, x_1 \leq q - 1$. Then define
\[
h(m) \equiv \alpha^{x_0} \beta^{x_1} \pmod{p}.
\]

The following shows that the function $h$ is probably strongly collision-free.

**Proposition.** If we know messages $m \neq m'$ with $h(m) = h(m')$, then we can determine the discrete logarithm $a = L_\alpha(\beta)$.

**Proof.** Write $m = x_0 + x_1 q$ and $m' = x_0' + x_1' q$. Suppose
\[
\alpha^{x_0} \beta^{x_1} \equiv \alpha^{x_0'} \beta^{x_1'} \pmod{p}.
\]
Using the fact that $\beta \equiv \alpha^a \pmod{p}$, we rewrite this as
\[
\alpha^{a(x_1 - x_1') - (x_0' - x_0)} \equiv 1 \pmod{p}.
\]
Since $\alpha$ is a primitive root mod $p$, we know that $\alpha^k \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p - 1}$. In our case, this means that
\[
a(x_1 - x_1') \equiv x_0' - x_0 \pmod{p - 1}.
\]
Let $d = \gcd(x_1 - x_1', p - 1)$. There are exactly $d$ solutions to the preceding congruence (see Section 3.3), and they can be found quickly. By the choice of $p$, the only factors of $p - 1$ are $1, 2, q, p - 1$. Since $0 \leq x_1, x_1' \leq q - 1$, it follows that $-(q - 1) \leq x_1 - x_1' \leq q - 1$. Therefore, if $x_1 - x_1' \neq 0$, then it is a nonzero multiple of $d$ of absolute value less than $q$. This means that $d \neq q, p - 1$, so $d = 1$ or $2$. Therefore there are at most 2 possibilities for $a$.

Calculate $\alpha^a$ for each possibility; only one of them will yield $\beta$. Therefore, we obtain $a$, as desired.

On the other hand, if $x_1 - x_1' = 0$, then the preceding yields $x_0' - x_0 \equiv 0 \pmod{p - 1}$. Since $-(q - 1) \leq x_0' - x_0 \leq q - 1$, we must have $x_0' = x_0$. Therefore, $m = m'$, contrary to our assumption. 
\[\square\]
It is now easy to show that $h$ is preimage resistant. Suppose we have an algorithm $g$ that starts with a message digest $y$ and quickly finds an $m$ with $h(m) = y$. In this case, it is easy to find $m_1 \neq m_2$ with $h(m_1) = h(m_2)$: Choose a random $m$ and compute $y = h(m)$, then compute $g(y)$. Since $h$ maps $q^2$ messages to $p - 1 = 2q$ message digests, there are many messages $m'$ with $h(m') = h(m)$. It is therefore not very likely that $m' = m$. If it is, try another random $m$. Soon, we should find a collision, that is, messages $m_1 \neq m_2$ with $h(m_1) = h(m_2)$. The preceding proposition shows that we can then solve a discrete log problem. Therefore, it is unlikely that such a function $g$ exists.

As we mentioned earlier, this hash function is good for illustrative purposes but is impractical because of its slow nature. Although it can be computed efficiently via repeated squaring, it turns out that even repeated squaring is too slow for practical applications. In applications such as electronic commerce, the extra time required to perform the multiplications in software is prohibitive.

There are several professional strength hash functions available. For example, there is the popular MD family due to Rivest. In particular, MD4 and its stronger version MD5 are widely used and produce 128-bit message digests for messages of arbitrary length. Another alternative is NIST’s Secure Hash Algorithm (SHA), which yields a 160-bit message digest.

Hash functions may also be employed as a check on data integrity. The question of data integrity comes up in basically two scenarios. The first is when the data (encrypted or not) are being transmitted to another person and a noisy communication channel introduces errors to the data. The second occurs when an observer rearranges the transmission in some manner before it gets to the receiver. Either way, the data have become corrupted.

For example, suppose Alice sends Bob long messages about financial transactions with Eve and encrypts them in blocks. Perhaps Eve deduces that the tenth block of each message lists the amount of money that is to be deposited to Eve’s account. She could easily substitute the tenth block from one message into another and increase the deposit.

In another situation, Alice might send Bob a message consisting of several blocks of data, but one of the blocks is lost during transmission. Bob might not ever realize that the block is missing.

Here is how hash functions can be used. Say we send $(m, h(m))$ over the communications channel and it is received as $(M, H)$. To check whether errors might have occurred, the recipient computes $h(M)$ and sees whether it equals $H$. If any errors occurred, it is likely that $h(M) \neq H$, because of the collision-free properties of $h$. 

8.2 A Simple Hash Example

There are many families of hash functions. The discrete log hash function that we described earlier is too slow to be of practical use. One reason it is slow is that it employs modular exponentiation, which makes its computational requirements about the same as RSA or ElGamal.

In order to make fast cryptographic hash functions, we need to steer away from modular exponentiation, and instead work on the message $m$ at the bit level. We now describe the basic idea behind many cryptographic hash functions by giving a simple hash function that shares many of the basic properties as hash functions that are used in practice. This hash function is not an industrial strength hash function and should never be used in any system.

Suppose we start with a message $m$ of arbitrary length $L$. We may break $m$ into $n$-bit blocks, where $n$ is much smaller than $L$. We shall denote these $n$-bit blocks by $m_j$, and thus represent $m = [m_1, m_2, \ldots, m_l]$. Here $l = \lceil L/n \rceil$, and the last block $m_l$ is padded with zeros to ensure that it has $n$ bits.

We write the $j$th block $m_j$ as a row vector

$$m_j = [m_{j1}, m_{j2}, m_{j3}, \ldots, m_{jn}],$$

where each $m_{ji}$ is a bit.

Now, we may stack these row vectors to form an array. Our hash $h(m)$ will have $n$ bits, where we calculate the $i$th bit as the XOR along the $i$th column of the matrix, that is $h_i = m_{1i} \oplus m_{2i} \oplus \cdots \oplus m_{li}$. We may visualize this as

$$
\begin{bmatrix}
  m_{11} & m_{12} & \cdots & m_{1n} \\
  m_{21} & m_{22} & \cdots & m_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  m_{l1} & m_{l2} & \cdots & m_{ln}
\end{bmatrix}
\begin{bmatrix}
  \oplus & \oplus & \cdots & \oplus \\
  \downarrow & \downarrow & \downarrow & \downarrow \\
  \downarrow & \downarrow & \downarrow & \downarrow
\end{bmatrix}
\begin{bmatrix}
  c_1 & c_2 & \cdots & c_n
\end{bmatrix} = h(m)
$$

This hash function is able to take an arbitrary length message and output an $n$-bit message digest. It is not considered cryptographically secure, though, since it is easy to find two messages that hash to the same value.

Practical cryptographic hash functions typically make use of several other bit-level operations in order to make it more difficult to find collisions.
One operation that is often used is bit rotation. We saw the use of bit rotation in DES. We define the left rotation operation

$$m \leftarrow y$$

as the result of shifting $m$ to the left $y$ bits and wrapping the leftmost $y$ bits around, placing them in rightmost $y$ bit locations.

We may modify our simple hash function above by requiring that block $m_j$ is left rotated by $j - 1$, to produce a new block $m'_j = m_j \leftarrow j - 1$. We may now arrange the $m'_j$ in columns and define a new, simple hash function by XORing these columns. Thus, we get

$$\begin{bmatrix}
m_{11} & m_{12} & \cdots & m_{1n} \\
m_{22} & m_{23} & \cdots & m_{21} \\
m_{33} & m_{34} & \cdots & m_{32} \\
\vdots & \vdots & \ddots & \vdots \\
m_{ll} & m_{l,l+1} & \cdots & m_{l,l-1}
\end{bmatrix}
\downarrow \downarrow \downarrow \downarrow
\oplus \oplus \oplus \oplus
\downarrow \downarrow \downarrow \downarrow
\begin{bmatrix}
c_1 & c_2 & \cdots & c_n
\end{bmatrix} = h(m).$$

This new hash function involving rotations makes it a little harder to find collisions than with the previous hash function. But it is still possible. Building a cryptographic hash requires considerably more tricks than just rotating. In the next section, we describe an example of a hash function that is used in practice. It uses the techniques of the present section, coupled with many more ways of mixing the bits.

### 8.3 The Secure Hash Algorithm

Now let us look at what is involved in making a real cryptographic hash function. Unlike block ciphers, where there are many block ciphers to choose from, there are only a few hash functions that are available. The most notable of these are the Secure Hash Algorithm (SHA-1), the Message Digest (MD) family, and the RIPEMD-160 message digest algorithm. The MD family has an interesting history. The original MD algorithm was never published, and the first MD algorithm to be published was MD2, followed by MD4 and MD5. Weaknesses in MD2 and MD4 were found, and MD5 was proposed by Ron Rivest as an improvement upon MD4. Collisions have been found for MD5, and the strength of MD5 is now less certain.
For this reason, we have chosen to discuss SHA-1 instead of the MD family. The reader is warned that discussion that follows is fairly technical and is provided in order to give the flavor of what happens inside a hash function.

The Secure Hash Algorithm was developed by the National Security Agency (NSA) for the National Institute of Standards and Technology (NIST). The original version, often referred to as SHA or SHA-0, was published in 1993 as a federal information processing standard (FIPS 180). SHA contained a weakness that was later uncovered by the NSA, which led to the a revised standards document (FIPS 180-1) that was released in 1995. This revised document describes the improved version, SHA-1, which is now the hash algorithm recommended by NIST.

SHA-1 produces a 160-bit hash and is built upon the same design principles as MD4 and MD5. These hash functions use an iterative procedure. Just as we did earlier, the original message \( m \) is broken into a set of fixed-size blocks, \( m = [m_1, m_2, \cdots, m_l] \), where the last block is padded to fill out the block. The message blocks are then processed via a sequence of rounds that use a compression function \( h' \), which combines the current block and the result from the previous round. That is, we start with an initial value \( X_0 \), and define \( X_j = h'(X_{j-1}, m_j) \). The final \( X_l \) is the message digest.

The trick behind building a hash function is to devise a good compression function. This compression function should be built in such a way as to make each input bit affect as many output bits as possible. One main difference between SHA-1 and the MD family is that for SHA-1 the input bits are used more often during the course of the hash function than they are for MD4 or MD5. This more conservative approach makes the design of SHA-1 more secure than either MD4 or MD5, but also makes it a little slower.

SHA-1 begins by taking the original message and padding it with a 1 bit followed by a sequence of 0 bits. Enough 0 bits are appended to make the new message 64 bits short of the next highest multiple of 512 bits in length. Following the appending of 1 and 0s, we append the 64-bit representation of the length \( T \) of the message. Thus, if the message is \( T \) bits, then the appending creates a message that consists of \( L = \lfloor T/512 \rfloor + 1 \) blocks of 512 bits. We break the appended message into \( L \) blocks \( m_1, m_2, \cdots, m_L \). The hash algorithm inputs these blocks one by one.

For example, if the original message has 2800 bits, we add a 1 and 207 0’s to obtain a new message of length 3008 = 6 × 512 − 64. Since 2800 = 101011110000₂ in binary, we append fifty-two 0’s followed by 101011110000 to obtain a message of length 3072. This is broken into six blocks of length 512.

In the description of the hash algorithm, we need the following operations on strings of 32 bits:
The SHA-1 Algorithm

1. Start with a message $m$. Append bits, as specified in the text, to obtain a message $y$ of the form $y = m_1 || m_2 || \cdots || m_L$, where each $m_i$ has 512 bits.
2. Initialize $H_0 = 67452301$, $H_1 = EFCDAB89$, $H_2 = 98BADCFE$, $H_3 = 10325476$, $H_4 = C3D2E1F0$.
3. For $i = 0$ to $L - 1$, do the following:
   (a) Write $m_i = W_0 || W_1 || \cdots || W_{15}$, where each $W_j$ has 32 bits.
   (b) For $t = 16$ to 79, let $W_t = (W_{t-3} \leftarrow 1) \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$
   (c) Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.
   (d) For $t = 0$ to 79, do the following steps in succession:
       $T = (A \leftarrow 5) + f_t(B, C, D) + E + W_t + K_t$, $E = D$, $D = C$, $C = (B \leftarrow 30)$, $B = A$, $A = T$.
   (e) Let $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_4 = H_4 + E$.
4. Output $H_0 || H_1 || H_2 || H_3 || H_4$. This is the 160-bit hash value.

1. $X \wedge Y =$ bitwise “and”, which is bitwise multiplication mod 2, or bitwise minimum.
2. $X \lor Y =$ bitwise “or”, which is bitwise maximum.
3. $X \oplus Y =$ bitwise addition mod 2.
4. $\neg X$ changes 1’s to 0’s and 0’s to 1’s.
5. $X + Y =$ addition of $X$ and $Y$ mod $2^{32}$, where $X$ and $Y$ are regarded as integers mod $2^{32}$.
6. $X \leftarrow r =$ shift of $X$ to the left by $r$ positions (and the beginning wraps around to the end).
8.3. The Secure Hash Algorithm

We also need the following functions:

\[ f_t(B, C, D) = \begin{cases} 
(B \land C) \lor ((\neg B) \land D) & \text{if } 0 \leq t \leq 19 \\
B \oplus C \oplus D & \text{if } 20 \leq t \leq 39 \\
(B \land C) \lor (B \land D) \lor (C \land D) & \text{if } 40 \leq t \leq 59 \\
B \oplus C \oplus D & \text{if } 60 \leq t \leq 79 
\end{cases} \]

Define constants \( K_0, \ldots, K_{79} \) as follows:

\[ K_t = \begin{cases} 
5A827999 & \text{if } 0 \leq t \leq 19 \\
6ED9EBA1 & \text{if } 20 \leq t \leq 39 \\
8F1BBCDC & \text{if } 40 \leq t \leq 59 \\
CA62C1D6 & \text{if } 60 \leq t \leq 79. 
\end{cases} \]

The above are written in hexagonal notation. Each digit or letter represents a string of 4 bits:

\[ 0 = 0000, \ 1 = 0001, \ 2 = 0010, \ldots, \ 9 = 1001, \]
\[ A = 1010, \ B = 1011, \ldots, \ F = 1111 \]

For example, BA1 equals \( 11 \times 16^2 + 10 \times 16^1 + 1 = 2977 \).

We summarize SHA-1 in the table on page 214. The core of the algorithm is step (3), which we present in Figure 8.2. All of the operations involved in the SHA-1 algorithm are elementary and very fast. Note that the basic procedure is iterated as many times as is needed to digest the whole message. This iterative procedure makes the algorithm very efficient in terms of reading and processing the message.

We now step through the algorithm. SHA-1 begins by first creating an initial 160-bit register \( X_0 \), that consists of five 32-bit subregisters \( H_0, H_1, H_2, H_3, H_4 \). These subregisters are initialized as follows:

\[ H_0 = 67452301 \]
\[ H_1 = EFCDBAB989 \]
\[ H_2 = 98BADCFFE \]
\[ H_3 = 10325476 \]
\[ H_4 = C3D2E1F0. \]

After the message block \( m_j \) is processed, the register \( X_j \) is updated to yield a register \( X_{j+1} \).

SHA-1 loops through each of the 512-bit message blocks \( m_j \). For each message block, \( m_j \), the register \( X_j \) is copied into subregisters \( A, B, C, D, E \).

Let's start with the first message block \( m_0 \), which is cut and mixed to yield \( W_0, \ldots, W_{79} \). These are fed into a sequence of four rounds, corresponding
to the four intervals \(0 \leq t \leq 19\), \(20 \leq t \leq 39\), \(40 \leq t \leq 59\), and \(60 \leq t \leq 79\). Each round takes as input the current value of the register \(X_0\) and the blocks \(W_t\) for that interval, and operates upon them for 20 iterations (that is, the counter \(t\) runs through the 20 values in the interval). Each iteration uses the round constant \(K_t\) and the operation \(f_t(B,C,D)\), which are the same for all iterations in that round. One after another, each round updates the \((A,B,C,D,E)\). Following the output of the fourth round, which is completed when \(t = 79\), the output subregisters \((A,B,C,D,E)\) are added to the input subregisters \((H_0,H_1,H_2,H_3,H_4)\) to produce 160 bits of output that become the next register \(X_1\), which will be copied into \((A,B,C,D,E)\) when processing the next message block \(m_1\). This output register \(X_1\) may be looked at as the output of the compression function \(h'\) when it is given input \(X_0\) and \(m_0\); that is, \(X_1 = h'(X_0,m_0)\).

We continue in this way for each of the of the 512-bit message blocks \(m_j\), using the previous register output \(X_j\) as input into calculating the next register output \(X_{j+1}\). Hence \(X_{j+1} = h'(X_j,m_j)\). In Figure 8.2, we depict the operation of the compression function \(h'\) on the \(j\)th message block \(m_j\) using the register \(X_j\). After completing all of the \(L\) message blocks, the

---

**Figure 8.2:** The operations performed by SHA-1 on a single message block \(m_j\).
8.4 Birthday Attacks

The final output is the 160-bit message digest.

The basic building block of the algorithm are the operations that take place on the subregisters in step (3d). These operations are pictured in Figure 8.3. These operations take the subregisters and operate on them using rotations and XORs, much like the method described in Section 8.2. However, SHA-1 also uses complicated mixing operations that are performed by \( f_t \) and the constants \( K_t \).

For more details on this and other hash functions, and for some of the theory involved in their construction, see [Stinson], [Schneier], and [Menezes et al.].

8.4 Birthday Attacks

If there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. This might seem surprising; it is called the \textbf{birthday paradox}. Let’s see why it’s true. We’ll ignore leap years and assume that all birthdays are equally likely (if not, the probabilities given would be slightly higher).

Consider the case of 23 people. We’ll compute the probability that they all have different birthdays. Line them up in a row. The first person uses up one day, so the second person has probability \( 1 - 1/365 \) of having a
different birthday. There are two days removed for the third person, so the probability is \((1 - 2/365)\) that the third birthday differs from the first two. Therefore, the probability of all three people having different birthdays is \((1 - 1/365)(1 - 2/365)\). Continuing in this way, we see that the probability that all 23 people have different birthdays is
\[
(1 - \frac{1}{365})(1 - \frac{2}{365}) \cdots (1 - \frac{22}{365}) = .493
\]
Therefore, the probability of at least two having the same birthday is
\[
1 - .493 = .507
\]

One way to understand the preceding calculation intuitively is to consider the case of 40 people. If the first 30 have a match, we’re done, so suppose the first 30 have different birthdays. Now we have to choose the last 10 birthdays. Since 30 birthdays are already chosen, we have approximately a 10% chance that a randomly chosen birthday will match one of the first 30. And we are choosing 10 birthdays. Therefore, it shouldn’t be too surprising that we get a match. In fact, the probability is 89% that there is a match among 40 people.

More generally, suppose we have \(N\) objects, where \(N\) is large. There are \(r\) people, and each chooses an object (with replacement, so several people could choose the same one). Then

\[
\text{Prob(there is a match)} \approx 1 - e^{-\lambda} \quad \text{when} \quad r \approx \sqrt{2\lambda N}.
\] (8.1)

Note that this is only an approximation that holds for large \(N\); for small \(n\) it is better to use the above product and obtain an exact answer. In Exercise 5, we derive this approximation. Letting \(\lambda = \ln 2\), we find that if \(r \approx 1.177\sqrt{N}\), then the probability is 50% that at least two people choose the same object.

For example, suppose we have 40 license plates, each ending in a 3-digit number. What is the probability that two of the license plates end in the same 3 digits? We have \(N = 1000\), the number of possible 3-digit numbers, and \(r = 40\), the number of license plates under consideration. Solve
\[
40 = \sqrt{2\lambda \cdot 1000}
\]
for \(\lambda\) to obtain \(\lambda = 0.8\). The approximate probability of a match is
\[
1 - e^{-0.8} = .551,
\]
so the probability is more than 50% that there is a match. We stress that this is only an approximation. The correct answer is obtained by calculating
\[
1 - \left(1 - \frac{1}{1000}\right)\left(1 - \frac{2}{1000}\right) \cdots \left(1 - \frac{39}{1000}\right) = .546.
\]
The next time you are stuck in traffic (and have a passenger to record numbers), check out this prediction.

But what is the probability that one of these 40 license plates has the same last 3 digits as yours (assuming that yours ends in 3 digits)? Each plate has probability \(1 - 1/1000\) of not matching yours, so the probability is \((1 - 1/1000)^{40} = .961\) that none of the 40 plates matches your plate. The reason the birthday paradox works is that we are not just looking for matches between one fixed plate, such as yours, and the other plates. We are looking for matches between any two plates in the set, so there are many more opportunities for matches.

The applications of these ideas to cryptology require a slightly different setup. Suppose there are two rooms, each with 30 people. What is the probability that someone in the first room has the same birthday as someone in the second room? More generally, suppose there are \(N\) objects and there are two groups of \(r\) people. Each person from each group selects an object (with replacement). What is the probability that someone from the first group chooses the same object as someone from the second group? Again, if \(r \approx \sqrt{\lambda N}\), then the probability is \(1 - e^{-\lambda}\) that there is a match. The probability of exactly \(i\) matches is \(\lambda^i e^{-\lambda}/i!\). An analysis of this problem, with generalizations, is given in [Girault et al.].

For example, if we take \(N = 365\) and \(r = 30\), then \(\sqrt{365\lambda} = 30\) yields \(\lambda = 2.466\). Since \(1 - e^{-\lambda} = .915\), there is approximately a 91.5% probability that someone in one group of 30 people has the same birthday as someone in a second group of 30 people.

The birthday attack can be used to find collisions for hash functions if the output of the hash function is not sufficiently large. Suppose that \(h\) is an \(n\)-bit hash function. Then there are \(N = 2^n\) possible outputs. Make a list \(h(x)\) for approximately \(r = \sqrt{N} = 2^{n/2}\) random choices of \(x\). Then we have the situation of \(r \approx \sqrt{N}\) “people” with \(N\) possible “birthdays,” so there is a good chance of having two values \(x_1\) and \(x_2\) with the same hash value. If we make the list longer, for example \(r = 10 \cdot 2^{n/2}\) values of \(x\), the probability becomes very high that there is a match.

Similarly, suppose we have two sets of inputs, \(S\) and \(T\). If we compute \(h(s)\) for approximately \(\sqrt{N}\) randomly chosen \(s \in S\) and \(h(t)\) for approximately \(\sqrt{N}\) randomly chosen \(t \in T\), then we expect some value \(h(s)\) to be equal to some value \(h(t)\). This situation will arise in an attack on signature schemes in Chapter 9, where \(S\) will be a set of good documents and \(T\) will be a set of fraudulent documents.

If the output of the hash function is around \(n = 60\) bits, the above attacks have a high chance of success. It is necessary to make lists of length approximately \(2^{n/2} = 2^{30} \approx 10^9\) and to store them. This is possible on most computers. However, if the hash function outputs 128-bit values, then the
lists have length around $2^{64} \approx 10^{19}$, which is too large, both in time and in memory.

\section*{A Birthday Attack on Discrete Logarithms}

Suppose we are working with a large prime $p$ and want to evaluate $L_\alpha(\beta)$. In other words, we want to solve $\alpha^x \equiv \beta \pmod{p}$. We can do this with high probability by a birthday attack.

Make two lists, both of length around $\sqrt{p}$:

1. The first list contains numbers $\alpha^k \pmod{p}$ for approximately $\sqrt{p}$ randomly chosen values of $k$.
2. The second list contains numbers $\beta \alpha^{-\ell} \pmod{p}$ for approximately $\sqrt{p}$ randomly chosen values of $\ell$.

There is a good chance that there is a match between some element on the first list and some element on the second list. If so, we have

$$\alpha^k \equiv \beta \alpha^{-\ell}, \text{ hence } \alpha^{k+\ell} \equiv \beta \pmod{p}.$$ 

Therefore, $x \equiv k + \ell \pmod{p - 1}$ is the desired discrete logarithm.

Let’s compare this method with the Baby Step - Giant Step (BSGS) method described in Section 7.2. Both methods have running time and storage space proportional to $\sqrt{p}$. However, the BSGS algorithm is \textbf{deterministic}, which means that it is guaranteed to produce an answer. The Birthday algorithm is probabilistic, which means that it probably produces an answer, but this is not guaranteed. Moreover, there is a computational advantage to the BSGS algorithm. Computing one member of a list from a previous one requires one multiplication (by $\alpha$ or by $\alpha^{-N}$). In the Birthday algorithm, the exponent $k$ is chosen randomly, so $\alpha^k$ must be computed each time. This makes the algorithm slower. Therefore, the BSGS algorithm is somewhat superior to the Birthday method.

\section*{8.5 Multicollisions}

In this section, we show that the iterative nature of most hash algorithms makes them less resistant than expected to finding multicollisions, namely inputs $x_1, \ldots, x_n$ all with the same hash value. This was pointed out by Joux \cite{Joux}, who also gave implications for properties of concatenated hash functions, which we discuss below.

Suppose there are $r$ people and there are $N$ possible birthdays. It can be shown that if $r \approx N^{(k-1)/k}$, then there is a good chance of at least $k$
8.5. Multicollisions

people having the same birthday. In other words, we expect a \( k \)-collision. If the output of a hash function is random, then we expect that this estimate would hold for \( k \)-collisions of hash function values. Namely, if a hash function has \( n \)-bit outputs, hence \( N = 2^n \) possible values, and if we calculate \( r = 2^{n(k-1)/k} \) values of the hash function, we expect a \( k \)-collision. However, in the following, we’ll show that often we can obtain collisions much more easily.

In many hash functions, for example, SHA-1, there is a compression function \( f \) that operates on inputs of a fixed length. Also, there is a fixed initial value \( IV \). The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message \( M \) into blocks \( M_1, M_2, \ldots, M_\ell \).
2. Let \( H_0 \) be the initial value \( IV \).
3. For \( i = 1, 2, \ldots, \ell \), let \( H_i = f(H_{i-1}, M_i) \).
4. Let \( H(M) = H_\ell \).

In SHA-1, the compression function is given in Figure 8.3. For each iteration, it takes a 160-bit input \( A || B || C || D || E \) from the preceding iteration along with a message block \( m_i \) of length 512 and outputs a new string \( A || B || C || D || E \) of length 160.

Suppose the output of the function \( f \), and therefore also of the hash function \( H \), has \( n \) bits. A birthday attack can find, in approximately \( 2^{n/2} \) steps, two blocks \( m_0 \) and \( m_0' \) such that \( f(H_0, m_0) = f(H_0, m_0') \). Let \( h_1 = f(H_0, m_0) \). A second birthday attack finds blocks \( m_1 \) and \( m_1' \) with \( f(h_1, m_1) = f(h_1, m_1') \). Continuing in this manner, we let

\[
h_i = f(h_{i-1}, m_{i-1})
\]

and use a birthday attack to find \( m_i \) and \( m_i' \) with

\[
f(h_i, m_i) = f(h_i, m_i').
\]

This process is continued until we have blocks \( m_0, m_0', m_1, m_1', \ldots, m_{t-1}, m_{t-1}' \), where \( t \) is some integer to be determined later.

We claim that each of the \( 2^t \) messages

\[
\begin{align*}
m_0 || m_1 || \cdots || m_{t-1} \\
m_0' || m_1' || \cdots || m_{t-1}' \\
m_0 || m_1' || \cdots || m_{t-1} \\
m_0' || m_1 || \cdots || m_{t-1}' \\
\vdots & \\
m_0' || m_1' || \cdots || m_{t-1}'
\end{align*}
\]
(all possible combinations with $m_i$ and $m'_i$) has the same hash value. This is because of the iterative nature of the hash algorithm. At each calculation $h_i = f(m, h_{i-1})$, the same value $h_i$ is obtained whether $m = m_{i-1}$ or $m = m'_{i-1}$. Therefore, the output of the function $f$ during each step of the hash algorithm is independent of whether an $m_{i-1}$ or an $m'_{i-1}$ is used. Therefore, the final output of the hash algorithm is the same for all messages. We thus have a $2^t$-collision.

The expected running time of this procedure is approximately a constant times $t n 2^{n/2}$ (see Exercise 6). Let $t = 2$, for example. Then it takes only around twice as long to find four messages with same hash value as it took to find two messages with the same hash. If the output of the hash function were truly random, rather than produced for example by an iterative algorithm, then the above procedure would not work. The expected time to find four messages with the same hash would then be approximately $2^{3n/4}$, which is much longer than the time it takes to find two colliding messages. Therefore, it is easier to find collisions with an iterative hash algorithm.

An interesting consequence of the preceding discussion relates to attempts to improve hash functions by concatenating their outputs. Suppose we have two hash functions $H_1$ and $H_2$. Before [Joux] appeared, the general wisdom was that the concatenation

$$H(M) = H_1(M) || H_2(M)$$

should be a significantly stronger hash function than either $H_1$ or $H_2$ individually. This would allow people to use somewhat weak hash functions to build much stronger ones. However, it now seems that this is not the case. Suppose the output of $H_i$ has $n_i$ bits. Also, assume that $H_1$ is calculated by an iterative algorithm, as in the preceding discussion. No assumptions are needed for $H_2$. We may even assume that it is a random oracle, in the sense of Section 8.6. In time approximately $n_2 n_1 2^{n_1/2}$, we can find $2^{n_2/2}$ messages that all have the same hash value for $H_1$. We then compute the value of $H_2$ for each of these $2^{n_2/2}$ messages. By the birthday paradox, we expect to find a match among these values of $H_2$. Since these messages all have the same $H_1$ value, we have a collision for $H_1 || H_2$. Therefore, in time proportional to $n_2 n_1 2^{n_1/2} + n_2 2^{n_2/2}$ (we’ll explain this estimate shortly), we expect to be able to find a collision for $H_1 || H_2$. This is not much longer than the time a birthday attack takes to find a collision for the longer of $H_1$ and $H_2$, and is much faster than the time $2^{(n_1+n_2)/2}$ that a standard birthday attack would take on this concatenated hash function.

How did we get the estimate $n_2 n_1 2^{n_1/2} + n_2 2^{n_2/2}$ for the running time? We used $n_2 n_1 2^{n_1/2}$ steps to get the $2^{n_2/2}$ messages with the same $H_1$ value. Each of these messages consisted of $n_2$ blocks of a fixed length. We then evaluated $H_2$ for each of these messages. For almost every hash function,
8.6 The Random Oracle Model

Ideally, a hash function is indistinguishable from a random function. The random oracle model, introduced in 1993 by Bellare and Rogaway, gives a convenient method for analyzing the security of cryptographic algorithms that use hash functions by treating hash functions as random oracles.

A random oracle acts as follows. Anyone can give it an input, and it will produce a fixed length output. If the input has already been asked previously by someone, then the oracle outputs the same value as it did before. If the input is not one that had previously been given to the oracle, then the oracle gives a randomly chosen output. For example, it could flip \( n \) fair coins and use the result to produce an \( n \)-bit output.

For practical reasons, a random oracle cannot be used in most cryptographic algorithms; however, assuming that a hash function behaves like a random oracle allows us to analyze the security of many cryptosystems that use hash functions.

We already made such an assumption in Section 8.4. When calculating the probability that a birthday attack finds collisions for a hash function, we assumed that the output of the hash function is randomly and uniformly distributed among all possible outcomes. If this is not the case, so the hash function has some values that tend to occur more frequently than others, then the probability of finding collisions is somewhat higher (for example, consider the extreme case of a really bad hash function that with high probability outputs only one value). Therefore, our estimate for the probability of collisions really only applies to an idealized setting. In practice, the use of actual hash functions probably produces very slightly more collisions.

In the following, we show how the random oracle model is used to analyze the security of a cryptosystem. Because the ciphertext is much longer than the plaintext, the system we describe is not as efficient as methods such as OAEP (see Section 6.2). However, the present system is a good illustration of the use of the random oracle model.

Let \( f \) be a one-way function that Bob knows how to invert. For example, \( f(x) = x^e \pmod{n} \), where \((e,n)\) is Bob’s public RSA key. Let \( H \) be a hash function. To encrypt a message \( m \), which is assumed to have the same bitlength as the output of \( H \), Alice chooses a random integer \( r \pmod{n} \) and lets the ciphertext be

\[
(y_1, y_2) = (f(r), H(r) \oplus m).
\]
When Bob receives \((y_1, y_2)\), he computes
\[
r = f^{-1}(y_1), \quad m = H(r) \oplus y_2.
\]
It is easy to see that this decryption produces the original message \(m\).

Now consider the following problem. Suppose Alice is shown two plain-
texts, \(m_1\) and \(m_2\), and one ciphertext, but she is not told which plaintext
encrypts to this ciphertext. Her job is to guess which one. If she cannot
do this with probability significantly better than 50%, then we say that the
cryptosystem has the **ciphertext indistinguishability** property.

Let's assume that the hash function is a random oracle. We'll show that
if Alice can succeed with significantly better than 50% probability , then she
can invert \(f\) with significantly better than zero probability . Therefore , if
\(f\) is truly a one-way function, the cryptosystem has the ciphertext indistin-
guishability property.

Suppose now that Alice has a ciphertext \((y_1, y_2)\) and two plaintexts, \(m_1\)
and \(m_2\). She is allowed to make a series of queries to the random oracle, each
time sending it a value \(r\) and receiving back the value \(H(r)\). Suppose that,
in the process of trying to figure out whether \(m_1\) or \(m_2\) yielded \((y_1, y_2)\), Alice
has asked for the hash values of each element of some set \(L = \{r_1, r_2, \ldots, r_ℓ\}\).

As Alice asks for each value \(H(x)\) for \(x \in L\), she computes \(f(x)\) for this
\(x\). If \(r \in L\), she eventually tries \(vx = r\) and finds that \(f(r) = y_1\). She then
knows this is the correct value of \(r\). Since \(r \in L\), she obtained \(H(r)\) from
the oracle, so she then computes \(H(r) \oplus y_2\) to obtain the plaintext, which is
either \(m_1\) or \(m_2\).

If \(r \not\in L\), then Alice does not know the value of \(H(r)\). Since \(H\) is a random
oracle, the possible values of \(H(r)\) are randomly and uniformly distributed
among all possible outputs. Therefore, the possible values for \(H(r) \oplus m\), for
any \(m\), are also randomly and uniformly distributed among all possibilities.
This means that \(y_2\) gives Alice no information about whether it comes from
\(m_1\) or from \(m_2\). So if \(r \not\in L\), Alice has probability 1/2 of guessing the correct
plaintext.

Now suppose \(r \in L\). Then Alice has obtained the value of \(H(r)\) from
the random oracle. She computes \(H(x) \oplus y_2\) for each \(x \in L\). If she gets \(m_1\)
or \(m_2\), then she computes \(f(x)\). If \(f(x) = y_1\), then Alice knows that \(x = r\)
(since \(f(r) = y_1\), too), and therefore that \(H(r) \oplus y_2\) is the plaintext that
was encrypted.

Let’s write this procedure in terms of probabilities. If \(r \not\in L\), Alice
guesses correctly half the time. If \(r \in L\), Alice always guesses correctly. Therefore

\[
\text{Prob}(\text{Alice guesses correctly}) = \frac{1}{2} \text{Prob}(r \not\in L) + \text{Prob}(r \in L).
\]
This is because Alice has 1/2 probability when \( r \not\in L \) and always succeeds when \( r \in L \).

Suppose now that Alice has probability at least \( \frac{1}{2} + \epsilon \) of guessing correctly, where \( \epsilon > 0 \) is some fixed number. Since \( \text{Prob}(r \not\in L) \leq 1 \) (this is true of all probabilities), we obtain

\[
\frac{1}{2} + \epsilon \leq \frac{1}{2} + \text{Prob}(r \in L).
\]

Therefore,

\[
\text{Prob}(r \in L) \geq \epsilon.
\]

But if \( r \in L \), then Alice discovers that \( f(r) = y_1 \), so the probability that she solves \( f(r) = y_1 \) for \( r \) is at least \( \epsilon \).

If we assume that it is computationally infeasible for Alice to find \( r \) with probability at least \( \epsilon \), then we conclude that it is computationally infeasible for Alice to guess correctly with probability at least \( \frac{1}{2} + \epsilon \). Therefore, if the function \( f \) is one-way, then the cryptosystem has the ciphertext indistinguishability property.

Note that it was important in the argument to assume that the values of \( H \) are randomly and uniformly distributed. If this were not the case, so the hash function had some bias, then Alice might have some method for guessing correctly with better than 50% probability, maybe with probability \( \frac{1}{2} + \epsilon \). This would reduce the conclusion to \( \text{Prob}(r \in L) \geq 0 \), which gives us no information. Therefore, the assumption that the hash function is a random oracle is important.

Of course, a good hash function is probably close to acting like a random oracle. In this case, the above argument shows that the cryptosystem with an actual hash function should be fairly resistant to Alice guessing correctly. However, it should be noted that Canetti, Goldreich, and Halevi [Canetti et al.] have constructed a cryptosystem that is secure in the random oracle model but which is not secure for any concrete choice of hash function. Fortunately, this construction is not one that would be used in practice.

The above procedure of reducing the security of a system to the solvability of some fundamental problem, such as the non-invertibility of a one-way function, is common in proofs of security. For example, in Section 7.5, we reduced certain questions for the ElGamal public key crytosystem to the solvability of Diffie-Hellman problems.

Section 8.5 shows that most hash functions do not behave as random oracles with respect to multicollisions. This indicates that some care is needed when applying the random oracle model.

The use of the random oracle model in analyzing a cryptosystem is somewhat controversial. However, many people feel that it gives some indication of the strength of the system. If a system is not secure in the random oracle
model, then it surely is not safe in practice. The controversy arises when a system is proved secure in the random oracle model. What does this say about the security of actual implementations? Different cryptographers will give different answers. However, at present, there seems to be no better method of analyzing the security that works widely.

8.7 Using Hash Functions to Encrypt

Cryptographic hash functions are some of the most widely used cryptographic tools, perhaps second only to block ciphers. They find application in many different areas of information security. Later, in Chapter 9 we shall see an application of hash functions to digital signatures, where the fact that they shrink the representation of data makes the operation of creating a digital signature more efficient. We shall now look at how they may be used to serve the role of a cipher by providing data confidentiality.

A cryptographic hash function takes an input of arbitrary length and provides a fixed-size output that appears random. In particular, if we have two inputs that are similar, then their hashes should be different. Generally, their hashes are very different. This is a property that hash functions share with good ciphers, and is a property that allows us to use a hash function to perform encryption.

The idea behind using a hash function to perform encryption is very similar to the operation of a Vernam-style cipher, which we saw an example of when we studied the output feedback mode (OFB) of a block cipher. Much like the block cipher did for OFB, the hash function creates a pseudorandom bit stream that is XORed with the plaintext to create a ciphertext.

In order to make a cryptographic hash function operate as a stream cipher, we need two components: a key shared between Alice and Bob, and an initialization vector. We shall soon address the issue of the initialization vector, but for now let us begin by assuming that Alice and Bob have established a shared secret $K_{AB}$.

Now, Alice could create a pseudorandom byte $x_1$ by taking the leftmost byte of the hash of $K_{AB}$, i.e. $x_1 = L_8(h(K_{AB}))$. She could then encrypt a byte of plaintext $p_1$ by XORing with the random byte $x_1$ to produce a byte of ciphertext

$$c_1 = p_1 \oplus x_1.$$

But if she has more than one byte of plaintext, then how should continue? We use feedback, much like we did in OFB mode. The next pseudorandom byte should be created by $x_2 = L_8(h(K_{AB} \parallel x_1))$. Then, the next ciphertext byte can be created by

$$c_2 = p_2 \oplus x_2.$$
8.7. Using Hash Functions to Encrypt

In general, the pseudorandom byte \( x_j \) is created by

\[
x_j = L_8 (h(K_{AB} || x_{j-1}))
\]

and encryption is simply XORing \( x_j \) with the plaintext \( p_j \). Decryption is a simple matter, as Bob must merely recreate the bytes \( x_j \) and XOR with the ciphertext \( c_j \) to get out the plaintext \( p_j \).

There is a simple problem with this procedure for encryption and decryption. What if Alice wants to encrypt a message on Monday, and a different message on Wednesday? How should she create the pseudorandom bytes? If she starts all over, then the pseudorandom sequence \( x_j \) on Monday and Wednesday will be the same. This is not desirable.

Instead, we must introduce some randomness to make certain the two bit streams are different. Thus, each time Alice sends a message, she should choose a random initialization vector, which we denote by \( x_0 \). She then starts by creating

\[
x_1 = L_8 (h(K_{AB} || x_0))
\]

and proceeding as before. But now, she must send \( x_0 \) to Bob, which she can do when she sends \( c_1 \). If Eve intercepts \( x_1 \), she is still not able to compute \( x_1 \) since she doesn’t know \( K_{AB} \). In fact, if \( h \) is a good hash function, then \( x_0 \) should give no information about \( x_1 \).

The idea of using a hash function to create an encryption procedure can be modified to create an encryption procedure that incorporates the plaintext, much in the same way as the CFB mode does.

Exercises

1. Let \( p \) be a prime and let \( \alpha \) be an integer with \( p \nmid \alpha \). Let \( h(x) \equiv \alpha^x \pmod{p} \). Explain why \( h(x) \) is not a good cryptographic hash function.

2. Let \( n = pq \) be the product of two distinct large primes and let \( h(x) = x^2 \pmod{n} \).
   (a) Why is \( h \) preimage resistant? (Of course, there are some values, such as 1, 4, 9, 16, \cdots for which it is easy to find a preimage. But usually it is difficult.)
   (b) Why is \( h \) not strongly collision-free?

3. Suppose a message \( m \) is divided into blocks of length 160 bits: \( m = M_1 || M_2 || \cdots || M_{\ell} \). Let \( h(x) = M_1 \oplus M_2 \oplus \cdots \oplus M_{\ell} \). Which of the properties (1), (2), (3) for a hash function does \( h \) satisfy?

4. In a family of four, what is the probability that no two people have birthdays in the same month? (Assume all months have equal probabilities.)

5. This problem derives the formula (8.1) for the probability of at least one match in a list of length \( r \) when there are \( N \) possible birthdays.
   (a) Let \( f(x) = \ln(1-x) + x \) and \( g(x) = \ln(1-x) + x + x^2 \). Show that
Chapter 8. Hash Functions

\[ f'(x) \leq 0 \text{ and } g'(x) \geq 0 \text{ for } 0 \leq x \leq 1/2. \]

(b) Using the facts that \( f(0) = g(0) = 0 \) and \( f \) is decreasing and \( g \) is increasing, show that

\[ -x - x^2 \leq \ln(1 - x) \leq -x \text{ for } 0 \leq x \leq 1/2. \]

(c) Show that if \( r \leq N/2 \), then

\[ -\frac{(r - 1)r}{2n} - \frac{r^3}{3N^2} \leq \sum_{j=1}^{r-1} \ln \left( 1 - \frac{j}{N} \right) \leq -\frac{(r - 1)r}{2N}. \]

\( (\text{Hint: } \sum_{j=1}^{r-1} j = (r - 1)r/2 \text{ and } \sum_{j=1}^{r-1} j^2 = (r - 1)r(2r - 1)/6 < r^3/3.) \)

(d) Assume \( r = \sqrt{2\lambda N} \), for some number \( \lambda \leq N/8 \) (this implies \( r \leq N/2 \)). Show that

\[ e^{-\lambda} e^{c_1/\sqrt{N}} \leq \prod_{j=1}^{r-1} \left( 1 - \frac{j}{N} \right) \leq e^{-\lambda} e^{c_2/\sqrt{N}}, \]

with \( c_1 = \sqrt{\lambda/2} - (2\lambda)^{3/2} \) and \( c_2 = \sqrt{\lambda/2} \).

(e) Observe that when \( N \) is large, \( e^{c/\sqrt{N}} \) is close to 1. Use this to show that if \( N \) is large, and \( r \) and \( N \) are as in part (d), then we have the approximation

\[ \prod_{j=1}^{r-1} \left( 1 - \frac{j}{N} \right) \approx e^{-\lambda}. \]

6. Suppose \( f(x) \) is a function with \( n \)-bit outputs and with inputs much larger than \( n \) bits (this implies that collisions must exist). We know that, with a birthday attack, we have probability 1/2 of finding a collision in approximately \( 2^{n/2} \) steps.

(a) Suppose we repeat the birthday attack until we find a collision. Show that the expected number of repetitions is

\[ \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 2 \]

(one way to evaluate the sum, call it \( S \), is to write \( S - \frac{1}{2}S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1 \).

(b) Assume that each evaluation of \( f \) takes time a constant times \( n \). (This is realistic since the inputs needed to find collisions can be taken to have \( 2n \) bits, for example.) Show that the expected time to find a collision for the function \( f \) is a constant times \( n 2^{n/2} \).
8.7. Using Hash Functions to Encrypt

(c) Show that the expected time to find the messages \(m_0, m'_0, \ldots, m_t, m'_t\) in Section 8.5 is a constant times \(tn^{2n/2}\).

7. Suppose we have an iterative hash function, as in Section 8.5, but suppose we adjust the function slightly at each iteration. For concreteness, suppose the algorithm proceeds as follows. There is a compression function \(f\) that operates on inputs of a fixed length. There is also a function \(g\) that yields outputs of a fixed length, and there is a fixed initial value \(IV\). The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message \(M\) into blocks \(M_1, M_2, \ldots, M_\ell\).
2. Let \(H_0\) be the initial value \(IV\).
3. For \(i = 1, 2, \ldots, \ell\), let \(H_i = f(H_{i-1}, M_i || g(i))\).
4. Let \(H(M) = H_\ell\).

Show that the method of Section 8.5 can be used to produce multicollisions.

8. The initial values \(K_t\) in SHA-1 might appear to be random. Here is how they were chosen.

(a) Compute \(\lfloor 2^{30} \sqrt{2} \rfloor\) and write the answer in hexadecimal. The answer should be \(K_0\).
(b) Do a similar computation with \(\sqrt{2}\) replaced by \(\sqrt{3}\), \(\sqrt{5}\), and \(\sqrt{10}\) and compare with \(K_{20}, K_{40}, \text{and } K_{60}\).

9. Let \(E_K\) be an encryption function with \(N\) possible keys \(K\), \(N\) possible plaintexts, and \(N\) possible ciphertexts. Suppose that, for each pair \((K_1, K_2)\) of keys, there is a key \(K_3\) such that \(E_{K_1}(E_{K_2}(m)) = E_{K_3}(m)\) for all plaintexts \(m\). Assume also that for every plaintext-ciphertext pair \((m, c)\), there is usually only one key \(K\) such that \(E_K(m) = c\). Suppose that you know a plaintext-ciphertext pair \((m, c)\). Give a birthday attack that usually finds the key \(K\) in approximately \(2\sqrt{N}\) steps. (It Remark: This is much faster than brute force.)

Computer Problems

1. (a) If there are 30 people in a classroom, what is the probability that at least two have the same birthday? Compare this to the approximation given by formula (8.1).
(b) How many people should there be in a classroom in order to have a 99% chance that at least two have the same birthday? (Hint: Use the
approximation to obtain an approximate answer. Then use the product, for various numbers of people, until you find the exact answer.)

(c) How many people should there be in a classroom in order to have 100% probability that at least two have the same birthday?

2. A professor posts the grades for a class using the last four digits of the Social Security number of each student. In a class of 200 students, what is the probability that at least two students have the same four digits?