Chapter 11 Hash Functions

11.1 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 13, hash functions will be used in digital signature algorithms. They also play a role in security protocols in Chapter 15, 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, 256 bits as depicted in Figure 11.1. Certain properties should be satisfied:

1. Given a message m, the message digest $h(m)$ can be calculated very quickly.
2. Given a 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). Note that if y is the message digest of some message, we are not trying to find this message. We are only looking for some m' with $h(m') = y$.
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 resistant**).

Figure 11.1 A Hash Function

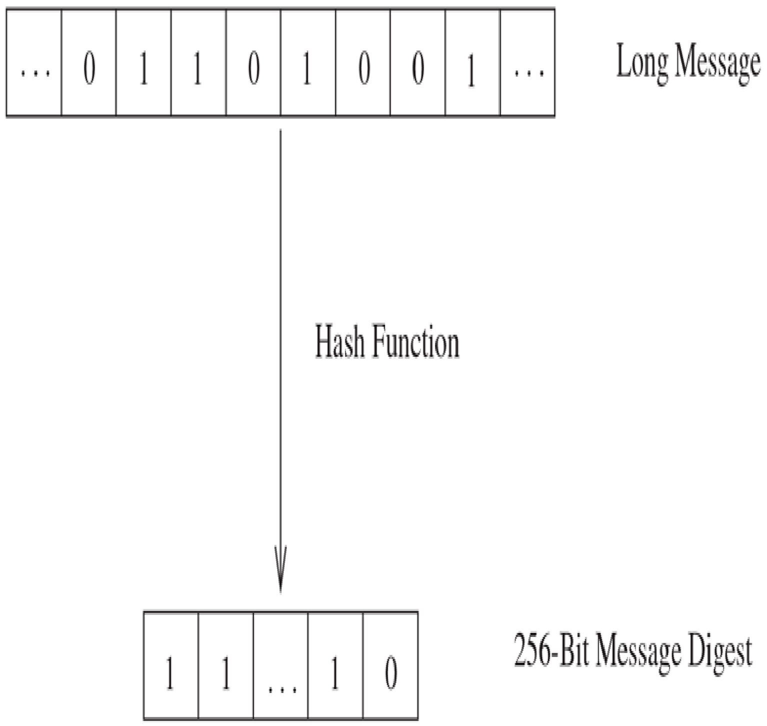


Figure 11.1 Full Alternative Text

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.

Preimage resistance and collision resistance are closely related, but we list them separately because they are used in slightly different circumstances. The following argument shows that, for our hash functions, collision resistance implies preimage resistance: Suppose H is not preimage resistant. Take a random x and compute $y = H(x)$. If H is not preimage resistant, we can quickly find x' with $H(x') = y = H(x)$. Because H is many-to-one, it is likely that $x \neq x'$, so we have a collision, contradicting the collision resistance of H. However, there are examples that show that for arbitrary functions, collision resistance does not imply preimage resistance. See Exercise 12.

In practice, it is sometimes sufficient to weaken (3) to require H to be **weakly collision resistant**. This means that given x, it is computationally infeasible to find $x' \neq x$ with $H(x') = H(x)$. This property is also called **second preimage resistance**.

Requirement (3) is the hardest one to satisfy. In fact, in 2004, Wang, Feng, Lai, and Yu (see [Wang et al.]) found many examples of collisions for the popular hash functions MD4, MD5, HAVAL-128, and RIPEMD. The MD5 collisions have been used by Ondrej Mikle [Mikle] to create two different and meaningful documents with the same hash, and the paper [Lenstra et al.] shows how to produce examples of X.509 certificates (see Section 15.5) with the same MD5 hash (see also Exercise 15). This means that a valid digital signature (see Chapter 13) on one certificate is also valid for the other certificate, hence it is impossible for someone to determine which is the certificate that was legitimately signed by a Certification Authority. It has been reported that weaknesses in MD5 were part of the design of the Flame malware, which attacked several computers in the Middle East, including Iran's oil industry, from 2010 to 2012.

In 2005, Wang, Yin, and Yu [Wang et al. 2] predicted that collisions could be found for the hash function SHA-1 with around 2^{69} calculations, which is much better than the expected 2^{80} calculations required by the birthday attack (see Section 12.1). In addition, they found collisions in a smaller 60-round version of SHA-1. These weaknesses were a cause for concern for using these hash algorithms and led to research into replacements. Finally, in 2017, a joint project between CWI Amsterdam and Google Research found collisions for SHA-1 [Stevens et al.]. Although SHA-1 is still common, it is starting to be used less and less.

One of the main uses of hash functions is in digital signatures. Since the length of a digital signature is often at least as long as the document being signed, it is much more efficient to sign the hash of a document rather than the full document. This will be discussed in Chapter 13.

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 never 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-resistance properties of h.

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 resistant.

Example

The following example, sometimes called the discrete log hash function, is due to Chaum, van Heijst, and Pfitzmann [Chaum et al.]. 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 15 in Chapter 13). We now choose two primitive roots α_1 and α_2 for p. Since α_1 is a primitive root, there exists a such that $\alpha_1^a \equiv \alpha_2 (\bmod\, p)$. 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 usually 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

$$h(m) \equiv \alpha_1^{x_0} \alpha_2^{x_1} \pmod{p}.$$

The following shows that the function h is probably strongly collision resistant.

Proposition

If we know messages $m \neq m'$ with $h(m) = h(m')$, then we can determine the discrete logarithm $a = L_{\alpha_1}(\alpha_2)$.

Proof

Write $m = x_0 + x_1 q$ and $m' = x'_0 + x'_1 q$. Suppose

$$\alpha_1^{x_0}\alpha_2^{x_1} \equiv \alpha_1^{x'_0}\alpha_2^{x'_1} \pmod{p}.$$

Using the fact that $\alpha_2 \equiv \alpha_1^a \pmod{p}$, we rewrite this as

$$\alpha_1^{a(x_1-x'_1)-(x'_0-x_0)} \equiv 1 \pmod{p}.$$

Since α_1 is a primitive root mod p, we know that $\alpha_1^k \equiv 1 \pmod{p}$ 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 $d = \gcd(x_1 - x\prime_1, p-1)$. There are exactly d solutions to the preceding congruence (see Subsection 3.3.1), 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 two possibilities for a. Calculate α_1^a for each possibility; only one of them will yield α_2. 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.

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 an algorithm 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.

11.2 Simple Hash Examples

There are many families of hash functions. The discrete log hash function that we described in the previous section is too slow to be of practical use. One reason is that it employs modular exponentiation, which makes its computational requirements about the same as RSA or ElGamal. Even though modular exponentiation is fast, it is not fast enough for the massive inputs that are used in some situations. The hash functions described in this section and the next are easily seen to involve only very basic operations on bits and therefore can be carried out much faster than procedures such as modular exponentiation.

We now describe the basic idea behind many cryptographic hash functions by giving a simple hash function that shares many of the basic properties of 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 denote these n-bit blocks by m_j, and thus represent $m = [m_1, m_2, \cdots, m_l]$. 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 jth block m_j as a row vector

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

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 ith column of the matrix, that is $h_i = m_{1i} \oplus m_{2i} \oplus \ \cdots \ \oplus m_{li}$. We may visualize this as

$$\begin{array}{cccc} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{l1} & m_{l2} & \cdots & m_{ln} \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ \oplus & \oplus & \oplus & \oplus \\ \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ [\ c_1 & c_2 & \cdots & c_n\ ] \end{array} \quad = \quad h(m).$$

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 (Exercise 9).

Practical cryptographic hash functions typically make use of several other bit-level operations in order to make it more difficult to find collisions. Section 11.4 contains many examples of such operations.

One operation that is often used is bit rotation. We define the right rotation operation

$$R^y(m)$$

as the result of shifting m to the right by y positions and wrapping the rightmost y bits around, placing them in leftmost y bit locations. Then $R^{-y}(m)$ gives a similar rotation of m by y places to the left.

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 $m'_j = R^{-(j-1)}(m_j)$. We may now arrange the m'_j in columns and define a new, simple hash function by XORing these columns. Thus, we get

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

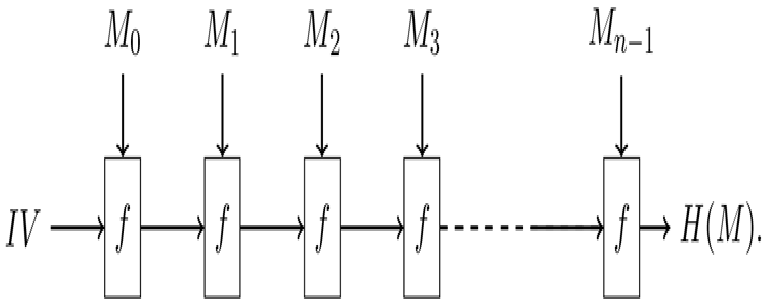
This new hash function involving rotations mixes the bits in one position with those in another, but it is still easy to find collisions (Exercise 9). Building a cryptographic hash requires considerably more tricks than just rotating. In later sections, we describe hash functions that are used in practice. They use the techniques of the present section, coupled with many more ways of mixing the bits.

11.3 The Merkle-Damgård Construction

Until recently, most hash functions used a form of the Merkle-Damgård construction. It was invented independently by Ralph Merkle in 1979 and Ivan Damgård in 1989. The main ingredient is a function f, usually called a **compression function**. It takes two bitstrings as inputs, call them H and M, and outputs a bitstring $H' = f(H, M)$ of the same length as H. For example, M could have length 512 and H could have length 256. These are the sizes that the hash function SHA-256 uses, and we'll use them for concreteness. The message that is to be hashed is suitably padded so that its length is a multiple of 512, and then broken into n blocks of length 512:

$$M_0||M_1||M_2||\cdots||M_{n-1}.$$

An initial value IV is set. Then the blocks are fed one-by-one into f and the final output is the hash value:



11.3-1 Full Alternative Text

This construction is very natural: The blocks are read from the message one at a time and stirred into the mix with the previous blocks. The final result is the hash value.

Over the years, some disadvantages of the method have been discovered. One is called the **length extension attack**. For example, suppose Alice wants to ensure that her message M to Bob has not been tampered with. They both have a secret key K, so Alice prepends M with K to get $K||M$. She sends both M and $H(K||M)$ to Bob. Since Bob also knows K, he computes $H(K||M)$ and checks that it agrees with the hash value sent by Alice. If so, Bob concludes that the message is authentic.

Since Eve does not know K, she cannot send her own message M' along with $H(K||M')$. But, because of the iterative form of the hash function, Eve can append blocks M'' to M if she intercepts Alice's communication. Then Eve sends

$$M||M'' \text{ and } H(K||M||M'') = f(H(K||M), M'')$$

to Bob. Since she knows $H(K||M)$, she can produce a message that Bob will regard as authentic. Of course, this attack can be thwarted by using $M||K$ instead of $K||M$, but it points to a weakness in the construction.

However, using $H(M||K)$ might also cause problems if Eve discovers a way of producing collisions for H. Namely, Eve finds M_1 and M_2 with $H(M_1) = H(M_2)$. If Eve can arrange this so that M_1 is a good message and M_2 is a bad message (see Section 12.1), then Eve arranges for Alice to authenticate M_1 by computing $H(M_1||K)$, which equals $H(M_2||K)$. This means that Alice has also authenticated M_2.

Another attack was given by Daum and Luks in 2005. Suppose Alice is using a high-level document language such as PostScript, which is really a program rather than just a text file. The file begins with a preamble that identifies the file as PostScript and gives some instructions. Then the content of the file follows.

Suppose Eve is able to find random strings R_1 and R_2 such that

$$H\,(\text{preamble; put}(R_1)) = H\,(\text{preamble; put}(R_2)),$$

where $\text{put}(R_i)$ instructs the PostScript program to put the string R_i in a certain register. In other words, we are assuming that Eve has found a collision of this form. If any string S is appended to these messages, there is still a collision

$$H\,(\text{preamble; put}(R_1)||S) = H\,(\text{preamble; put}(R_2)||S)$$

because of the iterative nature of the hash algorithm (we are ignoring the effects of padding).

Of course, Eve has an evil document T_1, perhaps saying that Alice (who is a bank president) gives Eve access to the bank's vault. Eve also produces a document T_2 that Alice will be willing to sign, for example, a petition to give bank presidents tax breaks. Eve then produces two messages:

$$\begin{aligned} Y_1 &= \text{preamble; put}(R_1); \text{put}(R_1); \text{if } (=) \text{ then } T_1 \text{ else } T_2 \\ Y_2 &= \text{preamble; put}(R_2); \text{put}(R_1); \text{if } (=) \text{ then } T_1 \text{ else } T_2. \end{aligned}$$

For example, Y_2 puts R_1 into a stack, then puts in R_2. They are not equal, so T_2 is produced. Eve now has two Postscript files, Y_1 and Y_2, with $H(Y_1) = H(Y_2)$. As we'll see in Chapter 13, it is standard for Alice to sign the hash of a message rather than the message itself. Eve shows Y_2 to Alice, who compiles it. The output is the petition that Alice is happy to sign. So Alice signs $H(Y_2)$. But this means Alice has also signed $H(Y_1)$. Eve takes Y_1 to the bank, along with Alice's signature on its hash value. The security officer at the bank checks that the signature is valid, then opens the document, which says that Alice grants Eve access to the bank's vault. This potentially costly forgery relies on Eve being able to find a collision, but again it shows a weakness in the construction if there is a possibility of finding collisions.

11.4 SHA-2

In this section and the next, we 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 used in practice. The most notable of these are the Secure Hash Algorithm (SHA) family, the Message Digest (MD) family, and the RIPEMD-160 message digest algorithm. 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.

The Secure Hash Algorithm was developed by the National Security Agency (NSA) and given to 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 a revised standards document (FIPS 180-1) that was released in 1995. This revised document describes the improved version, SHA-1, which for several years was the hash algorithm recommended by NIST. However, weaknesses started to appear and in 2017, a collision was found (see the discussion in Section 11.1). SHA-1 is now being replaced by a series of more secure versions called SHA-2. They still use the Merkle-Damgård construction. In the next section, we'll meet SHA-3, which uses a different construction.

The reader is warned that the discussion that follows is fairly technical and is provided in order to give the flavor of what happens inside a hash function.

The SHA-2 family consists of six algorithms: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The last three digits indicate the number of bits in the output. We'll describe SHA-256. The other five are very similar.

SHA-256 produces a 256-bit hash and is built upon the same design principles as MD4, MD5, and SHA-1. 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, $M = M^{(1)}||M^{(2)}||\cdots||M^{(N)}$, 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' that 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_N 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 the SHA family and the MD family is that for SHA the input bits are used more often during the course of the hash function than they are for MD4 and MD5. This more conservative approach makes the design of SHA-1 and SHA-2 more secure than either MD4 or MD5, but also makes it a little slower.

In the description of the hash algorithm, we need the following operations on strings of 32 bits:

1. $X \wedge Y =$ bitwise "and", which is bitwise multiplication mod 2, or bitwise minimum.

2. $X \vee Y$ = bitwise "or", which is bitwise maximum.

3. $X \oplus Y$ = bitwise addition mod 2.

4. $\neg X$ changes 1s to 0s and 0s to 1s .

5. $X + Y$ = addition of X and Y mod 2^{32}, where X and Y are regarded as integers mod 2^{32}.

6. $R^n(X)$ = rotation of X to the right by n positions (the end wraps around to the beginning).

7. $S^n(X)$ = shift of X to the right by n positions, with the first n bits becoming 0s (so the bits at the end disappear and do not wrap around).

We also need the following functions that operate on 32-bit strings:

$$\begin{array}{rcl} Ch(X,Y,Z) &=& (X \wedge Y) \oplus (\neg X \wedge Z) \\ Maj(X,Y,Z) &=& (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \\ \Sigma_0(X) &=& R^2(X) \oplus R^{13}(X) \oplus R^{22}(X) \\ \Sigma_1(X) &=& R^6(X) \oplus R^{11}(X) \oplus R^{25}(X) \\ \sigma_0(X) &=& R^7(X) \oplus R^{18}(X) \oplus S^3(X) \\ \sigma_1(X) &=& R^{17}(X) \oplus R^{19}(X) \oplus S^{10}(X). \end{array}$$

Define initial hash values $H_1^{(0)}, H_2^{(0)}, \ldots, H_8^{(0)}$ as follows:

$$\begin{array}{llll} H_1^{(0)} = \text{6A09E667} & H_2^{(0)} = \text{BB67AE85} & H_3^{(0)} = \text{3C6EF372} & H_4^{(0)} = \text{A54FF53A} \\ H_5^{(0)} = \text{510E527F} & H_6^{(0)} = \text{9B05688C} & H_7^{(0)} = \text{1F83D9AB} & H_8^{(0)} = \text{5BE0CD19} \end{array}$$

The preceding are written in **hexadecimal notation**. Each digit or letter represents a string of four bits:

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

For example, BA1 equals
$11 * 16^2 + 10 * 16^1 + 1 = 2977.$

These initial hash values are obtained by using the first eight digits of the fractional parts of the square roots of the first eight primes, expressed as "decimals" in base 16. See Exercise 7.

We also need sixty-four 32-bit words

$$K_0 = \text{428A2F98}, \quad K_1 = \text{71374491}, \quad \ldots, \quad K_{63} = \text{C67178f2}.$$

They are the first eight hexadecimal digits of the fractional parts of the cube roots of the first 64 primes.

Padding and Preprocessing

SHA-256 begins by taking the original message and padding it with the bit 1 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. (This restricts the messages to length less than $2^{64} \approx 10^{19}$ bits, which is not a problem in practice.)

For example, if the original message has 2800 bits, we add a 1 and 207 0s to obtain a new message of length $3008 = 6 \times 512 - 64$. Since $2800 = 101011110000_2$ in binary, we append fifty-two 0s followed by 101011110000 to obtain a message of length 3072. This is broken into six blocks of length 512.

Break the message with padding into N blocks of length 512:

$$M^{(1)} \big|\big| M^{(2)} \big|\big| \cdots \big|\big| M^{(N)}.$$

The hash algorithm inputs these blocks one by one. In the algorithm, each 512-bit block $M^{(i)}$ is divided into sixteen 32-bit blocks:

$$M^{(i)} \; = \; M_0^{(i)} \big|\big| M_1^{(i)} \big|\big| \; \cdots \; \big|\big| M_{15}^{(i)}.$$

The Algorithm

There are eight 32-bit registers, labeled a, b, c, d, e, f, g, h. These contain the intermediate hash values. The algorithm inputs a block of 512 bits from the message in Step 11, and in Steps 12 through 24, it stirs the bits of this block into a mix with the bits from the current intermediate hash values. After 64 iterations of this stirring, the algorithm produces an output that is added (mod 2^{32}) onto the previous intermediate hash values to yield the new hash values. After all of the blocks of the message have been processed, the final intermediate hash values give the hash of the message.

The basic building block of the algorithm is the set of operations that take place on the subregisters in Steps 15 through 24. They take the subregisters and operate on them using rotations, XORs, and other similar operations.

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

Algorithm 3 The SHA-256 algorithm

```
• 1: for $i$ from 1 to $N$ do
  ▹ This initializes the registers with the $(i-1)$st intermediate hash value
• 2: $a \leftarrow H_1^{(i-1)}$
• 3: $b \leftarrow H_2^{(i-1)}$
• 4: $c \leftarrow H_3^{(i-1)}$
• 5: $d \leftarrow H_4^{(i-1)}$
• 6: $e \leftarrow H_5^{(i-1)}$
• 7: $f \leftarrow H_6^{(i-1)}$
```

```
8: $g \leftarrow H_7^{(i-1)}$
9: $h \leftarrow H_8^{(i-1)}$
10: **for** $k$ from 0 to 15 **do**
11: $W_k \leftarrow M_k^{(i)}$ ▹ This is where the message blocks are entered.
12: **for** $j_1$ from 16 to 63 **do**
13: $W_{j_1} \leftarrow \sigma_1(W_{j_1-2}) + W_{j_1-7} + \sigma_0(W_{j_1-15}) + W_{j_1-16}$
14: **for** $j$ from 0 to 63 **do**
15: $T_1 \leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j$
16: $T_2 \leftarrow \Sigma_0(a) + Maj(a, b, c)$
17: $h \leftarrow g$
18: $g \leftarrow f$
19: $f \leftarrow e$
20: $e \leftarrow d + T_1$
21: $d \leftarrow c$
22: $c \leftarrow b$
23: $b \leftarrow a$
24: $a \leftarrow T_1 + T_2$
25: $H_1^{(i)} \leftarrow a + H_1^{(i-1)}$ ▹ These are the $i$th intermediate hash values
26: $H_2^{(i)} \leftarrow b + H_2^{(i-1)}$
27: $H_3^{(i)} \leftarrow c + H_3^{(i-1)}$
28: $H_4^{(i)} \leftarrow d + H_4^{(i-1)}$
29: $H_5^{(i)} \leftarrow e + H_5^{(i-1)}$
30: $H_6^{(i)} \leftarrow f + H_6^{(i-1)}$
31: $H_7^{(i)} \leftarrow g + H_7^{(i-1)}$
32: $H_8^{(i)} \leftarrow h + H_8^{(i-1)}$
33: $H(m) = H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)} \| H_8^{(N)}$
```

- 34: **return** $H(m)$

11.5 SHA-3/Keccak

In 2006, NIST announced a competition to produce a new hash function to serve alongside SHA-2. The new function was required to be at least as secure as SHA-2 and to have the same four output possibilities. Fifty-one entries were submitted, and in 2012, **Keccak** was announced as the winner. It was certified as a standard by NIST in 2012 in FIPS-202. (The name is pronounced “ketchak”. It has been suggested that the name is related to “Kecak,” a type of Balinese dance. Perhaps the movement of the dancers is analogous to the movement of the bits during the algorithm.) This algorithm became the hash function SHA-3.

The SHA-3 algorithm was developed by Guido Bertoni, Joan Daemen, and Gilles Van Assche from STMicroelectronics and Michaël Peeters from NXP Semiconductors. It differs from the Merkle-Damgård construction and is based on the theory of **Sponge Functions**. The idea is that the first part of the algorithm absorbs the message, and then the hash value is squeezed out. Here is how it works. The **state** of the machine is a string of b bits, which is fed to a function f that takes an input of b bits and outputs a string of b bits, thus producing a new state of the machine. In contrast to the compression functions in the Merkle-Damgård construction, the function f is a one-to-one function. Such a function could not be used in the Merkle-Damgård situation since the number of input bits (from M_i and the previous step) is greater than the number of output bits. But the different construction in the present case allows it.

Parameters r (“the rate”) and c (“the capacity”) are chosen so that $r + c = b$. The message (written in binary) is padded so that its length is a multiple of r, then is broken into n blocks of length r:

$$M = M_0||M_1||M_2|| \cdots ||M_{n-1}.$$

To start, the state is initialized to all 0s. The absorption stage is the first part of Figure 11.2.

Figure 11.2 A Sponge Function

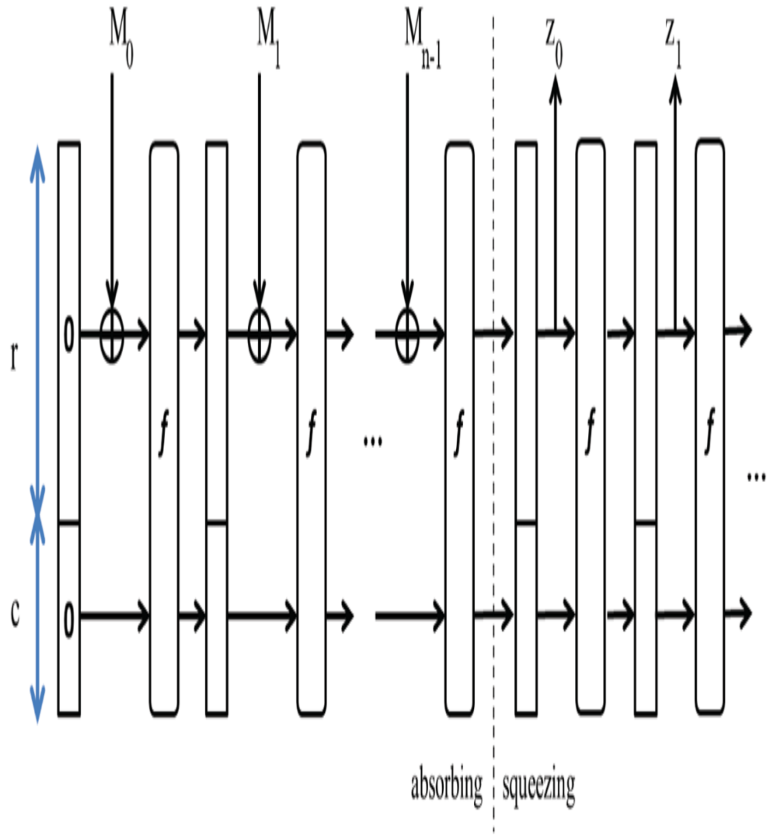


Figure 11.2 Full Alternative Text

After the absorption is finished, the hash value is squeezed out: r bits are output and truncated to the d bits that are used as the hash value. This is the second part of Figure 11.2.

Producing a SHA-3 hash value requires only one squeeze. However, the algorithm can also be used with multiple squeezes to produce arbitrarily long pseudorandom bitstrings. When it is used this way, it is often called SHAKE (= Secure Hash Algorithm with Keccak).

The “length extension” and collision-based attacks of Section 11.3 are less likely to succeed. Suppose two messages yield the same hash value. This means that when the absorption has finished and the squeezing stage is starting, there are d bits of the state that agree with these bits for the other message. But there are at least c bits that are not output, and there is no reason that these c bits match. If, instead of starting the squeezing, you do another round of absorption, the differing c bits will cause the subsequent states and the outputted hash values to differ. In other words, there are at least 2^c possible internal states for any given d-bit output H.

SHA-3 has four different versions, named SHA3-224, SHA3-256, SHA3-384, and SHA3-512. For SHA3-m, the m denotes the security level, measured in bits. For example, SHA3-256 is expected to require around 2^{256} operations to find a collision or a preimage. Since $2^{256} \approx 10^{77}$, this should be impossible well into the future. The parameters are taken to be

$$b = 1600, \quad d = m, \quad c = 2m, \quad r = 1600 - c.$$

For SHA3-256, these are

$$b = 1600, \quad d = 256, \quad c = 512, \quad r = 1088.$$

The same function f is used in all versions, which means that it is easy to change from one security level to another. Note that there is a trade-off between speed and security. If the security parameter m is increased, then r decreases, so the message is read slower, since it is read in blocks of r bits.

In the following, we concentrate on SHA3-256. The other versions are obtained by suitably varying the parameters.

For more details, see [FIPS 202].

The Padding. We start with a message M. The message is read in blocks of $r = 1088$ bits, so we want the message to have length that is a multiple of 1088. But first the message is padded to $M||01$. This is for “domain separation.” There are other uses of the Keccak algorithm such as SHAKE (mentioned above), and for these other purposes, M is padded differently, for example with 1111. This initial padding makes it very likely that using M in different situations yields different outputs. Next, “10*1 padding” is used. This means that first a 1 is appended to M011 to yield $M||$011 . Then sufficiently many 01 s are appended to make the total length one less than a multiple of 1088. Finally, a 1 is appended. We can now divide the result into blocks of length 1088.

Why are these choices made for the padding? Why not simply append enough 0s s to get the desired length? Suppose that $M_1 = 1010111$. Then $M_1||10 = 101011110$. Now append 1079 zeros to get the block to be hashed. If $M_2 = 1010$ is being used in SHAKE, then $M_2||1111$ is padded with 1080 zeros to yield the same block. This means that the outputs for M_1 and M_2 are equal. The padding is designed to avoid all such situations.

Absorption and Squeezing. From now on, we assume that the padding has been done and we have N blocks of length 1088:

$$M_0||M_1||\cdots||M_{N-1}.$$

The absorption now proceeds as in Figure 11.2 (we describe the function f later).

1. The initial state S is a string of 0s s of length 1600.
2. For $j = 0$ to $N-1$, let $S = f(S \oplus (M_j||0^c))$, where 0^c denotes a string of 0s of length $c = 512$. What this does is XOR the message block M_j with the first $r = 1088$ bits of S, and then

apply the function f. This yields an updated state S, which is modified during each iteration of the index j.

3. Return S.

The squeezing now proceeds as in Figure 11.2:

1. Input S and let Z be the empty string.
2. While $\mathrm{Length(Z)} < \mathrm{td}$ (where $d = 256$ is the output size)
    1. Let $Z = Z||Trunc_r(S)$, where $\mathrm{Trunc}_r\mathrm{(S)}$ denotes the first $r = 1088$ bits of S.
    2. $S = f(S)$.
3. Return $Trunc_d(Z)$.

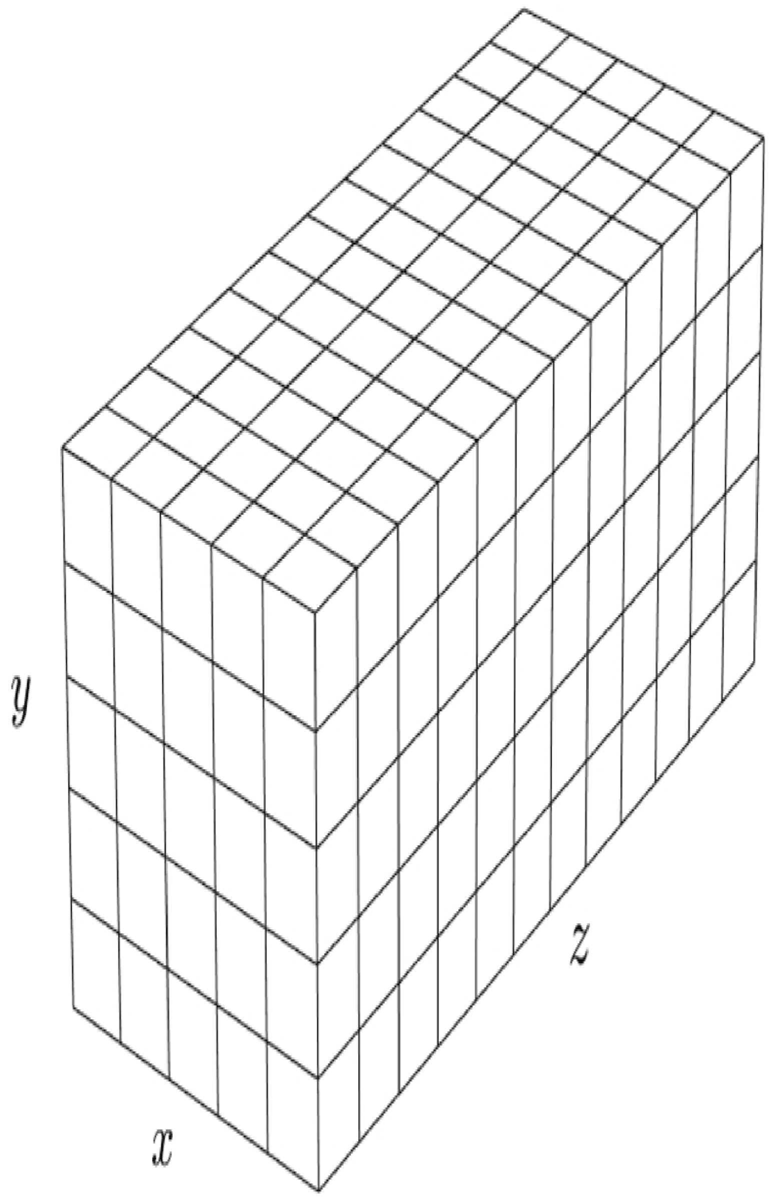
The bitstring $\mathrm{Trunc}_{256}(Z)$ is the 256-bit hash value SHA-$3(M)$. For the hash value, we need only one squeeze to obtain Z. But the algorithm could also be used to produce a much longer pseudorandom bitstring, in which case several squeezes might be needed.

The function $\boldsymbol{f}$. The main component of the algorithm is the function f, which we now describe. The input to f is the 1600-bit state of the machine

$$S = S[0] \,||\, S[1] \,||\, S[2] \,||\, \cdots \,||\, S[1599],$$

where each $S[j]$ is a bit. It's easiest to think of these bits as forming a three-dimensional $5 \times 5 \times 64$ array $A[x, y, z]$ with coordinates (x, y, z) satisfying

$$0 \le x \le 4, \quad 0 \le y \le 4, \quad 0 \le z \le 63.$$



A “column” consists of the five bits with fixed x, z. A “row” consists of the five bits with fixed y, z. A “lane” consists of the 64 bits with fixed x, y.

When we write “for all x, z” we mean for $0 \leq x \leq 4$ and $0 \leq z \leq 63$, and similarly for other combinations of x, y, z.

The correspondence between S and A is given by

$$A[x, y, z] = S[64(5y + x) + z]$$

for all $x,\ y,\ z$. For example, $A[1,\ 2,\ 3] = S[707]$. The ordering of the indices could be described as "lexicographic" order using $y,\ x,\ z$ (not $x,\ y,\ z$), since the index of S corresponding to $x_1,\ y_1,\ z_1$ is smaller than the index for $x_2,\ y_2,\ z_2$ if $y_1 x_1 z_1$ precedes $y_2 x_2 z_2$ in "alphabetic order."

The coordinates $x,\ y$ are taken to be numbers mod 5, and z is mod 64. For example $A[7,\ -1,\ 86]$ is taken to be $A[2,\ 4,\ 22]$, since $7 \equiv 2$ mod 5, $-1 \equiv 4$ mod 5, and $86 \equiv 22$ mod 64.

The computation of f proceeds in several steps. The steps receive the array A as input and they output a modified array A' to replace A.

The following steps I through V are repeated for $i = 0$ to $i = 23$:

1. The first step XORs the bits in a column with the parities of two nearby columns.
    1. For all $x,\ z$, let

       $C[x,\ z] = A[x,\ 0,\ z] \oplus A[x,\ 1,\ z] \oplus A[x,\ 2,\ z] \oplus A[x,\ 3,\ z] \oplus A[x,\ 4,\ z]$.

       This gives the "parity" of the bitstring formed by the five bits in the $x,\ z$ column.
    2. For all $x,\ z$, let
       $D[x,\ z] = C[x-1,\ z] \oplus C[x+1,\ z-1]$.
    3. For all $x,\ y,\ z$, let $A'[x,\ y,\ z] = A[x,\ y,\ z] \oplus D[x,\ z]$.
2. The second step rotates the 64 bits in each lane by an amount depending on the lane:
    1. For all z, let $A'[0,\ 0,\ z] = A[0,\ 0,\ z]$.
    2. Let $(x,\ y) = (1,\ 0)$.
    3. For $t = 0$ to 23
        1. For all z, let
           $A'[x,\ y,\ z] = A[x,\ y,\ z-(t+1)(t+2)/2]$
           .

2. Let $(x, y) = (y, 2x + 3y)$

4. Return A'.

For example, consider the bits with coordinates of the form $(1, 0, z)$. They are handled by the case $t = 0$ and we have $A'[1, 0, z] = A[1, 0, z - 1]$, so the bits in this lane are rotated by one position. Then, in Step 3(b), $(x, y) = (1, 0)$ is changed to $(0, 2)$ for the iteration with $t = 1$. We have $A'[0, 2, z] = A[0, 2, z - 3]$, so this lane is rotated by three positions. Then (x, y) is changed to $(2, 6)$, which is reduced mod 5 to $(2, 1)$, and we pass to $t = 2$, which gives a rotation by six (the rotations are by what are known as "triangular numbers"). After $t = 23$, all of the lanes have been rotated.

3. The third step rearranges the positions of the lanes:

   1. For all x, y, z, let $A'[x, y, z] = A[x + 3y, x, z]$.

   Again, the coordinate $x + 3y$ should be reduced mod 5.

4. The next step is the only nonlinear step in the algorithm. It XORs each bit with an expression formed from two other bits in its row.

   1. For all x, y, z, let

   2. $A'[x, y, z] = A[x, y, z] \oplus (A[x + 1, y, z] \oplus 1)(A[x + 2, y, z])$.

   3. The multiplication is multiplying two binary bits, hence is the same as the *AND* operator.

5. Finally, some bits in the $(0, 0)$ lane are modified.

   1. For all x, y, z, let $A'[x, y, z] = A[x, y, z]$.

   2. Set $RC = 0^{24}$.

   3. For $j = 0$ to 6, let $RC[2^j - 1] = rc(j + 7i)$, where rc is an auxiliary function defined below.

   4. For all z, let $A'[0, 0, z] = A'[0, 0, z] \oplus RC[z]$.

   5. Return A'.

After I through V are completed for one value of i, the next value of i is used and I through V are repeated for the new i, through $i = 23$. The final output is the new array A, which yields a new bitstring S of length 1600.

This completes the description of the function f, except that we still need to describe the auxiliary function rc.

The function rc takes an integer t mod 255 as input and outputs a bit according to the following algorithm:

```
1. If $t \equiv 0$ mod 255, return 1 . Else
2. $R =$10000000
3. For $k = 1$ to $t$ mod 255
    1. Let $R = 0||R$
    2. Let $R[0] = R[0] \oplus R[8]$
    3. Let $R[4] = R[4] \oplus R[8]$
    4. Let $R[5] = R[5] \oplus R[8]$
    5. Let $R[6] = R[6] \oplus R[8]$
    6. Let $R = \mathrm{Trunc}_8(R)$.
4. Return $R[0]$.
```

The bit that is outputted is $rc(t)$.

11.6 Exercises

1. Let p be a prime and let α 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.
    1. Alice claims that she knows who will win the next World Cup. She takes the name of the team, T, and encrypts it with a one-time pad K, and sends $C = T \oplus K$ to Bob. After the World Cup is finished, Alice reveals K, and Bob computes $T = C \oplus K$ to determine Alice's guess. Why should Bob not believe that Alice actually guessed the correct team, even if $T = C \oplus K$ is correct?

    2. To keep Alice from changing K, Bob requires Alice to send not only $C = T \oplus K$ but also $H(K)$, where H is a good cryptographic hash function. How does the use of the hash function convince Bob that Alice is not changing K?

    3. In the procedure in (b), Bob receives C and $H(K)$. Show how he can determine Alice's prediction, without needing Alice to send K? (*Hint:* There are fewer than 100 teams T that could win the World Cup.)

3. Let $n = pq$ be the product of two distinct large primes and let $h(x) = x^2 \pmod{n}$.

    1. 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.)

    2. Why is h not strongly collision resistant?

4. Let $H(x)$ be a cryptographic hash function. Nelson tries to make new hash functions.

    1. He takes a large prime p and a primitive root α for p. For an input x, he computes $\beta \equiv \alpha^x \pmod{p}$, then sets $H_2(x) = H(\beta)$. The function H_2 is not fast enough to be a hash function. Find one other property of hash functions that fails for H_2 and one that holds for H_2, and justify your answers.

    2. Since his function in part (a) is not fast enough, Nelson tries using $H_3 = H(x \bmod p)$. This is very fast. Find

one other property of hash functions that holds for H_3 and one that fails for H_3, and justify your answers.

5. 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 and which does it not satisfy? Justify your answers.

6. One way of storing and verifying passwords is the following. A file contains a user's login id plus the hash of the user's password. When the user logs in, the computer checks to see if the hash of the password is the same as the hash stored in the file. The password is not stored in the file. Assume that the hash function is a good cryptographic hash function.

    1. Suppose Eve is able to look at the file. What property of the hash function prevents Eve from finding a password that will be accepted as valid?

    2. When the user logs in, and the hash of the user's password matches the hash stored in the file, what property of the hash function says that the user probably entered the correct password? (*Hint:* Your answers to (a) and (b) should not be the same.)

7. The initial values $H_k^{(0)}$ in SHA-256 are extracted from the hexadecimal expansions of the fractional parts of the square roots of the first eight primes. Here is what that means.

    1. Compute $\lfloor 2^{32}(\sqrt{2}-1)\rfloor$ and write the answer in hexadecimal. The answer should be $H_1^{(0)}$.

    2. Do a similar computation with $\sqrt{2}$ replaced by $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{19}$ and compare with the appropriate values of $H_k^{(0)}$.

8. Alice and Bob (and no one else) share a key K. Each time that Alice wants to make sure that she is communicating with Bob, she sends him a random string S of 100 bits. Bob computes $B = H(S||K)$, where H is a good cryptographic hash function, and sends B to Alice. Alice computes $H(S||K)$. If this matches what Bob sent her, she is convinced that she is communicating with Bob.

    1. What property of H convinces Alice that she is communicating with Bob?

    2. Suppose Alice's random number generator is broken and she sends the same S each time she communicates with anyone. How can Eve (who doesn't know K, but who

intercepts all communications between Alice and Bob) convince Alice that she is Bob?

9.
    1. Show that neither of the two hash functions of Section 11.2 is preimage resistant. That is, given an arbitrary y (of the appropriate length), show how to find an input x whose hash is y.
    2. Find a collision for each of the two hash functions of Section 11.2.

10. An unenlightened professor asks his students to memorize the first 1000 digits of π for the exam. To grade the exam, he uses a 100-digit cryptographic hash function H. Instead of carefully reading the students' answers, he hashes each of them individually to obtain binary strings of length 100. Your score on the exam is the number of bits of the hash of your answer that agree with the corresponding bits of the hash of the correct answer.

    1. If someone gets 100% on the exam, why is the professor confident that the student's answer is correct?
    2. Suppose each student gets every digit of π wrong (a very unlikely occurrence!), and they all have different answers. Approximately what should the average on the exam be?

11. A bank in Tokyo is sending a terabyte of data to a bank in New York. There could be transmission errors. Therefore, the bank in Tokyo uses a cryptographic hash function H and computes the hash of the data. This hash value is sent to the bank in New York. The bank in New York computes the hash of the data received. If this matches the hash value sent from Tokyo, the New York bank decides that there was no transmission error. What property of cryptographic hash functions allows the bank to decide this?

12. (Thanks to Danna Doratotaj for suggesting this problem.)

    1. Let H_1 map 256-bit strings to 256-bit strings by $H_1(x) = x$. Show that H_1 is not preimage resistant but it is collision resistant.
    2. Let H be a good cryptographic hash function with a 256-bit output. Define a map H_2 from binary strings of arbitrary length to binary strings of length 257, as follows. If $0 \leq x < 2^{256}$, let $H_2(x) = x||$ (that is, x with appended). If $x \geq 2^{256}$, let $H_2(x) = H(x)||1$. Show that H_2 is collision resistant, and show that if y is a randomly chosen binary string of length 257, then the probability is at least 50% that you can easily find x with $H(x) = y$.

The functions H_1 and H_2 show that collision resistance does not imply preimage resistance quite as obviously as one might suspect.

13. Show that the computation of rc in Keccak (see the end of Section 11.5) can be given by an LFSR.

14. Let $Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$ be the function on 32-bit strings in the description of SHA-2.

    1. Suppose that the first bit of X and the first bit of Y are 1 and the first bit of Z is arbitrary. Show that the first bit of $\mathrm{Maj}(X,\ Y,\ Z)$ is 1.

    2. Suppose that the first bit of X and the first bit of Y are 0 and the first bit of Z is arbitrary. Show that the first bit of $\mathrm{Maj}(X, Y, Z)$ is 0.

    This shows that Maj gives the bitwise majority (for 0 vs. 1) for the strings X, Y, Z.

15. Let H be an iterative hash function that operates successively on input blocks of 512 bits. In particular, there is a compression function h and an initial value IV. The hash of a message $M_1 \parallel M_2$ of 1024 bits is computed by $X_1 = h(IV, M_1)$, and $H(M_1 \parallel M_2) = h(X_1, M_2)$. Suppose we have found a collision $h(IV, x_1) = h(IV, x_2)$ for some 512-bit blocks x_1 and x_2. Choose distinct primes p_1 and p_2, each of approximately 240 bits. Regard x_1 and x_2 as numbers between 0 and 2^{512}.

    1. Show that there exists an x_0 with $0 \le x_0 < p_1 p_2$ such that

       $$x_0 + 2^{512}x_1 \equiv 0 \pmod{p_1} \text{ and } x_0 + 2^{512}x_2 \equiv 0 \pmod{p_2}.$$

    2. Show that if $0 \le k < 2^{30}$, then $q_1 = (x_0 + 2^{512}x_1 + kp_1p_2)/p_1$ is approximately 2^{784}, and similarly for $q_2 = (x_0 + 2^{512}x_2 + kp_1p_2)/p_2$. (Assume that x_1 and x_2 are approximately 2^{512}.)

    3. Use the Prime Number Theorem (see Section 3.1) to show that the probability that q_1 is prime is approximately $1/543$ and the probability that both q_1 and q_2 are prime is about $1/300000$.

    4. Show that it is likely that there is some k with $0 \le k < 2^{30}$ such that both q_1 and q_2 are primes.

    5. Show that $n_1 = p_1 q_1$ and $n_2 = p_2 q_2$ satisfy $H(n_1) = H(n_2)$.

    This method of producing two RSA moduli with the same hash values is based on the method of [Lenstra et al.] for using a collision to produce two X.509 certificates with the same hashes. The method presented here produces moduli $n = pq$ with p and q of significantly different sizes (240 bits and 784 bits), but an adversary does not know this without factoring n.