11.4 HASH FUNCTIONS

A hash value h is generated by a function H of the form

$$h = \mathrm{H}(M)$$

where M is a variable-length message and $\mathrm{H}(M)$ is the fixed-length hash value. The hash value is appended to the message at the source at a time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the hash value. Because the hash function itself is not considered to be secret, some means is required to protect the hash value (Figure 11.5).

We begin by examining the requirements for a hash function to be used for message authentication. Because hash functions are, typically, quite complex, it is useful to examine next some very simple hash functions to get a feel for the issues involved. We then look at several approaches to hash function design.

Requirements for a Hash Function

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties (adapted from a list in [NECH92]):

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. $H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given value h, it is computationally infeasible to find x such that $H(x) = h$. This is sometimes referred to in the literature as the **one-way** property.
5. For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$. This is sometimes referred to as **weak collision resistance**.
6. It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$. This is sometimes referred to as **strong collision resistance**.[2]

The first three properties are requirements for the practical application of a hash function to message authentication.

The fourth property is the one-way property: It is easy to generate a code given a message but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value (Figure 11.5e). The secret value itself is not sent; however, if the hash function is not one way, an attacker can easily discover the secret value: If the attacker can observe or intercept a transmission, the attacker obtains the message M and the hash code $C = H(S_{AB} \| M)$. The attacker then inverts the hash function to obtain $S_{AB} \| M = H^{-1}(C)$. Because the attacker now has both M and $S_{AB} \| M$, it is a trivial matter to recover S_{AB}.

The fifth property guarantees that an alternative message hashing to the same value as a given message cannot be found. This prevents forgery when an encrypted hash code is used (Figures 11.5b and c). For these cases, the opponent can read the message and therefore generate its hash code. However, because the opponent does not have the secret key, the opponent should not be able to alter the message without detection. If this property were not true, an attacker would be capable of the following sequence: First, observe or intercept a message plus its encrypted hash code; second, generate an unencrypted hash code from the message; third, generate an alternate message with the same hash code.

The sixth property refers to how resistant the hash function is to a class of attack known as the birthday attack, which we examine shortly.

Birthday Attacks

Suppose that a 64-bit hash code is used. One might think that this is quite secure. For example, if an encrypted hash code C is transmitted with the corresponding unencrypted message M (Figure 11.5b or c), then an opponent would need to find an M' such that $H(M') = H(M)$ to substitute another message and fool the receiver. On average, the opponent would have to try about 2^{63} messages to find one that matches the hash code of the intercepted message [see Appendix 11A, Equation (11.1)].

However, a different sort of attack is possible, based on the birthday paradox (Appendix 11A). Yuval proposed the following strategy [YUVA79]:

1. The source, A, is prepared to "sign" a message by appending the appropriate m-bit hash code and encrypting that hash code with A's private key (Figure 11.5c).
2. The opponent generates $2^{m/2}$ variations on the message, all of which convey essentially the same meaning. The opponent prepares an equal number of messages, all of which are variations on the fraudulent message to be substituted for the real one.
3. The two sets of messages are compared to find a pair of messages that produces the same hash code. The probability of success, by the birthday paradox, is greater than 0.5. If no match is found, additional valid and fraudulent messages are generated until a match is made.
4. The opponent offers the valid variation to A for signature. This signature can then be attached to the fraudulent variation for transmission to the intended recipient. Because the two variations have the same hash code, they will produce the same signature; the opponent is assured of success even though the encryption key is not known.

Thus, if a 64-bit hash code is used, the level of effort required is only on the order of 2^{32} [see Appendix 11A, Equation (11.7)].

The generation of many variations that convey the same meaning is not difficult. For example, the opponent could insert a number of "space-space-backspace" character pairs between words throughout the document. Variations could then be

11.5 SECURITY OF HASH FUNCTIONS AND MACS

Just as with symmetric and public-key encryption, we can group attacks on hash functions and MACs into two categories: brute-force attacks and cryptanalysis.

Brute-Force Attacks

The nature of brute-force attacks differs somewhat for hash functions and MACs.

Hash Functions

The strength of a hash function against brute-force attacks depends solely on the length of the hash code produced by the algorithm. Recall from our discussion of hash functions that there are three desirable properties:

- **One-way:** For any given code h, it is computationally infeasible to find x such that $H(x) = h$.
- **Weak collision resistance:** For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
- **Strong collision resistance:** It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$.

For a code of length n, the level of effort required, as we have seen is proportional to the following:

One way	2^n
Weak collision resistance	2^n
Strong collision resistance	$2^{n/2}$

If strong collision resistance is required (and this is desirable for a general-purpose secure hash code), then the value $2^{n/2}$ determines the strength of the hash code against brute-force attacks. Oorschot and Wiener [OORS94] presented a design for a $10 million collision search machine for MD5, which has a 128-bit hash length, that could find a collision in 24 days. Thus a 128-bit code may be viewed as inadequate. The next step up, if a hash code is treated as a sequence of 32 bits, is a 160-bit hash length. With a hash length of 160 bits, the same search machine would require over four thousand years to find a collision. Currently, the two most popular hash codes, SHA-1 and RIPEMD-160, discussed in Chapter 12, provide a 160-bit hash code length.