PREV NEXT

## 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 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 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 $y \neq x$ such that 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]

[2] Unfortunately, these terms are not used consistently. Alternate terms used in the literature include *one-way hash function*: (properties 4 and 5); *collision-resistant hash function*: (properties 4, 5, and 6); *weak one-way hash function*: (properties 4 and 5); *strong one-way hash function*: (properties 4, 5, and 6). The reader must take care in reading the literature to determine the meaning of the particular terms used.

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

The fourth property, the one-way property, states that 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 $C = \mathrm{H}(S_{AB}||M)$. The attacker then inverts the hash function to obtain $S_{AB}||M = \mathrm{H}^{1}(C)$. 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 type of attack known as the birthday attack, which we examine shortly.

---

## Simple Hash Functions

All hash functions operate using the following general principles. The input (message, file, etc.) is viewed as a sequence of $n$-bit blocks. The input is processed one block at a time in an iterative fashion to produce an $n$-bit hash function.

One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block. This can be expressed as follows:

$$C_i = b_{i1} \oplus b_{i1} \oplus \ldots \oplus b_{im}$$

where

| | |
|---|---|
| $C_i$ | = $i$th bit of the hash code, $1 \leq i \leq n$ |
| $m$ | = number of $n$-bit blocks in the input |
| $b_{ij}$ | = $i$th bit in $j$th block |
| $\oplus$ | = XOR operation |

This operation produces a simple parity for each bit position and is known as a longitudinal redundancy check. It is reasonably effective for random data as a data integrity check. Each $n$-bit hash value is equally likely. Thus, the probability that a data error will result in an unchanged hash value is $2^{n}$. With more predictably formatted data, the function is less effective. For example, in