A hash function \( H \) accepts a variable-length block of data \( M \) as input and produces a fixed-size hash value \( h = H(M) \). A “good” hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random. In general terms, the principal object of a hash function is data integrity. A change to any bit or bits in \( M \) results, with high probability, in a change to the hash code. Hash functions are often used to determine whether or not data has changed.

Figure 11.1 depicts the general operation of a cryptographic hash function. Typically, the input is padded out to an integer multiple of some fixed length (e.g., 1024 bits) and the padding includes the value of the length of the original message in bits. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value.
11.1 APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

Message Authentication:

Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., contain no modification, insertion, deletion, or replay).

Figure 11.2 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows:

(a) Symmetric encryption: confidentiality and authentication

(b) Public-key encryption: confidentiality

(c) Public-key encryption: authentication and signature
a. The message plus concatenated hash code is encrypted using symmetric encryption. Because only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication. Because encryption is applied to the entire message plus hash code, confidentiality is also provided.

b. Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality.

c. It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value. A computes the hash value over the concatenation of $M$ and $S$ appends the resulting hash value to $M$. Because B possesses $S$, it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.

d. Confidentiality can be added to the approach of method (c) by encrypting the entire message plus the hash code

When confidentiality is not required, method (b) has an advantage over methods (a) and (d), which encrypts the entire message, in that less computation is required.

**Digital Signatures:**

Another important application, which is similar to the message authentication application, is the digital signature. The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, the hash value of a message is encrypted with a user's private key. Anyone who knows the user's public key can verify the integrity
of the message that is associated with the digital signature. In this case an attacker who wishes to alter the message would need to know the user's private key. Figure 11.3 illustrates, in a simplified fashion, how a hash code is used to provide a digital signature:

a. The hash code is encrypted, using public-key encryption and using the sender's private key. As with Figure 11.2b, this provides authentication. It also provides a digital signature, because only the sender could have produced the encrypted hash code. In fact, this is the essence of the digital signature technique.

b. If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.

Other Hash Function Uses:

Hash functions are commonly used to create a one-way password file. A hash of a password is stored by an operating system rather than the password itself. When a user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.
Hash functions can be used for **intrusion detection and virus detection**. Store \( H(F) \) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing \( H(F) \). An intruder would need to change \( F \) without changing \( H(F) \).

A cryptographic hash function can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG). A common application for a hash-based PRF is for the generation of symmetric keys.

**Hash Function Requirements:**

Before proceeding, we need to define two terms. For a hash value \( h = H(x) \), we say that \( x \) is the **preimage** of \( h \). That is, \( x \) is a data block whose hash function, using the function \( H \), is \( h \). A collision occurs if we have and 
\[
\exists \neq \text{ and } ( ) = ( )
\]

\[
h = ( ) ( )
\]

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable input size</td>
<td>( H ) can be applied to a block of data of any size.</td>
</tr>
<tr>
<td>Fixed output size</td>
<td>( H ) produces a fixed-length output.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>( H(x) ) is relatively easy to compute for any given ( x ), making both hardware and software implementations practical.</td>
</tr>
<tr>
<td>Preimage resistant (one-way property)</td>
<td>For any given hash value ( h ), it is computationally infeasible to find ( y ) such that ( H(y) = h ).</td>
</tr>
<tr>
<td>Second preimage resistant (weak collision resistant)</td>
<td>For any given block ( x ), it is computationally infeasible to find ( y \neq x ) with ( H(y) = H(x) ).</td>
</tr>
<tr>
<td>Collision resistant (strong collision resistant)</td>
<td>It is computationally infeasible to find any pair ( (x, y) ) such that ( H(x) = H(y) ).</td>
</tr>
<tr>
<td>Pseudorandomness</td>
<td>Output of ( H ) meets standard tests for pseudorandomness.</td>
</tr>
</tbody>
</table>
The first three properties are requirements for the practical application of a hash function.

The fourth property, preimage (for a hash value \( h = H(x) \), we say that \( x \) is the preimage of \( h \)) resistant, 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 11.2c).

The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used (Figure 11.2b and Figure 11.3a). A hash function that satisfies the first five properties in Table 11.1 is referred to as a weak hash function.

If the sixth property, collision resistant, is also satisfied, then it is referred to as a strong hash function. A strong hash function protects against an attack in which one party generates a message for another party to sign.

The final requirement, pseudorandomness, has not traditionally been listed as a requirement of cryptographic hash functions, but is more or less implied.

**Attacks on Hash Functions**

As with encryption algorithms, there are two categories of attacks on hash functions: brute-force attacks and cryptanalysis.

A brute-force attack does not depend on the specific algorithm but depends only on bit length. In the case of a hash function, a brute-force attack depends only on the bit length of the hash value.

A cryptanalysis, in contrast, is an attack based on weaknesses in a particular cryptographic algorithm.
Preimage and second Preimage attack:

For a preimage or second preimage attack, an adversary wishes to find a value \( y \) such that \( H(y) \) is equal to a given hash value \( h \). The brute force method is to pick values of \( y \) at random and try each value until a collision occurs. For an \( m \)-bit hash value, the level of effort is proportional to \( 2^m \). Specifically, the adversary would have to try, on average, \( 2^{m-1} \) values of \( y \) to find one that generates a given hash value \( h \).

Collision resistant attack:

For a collision resistant attack, an adversary wishes to find two messages or data blocks, \( x \) and \( y \), that yield the same hash function: \( H(x) = H(y) \). This requires much less effort than a preimage or second preimage attack. The effort required is explained by a mathematical result referred to as the birthday paradox (next slide).

If collision resistance is required, then the value \( 2^{m/2} \) determines the strength of the hash code against brute-force attacks.

Birthday Attacks:

The Birthday Attack exploits the birthday paradox – the chance that in a group of people two will share the same birthday. One can generalize the problem to one wanting a matching pair from any two sets, and show need \( 2^{m/2} \) in each to get a matching \( m \)-bit hash.

It exploit the birthday paradox in a collision resistant attack. Note that creating many message variants is relatively easy, either by rewording or just varying the amount of white-space in the message. All of which indicates that larger MACs/Hashes are needed.
**Birthday attack** works thus:

1. The source, A, is prepared to sign a legitimate message x by appending the appropriate m-bit hash code and encrypting that hash code with A’s private key (Figure 11.3a).
2. The opponent generates $2^{m/2}$ variations of x, all of which convey essentially the same meaning, and stores the messages and their hash values.
3. The opponent prepares a fraudulent message y for which A’s signature is desired.
4. The opponent generates minor variations y’ of y, all of which convey essentially the same meaning. For each y’, the opponent computes $H(y’)$, checks for matches with any of the $H(x’)$values, and continues until a match is found. That is, the process continues until a y’ is generated with a hash value equal to the hash value of one of the x’ values.
5. 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.

**Hash Function Cryptanalysis:**

As with encryption algorithms, cryptanalytic attacks on hash functions seek to exploit some property of the algorithm to perform some attack other than an exhaustive search.
The hash function takes an input message and partitions it into $L$ fixed-sized blocks of $b$ bits each. If necessary, the final block is padded to $b$ bits. The final block also includes the value of the total length of the input to the hash function. The inclusion of the length makes the job of the opponent more difficult. The hash algorithm involves repeated use of a compression function $f$ that takes two inputs (an $n$-bit input from the previous step, called the chaining variable, and a $b$-bit block) and produces an $n$-bit output. At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm. The final value of the chaining variable is the hash value. Often, $b > n$; hence the term compression.

$$CV_0 = IV = \text{initial } n\text{-bit value}$$

$$CV_i = (f, CV_{i-1}, Y_{i-1}) \quad 1 \leq i \leq L$$

$$H(M) = CV_L$$

If the compression function is collision resistant, then so is the resultant iterated hash function. Therefore, the structure can be used to produce a secure hash function to operate on a message of any length.

Cryptanalysis of hash functions focuses on the internal structure of $f$ and is based on attempts to find efficient techniques for producing collisions for a single execution of $f$. Once that is done, the attack must take into account the fixed value of $IV$. The attack on $f$ depends on exploiting its internal structure. The attacks that have been mounted on hash functions are rather complex and beyond our scope here.

- MD5 Algorithm is posted Separately.
12.1. Secure Hash Algorithm

- The actual standards document is entitled Secure Hash Standard. SHA is based on the hash function MD4 and its design closely models MD4.
- SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512.
- These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1.

<table>
<thead>
<tr>
<th>Table 12.1 Comparison of SHA Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message digest size</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>Message size</td>
</tr>
<tr>
<td>Block size</td>
</tr>
<tr>
<td>Word size</td>
</tr>
<tr>
<td>Number of steps</td>
</tr>
<tr>
<td>Security</td>
</tr>
</tbody>
</table>

Notes: 1. All sizes are measured in bits.
2. Security refers to the fact that a birthday attack on a message digest of size $n$ produces a collision with a workfactor of approximately $2^{n/2}$.

SHA-512 Logic

The algorithm takes as input a message with a maximum length of less than $2^{128}$ bits and produces as **output a 512-bit message digest**. The input is processed in 1024-bit blocks.

The processing consists of the following steps:

- **Step 1: Append padding bits.** The message is padded so that its length is congruent to 896 modulo 1024 [$\text{length} \equiv 896 \pmod{1024}$]. Padding is always added, even if the
message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

- **Step 2: Append length.** A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).

  The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. In Figure 12.1, the expanded message is represented as the sequence of 1024 bit blocks $M_1, M_2, ..., M_N$, so that the total length of the expanded message is $N \times 1024$ bits.

- **Step 3: Initialize hash buffer.** A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers ($a, b, c, d, e, f, g, h$). These registers are initialized to the following 64-bit integers (hexadecimal values):

  a = 6A09E667F3BCC908   e = 510E527FADE682D1
  b = BB67AE8584CAA73B   f = 9B05688C2B3E6C1F
  c = 3C6EF372FE94F82B   g = 1F83D9ABFB41BD6B
  d = A54FF53A5F1D36F1   h = 5BE0CDI9137E2179

  These values are stored in big-endian format, which is the most significant byte of a word in the low-address (leftmost) byte position. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

- **Step 4: Process message in 1024-bit (128-word) blocks.** The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 12.1. The logic is illustrated in Figure 12.2.
In this Step 4, it processes the message in 1024-bit (128-word) blocks, using a module that consists of 80 rounds, labeled F.

Each round takes as input the 512-bit buffer value, and updates the contents of the buffer. Each round t makes use of a 64-bit value $W_t$ derived using a message schedule from the current 1024-bit block being processed.

Each round also makes use of an additive constant $K_t$, where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers.

The output of the eightieth round is added to the input to the first round to produce the final hash value for this message block, which forms the input to the next iteration of this compression function, as shown on the previous slide.
Step 5: Output. After all $N$ 1024-bit blocks have been processed; the output from the $N$th stage is the 512-bit message digest.

We can summarize the behavior of SHA-512 as follows:

$$H_0 = IV$$
\[ H_0 = IV \]
\[ H_i = \text{SUM}_{64}(H_{i-1}, \text{abcdefgh}_i) \]
\[ \text{MD} = H_N \]

where

\[ \text{IV} \] = initial value of the abcdefgh buffer, defined in step 3
\[ \text{abcdefgh}_i \] = the output of the last round of processing of the \( i \)th message block
\[ N \] = the number of blocks in the message (including padding and length fields)
\[ \text{SUM}_{64} \] = Addition modulo \( 2^{64} \) performed separately on each word of the pair of inputs
\[ \text{MD} \] = final message digest value

**SHA-512 Round Function**

Let us look in more detail at the logic in each of the 80 steps of the processing of one 512-bit block (Figure 12.3). Each round is defined by the following set of equations:

**SHA-512 Round Function**

Let us look in more detail at the logic in each of the 80 steps of the processing of one 512-bit block (Figure 11.10). Each round is defined by the following set of equations:

where

\[ h = \]
Chapter 11 Cryptographic Hash Functions

\[
\begin{align*}
    &= + \\
    &= \\
    &= \\
    &= \\
    &= +
\end{align*}
\]

Where

\( t \) = step number; \( 0 \leq t \leq 79 \)

\( Ch(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g) \)

*the conditional function: If \( e \) then \( f \) else \( g \)*

\( Maj(a, b, c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c) \)

*the function is true only of the majority (two or three) of the arguments are true*

\[
\begin{align*}
    &= \text{ROTR} \left( \right) \oplus \text{ROTR} \left( \right) \oplus \text{ROTR} \left( \right)
\end{align*}
\]

\[
\begin{align*}
    &= \text{ROTR} \left( \right) \oplus \text{ROTR} \left( \right) \oplus \text{ROTR} \left( \right)
\end{align*}
\]

\( \text{ROTR} \left( \right) = \) circular right shift (rotation) of the 64-bit argument by \( t \) bits

\( = 64 \)-bit word derived from the current 512-bit input block

\( = 64 \)

\( + = \text{addition modulo } 2^{64} \)
It remains to indicate how the 64-bit word values are derived from the 1024-bit message. Figure 11.11 illustrates the mapping. The first 16 values of are taken directly from the 16 words of the current block. The remaining values are defined as

$$ W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16} $$

where

$$ \sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x) $$

$$ \sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x) $$

\text{ROTR}^n(x) = \text{circular right shift (rotation) of the 64-bit argument } x \text{ by } n \text{ bits}

\text{SHR}^n(x) = \text{left shift of the 64-bit argument } x \text{ by } n \text{ bits with padding by zeros on the right}

+ = \text{addition modulo } 2^{64} \]
Thus, in the first 16 steps of processing, the value of $M_i$ is equal to the corresponding word in the message block. For the remaining 64 steps, the value of $M_i$ consists of the circular left shift by one bit of the XOR of four of the preceding values of $W_t$, with two of those values subjected to shift and rotate operations. This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.

******