PART 3: CRYPTOGRAPHIC DATA INTEGRITY ALGORITHMS

CHAPTER 11

CRYPTOGRAPHIC HASH FUNCTIONS
11.1 Applications of Cryptographic Hash Functions
   Message Authentication
   Digital Signatures
   Other Applications

11.2 Two Simple Hash Functions

11.3 Requirements and Security
   Security Requirements for Cryptographic Hash Functions
   Brute-Force Attacks
   Cryptanalysis

11.4 Hash Functions Based on Cipher Block Chaining

11.5 Secure Hash Algorithm (SHA)
   SHA-512 Logic
   SHA-512 Round Function
   Example

11.6 SHA-3
KEY POINTS

- A **hash function** maps a variable-length message into a **fixed-length hash value**, or **message digest**.
- Virtually all **cryptographic hash functions** involve the **iterative use of a compression function**.
- The **compression function** used in secure hash algorithms falls into one of two categories: a function specifically designed for the hash function or an algorithm based on a symmetric block cipher. **SHA** and **Whirlpool** are examples of these two approaches, respectively.
Figure 11.1  Black Diagram of Cryptographic Hash Function; $h = H(M)$
11.1 APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

Perhaps the most versatile cryptographic algorithm is the cryptographic hash function. It is used in a wide variety of security applications and Internet protocols. To better understand some of the requirements and security implications for cryptographic hash functions, it is useful to look at the range of applications in which it is employed.

**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). In many cases, there is a requirement that the authentication mechanism assures that purported identity of the sender is valid. When a hash function is used to provide message authentication, the hash function value is often referred to as a *message digest*.

Figure 11.2 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows.
Figure 11.2  Simplified Examples of the Use of a Hash Function for Message Authentication
More commonly, message authentication is achieved using a **message authentication code (MAC)**, also known as a **keyed hash function**. Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC. This can then be transmitted with or stored with the protected message. If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the stored MAC value. An attacker who alters the message will be unable to alter the MAC value without knowledge of the secret key. Note that the verifying party also knows who the sending party is because no one else knows the secret key.
Note that the combination of hashing and encryption results in an overall function that is, in fact, a MAC (Figure 11.2b). That is, $E(K, H(M))$ is a function of a variable-length message $M$ and a secret key $K$, and it produces a fixed-size output that is secure against an opponent who does not know the secret key. In practice, specific MAC algorithms are designed that are generally more efficient than an encryption algorithm.

We discuss MACs in Chapter 12.
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. As we shall see in Chapter 14, the implications of digital signatures go beyond just message authentication.

Figure 11.3 illustrates, in a simplified fashion, how a hash code is used to provide a digital signature.
Figure 11.3  Simplified Examples of Digital Signatures
<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>
Figure 11.5  Relationship Among Hash Function Properties
<table>
<thead>
<tr>
<th>Application</th>
<th>Preimage Resistant</th>
<th>Second Preimage Resistant</th>
<th>Collision Resistant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash + digital signature</td>
<td>yes</td>
<td></td>
<td>yes*</td>
</tr>
<tr>
<td>Intrusion detection and virus detection</td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Hash + symmetric encryption</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-way password file</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
</tr>
</tbody>
</table>

* Resistance required if attacker is able to mount a chosen message attack
To summarize, for a hash code of length $m$, the level of effort required, as we have seen, is proportional to the following.

<table>
<thead>
<tr>
<th>Property</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preimage resistant</td>
<td>$2^m$</td>
</tr>
<tr>
<td>Second preimage resistant</td>
<td>$2^m$</td>
</tr>
<tr>
<td>Collision resistant</td>
<td>$2^{\frac{m}{2}}$</td>
</tr>
</tbody>
</table>
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

![Diagram](https://via.placeholder.com/150)

**Figure 11.7** General Structure of Secure Hash Code

- \( IV = \) Initial value
- \( CV_i = \) Chaining variable
- \( Y_i = \) \( i \)th input block
- \( f = \) Compression algorithm
- \( L = \) Number of input blocks
- \( n = \) Length of hash code
- \( b = \) Length of input block
of the chaining variable is the hash value. Often, \( b > n \); hence the term compression. The hash function can be summarized as

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

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

\[
H(M) = CV_L
\]

where the input to the hash function is a message \( M \) consisting of the blocks \( Y_0, Y_1, \ldots, Y_{L-1} \).
11.4 HASH FUNCTIONS BASED ON CIPHER BLOCK CHAINING

A number of proposals have been made for hash functions based on using a cipher block chaining technique, but without using the secret key. One of the first such proposals was that of Rabin [RABI78]. Divide a message $M$ into fixed-size blocks $M_1, M_2, \ldots, M_N$ and use a symmetric encryption system such as DES to compute the hash code $G$ as

\[
\begin{align*}
H_0 & = \text{initial value} \\
H_i & = E(M_i, H_{i-1}) \\
G & = H_N
\end{align*}
\]

This is similar to the CBC technique, but in this case, there is no secret key. As with any hash code, this scheme is subject to the birthday attack, and if the encryption algorithm is DES and only a 64-bit hash code is produced, then the system is vulnerable.
Table 11.3  Comparison of SHA Parameters

<table>
<thead>
<tr>
<th></th>
<th>SHA-1</th>
<th>SHA-224</th>
<th>SHA-256</th>
<th>SHA-384</th>
<th>SHA-512</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Digest Size</strong></td>
<td>160</td>
<td>224</td>
<td>256</td>
<td>384</td>
<td>512</td>
</tr>
<tr>
<td><strong>Message Size</strong></td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{128}$</td>
<td>$&lt;2^{128}$</td>
</tr>
<tr>
<td><strong>Block Size</strong></td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td><strong>Word Size</strong></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td><strong>Number of Steps</strong></td>
<td>80</td>
<td>64</td>
<td>64</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

*Note: All sizes are measured in bits.*
11.5 SECURE HASH ALGORITHM (SHA)

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). Indeed, because virtually every other widely used hash function had been found to have substantial cryptanalytic weaknesses, SHA was more or less the last remaining standardized hash algorithm by 2005. SHA was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993. When weaknesses were discovered in SHA, now known as SHA-0, a revised version was issued as FIPS 180-1 in 1995 and is referred to as SHA-1. 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 is also specified in RFC 3174, which essentially duplicates the material in FIPS 180-1 but adds a C code implementation.
SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively. Collectively, these hash algorithms are known as SHA-2. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version (Table 11.3). SHA-2 is also specified in RFC 4634, which essentially duplicates the material in FIPS 180-3 but adds a C code implementation.

In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on SHA-2 by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using $2^{69}$ operations, far fewer than the $2^{80}$ operations previously thought needed to find a collision with an SHA-1 hash [WANG05]. This result should hasten the transition to SHA-2.

In this section, we provide a description of SHA-512. The other versions are quite similar.
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. Figure 11.8 depicts the overall processing of a message to produce a digest. This follows the general structure depicted in Figure 11.7. The processing consists of the following steps.

### Table 11.3 Comparison of SHA Parameters

<table>
<thead>
<tr>
<th></th>
<th>SHA-1</th>
<th>SHA-224</th>
<th>SHA-256</th>
<th>SHA-384</th>
<th>SHA-512</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Digest Size</strong></td>
<td>160</td>
<td>224</td>
<td>256</td>
<td>384</td>
<td>512</td>
</tr>
<tr>
<td><strong>Message Size</strong></td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{128}$</td>
<td>$&lt;2^{128}$</td>
</tr>
<tr>
<td><strong>Block Size</strong></td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td><strong>Word Size</strong></td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td><strong>Number of Steps</strong></td>
<td>80</td>
<td>64</td>
<td>64</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

*Note: All sizes are measured in bits.*
+ = word-by-word addition mod $2^{512}$
Step 1  **Append padding bits.** The message is padded so that its length is congruent to 896 modulo 1024 \([\text{length} \equiv 896(\text{mod } 1024)]\). 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 11.8, the expanded message is represented as the sequence of 1024-bit blocks \(M_1, M_2, \ldots, M_N\), 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):

\[
\begin{align*}
    &a = \text{6A09E667F3BCC908} \quad &e = \text{510E527FADE682D1} \\
    &b = \text{BB67AE8584CAA73B} \quad &f = \text{9B05688C2B3E6C1F} \\
    &c = \text{3C6EF372FE94F82B} \quad &g = \text{1F83D9ABFB41BD6B} \\
    &d = \text{A54FF53A5F1D36F1} \quad &h = \text{5BE0CD19137E2179}
\end{align*}
\]

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 11.8. The logic is illustrated in Figure 11.9.
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:

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

where

- **IV** = initial value of the abcdefgh buffer, defined in step 3
- **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)
- **SUM\(_{64}\)** = addition modulo \( 2^{64} \) performed separately on each word of the pair of inputs
- **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 11.10). Each round is defined by the following set of equations:

\[ T_1 = h + \text{Ch}(e, f, g) + \left( \sum_{i=1}^{512} e \right) + W_t + K_t \]
\[ T_2 = \left( \sum_{i=0}^{512} a \right) + \text{Maj}(a, b, c) \]

- \[ h = g \]
- \[ g = f \]
- \[ f = e \]
- \[ e = d + T_1 \]
- \[ d = c \]
- \[ c = b \]
- \[ b = a \]
- \[ a = T_1 + T_2 \]

where

- \[ t = \text{step number; } 0 \leq t \leq 79 \]
- \[ \text{Ch}(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g) \]

the conditional function: If \(e\) then \(f\) else \(g\)
<table>
<thead>
<tr>
<th>SHA-512 Constants</th>
<th>428a2f98d728ae22</th>
<th>7137449123ef65cd</th>
<th>b8c0fbcfec4d3b2f</th>
<th>e9b5dba58189d9bb</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3956c25bf348b538</td>
<td>59f111f1b605d019</td>
<td>923f82a4af194f9b</td>
<td>ab1c5ed5da6d8118</td>
</tr>
<tr>
<td></td>
<td>d807aa98a3030242</td>
<td>12835b0145706f6e</td>
<td>243185be4ee4b28c</td>
<td>550c7dc3d5ff46a2</td>
</tr>
<tr>
<td></td>
<td>72be5d74f27b896f</td>
<td>80deb1fe3b1696b1</td>
<td>9bdc06a725c71235</td>
<td>c19bf174cf692694</td>
</tr>
<tr>
<td></td>
<td>e49b69c19ef14ad2</td>
<td>efbe4786384f25e3</td>
<td>0fc19dc68b8cd5b5</td>
<td>240calcc77ac9c65</td>
</tr>
<tr>
<td></td>
<td>2de92c6f592b0275</td>
<td>4a7a484aa6e6e483</td>
<td>5cb0a9dcb341fb4d</td>
<td>76f988da831153b5</td>
</tr>
<tr>
<td></td>
<td>983e5152e66dfab</td>
<td>ad831c662db43210</td>
<td>fd0327c898f2b13f</td>
<td>bf597fc7bee0ee4</td>
</tr>
<tr>
<td></td>
<td>c600bf33da88fc2</td>
<td>d5a79147b03aa725</td>
<td>06ca6351e003826f</td>
<td>142929670a0e6e70</td>
</tr>
<tr>
<td></td>
<td>27b70a8546d22f2c</td>
<td>2e1b21385c26c926</td>
<td>4d2c6dfc5ac42aed</td>
<td>53380d139d95b3df</td>
</tr>
<tr>
<td></td>
<td>650a73548baf63de</td>
<td>766a0a6bb377b2a8</td>
<td>81c9c2e47edae6e</td>
<td>92722c851482353b</td>
</tr>
<tr>
<td></td>
<td>a2bfe8a14cf10364</td>
<td>a81664bbcc423001</td>
<td>c248b3b7b00f89791</td>
<td>c76c51a30654be30</td>
</tr>
<tr>
<td></td>
<td>d192e819d6f5218</td>
<td>d69906245565a910</td>
<td>f40e35855771202a</td>
<td>106aa07032b2b0b</td>
</tr>
<tr>
<td></td>
<td>19a4c116ba8d20c8</td>
<td>1e376c08514ab53</td>
<td>2748774df8ebeb99</td>
<td>34b0cb5e19b48a8</td>
</tr>
<tr>
<td></td>
<td>391c0cb3c5c95a63</td>
<td>4ed8aa4ae3418acb</td>
<td>5b9cca4f7763e373</td>
<td>682e6ff3d6b2ba3</td>
</tr>
<tr>
<td></td>
<td>748f8e5e5debf2fc</td>
<td>78a5636f3172f60</td>
<td>8dc87814a1f0ab72</td>
<td>8cc702081a6d49ec</td>
</tr>
<tr>
<td></td>
<td>90befffa23631e28</td>
<td>a4506cebe82bde9</td>
<td>bef9a3f7b2c67915</td>
<td>c67178f2e372532b</td>
</tr>
<tr>
<td></td>
<td>ca27eceea26619c</td>
<td>d186b8c721c0c207</td>
<td>eada7dd6cde0ebe</td>
<td>f57d4f7f0e6ed178</td>
</tr>
<tr>
<td></td>
<td>06f067aa72176fba</td>
<td>0a637dc5a2e898a6</td>
<td>113f9804bef90dae</td>
<td>1b710b35131c471b</td>
</tr>
<tr>
<td></td>
<td>28db77f523047d84</td>
<td>32caab7b40c72493</td>
<td>3c9ebe0a159bebc</td>
<td>431d67c49c100d4c</td>
</tr>
<tr>
<td></td>
<td>4cc5d4becb3e42b6</td>
<td>597f299cfc657e2a</td>
<td>5fcb6fab3a6f4e</td>
<td>6c44198c4a475817</td>
</tr>
</tbody>
</table>
\[ \text{Maj}(a, b, c) = (a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c) \]

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

\[ \left( \sum_{0}^{512} a \right) = \text{ROTR}^{28}(a) \oplus \text{ROTR}^{34}(a) \oplus \text{ROTR}^{39}(a) \]

\[ \left( \sum_{1}^{512} e \right) = \text{ROTR}^{14}(e) \oplus \text{ROTR}^{18}(e) \oplus \text{ROTR}^{41}(e) \]

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

\text{W}_{t} = \text{a 64-bit word derived from the current 512-bit input block}

\text{K}_{t} = \text{a 64-bit additive constant}

+ = \text{addition modulo } 2^{64}
Figure 11.10  Elementary SHA-512 Operation (single round)
It remains to indicate how the 64-bit word values $W_t$ are derived from the 1024-bit message. Figure 11.11 illustrates the mapping. The first 16 values of $W_t$ 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

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

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

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

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

$+ = \text{addition modulo } 2^{64}$
Figure 11.11  Creation of 80-word Input Sequence for SHA-512 Processing of Single Block
Example

We include here an example based on one in FIPS 180. We wish to hash a one-block message consisting of three ASCII characters: “abc”, which is equivalent to the following 24-bit binary string:

01100001 01100010 01100011

Recall from step 1 of the SHA algorithm, that the message is padded to a length congruent to 896 modulo 1024. In this case of a single block, the padding consists of \(896 - 24 = 872\) bits, consisting of a “1” bit followed by 871 “0” bits. Then a 128-bit length value is appended to the message, which contains the length of the original message (before the padding). The original length is 24 bits, or a hexadecimal value of 18. Putting this all together, the 1024-bit message block, in hexadecimal, is

\[24 = (1, 1, 0, 0, 0)_2 = (1, 8)_{16}\]
The resulting 512-bit message digest is

`ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a 2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f`

Suppose now that we change the input message by one bit, from “abc” to “cbc”. Then, the 1024-bit message block is

```
6362638000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

And the resulting 512-bit message digest is

`531668966ee79b70 0b8e593261101354 4273f7ef7b31f279 2a7ef68d53f93264 319c165ad96d9187 55e6a204c2607e27 6e05c9f993a64c85 ef9e1e125c0f925f`
As of this writing, SHA-1 has not yet been “broken.” That is, no one has demonstrated a technique for producing collisions in less than brute-force time. However, because SHA-1 is very similar in structure and in the basic mathematical operations used to MD5 and SHA-0, both of which have been broken, SHA-1 is considered insecure and has been phased out for SHA-2.

SHA-2, particularly the 512-bit version, would appear to provide unassailable security. However, SHA-2 shares the same structure and mathematical operations as its predecessors, and this is a cause for concern. Because it will take years to find a suitable replacement for SHA-2, should it become vulnerable, NIST decided to begin the process of developing a new hash standard.

Accordingly, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. NIST would like to have a new standard in place by the end of 2012, but emphasizes that this is not a fixed timeline and that the schedule could slip well beyond that date. The basic requirements that must be satisfied by any candidate for SHA-3 are the following.
1. It must be possible to replace SHA-2 with SHA-3 in any application by a simple drop-in substitution. Therefore, SHA-3 must support hash value lengths of 224, 256, 384, and 512 bits.

2. SHA-3 must preserve the online nature of SHA-2. That is, the algorithm must process comparatively small blocks (512 or 1024 bits) at a time instead of requiring that the entire message be buffered in memory before processing it.

Beyond these basic requirements, NIST has defined a set of evaluation criteria. These criteria are designed to reflect the requirements for the main applications supported by SHA-2, which include digital signatures, hashed message authentication

Keccak
codes, key generation, and pseudorandom number generation. The evaluation criteria for the new hash function, in decreasing order of importance, are as follows.

- **Security**: The security strength of SHA-3 should be close to the theoretical maximum for the different required hash sizes and for both preimage resistance and collision resistance. SHA-3 algorithms must be designed to resist any potentially successful attack on SHA-2 functions. In practice, this probably means that SHA-3 must be fundamentally different than the SHA-1, SHA-2, and MD5 algorithms in either structure, mathematical functions, or both.

- **Cost**: SHA-3 should be both time and memory efficient over a range of hardware platforms.

- **Algorithm and implementation characteristics**: Consideration will be given to such characteristics as flexibility (e.g., tunable parameters for security/performance tradeoffs, opportunity for parallelization, and so on) and simplicity. The latter characteristic makes it easier to analyze the security properties of the algorithm.