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. Figure 11.13 summarizes the SHA-512 logic.

The SHA-512 algorithm has the property that every bit of the hash code is a function of every bit of the input. The complex repetition of the basic function F produces results that are well mixed; that is, it is unlikely that two messages chosen at random, even if they exhibit similar regularities, will have the same hash code. Unless there is some hidden weakness in SHA-512, which has not so far been published, the difficulty of coming up with two messages having the same message digest is on the order of 2^{256} operations, while the difficulty of finding a message with a given digest is on the order of 2^{512} operations.

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 in bits (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

```
6162638000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018
```

This block is assigned to the words W0, . . . , W15 of the message schedule, which appears as follows.

$W_0 = 6162638000000000$ $W_8 = 0000000000000000$
$W_1 = 0000000000000000$ $W_9 = 0000000000000000$
$W_2 = 0000000000000000$ $W_{10} = 0000000000000000$
$W_3 = 0000000000000000$ $W_{11} = 0000000000000000$
$W_4 = 0000000000000000$ $W_{12} = 0000000000000000$
$W_5 = 0000000000000000$ $W_{13} = 0000000000000000$
$W_6 = 0000000000000000$ $W_{14} = 0000000000000000$
$W_7 = 0000000000000000$ $W_{15} = 0000000000000018$

The padded message consists blocks $M_1, M_2, \ldots, M_N$. Each message block M_i consists of 16 64-bit words $M_{i,0}, M_{i,1}, \ldots, M_{i,15}$. All addition is performed modulo 2^{64}.

$H_{0,0} = 6A09E667F3BCC908$ $H_{0,4} = 510E527FADE682D1$
$H_{0,1} = BB67AE8584CAA73B$ $H_{0,5} = 9B05688C2B3E6C1F$
$H_{0,2} = 3C6EF372FE94F82B$ $H_{0,6} = 1F83D9ABFB41BD6B$
$H_{0,3} = A54FF53A5F1D36F1$ $H_{0,7} = 5BE0CD19137E2179$

for $i = 1$ **to** N

1. Prepare the message schedule W
for $t = 0$ **to** 15
$W_t = M_{i,t}$
for $t = 16$ **to** 79
$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$

2. Initialize the working variables

$a = H_{i-1,0}$ $e = H_{i-1,4}$
$b = H_{i-1,1}$ $f = H_{i-1,5}$
$c = H_{i-1,2}$ $g = H_{i-1,6}$
$d = H_{i-1,3}$ $h = H_{i-1,7}$

3. Perform the main hash computation
for $t = 0$ **to** 79

$$T_1 = h + \text{Ch}(e, f, g) + \left(\Sigma_1^{512} e\right) + W_t + K_t$$

$$T_2 = \left(\Sigma_0^{512} a\right) + \text{Maj}(a, b, c)$$

$h = g$
$g = f$
$f = e$
$e = d + T_1$
$d = c$
$c = b$
$b = a$
$a = T_1 + T_2$

4. Compute the intermediate hash value

$H_{i,0} = a + H_{i-1,0}$ $H_{i,4} = e + H_{i-1,4}$
$H_{i,1} = b + H_{i-1,1}$ $H_{i,5} = f + H_{i-1,5}$
$H_{i,2} = c + H_{i-1,2}$ $H_{i,6} = g + H_{i-1,6}$
$H_{i,3} = d + H_{i-1,3}$ $H_{i,7} = h + H_{i-1,7}$

return $\{H_{N,0} \| H_{N,1} \| H_{N,2} \| H_{N,3} \| H_{N,4} \| H_{N,5} \| H_{N,6} \| H_{N,7}\}$

Figure 11.13 SHA-512 Logic

As indicated in Figure 11.13, the eight 64-bit variables, a through h, are initialized to values $H_{0,0}$ through $H_{0,7}$. The following table shows the initial values of these variables and their values after each of the first two rounds.

a	6a09e667f3bcc908	f6afceb8bcfcddf5	1320f8c9fb872cc0
b	bb67ae8584caa73b	6a09e667f3bcc908	f6afceb8bcfcddf5
c	3c6ef372fe94f82b	bb67ae8584caa73b	6a09e667f3bcc908
d	a54ff53a5f1d36f1	3c6ef372fe94f82b	bb67ae8584caa73b
e	510e527fade682d1	58cb02347ab51f91	c3d4ebfd48650ffa
f	9b05688c2b3e6c1f	510e527fade682d1	58cb02347ab51f91
g	1f83d9abfb41bd6b	9b05688c2b3e6c1f	510e527fade682d1
h	5be0cd19137e2179	1f83d9abfb41bd6b	9b05688c2b3e6c1f

Note that in each of the rounds, six of the variables are copied directly from variables from the preceding round.

The process continues through 80 rounds. The output of the final round is

```
73a54f399fa4b1b2 10d9c4c4295599f6 d67806db8b148677 654ef9abec389ca9
d08446aa79693ed7 9bb4d39778c07f9e 25c96a7768fb2aa3 ceb9fc3691ce8326
```

The hash value is then calculated as

$$H_{1,0} = \texttt{6a09e667f3bcc908} + \texttt{73a54f399fa4b1b2} = \texttt{ddaf35a193617aba}$$
$$H_{1,1} = \texttt{bb67ae8584caa73b} + \texttt{10d9c4c4295599f6} = \texttt{cc417349ae204131}$$
$$H_{1,2} = \texttt{3c6ef372fe94f82b} + \texttt{d67806db8b148677} = \texttt{12e6fa4e89a97ea2}$$
$$H_{1,3} = \texttt{a54ff53a5f1d36f1} + \texttt{654ef9abec389ca9} = \texttt{0a9eeee64b55d39a}$$
$$H_{1,4} = \texttt{510e527fade682d1} + \texttt{d08446aa79693ed7} = \texttt{2192992a274fc1a8}$$
$$H_{1,5} = \texttt{9b05688c2b3e6c1f} + \texttt{9bb4d39778c07f9e} = \texttt{36ba3c23a3feebbd}$$
$$H_{1,6} = \texttt{1f83d9abfb41bd6b} + \texttt{25c96a7768fb2aa3} = \texttt{454d4423643ce80e}$$
$$H_{1,7} = \texttt{5be0cd19137e2179} + \texttt{ceb9fc3691ce8326} = \texttt{2a9ac94fa54ca49f}$$

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 0000000000000000 0000000000000018
```

And the resulting 512-bit message digest is

```
531668966ee79b70 0b8e593261101354 4273f7ef7b31f279 2a7ef68d53f93264
319c165ad96d9187 55e6a204c2607e27 6e05cdf993a64c85 ef9e1e125c0f925f
```

The number of bit positions that differ between the two hash values is 253, almost exactly half the bit positions, indicating that SHA-512 has a good avalanche effect.

11.6 SHA-3

As of this writing, the Secure Hash Algorithm (SHA-1) has not yet been "broken." That is, no one has demonstrated a technique for producing collisions in a practical amount of 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. The winning design for SHA-3 was announced by NIST in October 2012 and published as FIP 102 in August 2015. SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications.

Appendix V looks at the evaluation criteria used by NIST to select from among the candidates for AES, plus the rationale for picking Keccak, which was the winning candidate. This material is useful in understanding not just the SHA-3 design but also the criteria by which to judge any cryptographic hash algorithm.

The Sponge Construction

The underlying structure of SHA-3 is a scheme referred to by its designers as a **sponge construction** [BERT07, BERT11]. The sponge construction has the same general structure as other iterated hash functions (Figure 11.8). The sponge function takes an input message and partitions it into fixed-size blocks. Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block.

The sponge function is defined by three parameters:

f = the internal function used to process each input block[3]

r = the size in bits of the input blocks, called the **bitrate**

pad = the padding algorithm

A sponge function allows both variable length input and output, making it a flexible structure that can be used for a hash function (fixed-length output), a pseudorandom number generator (fixed-length input), and other cryptographic functions. Figure 11.14 illustrates this point. An input message of n bits is partitioned into k fixed-size blocks of r bits each. The message is padded to achieve a length that is an integer multiple of r bits. The resulting partition is the sequence of blocks $P_0, P_1, \ldots, P_{k-1}$, with length $k \times r$. For uniformity, padding is always added, so

[3]The Keccak documentation refers to f as a permutation. As we shall see, it involves both permutations and substitutions. We refer to f as the **iteration function**, because it is the function that is executed once for each iteration, that is, once for each block of the message that is processed.