Cryptography and Network Security (CS435/890BN)

Part Nine

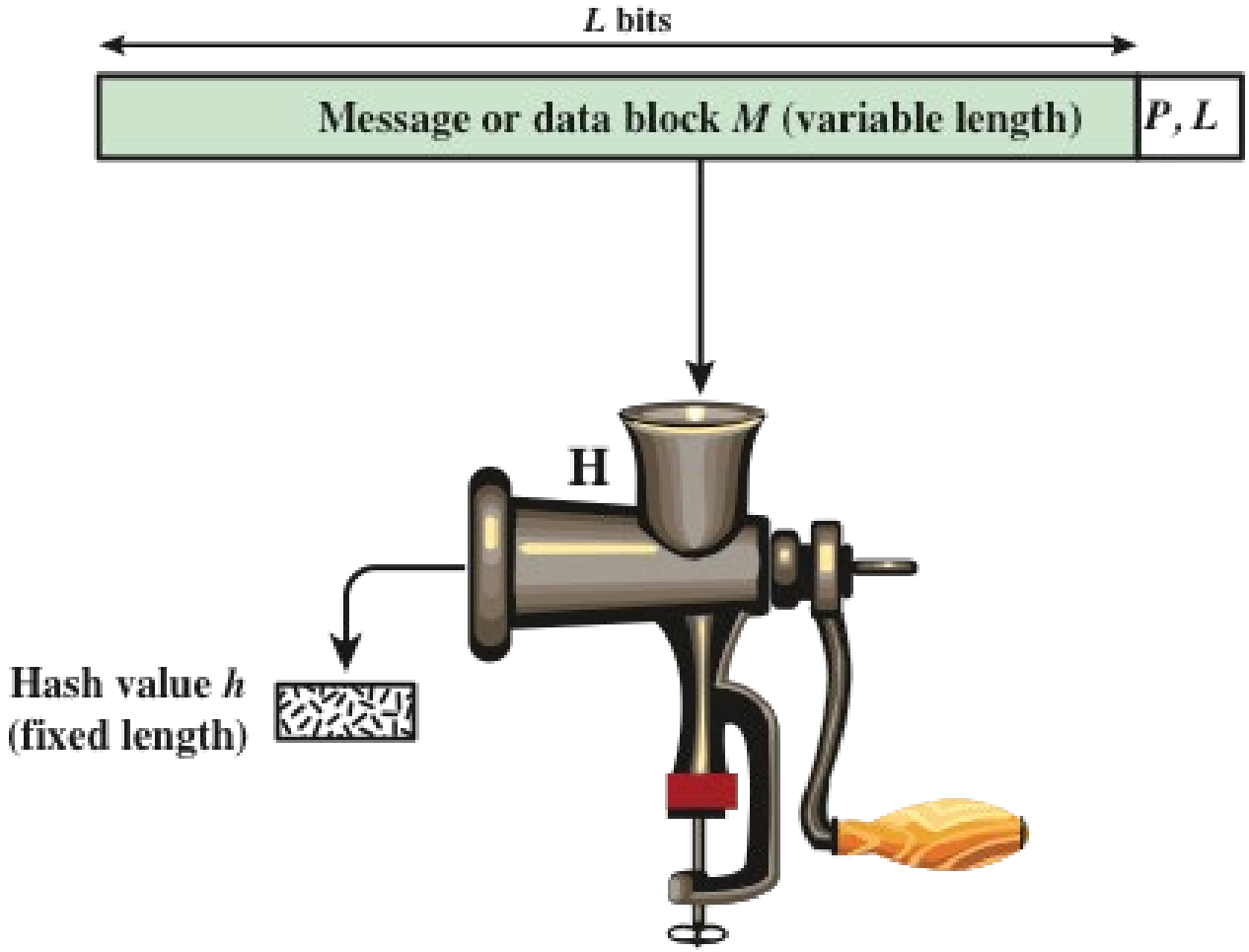
(Cryptographic Hash Functions)

Hash Functions

- A hash function H accepts a variable-length block of data *M* as input and produces a fixed-size hash value
  - $h = H(M)$
  - Principal object is data integrity
- Cryptographic hash function
  - An algorithm for which it is computationally infeasible to find either:

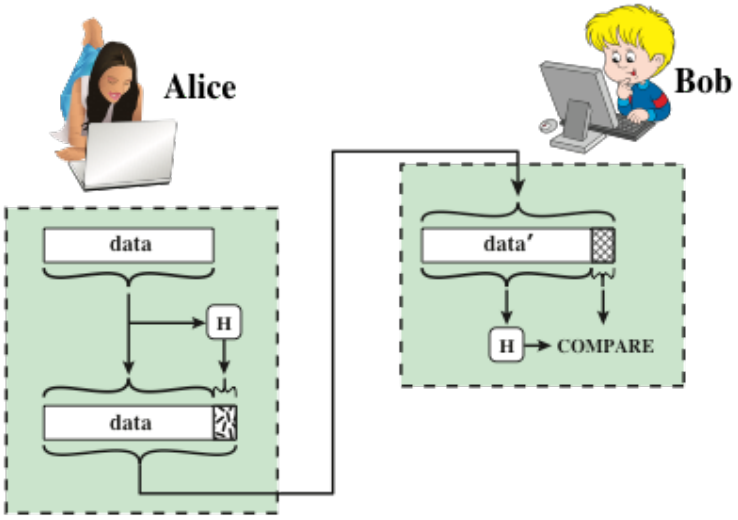
    (a) a data object that maps to a pre-specified hash result (the one-way property)

    (b) two data objects that map to the same hash result (the collision-free property)

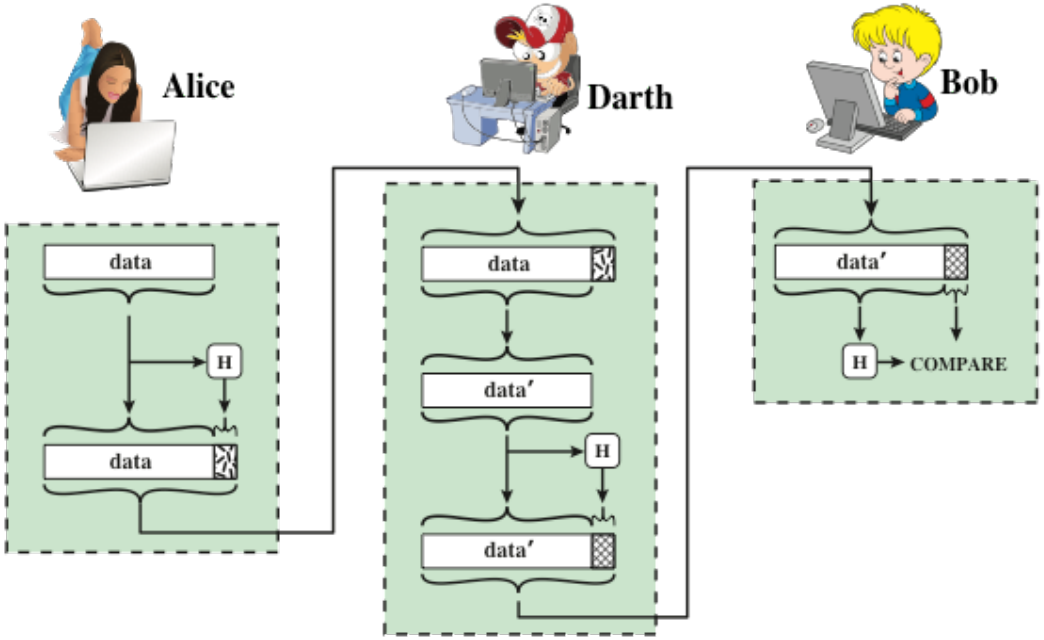


P, *L* = padding plus length field

Figure 11.1 Cryptographic Hash Function; $h = H(M)$



(a) Use of hash function to check data integrity



(b) Man-in-the-middle attack

Figure 11.2 Attack Against Hash Function

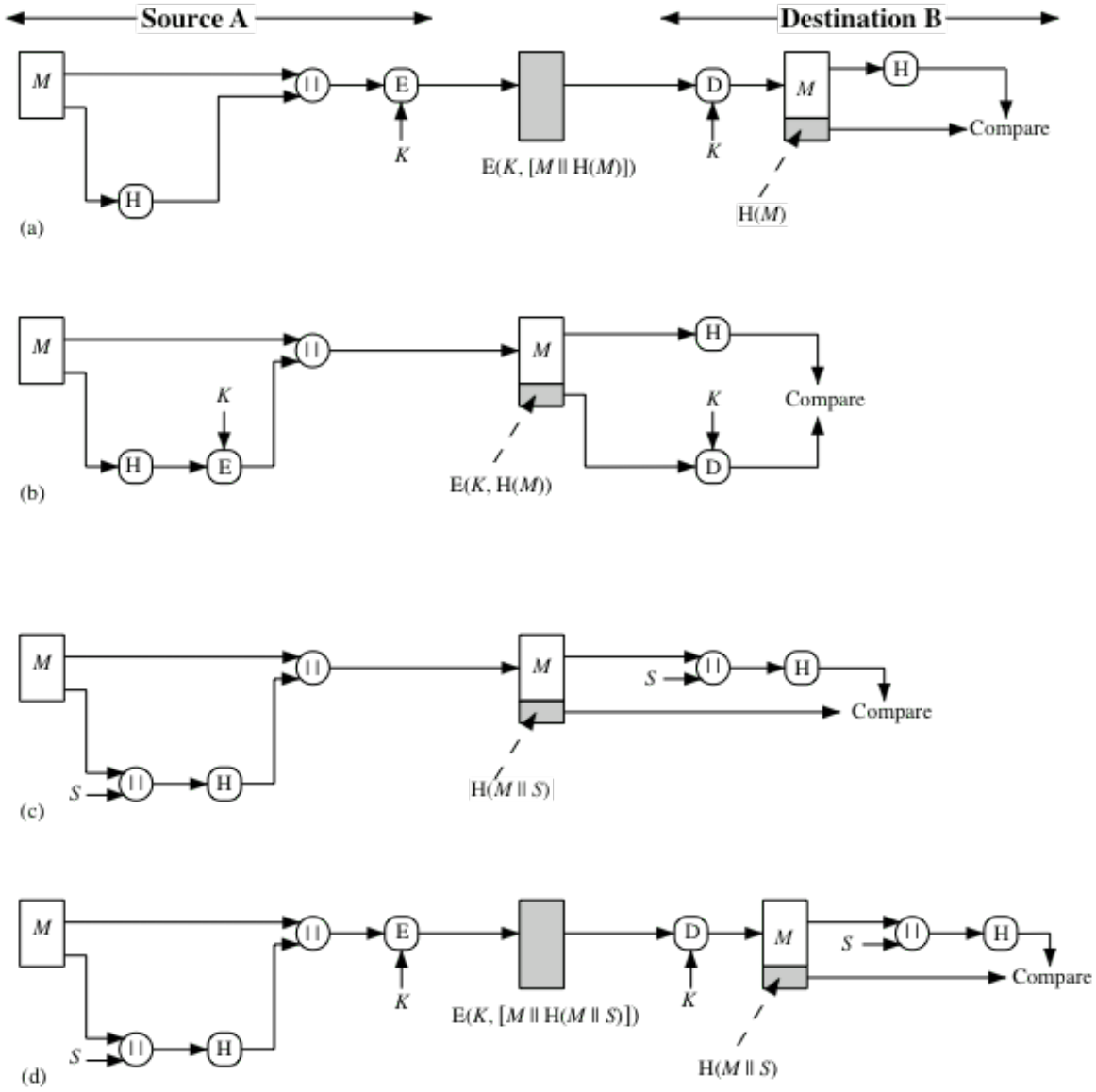


Figure 11.3 Simplified Examples of the Use of a Hash Function for Message Authentication

Message Authentication Code (MAC)

- Also known as a *keyed hash function*
- Typically used between two parties that share a secret key to authenticate information exchanged between those parties

Takes as input a secret key and a data block and produces a hash value (MAC) which is associated 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 associated MAC value
- An attacker who alters the message will be unable to alter the associated MAC value without knowledge of the secret key

Digital Signature

- Operation is similar to that of the MAC
- 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
- An attacker who wishes to alter the message would need to know the user's private key
- Implications of digital signatures go beyond just message authentication

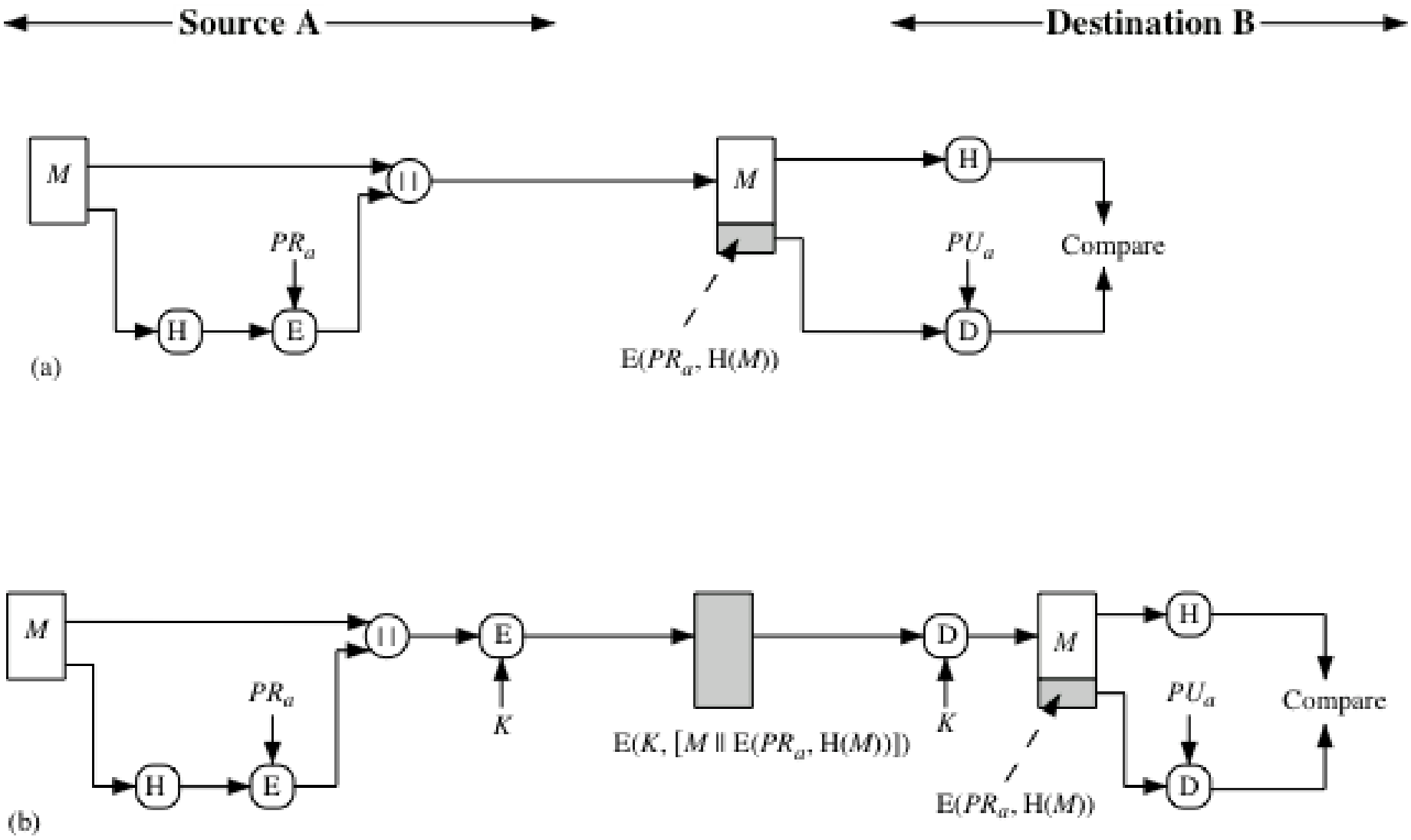


Figure 11.4 Simplified Examples of Digital Signatures

Other Hash Function Uses

Commonly used to create a one-way password file

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

Can be used for intrusion and virus detection

Store H(F) for each file on a system and secure the hash values

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)

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

Two Simple Hash Functions

- Consider two simple insecure hash functions that operate using the following general principles:
  - The input 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
- Bit-by-bit exclusive-OR (XOR) of every block
  - $C_i = b_{i1}\ xor\ b_{i2}\ xor\ .\ .\ .\ xor\ b_{im}$
  - Produces a simple parity for each bit position and is known as a longitudinal redundancy check
  - Reasonably effective for random data as a data integrity check
- Perform a one-bit circular shift on the hash value after each block is processed
  - Has the effect of randomizing the input more completely and overcoming any regularities that appear in the input

First Simple Hash Function

$$C_i = b_{i1} \oplus b_{i2} \oplus \cdots \oplus b_{im}$$

where

C_i = *i*th bit of the hash value, 1≤ i ≤n

m = number of n-bit blocks in the input

b_{ij}= *i*th bit in *j*th block

$\oplus$ = XOR operation

Example

- Input: 10010101 11001110 01101101

$$C_1 = 1 \oplus 1 \oplus 0 = 0$$
$$C_2 = 0 \oplus 1 \oplus 1 = 0$$
$$C_3 = 0 \oplus 0 \oplus 1 = 1$$
$$C_4 = 1 \oplus 0 \oplus 0 = 1$$
$$C_5 = 0 \oplus 1 \oplus 1 = 0$$
$$C_6 = 1 \oplus 1 \oplus 1 = 1$$
$$C_7 = 0 \oplus 1 \oplus 0 = 1$$
$$C_8 = 1 \oplus 0 \oplus 1 = 0$$

- Output: 00110110

Second Simple Hash Function

- Initially set the n-bit hash value to zero.
- Process each successive n-bit block of data as follows:
  - Rotate the current hash value to the left by one-bit
  - XOR the block into the hash value

Example of the Improvement

- Input: 10010101 11001110 01101101

Initial		0 0 0 0 0 0 0 0
B_1	$\oplus$	1 0 0 1 0 1 0 1
		1 0 0 1 0 1 0 1
$\leftarrow$		0 0 1 0 1 0 1 1
B_2	$\oplus$	1 1 0 0 1 1 1 0
		1 1 1 0 0 1 0 1
$\leftarrow$		1 1 0 0 1 0 1 1
B_3	$\oplus$	0 1 1 0 1 1 0 1
Output		1 0 1 0 0 1 1 0

Two Simple Hash Functions

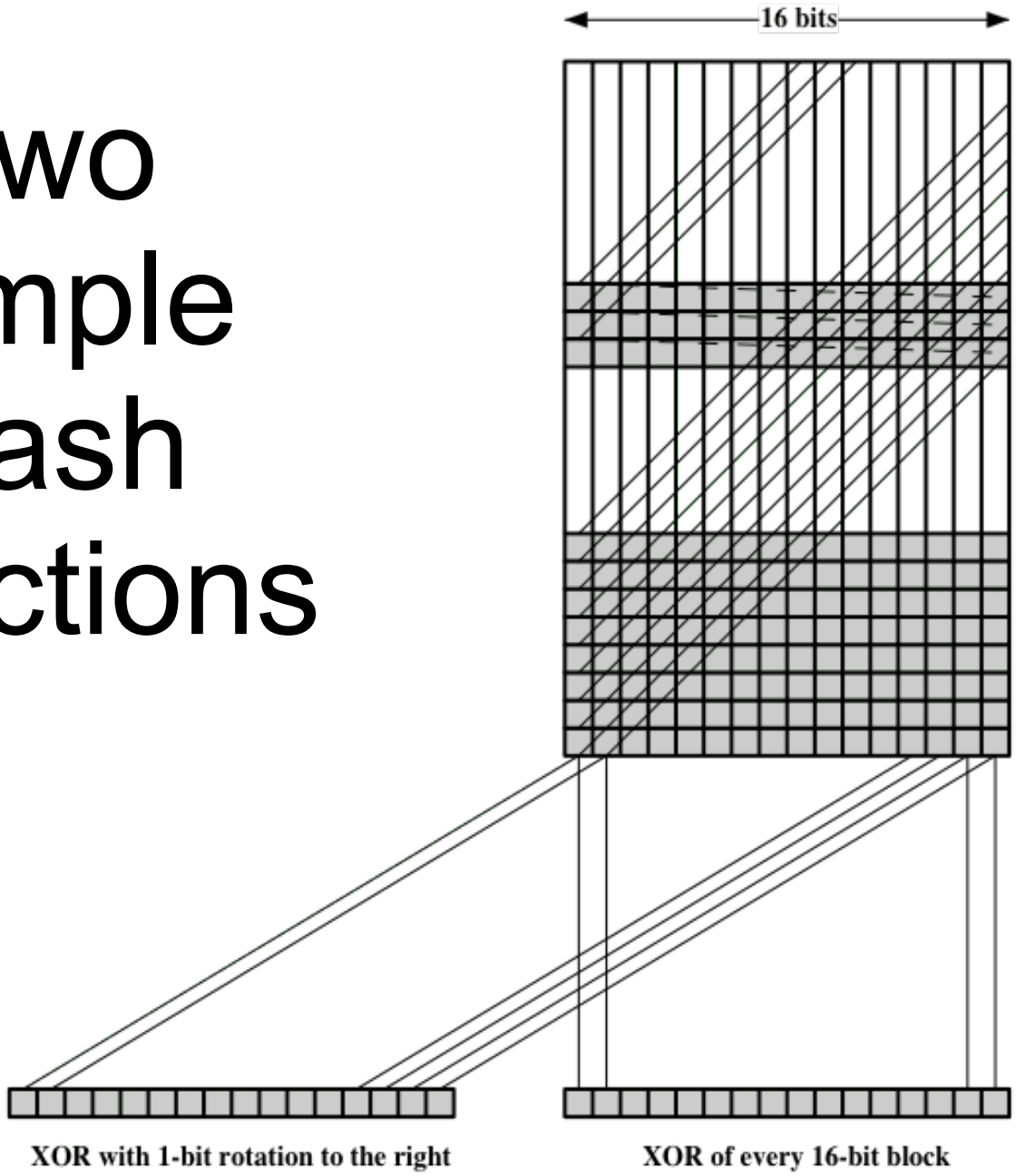
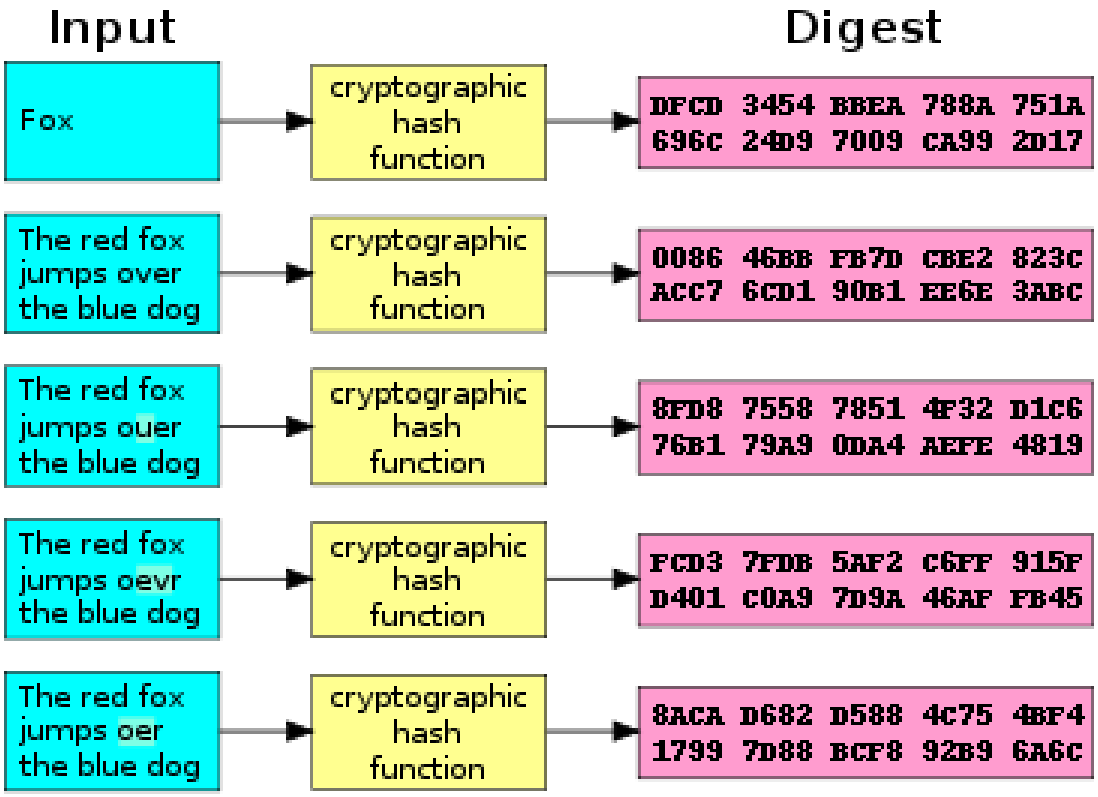


Figure 11.5 Two Simple Hash Functions

An example of the Input/Output



The hash function I used here is SHA-1. Note that even small changes in the source input (here in the word "over") drastically change the resulting output

Requirements and Security

Preimage

- *x* is the preimage of *h* for a hash value *h* = H(*x)*
- Is a data block whose hash function, using the function H, is *h*
- Because H is a many-to-one mapping, for any given hash value *h,* there will in general be multiple preimages

Collision

- Occurs if we have $x \neq y$ and H(*x)* = H(*y)*
- Because we are using hash functions for data integrity, collisions are clearly undesirable

Table 11.1

Requirements for a Cryptographic Hash Function H

Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	$H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that $H(y) = h$.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness

(Table can be found on page 323 in textbook.)

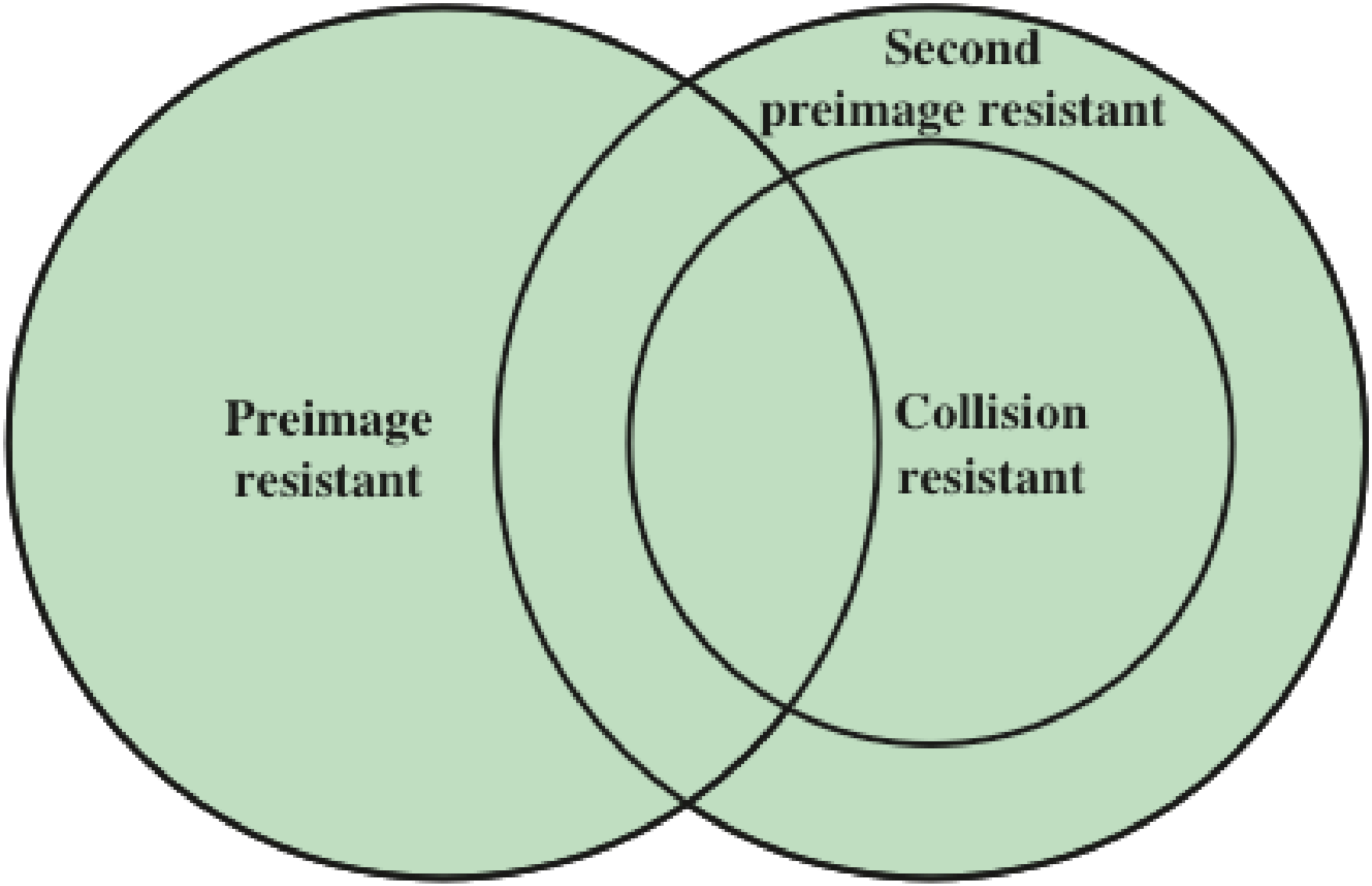


Figure 11.6 Relationship Among Hash Function Properties

Table 11.2
Hash Function Resistance Properties Required for Various Data Integrity Applications

	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + digital signature	yes	yes	yes*
Intrusion detection and virus detection		yes	
Hash + symmetric encryption			
One-way password file	yes		
MAC	yes	yes	yes*

It shows the resistant properties required for various hash function applications.

Attacks on Hash Functions

Brute-Force Attacks

- Does not depend on the specific algorithm, only depends on bit length
- In the case of a hash function, attack depends only on the bit length of the hash value
- Method is to pick values at random and try each one until a collision occurs

Cryptanalysis

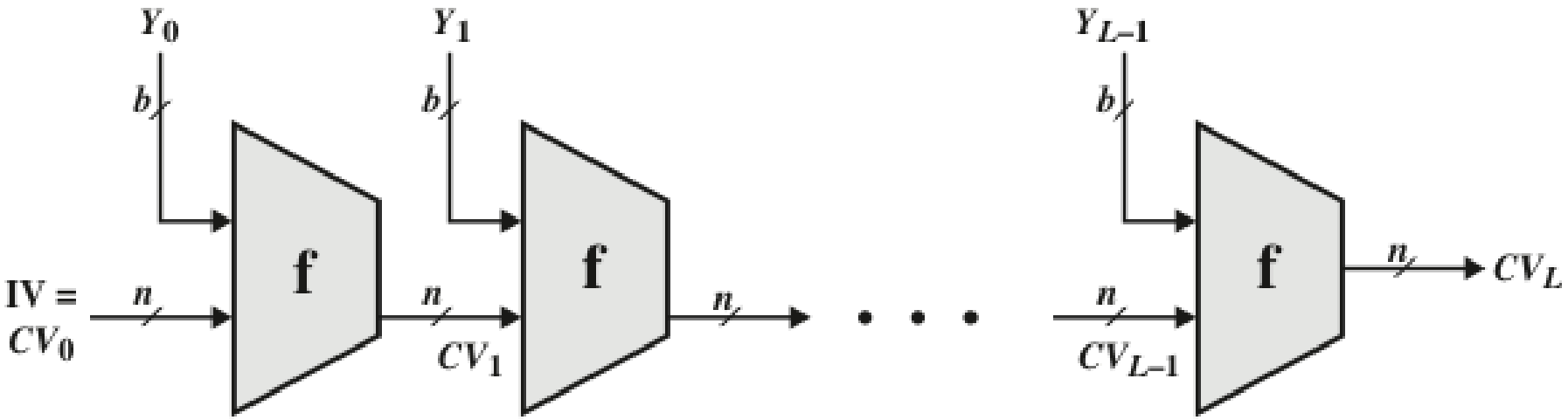
- An attack based on weaknesses in a particular cryptographic algorithm
- Seek to exploit some property of the algorithm to perform some attack other than an exhaustive search

Well-Known Hash Functions

- MD2, MD4, MD5
- SHA-0, SHA-1, SHA-2
- RIPEMD, RIPEMD-128/256, RIPEMD-160/320
- GOST
- HAVAL
- PANAMA
- RadioGatun
- WHIRLPOOL
- Tiger(2) – 192/160/128

Well-Known Hash Functions

- ~~MD2~~, ~~MD4~~, ~~MD5~~
- ~~SHA-0~~, SHA-1, SHA-2 → SHA-3
- ~~RIPEMD~~, RIPEMD-128/256, RIPEMD-160/320
- ~~GOST~~
- ~~HAVAL~~
- ~~PANAMA~~
- ~~RadioGatun~~
- WHIRLPOOL
- ~~Tiger(2) – 192/160/128~~



IV = Initial value
CV_i = chaining variable
Y_i = ith input block
f = compression algorithm
L = number of input blocks
n = length of hash code
b = length of input block

Figure 11.8 General Structure of Secure Hash Code

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 proposals was that of Rabin
  - 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

    H_0= initial value
    $H_i = E(M_i, H_{i-1})$
    $G = H_N$
  - 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
  - If the encryption algorithm is DES and only a 64-bit hash code is produced, the system is vulnerable
- Meet-in-the-middle-attack
  - Another version of the birthday attack used even if the opponent has access to only one message and its valid signature and cannot obtain multiple signings
- It can be shown that some form of birthday attack will succeed against any hash scheme involving the use of cipher block chaining without a secret key, provided that either the resulting hash code is small enough or that a larger hash code can be decomposed into independent subcodes

Secure Hash Algorithm (SHA)

- SHA was originally designed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993
- Was revised in 1995 as SHA-1
- Based on the hash function MD4 and its design closely models MD4
- Produces 160-bit hash values
- In 2002 NIST produced a revised version of the standard that defined three new versions of SHA with hash value lengths of 256, 384, and 512
  - Collectively known as SHA-2

Table 11.3
Comparison of SHA Parameters

Algorithm	Message Size	Block Size	Word Size	Message Digest Size
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Note: All sizes are measured in bits.

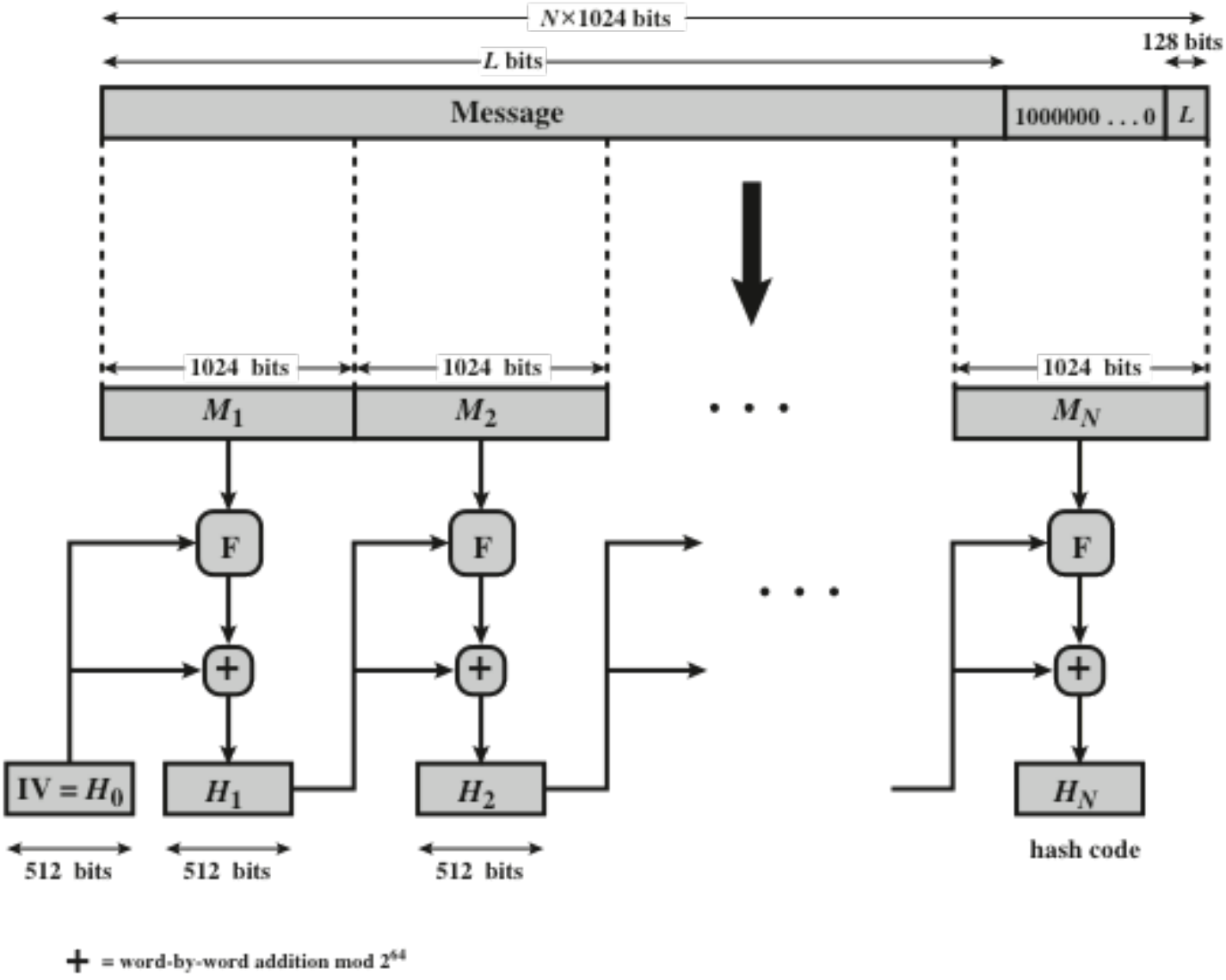


Figure 11.9 Message Digest Generation Using SHA-512

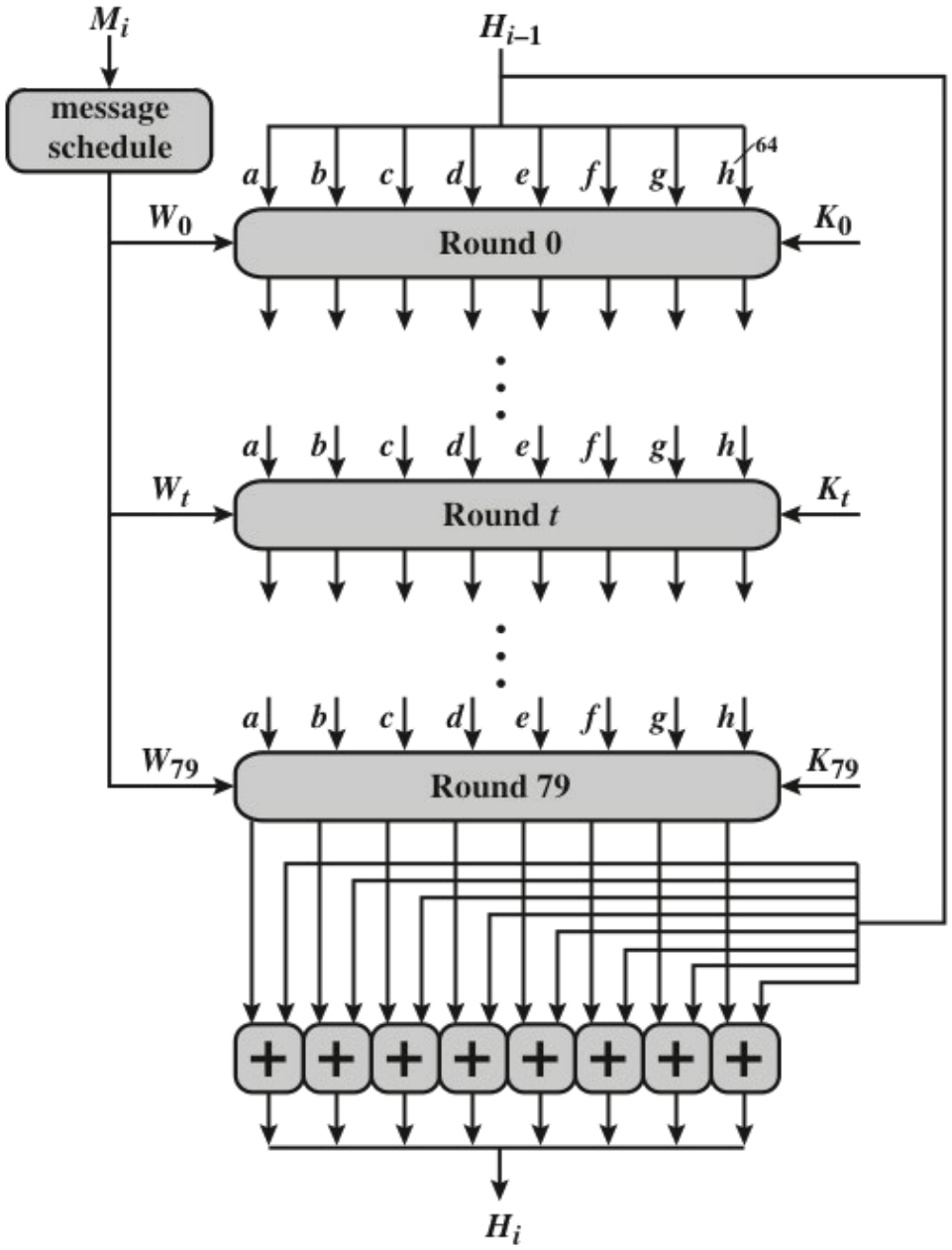


Figure 11.10 SHA-512 Processing of a Single 1024-Bit Block

Table 11.4 ---- SHA-512 Constants

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

(Table can be found on page 341 in textbook)

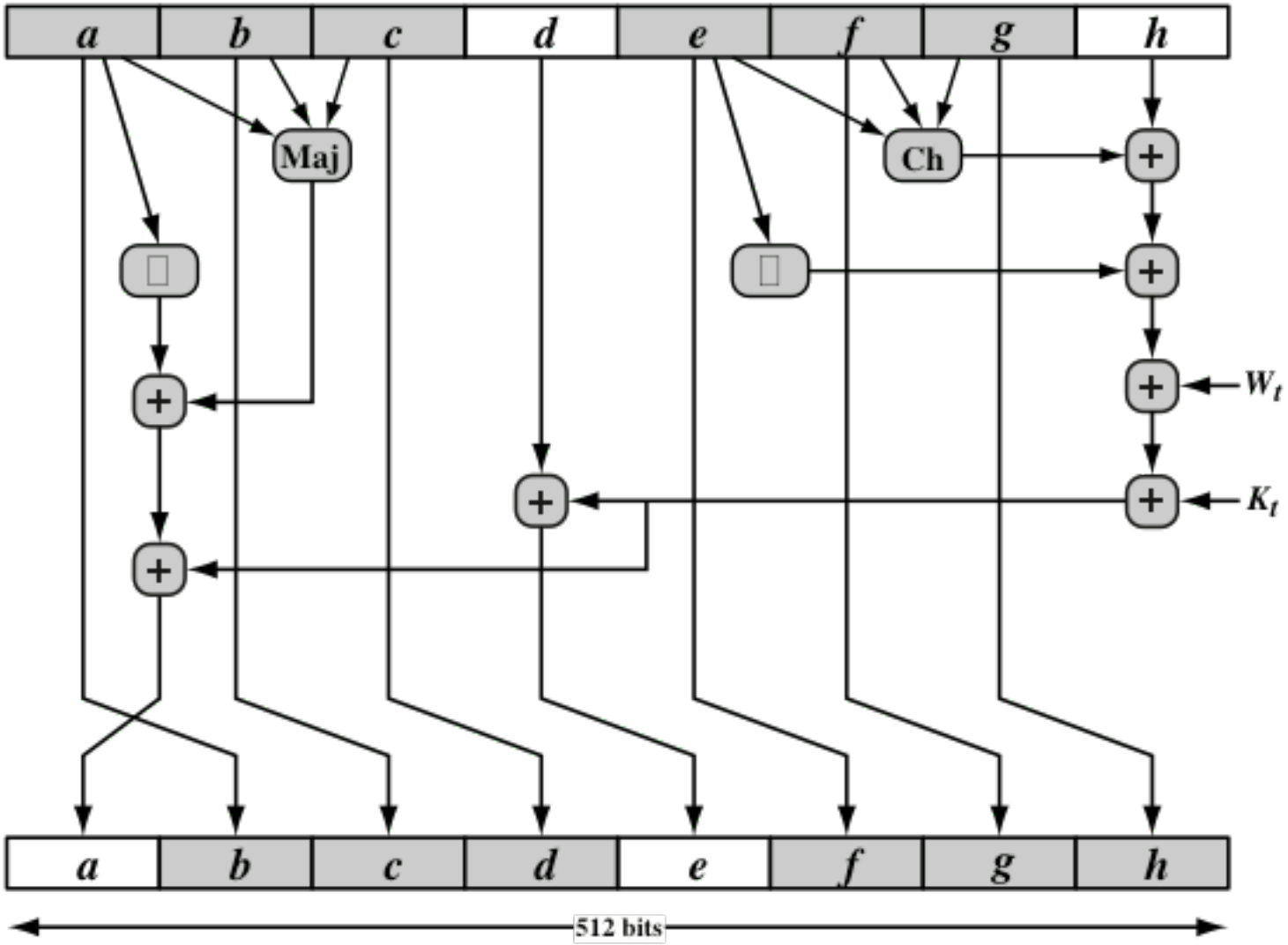


Figure 11.11 Elementary SHA-512 Operation (single round)

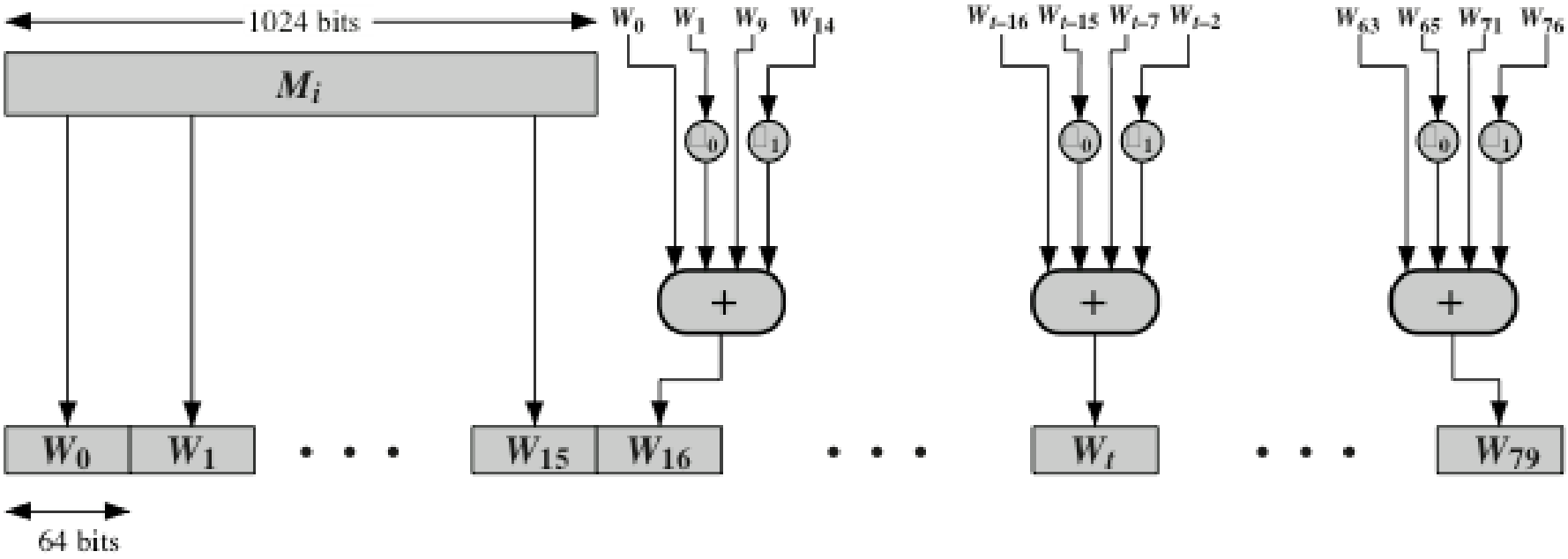


Figure 11.12 Creation of 80-word Input Sequence for SHA-512 Processing of Single Block

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}.

$H_{0,0} = \texttt{6A09E667F3BCC908}$ $\quad$ $H_{0,4} = \texttt{510E527FADE682D1}$

$H_{0,1} = \texttt{BB67AE8584CAA73B}$ $\quad$ $H_{0,5} = \texttt{9B05688C2B3E6C1F}$

$H_{0,2} = \texttt{3C6EF372FE94F82B}$ $\quad$ $H_{0,6} = \texttt{1F83D9ABFB41BD6B}$

$H_{0,3} = \texttt{A54FF53A5F1D36F1}$ $\quad$ $H_{0,7} = \texttt{5BE0CDI9137E2179}$

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
   $a = H_{i-1,0}$ $\quad$ $e = H_{i-1,4}$
   $b = H_{i-1,1}$ $\quad$ $f = H_{i-1,5}$
   $c = H_{i-1,2}$ $\quad$ $g = H_{i-1,6}$
   $d = H_{i-1,3}$ $\quad$ $h = H_{i-1,7}$
3. Perform the main hash computation
   for $t = 0$ **to** 79
   $T_1 = h + \text{Ch}(e, f, g) + \left(\sum_1^{512} e\right) + W_t + K_t$
   $T_2 = \left(\sum_0^{512} a\right) + \text{Maj}(a, b, c)$
   $h = g$
   $g = f$
   $f = e$
   $e = d + T_1$
   $d = c$
   $c = b$
   $b = a$
   $a = T_1 + T_2$
4. Compute the intermediate hash value
   $H_{i,0} = a + H_{i-1,0}$ $\quad$ $H_{i,4} = e + H_{i-1,4}$
   $H_{i,1} = b + H_{i-1,1}$ $\quad$ $H_{i,5} = f + H_{i-1,5}$
   $H_{i,2} = c + H_{i-1,2}$ $\quad$ $H_{i,6} = g + H_{i-1,6}$
   $H_{i,3} = d + H_{i-1,3}$ $\quad$ $H_{i,7} = h + H_{i-1,7}$

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 can be found on page 345 in textbook)

Figure 11.13 SHA-512 Logic

SHA-3

SHA-1 has not yet been "broken"

- **No one has demonstrated a technique for producing collisions in a practical amount of time**
- **Considered to be insecure and has been phased out for SHA-2**

NIST announced in 2007 a competition for the SHA-3 next generation NIST hash function

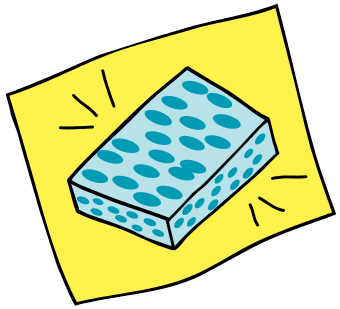
- **Winning design was announced by NIST in October 2012**
- **SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications**

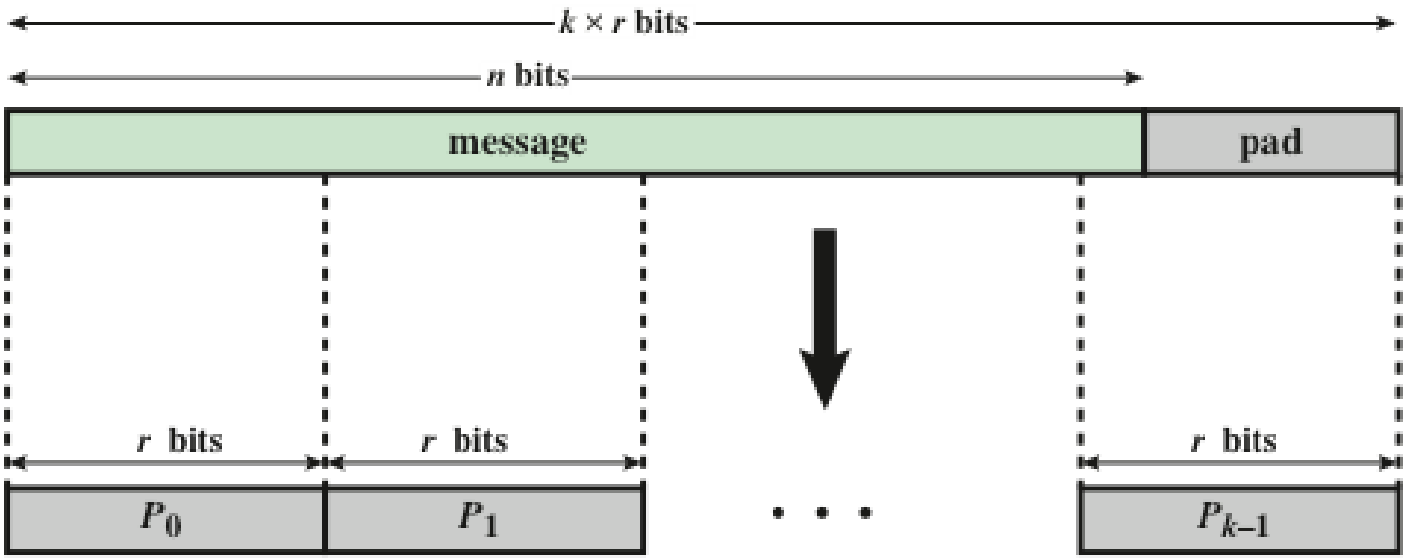
SHA-2 shares the same structure and mathematical operations as its predecessors so this is a cause for concern

- **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**

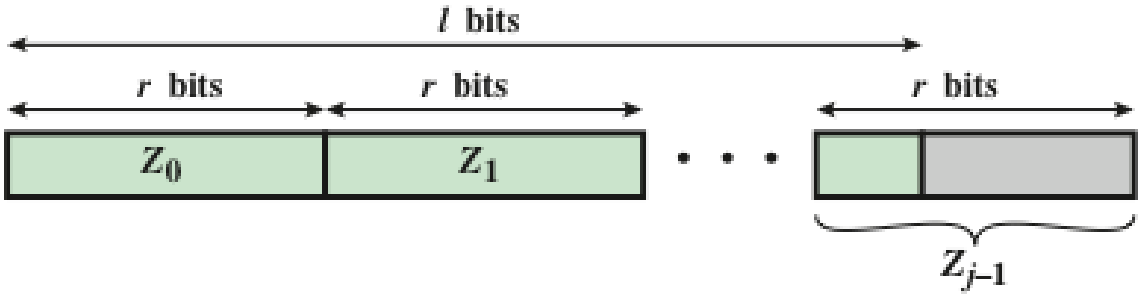
The Sponge Construction

- Underlying structure of SHA-3 is a scheme referred to by its designers as a *sponge construction*
- 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
  - r = the size in bits of the input blocks, called the *bitrate*
  - pad = the padding algorithm





(a) Input



(b) Output

Figure 11.14 Sponge Function Input and Output

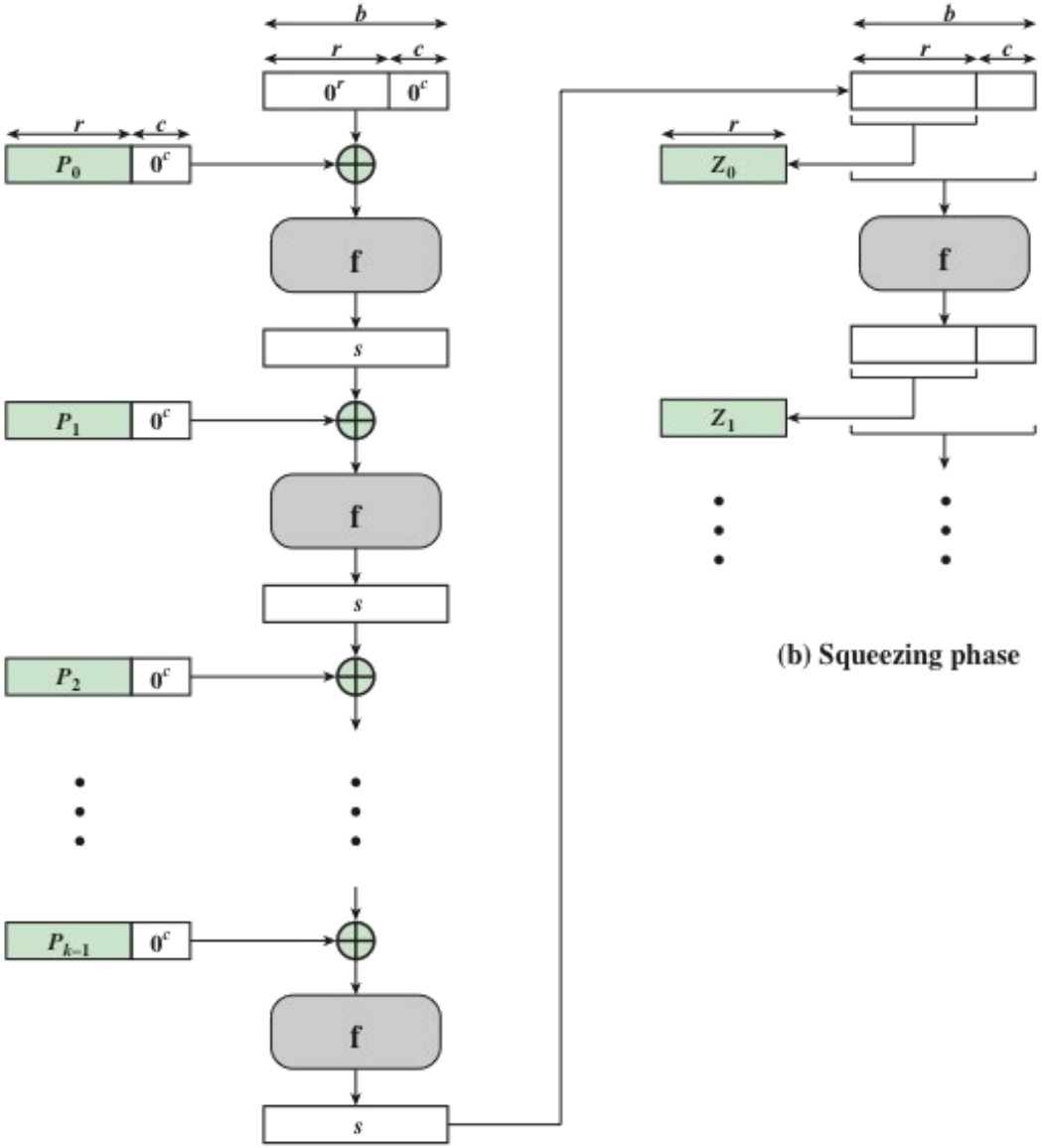


Figure 11.15 Sponge Construction

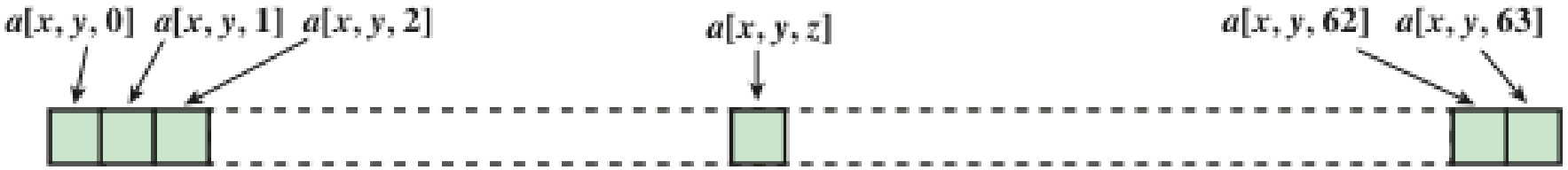
Table 11.5 SHA-3 Parameters

Message Digest Size	224	256	384	512
Message Size	no maximum	no maximum	no maximum	no maximum
Block Size (bitrate r)	1152	1088	832	576
Word Size	64	64	64	64
Number of Rounds	24	24	24	24
Capacity c	448	512	768	1024
Collision resistance	2^{112}	2^{128}	2^{192}	2^{256}
Second preimage resistance	2^{224}	2^{256}	2^{384}	2^{512}

It shows the supported values of r and c.

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	$L[0, 4]$	$L[1, 4]$	$L[2, 4]$	$L[3, 4]$	$L[4, 4]$
$y = 3$	$L[0, 3]$	$L[1, 3]$	$L[2, 3]$	$L[3, 3]$	$L[4, 3]$
$y = 2$	$L[0, 2]$	$L[1, 2]$	$L[2, 2]$	$L[3, 2]$	$L[4, 2]$
$y = 1$	$L[0, 1]$	$L[1, 1]$	$L[2, 1]$	$L[4, 1]$	$L[4, 1]$
$y = 0$	$L[0, 0]$	$L[1, 0]$	$L[2, 0]$	$L[3, 0]$	$L[4, 0]$

(a) State variable as 5 ×5 matrix A of 64-bit words



(b) Bit labeling of 64-bit words

Figure 11.16 SHA-3 State Matrix

SHA-3
Iteration Function *f*

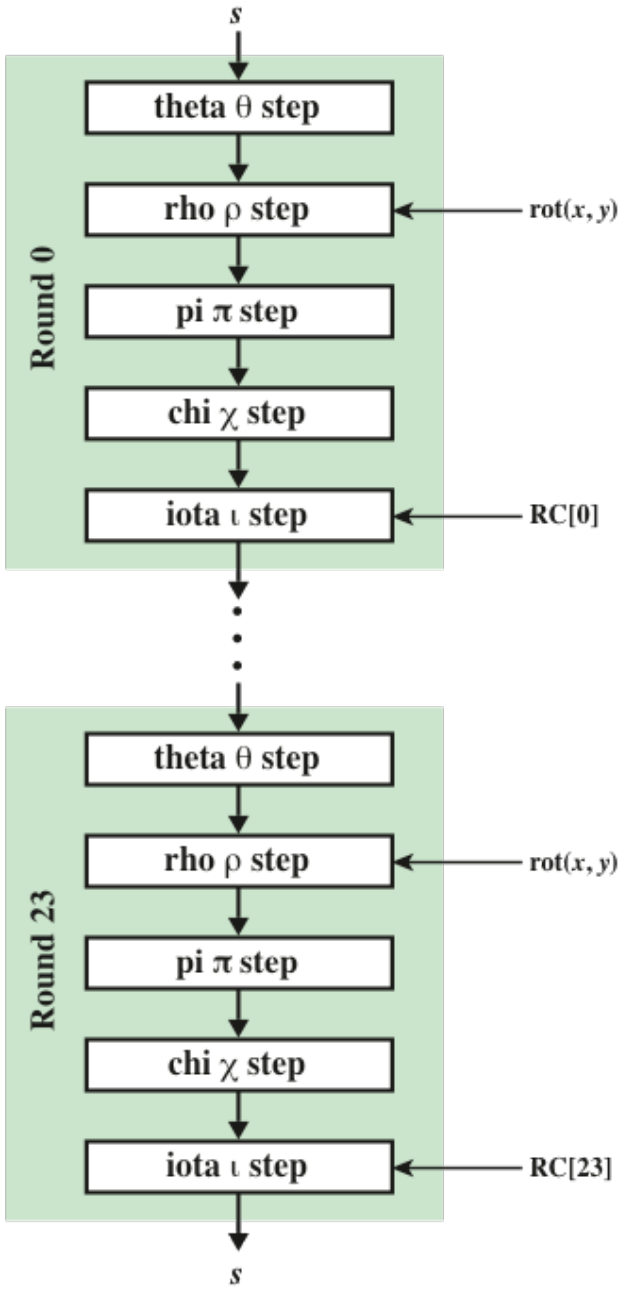
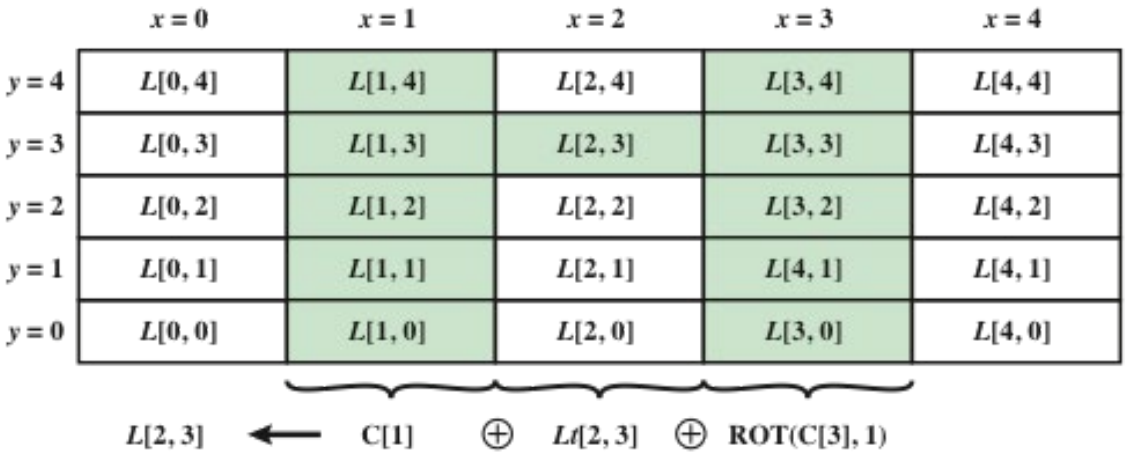


Figure 11.17 SHA-3 Iteration Function *f*

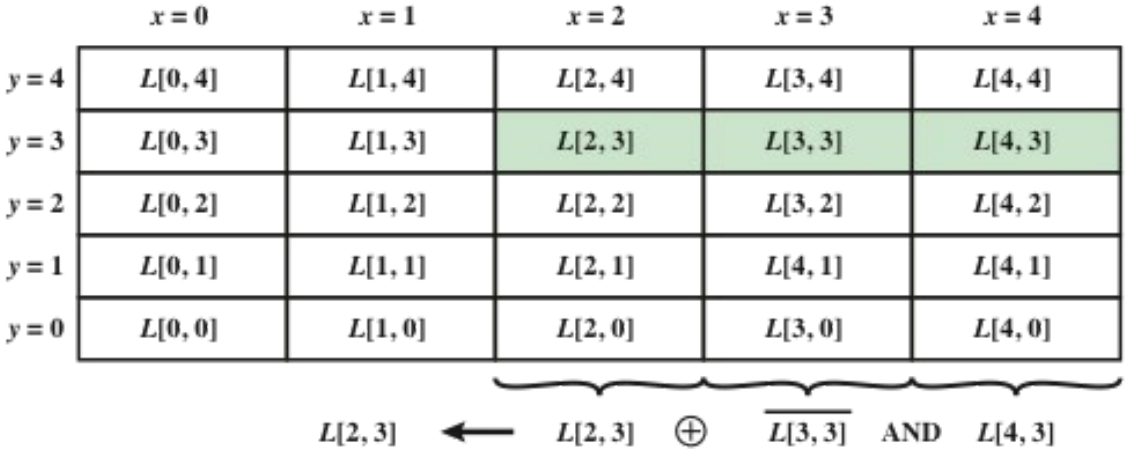
Table 11.6

Step Functions in SHA-3

Function	Type	Description
θ	**Substitution**	**New value of each bit in each word depends its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.**
ρ	**Permutation**	**The bits of each word are permuted using a circular bit shift. $W[0, 0]$ is not affected.**
π	**Permutation**	**Words are permuted in the 5×5 matrix. $W[0, 0]$ is not affected.**
χ	**Substitution**	**New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.**
ι	**Substitution**	**$W[0, 0]$ is updated by XOR with a round constant.**

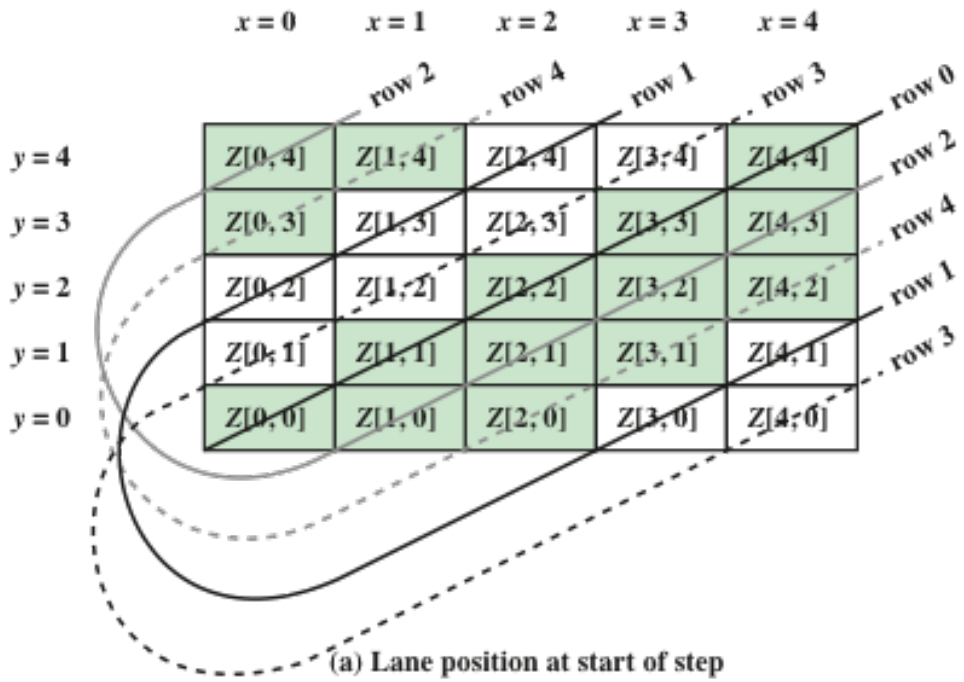


(a) θ step function



(b) χ step function

Figure 11.18 Theta and Chi Step Functions



(a) Lane position at start of step

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 4	Z[2, 0]	Z[3, 1]	Z[4, 2]	Z[0, 3]	Z[1, 4]
y = 3	Z[4, 0]	Z[0, 1]	Z[1, 2]	Z[2, 3]	Z[3, 4]
y = 2	Z[1, 0]	Z[2, 1]	Z[3, 2]	Z[4, 3]	Z[0, 4]
y = 1	Z[3, 0]	Z[4, 1]	Z[0, 2]	Z[1, 3]	Z[2, 4]
y = 0	Z[0, 0]	Z[1, 1]	Z[2, 2]	Z[3, 3]	Z[4, 4]

(b) Lane position after permutation

Figure 11.19 Pi Step Function

Table 11.8
Round Constants in SHA-3

Round	Constant (hexadecimal)	Number of 1 bits
0	0000000000000001	1
1	0000000000008082	3
2	800000000000808A	5
3	8000000080008000	3
4	000000000000808B	5
5	0000000080000001	2
6	8000000080008081	5
7	8000000000008009	4
8	000000000000008A	3
9	0000000000000088	2
10	0000000080008009	4
11	000000008000000A	3

Round	Constant (hexadecimal)	Number of 1 bits
12	000000008000808B	6
13	800000000000008B	5
14	8000000000008089	5
15	8000000000008003	4
16	8000000000008002	3
17	8000000000000080	2
18	000000000000800A	3
19	800000008000000A	4
20	8000000080008081	5
21	8000000000008080	3
22	0000000080000001	2
23	8000000080008008	4