10.6 Suppose Alice and Bob use an Elgamal scheme with a common prime $q = 157$ and a primitive root $\alpha = 5$.

a. If Bob has public key $Y_B = 10$ and Alice chose the random integer $k = 3$, what is the ciphertext of $M = 9$?

b. If Alice now chooses a different value of k so that the encoding of $M = 9$ is $C = (25, C_2)$, what is the integer C_2?

10.7 Rule (5) for doing arithmetic in elliptic curves over real numbers states that to double a point Q_2, draw the tangent line and find the other point of intersection S. Then $Q + Q = 2Q = -S$. If the tangent line is not vertical, there will be exactly one point of intersection. However, suppose the tangent line is vertical? In that case, what is the value $2Q$? What is the value $3Q$?

10.8 Demonstrate that the two elliptic curves of Figure 10.4 each satisfy the conditions for a group over the real numbers.

10.9 Is $(5, 12)$ a point on the elliptic curve $y^2 = x^3 + 4x - 1$ over real numbers?

10.10 On the elliptic curve over the real numbers $y^2 = x^3 - \frac{17}{12}x + 1$, let $P = (0, 1)$ and $Q = (1.5, 1.5)$. Find $P + Q$ and $2P$.

10.11 Does the elliptic curve equation $y^2 = x^3 + x + 2$ define a group over Z_7?

10.12 Consider the elliptic curve $E_7(2, 1)$; that is, the curve is defined by $y^2 = x^3 + 2x + 1$ with a modulus of $p = 7$. Determine all of the points in $E_7(2, 1)$. *Hint:* Start by calculating the right-hand side of the equation for all values of x.

10.13 What are the negatives of the following elliptic curve points over Z_7? $P = (3, 5)$; $Q = (2, 5)$; and $R = (5, 0)$.

10.14 For $E_{11}(1, 7)$, consider the point $G = (3, 2)$. Compute the multiple of G from $2G$ through $13G$.

10.15 This problem performs elliptic curve encryption/decryption using the scheme outlined in Section 10.4. The cryptosystem parameters are $E_{11}(1, 7)$ and $G = (3, 2)$. B's private key is $n_B = 7$.

a. Find B's public key P_B.

b. A wishes to encrypt the message $P_m = (10, 7)$ and chooses the random value $k = 5$. Determine the ciphertext C_m.

c. Show the calculation by which B recovers P_m from C_m.

10.16 The following is a first attempt at an elliptic curve signature scheme. We have a global elliptic curve, prime p, and "generator" G. Alice picks a private signing key X_A and forms the public verifying key $Y_A = X_A G$. To sign a message M:

- Alice picks a value k.
- Alice sends Bob M, k, and the signature $S = M - kX_A G$.
- Bob verifies that $M = S + kY_A$.

a. Show that this scheme works. That is, show that the verification process produces an equality if the signature is valid.

b. Show that the scheme is unacceptable by describing a simple technique for forging a user's signature on an arbitrary message.

10.17 Here is an improved version of the scheme given in the previous problem. As before, we have a global elliptic curve, prime p, and "generator" G. Alice picks a private signing key X_A and forms the public verifying key $Y_A = X_A G$. To sign a message M:

- Bob picks a value k.
- Bob sends Alice $C_1 = kG$.
- Alice sends Bob M and the signature $S = M - X_A C_1$.
- Bob verifies that $M = S + kY_A$.

a. Show that this scheme works. That is, show that the verification process produces an equality if the signature is valid.

b. Show that forging a message in this scheme is as hard as breaking (ElGamal) elliptic curve cryptography. (Or find an easier way to forge a message?)

c. This scheme has an extra "pass" compared to other cryptosystems and signature schemes we have looked at. What are some drawbacks to this?

PART FOUR: CRYPTOGRAPHIC DATA INTEGRITY ALGORITHMS

CHAPTER 11

CRYPTOGRAPHIC HASH FUNCTIONS

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Summarize the applications of cryptographic hash functions.
- Explain why a hash function used for message authentication needs to be secured.
- Understand the differences among preimage resistant, second preimage resistant, and collision resistant properties.
- Present an overview of the basic structure of cryptographic hash functions.
- Describe how cipher block chaining can be used to construct a hash function.
- Understand the operation of SHA-512.

A **hash function** H accepts a variable-length block of data M as input and produces a fixed-size result $h = \mathrm{H}(M)$, referred to as a **hash value** or a **hash code**. 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 value.

The kind of hash function needed for security applications is referred to as a **cryptographic hash function**. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly more efficient than brute force) to find either (a) a data object that maps to a pre-specified hash result (the one-way property) or (b) two data objects that map to the same hash result (the collision-free property). Because of these characteristics, 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, as explained subsequently.

This chapter begins with a discussion of the wide variety of applications for cryptographic hash functions. Next, we look at the security requirements for such functions. Then we look at the use of cipher block chaining to implement a cryptographic hash function. The remainder of the chapter is devoted to the most important and widely used family of cryptographic hash functions, the Secure Hash Algorithm (SHA) family.

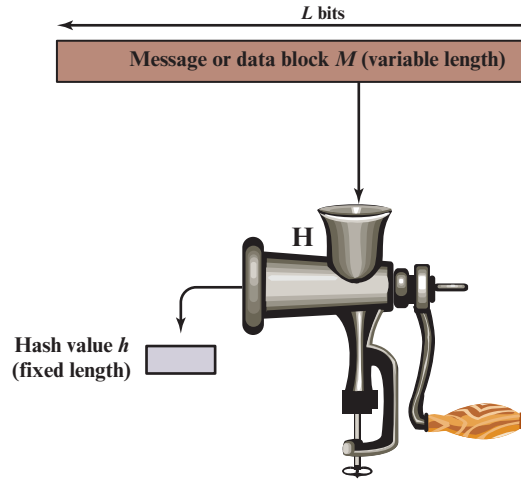


Figure 11.1 Cryptographic Hash Function; $h = \mathrm{H}(M)$

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

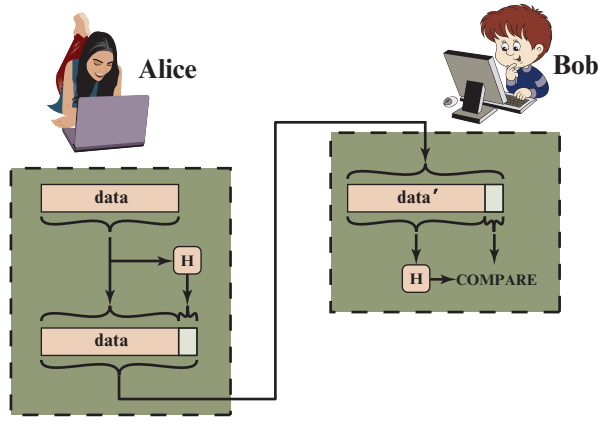
Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., there is no modification, insertion, deletion, or replay). In many cases, there is a requirement that the authentication mechanism assures that the 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**.[1]

The essence of the use of a hash function for message integrity is as follows. The sender computes a hash value as a function of the bits in the message and transmits both the hash value and the message. The receiver performs the same hash calculation on the message bits and compares this value with the incoming hash value.

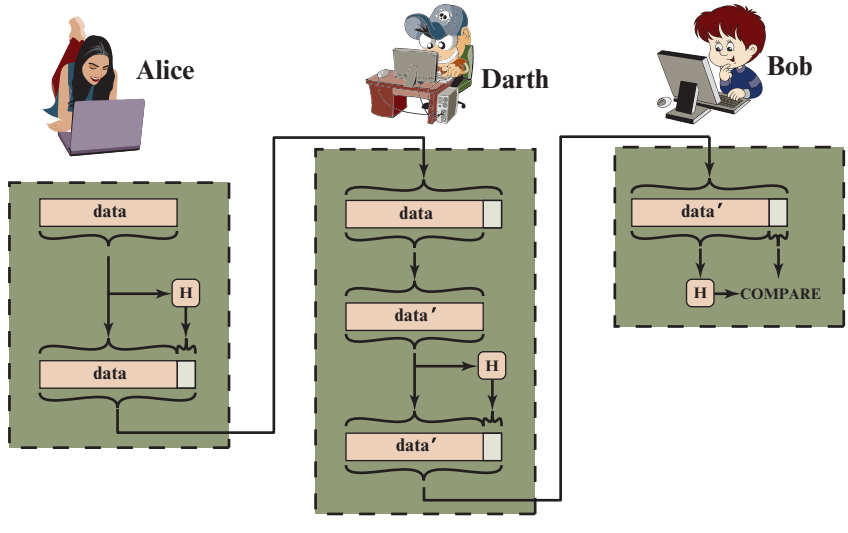
[1]The topic of this section is invariably referred to as message authentication. However, the concepts and techniques apply equally to data at rest. For example, authentication techniques can be applied to a file in storage to assure that the file is not tampered with.

If there is a mismatch, the receiver knows that the message (or possibly the hash value) has been altered (Figure 11.2a).

The hash value must be transmitted in a secure fashion. That is, the hash value must be protected so that if an adversary alters or replaces the message, it is not feasible for adversary to also alter the hash value to fool the receiver. This type of attack is shown in Figure 11.2b. In this example, Alice transmits a data block and attaches a hash value. Darth intercepts the message, alters or replaces the data block, and calculates and attaches a new hash value. Bob receives the altered data with the new hash value and does not detect the change. To prevent this attack, the hash value generated by Alice must be protected.



(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 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows.

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.

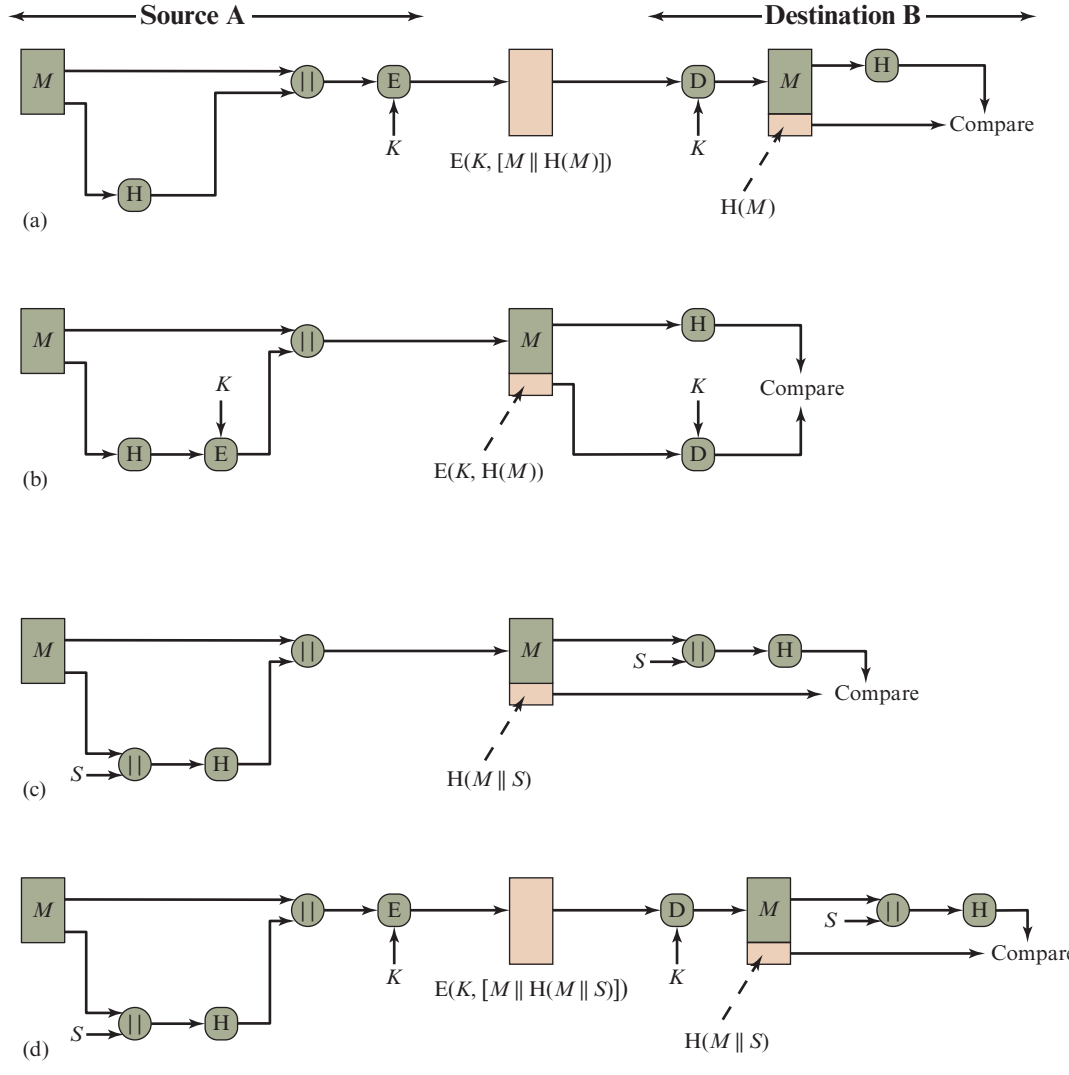


Figure 11.3 Simplified Examples of the Use of a Hash Function for Message Authentication

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 S. A computes the hash value over the concatenation of M and S and 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. Nevertheless, there has been growing interest in techniques that avoid encryption (Figure 11.3c). Several reasons for this interest are pointed out in [TSUD92].

- Encryption software is relatively slow. Even though the amount of data to be encrypted per message is small, there may be a steady stream of messages into and out of a system.
- Encryption hardware costs are not negligible. Low-cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have this capability.
- Encryption hardware is optimized toward large data sizes. For small blocks of data, a high proportion of the time is spent in initialization/invocation overhead.
- Encryption algorithms may be covered by patents, and there is a cost associated with licensing their use.

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, 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. 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 11.3b). 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

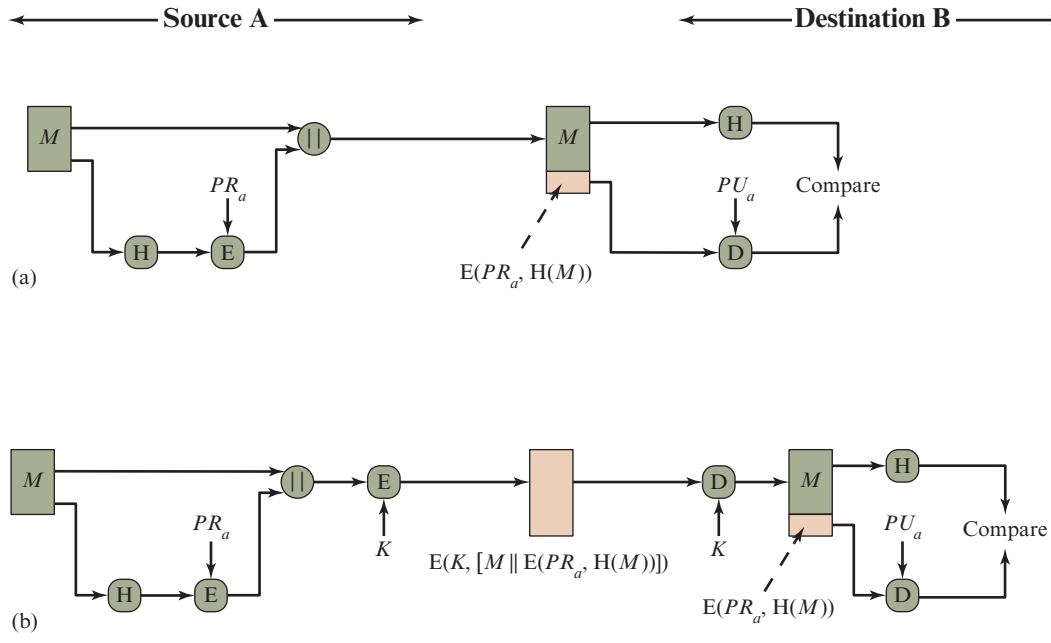


Figure 11.4 Simplified Examples of Digital Signatures

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 11.4 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 with the sender's private key. As with Figure 11.3b, 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 Applications

Hash functions are commonly used to create a one-way password file. Chapter 24 explains a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, 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. We discuss this application in Chapter 12.

11.2 TWO SIMPLE HASH FUNCTIONS

To get some feel for the security considerations involved in cryptographic hash functions, we present two simple, insecure hash functions in this section. 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

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

where

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

This operation produces a simple parity bit 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 most normal text files, the high-order bit of each octet is always zero. So if a 128-bit hash value is used, instead of an effectiveness of 2^{-128}, the hash function on this type of data has an effectiveness of 2^{-112}.

A simple way to improve matters is to perform a one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as follows.

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

This has the effect of "randomizing" the input more completely and overcoming any regularities that appear in the input. Figure 11.5 illustrates these two types of hash functions for 16-bit hash values.

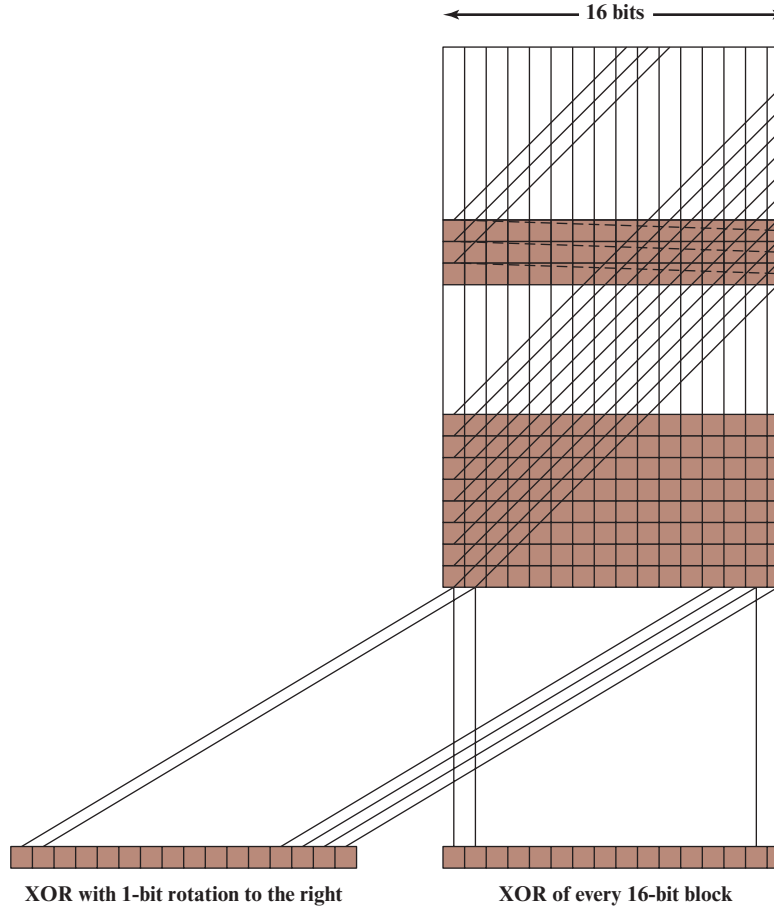


Figure 11.5 Two Simple Hash Functions

Although the second procedure provides a good measure of data integrity, it is virtually useless for data security when an encrypted hash code is used with a plaintext message, as in Figures 11.3b and 11.4a. Given a message, it is an easy matter to produce a new message that yields that hash code: Simply prepare the desired alternate message and then append an n-bit block that forces the new message plus block to yield the desired hash code.

Although a simple XOR or rotated XOR (RXOR) is insufficient if only the hash code is encrypted, you may still feel that such a simple function could be useful when the message together with the hash code is encrypted (Figure 11.3a). But you must be careful. A technique originally proposed by the National Bureau of Standards used the simple XOR applied to 64-bit blocks of the message and then an encryption of the entire message that used the cipher block chaining (CBC) mode. We can define the scheme as follows: Given a message M consisting of a sequence of 64-bit blocks $X_1, X_2, \ldots, X_N$, define the hash code $h = \mathrm{H}(M)$

as the block-by-block XOR of all blocks and append the hash code as the final block:

$$h = X_{N+1} = X_1 \oplus X_2 \oplus \ \ldots \ \oplus X_N$$

Next, encrypt the entire message plus hash code using CBC mode to produce the encrypted message $Y_1, Y_2, \ldots, Y_{N+1}$. [JUEN85] points out several ways in which the ciphertext of this message can be manipulated in such a way that it is not detectable by the hash code. For example, by the definition of CBC (Figure 6.4), we have

$$\begin{aligned} X_1 &= IV \oplus \mathrm{D}(K, Y_1) \\ X_i &= Y_{i-1} \oplus \mathrm{D}(K, Y_i) \\ X_{N+1} &= Y_N \oplus \mathrm{D}(K, Y_{N+1}) \end{aligned}$$

But X_{N+1} is the hash code:

$$\begin{aligned} X_{N+1} &= X_1 \oplus X_2 \oplus \ \ldots \ \oplus X_N \\ &= [IV \oplus \mathrm{D}(K, Y_1)] \oplus [Y_1 \oplus \mathrm{D}(K, Y_2)] \oplus \ \ldots \ \oplus [Y_{N-1} \oplus \mathrm{D}(K, Y_N)] \end{aligned}$$

Because the terms in the preceding equation can be XORed in any order, it follows that the hash code would not change if the ciphertext blocks were permuted.

11.3 REQUIREMENTS AND SECURITY

Before proceeding, we need to define two terms. For a hash value $h = \mathrm{H}(x)$, we say that x is the **preimage** of h. That is, x is a data block whose hash value, 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. A **collision** occurs if we have $x \neq y$ and $\mathrm{H}(x) = \mathrm{H}(y)$. Because we are using hash functions for data integrity, collisions are clearly undesirable.

Let us consider how many preimages are there for a given hash value, which is a measure of the number of potential collisions for a given hash value. Suppose the length of the hash code is n bits, and the function H takes as input messages or data blocks of length b bits with $b > n$. Then, the total number of possible messages is 2^b and the total number of possible hash values is 2^n. On average, each hash value corresponds to 2^{b-n} preimages. If H tends to uniformly distribute hash values then, in fact, each hash value will have close to 2^{b-n} preimages. If we now allow inputs of arbitrary length, not just a fixed length of some number of bits, then the number of preimages per hash value is arbitrarily large. However, the security risks in the use of a hash function are not as severe as they might appear from this analysis. To understand better the security implications of cryptographic hash functions, we need to precisely define their security requirements.

Security Requirements for Cryptographic Hash Functions

Table 11.1 lists the generally accepted requirements for a cryptographic hash function. The first three properties are requirements for the practical application of a hash function.

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) with $x \neq y$, such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

The fourth property, preimage 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 11.3c). 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 $h = H(S \| M)$. The attacker then inverts the hash function to obtain $S \| M = H^{-1}(MD_M)$. Because the attacker now has both M and $S \| M$, it is a trivial matter to recover S.

The fifth property, second preimage resistant, guarantees that it is infeasible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used (Figures 11.3b and 11.4a). 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.

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. For example, suppose Bob writes an IOU message, sends it to Alice, and she signs it. Bob finds two messages with the same hash, one of which requires Alice to pay a small amount and one that requires a large payment. Alice signs the first message, and Bob is then able to claim that the second message is authentic.

Figure 11.6 shows the relationships among the three resistant properties. A function that is collision resistant is also second preimage resistant, but the reverse is not necessarily true. A function can be collision resistant but not preimage resistant and vice versa. A function can be preimage resistant but not second preimage resistant and vice versa. See [MENE97] for a discussion.

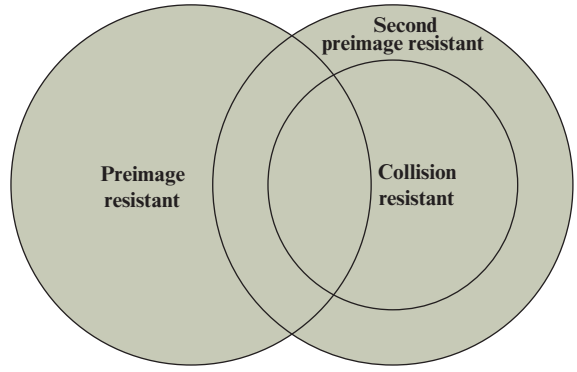


Figure 11.6 Relationship Among Hash Function Properties

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*

*Resistance required if attacker is able to mount a chosen message attack

Table 11.2 shows the resistant properties required for various hash function applications.

The final requirement in Table 11.1, **pseudorandomness**, has not traditionally been listed as a requirement of cryptographic hash functions but is more or less implied. [JOHN05] points out that cryptographic hash functions are commonly used for key derivation and pseudorandom number generation, and that in message integrity applications, the three resistant properties depend on the output of the hash function appearing to be random. Thus, it makes sense to verify that in fact a given hash function produces pseudorandom output.

Brute-Force Attacks

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. We look first at brute-force attacks.

Preimage and Second Preimage Attacks 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. This result is derived in Appendix E [Equation (E.1)].

Collision Resistant Attacks 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 turns out to require considerably less effort than a preimage or second preimage attack. The effort required is explained by a mathematical result referred to as the birthday paradox. In essence, if we choose random variables from a uniform distribution in the range 0 through $N - 1$, then the probability that a repeated element is encountered exceeds 0.5 after $\sqrt{N}$ choices have been made. Thus, for an m-bit hash value, if we pick data blocks at random, we can expect to find two data blocks with the same hash value within $\sqrt{2^m} = 2^{m/2}$ attempts. The mathematical derivation of this result is found in Appendix E.

Yuval proposed the following strategy to exploit the birthday paradox in a collision resistant attack [YUVA79].

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 11.4a).
2. The opponent generates $2^{m/2}$ variations x' 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.

Thus, if a 64-bit hash code is used, the level of effort required is only on the order of 2^{32} [see Appendix E, Equation (E.7)].

The generation of many variations that convey the same meaning is not difficult. For example, the opponent could insert a number of "space-space-backspace" character pairs between words throughout the document. Variations could then be generated by substituting "space-backspace-space" in selected instances. Alternatively,

the opponent could simply reword the message but retain the meaning. Figure 11.7 provides an example.

To summarize, for a hash code of length m, the level of effort required, as we have seen, is proportional to the following.

Preimage resistant	2^m
Second preimage resistant	2^m
Collision resistant	$2^{m/2}$

As { the / — } Dean of Blakewell College, I have { had the pleasure of knowing / known } Cherise Rosetti for the { last / past } four years. She { has been / was } { a tremendous / an outstanding } { asset to / role model in } { our / the } school. I { would like to take this opportunity to / wholeheartedly } recommend Cherise for your { school's / — } graduate program. I { am / feel } { confident / certain } { that / — } { she / Cherise } will { continue to / — } succeed in her studies. { She / Cherise } is a dedicated student and { thus far her grades / her grades thus far } { have been / are } { exemplary / excellent }. In class, { she / Cherise } { has proven to be / has been } a take-charge { person / individual } { who is / — } able to successfully develop plans and implement them.

{ She / Cherise } has also assisted { us / — } in our admissions office. { She / Cherise } has { successfully / — } demonstrated leadership ability by counseling new and prospective students. { Her / Cherise's } advice has been { a great / of considerable } help to these students, many of whom have { taken time to share / shared } their comments with me regarding her pleasant and { encouraging / reassuring } attitude. { For these reasons / It is for these reasons that } I { highly recommend / offer high recommendations for } Cherise { without reservation / unreservedly }. Her { ambition / drive } and { abilities / potential } will { truly / surely } be an { asset to / plus for } your { establishment / school }.

Figure 11.7 A Letter in 2^{38} Variations

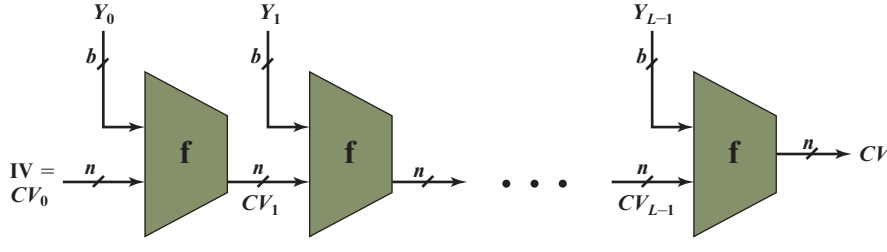
If collision resistance is required (and this is desirable for a general-purpose secure hash code), then the value $2^{m/2}$ determines the strength of the hash code against brute-force attacks. Van Oorschot and Wiener [VANO94] presented a design for a $10 million collision search machine for MD5, which has a 128-bit hash length, that could find a collision in 24 days. Thus, a 128-bit code may be viewed as inadequate. The next step up, if a hash code is treated as a sequence of 32 bits, is a 160-bit hash length. With a hash length of 160 bits, the same search machine would require over four thousand years to find a collision. With today's technology, the time would be much shorter, so that 160 bits now appears suspect.

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 way to measure the resistance of a hash algorithm to cryptanalysis is to compare its strength to the effort required for a brute-force attack. That is, an ideal hash algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.

In recent years, there has been considerable effort, and some successes, in developing cryptanalytic attacks on hash functions. To understand these, we need to look at the overall structure of a typical secure hash function, indicated in Figure 11.8. This structure, referred to as an iterated hash function, was proposed by Merkle [MERK79, MERK89] and is the structure of most hash functions in use today, including SHA, which is discussed later in this chapter. 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. Either the opponent must find two messages of equal length that hash to the same value or two messages of differing lengths that, together with their length values, hash to the same value.

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*,



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

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*. The hash function can be summarized as

$$\begin{aligned} CV_0 &= IV = \text{initial } n\text{-bit value} \\ CV_i &= \text{f}(CV_{i-1}, Y_{i-1}) \quad 1 \leq i \leq L \\ \text{H}(M) &= CV_L \end{aligned}$$

where the input to the hash function is a message M consisting of the blocks $Y_0, Y_1, \ldots, Y_{L-1}$.

The motivation for this iterative structure stems from the observation by Merkle [MERK89] and Damgard [DAMG89] that if the length field is included in the input, and if the compression function is collision resistant, then so is the resultant iterated hash function.[2] Therefore, the structure can be used to produce a secure hash function to operate on a message of any length. The problem of designing a secure hash function reduces to that of designing a collision-resistant compression function that operates on inputs of some fixed size.

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. Typically, as with symmetric block ciphers, f consists of a series of rounds of processing, so that the attack involves analysis of the pattern of bit changes from round to round.

Keep in mind that for any hash function there must exist collisions, because we are mapping a message of length at least equal to twice the block size b (because we must append a length field) into a hash code of length n, where $b \geq n$. What is required is that it is computationally infeasible to find collisions.

The attacks that have been mounted on hash functions are rather complex and beyond our scope here. For the interested reader, useful surveys of cryptanalysis of hash functions include [PREN10], [ROGA04b], and [LUCK04].

11.4 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.

[2]The converse is not necessarily true.

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.

SHA-1 produces a hash value of 160 bits. A simple brute-force technique for "breaking" SHA-1, that is, on being able to produce two different messages that produce the same hash function, would require on average 280 SHA-1 compressions. This appears prohibitive with current and foreseeable computational capacity. However, due to concern that cryptanalytic techniques might soon make SHA-1 vulnerable, NIST published a revised version of the standard in 2002, 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-1 and SHA-2 are also specified in RFC 6234, 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. Despite this, SHA-1 continued to be used for digital signature and other applications by numerous applications, such as web browsers. The reluctance to go through the expense and effort of transitioning to SHA-2 has been overcome by a breakthrough announced by a research team in 2017 [STEV17, CONS17]. The team demonstrated that SHA-1 collision attacks have finally become practical by providing the first known instance of a collision. In total, the computational effort spent is equivalent to 263.1 SHA-1 compressions and took approximately 6500 CPU years and 100 GPU years. As a result, Microsoft, Google, Apple, and Mozilla have all announced that their respective browsers have stopped accepting SHA-1 SSL certificates in 2017.

In this section, we provide a description of SHA-512. The other versions are quite similar. [SMIT15] provides a good description of SHA-256.

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 11.9 depicts the overall processing of a message to produce a digest. This follows the general structure depicted in Figure 11.8. The processing consists of the following steps.

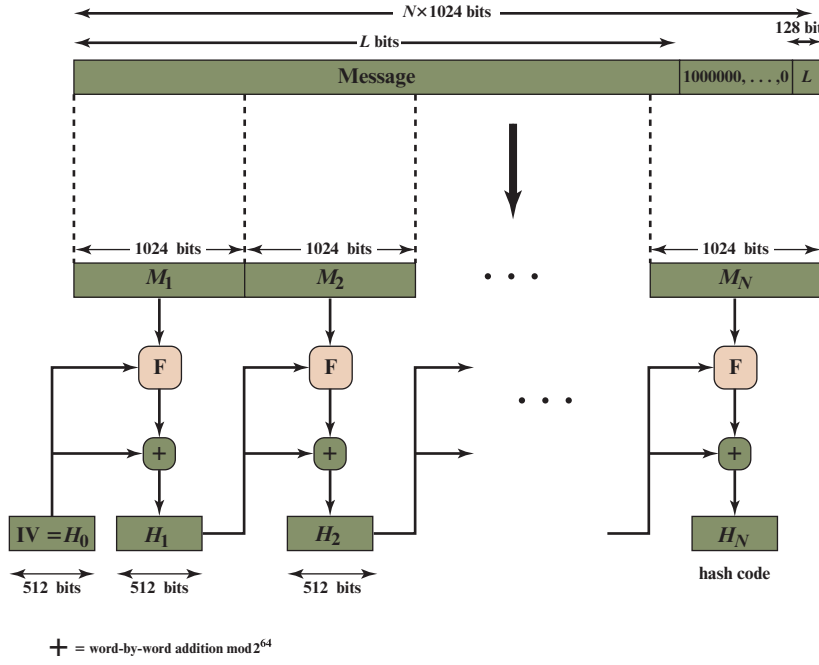


Figure 11.9 Message Digest Generation Using SHA-512

Step 1 **Append padding bits.** The message is padded so that its length is congruent to 896 modulo 1024 [length ≡ 896(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 in bits (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 11.9, 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):

```
a = 6A09E667F3BCC908   e = 510E527FADE682D1
b = BB67AE8584CAA73B   f = 9B05688C2B3E6C1F
c = 3C6EF372FE94F82B   g = 1F83D9ABFB41BD6B
d = A54FF53A5F1D36F1   h = 5BE0CD19137E2179
```

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. The values are stored in **big-endian format**, which is the most significant byte of a word in the low-address (leftmost) byte position. In contrast, in **little-endian format**, the least significant byte is stored in the lowest address.

Step 4 **Process message in 1024-bit (128-byte) blocks.** The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 11.9. The logic is illustrated in Figure 11.10.

Each round takes as input the 512-bit buffer value, abcdefgh, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, H_{i-1}. Each round t makes use of a 64-bit value W_t, derived from the current 1024-bit block being processed (M_i). These values are derived using a message schedule described subsequently. 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 constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data. Table 11.4 shows these constants in hexadecimal format (from left to right).

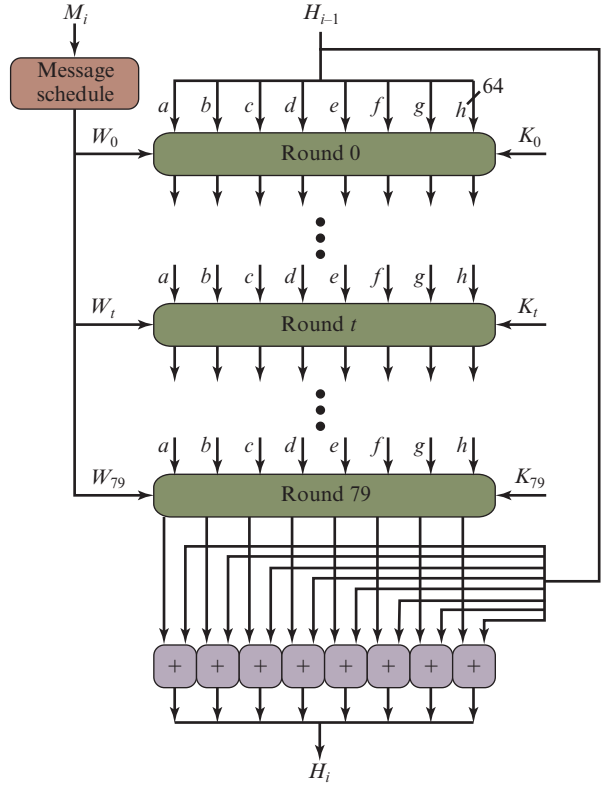


Figure 11.10 SHA-512 Processing of a Single 1024-Bit Block

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

The output of the eightieth round is added to the input to the first round (H_{i-1}) to produce H_i. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in H_{i-1}, using addition modulo 2^{64}.

Step 5 Output. After all N 1024-bit blocks have been processed, the output from the Nth stage is the 512-bit message digest.

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

$$\begin{aligned} H_0 &= \text{IV} \\ H_i &= \text{SUM}_{64}(H_{i-1}, \text{abcdefgh}_i) \\ MD &= H_N \end{aligned}$$

where

IV = initial value of the abcdefgh buffer, defined in step 3

abcdefgh_i = the output of the last round of processing of the ith 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 11.11). Each round is defined by the following set of equations:

$$T_1 = h + \text{Ch}(e, f, g) + (\sum\nolimits_1^{512} e) + W_t + K_t$$
$$T_2 = (\sum\nolimits_0^{512} a) + \text{Maj}(a, b, c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

where

t = step number; $0 \le t \le 79$

$\text{Ch}(e, f, g)$ = $(e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$
the conditional function: If e then f else g

$\text{Maj}(a, b, c)$ = $(a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$
the function is true only of the majority (two or three) of the arguments are true

$(\Sigma_0^{512} a)$ = $\text{ROTR}^{28}(a) \oplus \text{ROTR}^{34}(a) \oplus \text{ROTR}^{39}(a)$

$(\Sigma_1^{512} e)$ = $\text{ROTR}^{14}(e) \oplus \text{ROTR}^{18}(e) \oplus \text{ROTR}^{41}(e)$

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

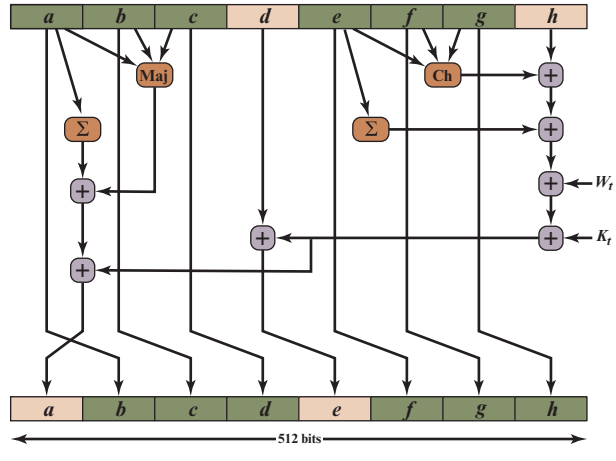


Figure 11.11 Elementary SHA-512 Operation (single round)

W_t = a 64-bit word derived from the current 1024-bit input block
K_t = a 64-bit additive constant
$+$ = addition modulo 2^{64}

Two observations can be made about the round function.

1. Six of the eight words of the output of the round function involve simply permutation (b, c, d, f, g, h) by means of rotation. This is indicated by shading in Figure 11.11.
2. Only two of the output words (a, e) are generated by substitution. Word e is a function of input variables (d, e, f, g, h), as well as the round word W_t and the constant K_t. Word a is a function of all of the input variables except d, as well as the round word W_t and the constant K_t.

It remains to indicate how the 64-bit word values W_t are derived from the 1024-bit message. Figure 11.12 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

$$\sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$$
$$\sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$$
$$\text{ROTR}^n(x) = \text{circular right shift (rotation) of the 64-bit argument } x \text{ by } n \text{ bits}$$
$$\text{SHR}^n(x) = \text{right shift of the 64-bit argument } x \text{ by } n \text{ bits with padding by zeros on the left}$$
$$+ = \text{addition modulo } 2^{64}$$

Thus, in the first 16 steps of processing, the value of W_t is equal to the corresponding word in the message block. For the remaining 64 steps, the value of W_t 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.

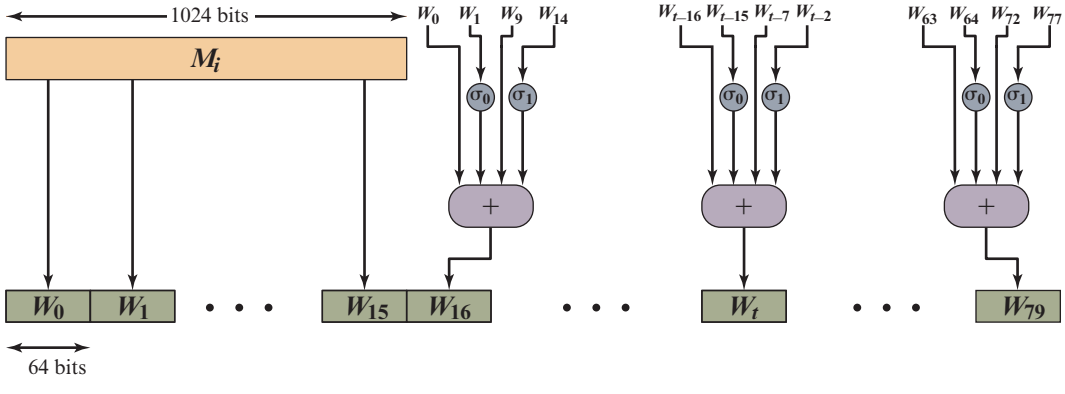


Figure 11.12 Creation of 80-word Input Sequence for SHA-512 Processing of Single Block

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 undisclosed weakness in SHA-512 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.

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

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}.

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

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

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

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

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

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.

11.5 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.

NISTIR 7896 (*Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*) summarizes the evaluation criteria used by NIST to select from among the candidates for SHA-3, 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.

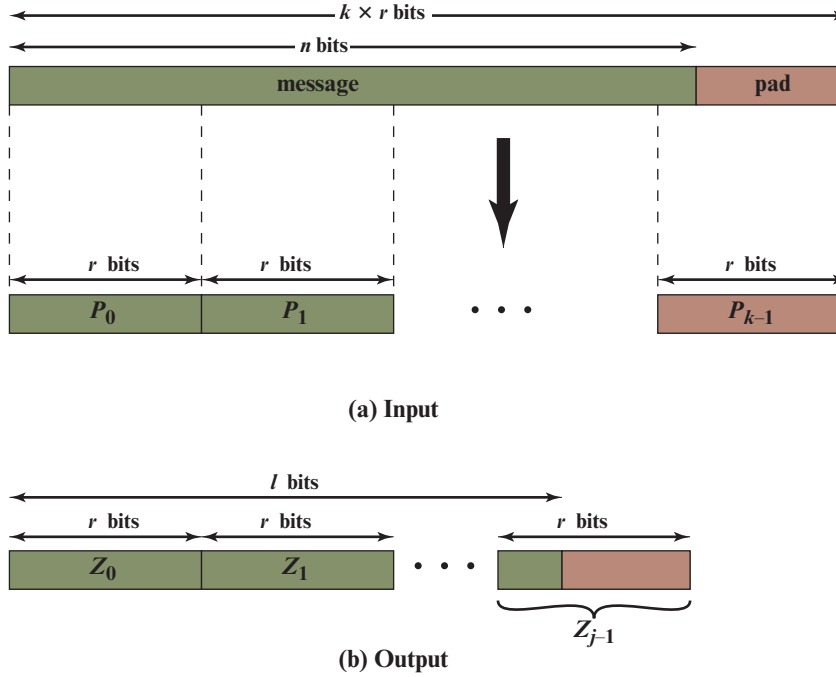


Figure 11.14 Sponge Function Input and Output

that if $n \bmod r = 0$, a padding block of r bits is added. The actual padding algorithm is a parameter of the function. The sponge specification proposes two padding schemes, The following definitions are based on [BERT11]

- **Simple padding (pad10*):** The minimum padding is added so that the block length divides the padded message length. The padding is all zeros except the first padding bit is a binary one.
- **Multirate padding (pad10*1):** The minimum padding is added so that the block length divides the padded message length. The padding is all zeros except the first and last padding bit are binary ones. Unlike simple padding, multirate padding is secure even if the rate r is changed for a given f. FIPS 202 uses multirate padding.

After processing all of the blocks, the sponge function generates a sequence of output blocks $Z_0, Z_1, \ldots, Z_{j-1}$. The number of output blocks generated is determined by the number of output bits desired. If the desired output is ℓ bits, then j blocks are produced, such that $(j - 1) \times r < \ell \leq j \times r$.

Figure 11.15 shows the iterated structure of the sponge function. The sponge construction operates on a state variable s of $b = r + c$ bits, which is initialized to all zeros and modified at each iteration. The value r is called the bitrate. This value is the block size used to partition the input message. The term *bitrate* reflects the fact that r is the number of bits processed at each iteration: the larger the value of r, the greater the rate at which message bits are processed by the sponge construction.

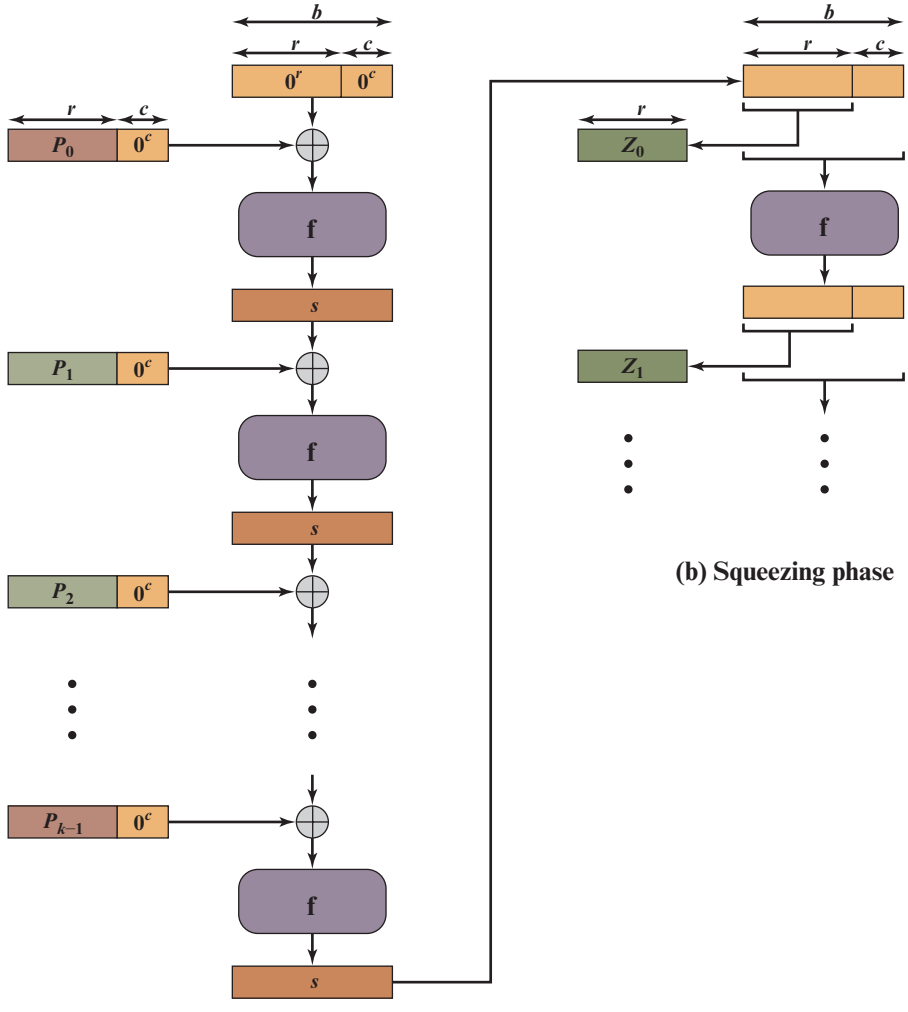


Figure 11.15 Sponge Construction

The value c is referred to as the capacity. A discussion of the security implications of the capacity is beyond our scope. In essence, the capacity is a measure of the achievable complexity of the sponge construction and therefore the achievable level of security. A given implementation can increase claimed security and reduce speed by increasing the capacity c and decreasing the bitrate r accordingly, or vice versa. The default values for Keccak are $c = 1024$ bits, $r = 576$ bits, and therefore $b = 1600$ bits.

The sponge construction consists of two phases. The absorbing phase proceeds as follows: For each iteration, the input block to be processed is padded with zeroes to extend its length from r bits to b bits. Then, the bitwise XOR of the extended

message block and s is formed to create a b-bit input to the iteration function f. The output of f is the value of s for the next iteration.

If the desired output length ℓ satisfies $\ell \leq b$, then at the completion of the absorbing phase, the first r bits of s are returned and the sponge construction terminates. Otherwise, the sponge construction enters the squeezing phase. To begin, the first r bits of s are retained as block Z_0. Then, the value of s is updated with repeated executions of f, and at each iteration, the first r bits of s are retained as block Z_i and concatenated with previously generated blocks. The process continues through $(j - 1)$ iterations until we have $(j - 1) \times r < \ell \leq j \times r$. At this point the first ℓ bits of the concatenated block Z are returned.

Note that the absorbing phase has the structure of a typical hash function. A common case will be one in which the desired hash length is less than or equal to the input block length; that is, $\ell \leq r$. In that case, the sponge construction terminates after the absorbing phase. If a longer output than b bits is required, then the squeezing phase is employed. Thus the sponge construction is quite flexible. For example, a short message with a length r could be used as a seed and the sponge construction would function as a pseudorandom number generator.

To summarize, the sponge construction is a simple iterated construction for building a function F with variable-length input and arbitrary output length based on a fixed-length transformation or permutation f operating on a fixed number b of bits. The sponge construction is defined formally in [BERT11] as follows:

Algorithm The sponge construction SPONGE[f, pad, r]
Require: $r < b$

  Interface: Z = sponge(M, ℓ) with $M \in Z_2^*$, integer $\ell > 0$ and $Z \in Z_2^\ell$
  $P = M \| \text{pad}[r](|M|)$
  $s = 0^b$
  for $i = 0$ to $|P|_r - 1$ **do**
    $s = s \oplus (Pi \| 0^{b-r})$
    $s = f(s)$
  end for
  $Z = \lfloor s \rfloor_r$
  while $|Z|_r \; r < \ell$ **do**
    $s = f(s)$
    $Z = Z \| \lfloor s \rfloor_r$
  end while
  return $\lfloor Z \rfloor_\ell$

In the algorithm definition, the following notation is used: $|M|$ is the length in bits of a bit string M. A bit string M can be considered as a sequence of blocks of some fixed length x, where the last block may be shorter. The number of blocks of M is denoted by $|M|_x$. The blocks of M are denoted by M_i and the index ranges from 0 to $|M|_x - 1$. The expression $\lfloor M \rfloor_\ell$ denotes the truncation of M to its first ℓ bits.

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}

Note: All sizes and security levels—are measured in bits.

SHA-3 makes use of the iteration function f, labeled Keccak-f, which is described in the next section. The overall SHA-3 function is a sponge function expressed as Keccak[r, c] to reflect that SHA-3 has two operational parameters, r, the message block size, and c, the capacity, with the default of $r + c = 1600$ bits. Table 11.5 shows the supported values of r and c. As Table 11.5 shows, the hash function security associated with the sponge construction is a function of the capacity c.

In terms of the sponge algorithm defined above, Keccak[r, c] is defined as

$$\text{Keccak}\,[r, c] \underline{\underline{\Delta}}\ \text{SPONGE}\,[\text{Keccak-}f\,[r + c], \text{pad } 10^*1, r]$$

We now turn to a discussion of the iteration function Keccak-f.

The SHA-3 Iteration Function *f*

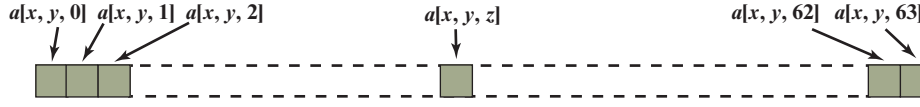
We now examine the iteration function Keccak-f used to process each successive block of the input message. Recall that f takes as input a 1600-bit variable s consisting of r bits, corresponding to the message block size followed by c bits, referred to as the capacity. For internal processing within f, the input state variable s is organized as a $5 \times 5 \times 64$ array a. The 64-bit units are referred to as lanes. For our purposes, we generally use the notation $a[x, y, z]$ to refer to an individual bit with in the state array. When we are more concerned with operations that affect entire lanes, we designate the 5×5 matrix as $L[x, y]$, where each entry in L is a 64-bit lane. The use of indices within this matrix is shown in Figure 11.16.[4] Thus, the columns are labeled $x = 0$ through $x = 4$, the rows are labeled $y = 0$ through $y = 4$, and the individual bits within a lane are labeled $z = 0$ through $z = 63$. The mapping between the bits of s and those of a is

$$s[64(5y + x) + z] = a[x, y, z]$$

[4]Note that the first index (x) designates a column and the second index (y) designates a row. This is in conflict with the convention used in most mathematics sources, where the first index designates a row and the second index designates a column (e.g., Knuth, D. *The Art of Computing Programming, Volume 1, Fundamental Algorithms*; and Korn, G., and Korn, T. *Mathematical Handbook for Scientists and Engineers*).

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

(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

We can visualize this with respect to the matrix in Figure 11.16. When treating the state as a matrix of lanes, the first lane in the lower left corner, $L[0, 0]$, corresponds to the first 64 bits of s. The lane in the second column, lowest row, $L[1, 0]$, corresponds to the next 64 bits of s. Thus, the array a is filled with the bits of s starting with row $y = 0$ and proceeding row by row.

Structure of f The function f is executed once for each input block of the message to be hashed. The function takes as input the 1600-bit state variable and converts it into a 5×5 matrix of 64-bit lanes. This matrix then passes through 24 rounds of processing. Each round consists of five steps, and each step updates the state matrix by permutation or substitution operations. As shown in Figure 11.17, the rounds are identical with the exception of the final step in each round, which is modified by a round constant that differs for each round.

The application of the five steps can be expressed as the composition[5] of functions:

$$\mathrm{R} = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

Table 11.6 summarizes the operation of the five steps. The steps have a simple description leading to a specification that is compact and in which no trapdoor can be hidden. The operations on lanes in the specification are limited to bitwise Boolean operations (XOR, AND, NOT) and rotations. There is no need for table lookups, arithmetic operations, or data-dependent rotations. Thus, SHA-3 is easily and efficiently implemented in either hardware or software.

We examine each of the step functions in turn.

[5]If f and g are two functions, then the function F with the equation $y = \mathrm{F}(x) = \mathrm{g}[\mathrm{f}(x)]$ is called the **composition** of f and g and is denoted as $\mathrm{F} = \mathrm{g} \circ \mathrm{f}$.

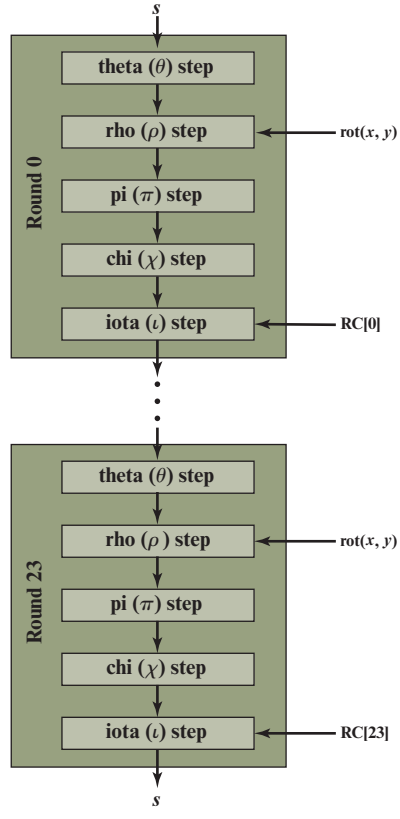


Figure 11.17 SHA-3 Iteration Function f

Table 11.6 Step Functions in SHA-3

Function	Type	Description
θ	Substitution	New value of each bit in each word depends on 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.

THETA STEP FUNCTION The Keccak reference defines the θ function as follows. For bit z in column x, row y,

$$\theta: a[x, y, z] \leftarrow a[x, y, z] \oplus \sum_{y'=0}^{4} a[(x - 1), y', z] \oplus \sum_{y'=0}^{4} a[(x + 1), y', (z - 1)] \quad \textbf{(11.1)}$$

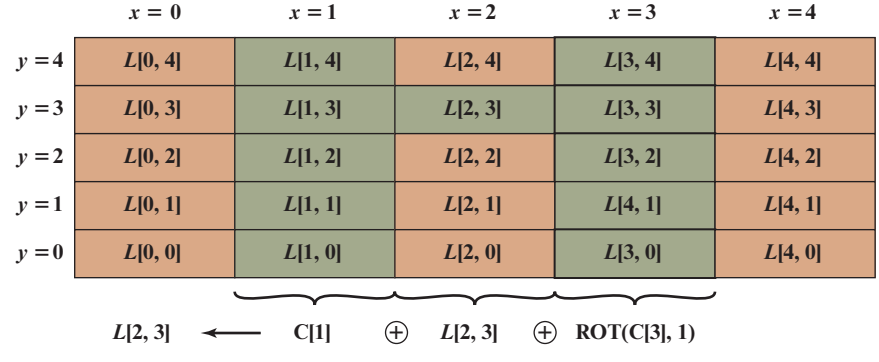
where the summations are XOR operations. We can see more clearly what this operation accomplishes with reference to Figure 11.18a. First, define the bitwise XOR of the lanes in column x as

$$C[x] = L[x, 0] \oplus L[x, 1] \oplus L[x, 2] \oplus L[x, 3] \oplus L[x, 4]$$

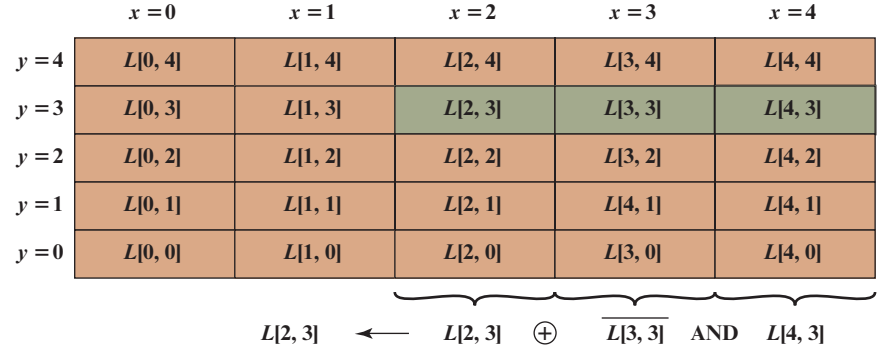
Consider lane $L[x, y]$ in column x, row y. The first summation in Equation 11.1 performs a bitwise XOR of the lanes in column $(x - 1)$ mod 4 to form the 64-bit lane $C[x - 1]$. The second summation performs a bitwise XOR of the lanes in column $(x + 1)$ mod 4, and then rotates the bits within the 64-bit lane so that the bit in position z is mapped into position $z + 1$ mod 64. This forms the lane ROT $(C[x + 1], 1)$. These two lanes and $L[x, y]$ are combined by bitwise XOR to form the updated value of $L[x, y]$. This can be expressed as

$$L[x, y] \leftarrow L[x, y] \oplus C[x - 1] \oplus \text{ROT}(C[x + 1], 1)$$

Figure 11.18.a illustrates the operation on $L[3, 2]$. The same operation is performed on all of the other lanes in the matrix.



(a) θ step function



(b) χ step function

Figure 11.18 Theta and Chi Step Functions

Several observations are in order. Each bit in a lane is updated using the bit itself and one bit in the same bit position from each lane in the preceding column and one bit in the adjacent bit position from each lane in the succeeding column. Thus the updated value of each bit depends on 11 bits. This provides good mixing. Also, the theta step provides good diffusion, as that term was defined in Chapter 4. The designers of Keccak state that the theta step provides a high level of diffusion on average and that without theta, the round function would not provide diffusion of any significance.

RHO STEP FUNCTION The ρ function is defined as follows:

$$\rho\colon a[x, y, z] \leftarrow a[x, y, z] \quad \text{if } x = y = 0$$

otherwise,

$$\rho\colon a[x, y, z] \leftarrow a\left[x, y, \left(z - \frac{(t+1)(t+2)}{2}\right)\right] \tag{11.2}$$

with t satisfying $0 \leq t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\text{GF}(5)^{2\times 2}$

It is not immediately obvious what this step performs, so let us look at the process in detail.

1. The lane in position $(x, y) = (0, 0)$, that is $L[0, 0]$, is unaffected. For all other words, a circular bit shift within the lane is performed.
2. The variable t, with $0 \leq t < 24$, is used to determine both the amount of the circular bit shift and which lane is assigned which shift value.
3. The 24 individual bit shifts that are performed have the respective values $\frac{(t+1)(t+2)}{2} \bmod 64$.
4. The shift determined by the value of t is performed on the lane in position (x, y) in the 5×5 matrix of lanes. Specifically, for each value of t, the corresponding matrix position is defined by $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. For example, for $t = 3$, we have

$$\begin{aligned}
\begin{pmatrix} x \\ y \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^3 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 6 \end{pmatrix} \bmod 5 = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \bmod 5 \\
&= \begin{pmatrix} 1 \\ 7 \end{pmatrix} \bmod 5 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}
\end{aligned}$$

Table 11.7 Rotation Values Used in SHA-3

(a) Calculation of values and positions

t	g(t)	g (t) mod 64	x, y
0	1	1	1, 0
1	3	3	0, 2
2	6	6	2, 1
3	10	10	1, 2
4	15	15	2, 3
5	21	21	3, 3
6	28	28	3, 0
7	36	36	0, 1
8	45	45	1, 3
9	55	55	3, 1
10	66	2	1, 4
11	78	14	4, 4

t	g(t)	g (t) mod 64	x, y
12	91	27	4, 0
13	105	41	0, 3
14	120	56	3, 4
15	136	8	4, 3
16	153	25	3, 2
17	171	43	2, 2
18	190	62	2, 0
19	210	18	0, 4
20	231	39	4, 2
21	253	61	2, 4
22	276	20	4, 1
23	300	44	1, 1

Note: $g(t) = (t + 1)(t + 2)/2$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} \bmod 5$$

(b) Rotation values by word position in matrix

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	18	2	61	56	14
$y = 3$	41	45	15	21	8
$y = 2$	3	10	43	25	39
$y = 1$	36	44	6	55	20
$y = 0$	0	1	62	28	27

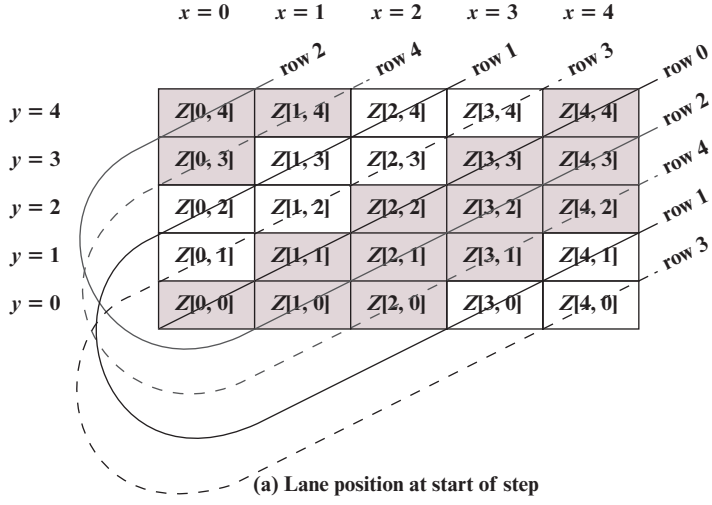
Table 11.7 shows the calculations that are performed to determine the amount of the bit shift and the location of each bit shift value. Note that all of the rotation amounts are different.

The ρ function thus consists of a simple permutation (circular shift) within each lane. The intent is to provide diffusion within each lane. Without this function, diffusion between lanes would be very slow.

Pi Step Function The π function is defined as follows:

$$\pi: \quad a[x, y] \leftarrow a[x', y'], \quad with \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{11.3}$$

This can be rewritten as $(x, y) \times (y, (2x + 3y))$. Thus, the lanes within the 5×5 matrix are moved so that the new x position equals the old y position and



(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

the new y position is determined by $(2x + 3y)$ mod 5. Figure 11.19 helps in visualizing this permutation. Lanes that are along the same diagonal (increasing in y value, going from left to right) prior to π are arranged on the same row in the matrix after π is executed. Note that the position of $L[0, 0]$ is unchanged.

Thus the π step is a permutation of lanes: The lanes move position within the 5×5 matrix. The ρ step is a permutation of bits: Bits within a lane are rotated. Note that the π step matrix positions are calculated in the same way that, for the ρ step, the one-dimensional sequence of rotation constants is mapped to the lanes of the matrix.

CHI STEP FUNCTION The χ function is defined as follows:

$$\chi: a[x] \leftarrow a[x] \oplus ((a[x + 1] \oplus 1) \text{ AND } a[x + 2]) \qquad \textbf{(11.4)}$$

This function operates to update each bit based on its current value and the value of the corresponding bit position in the next two lanes in the same row. The

operation is more clearly seen if we consider a single bit $a[x, y, z]$ and write out the Boolean expression:

$$a[x, y, z] \leftarrow a[x, y, z] \oplus (\text{NOT}(a[x + 1, y, z])) \text{ AND } (a[x + 2, y, z])$$

Figure 11.18b illustrates the operation of the χ function on the bits of the lane $L[3, 2]$. This is the only one of the step functions that is a nonlinear mapping. Without it, the SHA-3 round function would be linear.

Iota Step Function The ι function is defined as follows:

$$\iota: a \leftarrow a \oplus RC[i_r] \tag{11.5}$$

This function combines an array element with a round constant that differs for each round. It breaks up any symmetry induced by the other four step functions. In fact, Equation 11.5 is somewhat misleading. The round constant is applied only to the first lane of the internal state array. We express this is as follows:

$$L[0, 0] \leftarrow L[0, 0] \oplus \text{RC}[i_r] \quad 0 \leq i_r \leq 23$$

Table 11.8 lists the 24 64-bit round constants. Note that the Hamming weight, or number of 1 bits, in the round constants ranges from 1 to 6. Most of the bit positions are zero and thus do not change the corresponding bits in $L[0, 0]$. If we take the cumulative OR of all 24 round constants, we get

$$\text{RC}[0] \text{ OR } \text{RC}[1] \text{ OR } \ldots \text{ OR } \text{RC}[23] = \texttt{8000000080008008B}$$

Thus, only 7 bit positions are active and can affect the value of $L[0, 0]$. Of course, from round to round, the permutations and substitutions propagate the effects of the ι function to all of the lanes and all of the bit positions in the matrix. It is easily seen that the disruption diffuses through θ and χ to all lanes of the state after a single round.

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
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

11.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

big endian	hash code	message authentication code (MAC)
compression function	hash function	message digest
cryptographic hash function	hash value	
	little endian	

Review Questions

11.1 What characteristics are needed in a secure hash function?

11.2 Alice sends Bob a message with an attached hash value. If the message digest is sent in a secure fashion, then how would Bob know if there was a possible man-in-the-middle attack?

11.3 What is the role of a compression function in a hash function?

11.4 How safe is it to use a hash function without encryption in terms of integrity?

11.5 What basic arithmetical and logical functions are used in SHA?

11.6 Define a collision and explain how to deal with collision attacks.

11.7 Define the term *sponge construction*.

11.8 Summarise the five main steps of SHA-512 and its main functions.

11.9 List and briefly describe the step functions that comprise the iteration function *f*.

Problems

11.1 The high-speed transport protocol XTP (Xpress Transfer Protocol) uses a 32-bit checksum function defined as the concatenation of two 16-bit functions: XOR and RXOR, defined in Section 11.4 as "two simple hash functions" and illustrated in Figure 11.5.

a. Will this checksum detect all errors caused by an odd number of error bits? Explain.

b. Will this checksum detect all errors caused by an even number of error bits? If not, characterize the error patterns that will cause the checksum to fail.

c. Comment on the effectiveness of this function for use as a hash function for authentication.

11.2 **a.** 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 example, proposed in [DAVI89], is as follows. 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 H as

$$\begin{aligned} H_0 &= \text{initial value} \\ H_i &= H_{i-1} \oplus \mathrm{E}(M_i, \mathrm{H}_{i-1}) \\ H &= H_N \end{aligned}$$

Assume that DES is used as the encryption algorithm. Recall the complementarity property of DES (Problem 4.14): If $Y = \mathrm{E}(K, X)$, then $Y' = \mathrm{E}(K', X')$. Use this property to show how a message consisting of blocks $M_1, M_2, \ldots, M_N$ can be altered without altering its hash code.

b. A variation of the scheme above is proposed in [MEYE88], with the following formula:

$$H_i = M_i \oplus \mathrm{E}(H_{i-1}, M_i)$$

Show that a similar attack that of Problem 11.2a will succeed against this scheme.

11.3 **a.** Consider the following hash function. Messages are in the form of a sequence of numbers in Z_n, $M = (a_1, a_2, \ldots a_t)$. The hash value h is calculated as $\left(\sum_{i=1}^{t} a_i\right)$ for some predefined value n. Does this hash function satisfy any of the requirements for a hash function listed in Table 11.1? Explain your answer.

b. Repeat part (a) for the hash function $h = \left(\sum_{i=1}^{t} (a_i)^2\right) \bmod n$.

c. Calculate the hash function of part (b) for $M = (189, 632, 900, 722, 349)$ and $n = 989$.

11.4 For a message digest where hash functions are used to provide message authentication and integrity, what is the most appropriate way to protect the hash values?

11.5 Encryption assists in providing confidentiality to the data being sent from party A to party B. However, in recent years, there is an interest in avoiding encryption depending on the application. Why?

11.6 Suppose H(m) is a collision-resistant hash function that maps a message of arbitrary bit length into an n-bit hash value. Is it true that, for all messages x, x' with $x \neq x'$, we have $H(x) \neq H(x')$ Explain your answer.

11.7 Given the rotation values used in SHA-3 in Table 11.7, if $x = 3$ and $y = 2$, how many bit shifts are necessary for a rotation and how random can this operation be?

11.8 For SHA-512, show the equations for the values of W_{16}, W_{18}, W_{23}, and W_{31}.

11.9 State the value of the padding field in SHA-512 if the length of the message is

a. 2942 bits.

b. 2943 bits.

c. 2944 bits.

11.10 State the value of the length field in SHA-512 if the length of the message is

a. 2942 bits.

b. 2943 bits.

c. 2944 bits.

11.11 Suppose $a_1a_2a_3a_4$ are the 4 bytes in a 32-bit word. Each a_i can be viewed as an integer in the range 0 to 255, represented in binary. In a big-endian architecture, this word represents the integer

$$a_1 2^{24} + a_2 2^{16} + a_3 2^8 + a_4$$

In a little-endian architecture, this word represents the integer

$$a_4 2^{24} + a_3 2^{16} + a_2 2^8 + a_1$$

a. Some hash functions, such as MD5, assume a little-endian architecture. It is important that the message digest be independent of the underlying architecture. Therefore, to perform the modulo 2 addition operation of MD5 or RIPEMD-160 on a big-endian architecture, an adjustment must be made. Suppose $X = x_1\, x_2\, x_3\, x_4$ and $Y = y_1\, y_2\, y_3\, y_4$. Show how the MD5 addition operation (X + Y) would be carried out on a big-endian machine.

b. SHA assumes a big-endian architecture. Show how the operation (X + Y) for SHA would be carried out on a little-endian machine.

11.12 This problem introduces a hash function similar in spirit to SHA that operates on letters instead of binary data. It is called the *toy tetragraph hash* (tth).[6] Given a message consisting of a sequence of letters, tth produces a hash value consisting of four letters. First, tth divides the message into blocks of 16 letters, ignoring spaces, punctuation, and capitalization. If the message length is not divisible by 16, it is padded out with nulls. A four-number running total is maintained that starts out with the value (0, 0, 0, 0); this is input to the compression function for processing the first block. The compression function consists of two rounds.

Round 1 Get the next block of text and arrange it as a row-wise 4×4 block of text, and convert it to numbers ($A = 0, B = 1$, etc.). For example, for the block ABCDEFGHIJKLMNOP, we have

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Then, add each column mod 26 and add the result to the running total, mod 26. In this example, the running total is (24, 2, 6, 10).

Round 2 Using the matrix from round 1, rotate the first row left by 1, second row left by 2, third row left by 3, and reverse the order of the fourth row. In our example:

B	C	D	A
G	H	E	F
L	I	J	K
P	O	N	M

1	2	3	0
6	7	4	5
11	8	9	10
15	14	13	12

Now, add each column mod 26 and add the result to the running total. The new running total is (5, 7, 9, 11). This running total is now the input into the first round of the compression function for the next block of text. After the final block is processed, convert the final running total to letters. For example, if the message is ABCDEFGHIJKLMNOP, then the hash is FHJL.

a. Draw figures comparable to Figures 11.9 and 11.10 to depict the overall tth logic and the compression function logic.

b. Calculate the hash function for the 22-letter message "Practice makes us perfect."

c. To demonstrate the weakness of tth, find a message of length 32-letter that produces the same hash.

11.13 For each of the possible capacity values of SHA-3 (Table 11.5), which lanes in the internal 55 state matrix start out as lanes of all zeros?

11.14 During the permutation phase in SHA-3, if a new position is determined by $(2x + 3y)$ mod 5, how big is the matrix in the permutation of lanes? Illustrate your answer with the aid of a diagram.

---

[6] I thank William K. Mason, of the magazine staff of *The Cryptogram*, for providing this example.

11.15 Consider the state matrix as illustrated in Figure 11.16a. Now rearrange the rows and columns of the matrix so that $L[0, 0]$ is in the center. Specifically, arrange the columns in the left-to-right order ($x = 3, x = 4, x = 0, x = 1, x = 2$) and arrange the rows in the top-to-bottom order ($y = 2, y = 1, y = 0, y = 4, y = 6$). This should give you some insight into the permutation algorithm used for the function and for permuting the rotation constants in the function. Using this rearranged matrix, describe the permutation algorithm.

11.16 The function only affects $L[0, 0]$. Section 11.6 states that the changes to $L[0, 0]$ diffuse through θ and to all lanes of the state after a single round.

a. Show that this is so.

b. How long before all of the bit positions in the matrix are affected by the changes to $L[0, 0]$?