CHAPTER 11

Message Integrity and Message Authentication

Objectives

This chapter has several objectives:

- ❏ To define message integrity
- ❏ To define message authentication
- ❏ To define criteria for a cryptographic hash function
- ❏ To define the Random Oracle Model and its role in evaluating the security of cryptographic hash functions
- ❏ To distinguish between an MDC and a MAC
- ❏ To discuss some common MACs

This is the first of three chapters devoted to message integrity, message authentication, and entity authentication. This chapter discusses general ideas related to cryptographic hash functions that are used to create a message digest from a message. Message digests guarantee the integrity of the message. We then discuss how simple message digests can be modified to authenticate the message. The standard cryptography cryptographic hash functions are developed in Chapter 12.

11.1 MESSAGE INTEGRITY

The cryptography systems that we have studied so far provide *secrecy,* or *confidentiality,* but not *integrity.* However, there are occasions where we may not even need secrecy but instead must have integrity. For example, Alice may write a will to distribute her estate upon her death. The will does not need to be encrypted. After her death, anyone can examine the will. The integrity of the will, however, needs to be preserved. Alice does not want the contents of the will to be changed.

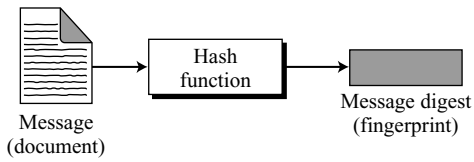
Document and Fingerprint

One way to preserve the integrity of a document is through the use of a *fingerprint*. If Alice needs to be sure that the contents of her document will not be changed, she can put her fingerprint at the bottom of the document. Eve cannot modify the contents of this document or create a false document because she cannot forge Alice's fingerprint. To ensure that the document has not been changed, Alice's fingerprint on the document can be compared to Alice's fingerprint on file. If they are not the same, the document is not from Alice.

Message and Message Digest

The electronic equivalent of the document and fingerprint pair is the *message* and *digest* pair. To preserve the integrity of a message, the message is passed through an algorithm called a **cryptographic hash function.** The function creates a compressed image of the message that can be used like a fingerprint. Figure 11.1 shows the message, cryptographic hash function, and **message digest.**

Figure 11.1 *Message and digest*



Difference

The two pairs (document/fingerprint) and (message/message digest) are similar, with some differences. The document and fingerprint are physically linked together. The message and message digest can be unlinked (or sent) separately, and, most importantly, the message digest needs to be safe from change.

The message digest needs to be safe from change.

Checking Integrity

To check the integrity of a message, or document, we run the cryptographic hash function again and compare the new message digest with the previous one. If both are the same, we are sure that the original message has not been changed. Figure 11.2 shows the idea.

Cryptographic Hash Function Criteria

A cryptographic hash function must satisfy three criteria: **preimage resistance, second preimage resistance,** and **collision resistance,** as shown in Figure 11.3.

Figure 11.2 *Checking integrity*

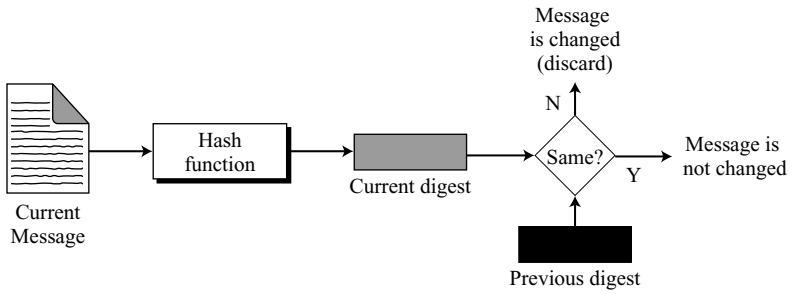
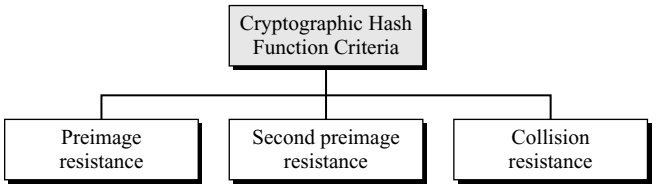


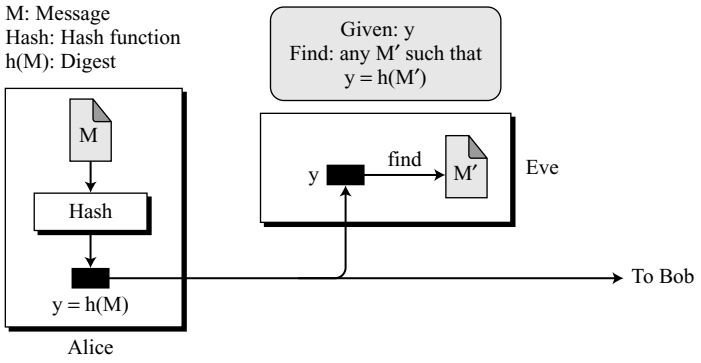
Figure 11.3 *Criteria of a cryptographic hash function*



Preimage Resistance

A cryptographic hash function must be preimage resistant. Given a hash function h and y = h(M), it must be extremely difficult for Eve to find any message, M′, such that y = h(M′). Figure 11.4 shows the idea.

Figure 11.4 *Preimage*



If the hash function is not preimage resistant, Eve can intercept the digest h(M) and create a message M′. Eve can then send M′ to Bob pretending it is M.

Preimage Attack	
Given: y = h(M)	**Find: M′ such that y = h(M′)**

Example 11.1

Can we use a conventional lossless compression method such as StuffIt as a cryptographic hash function?

Solution

We cannot. A lossless compression method creates a compressed message that is reversible. You can uncompress the compressed message to get the original one.

Example 11.2

Can we use a checksum function as a cryptographic hash function?

Solution

We cannot. A checksum function is not preimage resistant, Eve may find several messages whose checksum matches the given one.

Second Preimage Resistance

The second criterion, **second preimage resistance,** ensures that a message cannot easily be forged. If Alice creates a message and a digest and sends both to Bob, this criterion ensures that Eve cannot easily create another message that hashes to the exact same digest. In other words, given a specific message and its digest, it is impossible (or at least very difficult) to create another message with the same digest. Figure 11.5 shows the idea.

Eve intercepts (has access to) a message M and its digest h(M). She creates another message M′≠ M, but h(M) = h(M′). Eve sends the M′ and h(M′) to Bob. Eve has forged the message.

Second Preimage Attack	
Given: M and h(M)	**Find: M′ ≠ M such that h(M) = h(M′)**

Collision Resistance

The third criterion, **collision resistance,** ensures that Eve cannot find two messages that hash to the same digest. Here the adversary can create two messages (out of scratch) and hashed to the same digest. We will see later how Eve can benefit from this weakness in the hash function. For the moment, suppose two different wills can be created that hash to the same digest. When the time comes for the execution of the will, the second (forged) will is presented to the heirs. Because the digest matches both wills, the substitution is undetected. Figure 11.6 shows the idea. We will see later that this type of attack is much easier to launch than the two previous kinds. In other words, we need particularly be sure that a hash function is collision resistant.

Figure 11.5 *Second preimage*

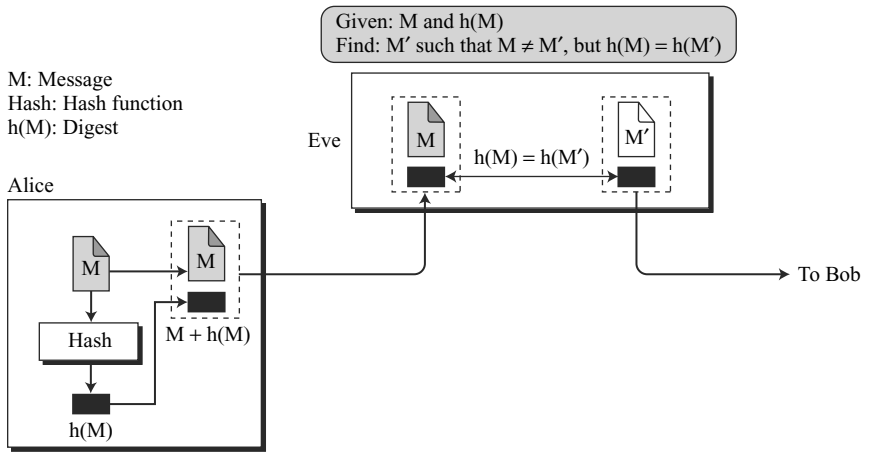
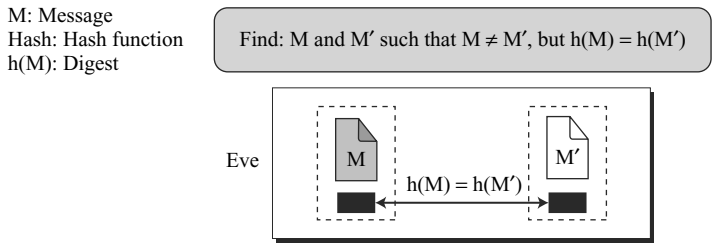


Figure 11.6 *Collision*



Collision Attack	
Given: none	**Find: M′ ≠ M such that h(M) = h(M′)**

11.2 RANDOM ORACLE MODEL

The **Random Oracle Model,** which was introduced in 1993 by Bellare and Rogaway, is an ideal mathematical model for a hash function. A function based on this model behaves as follows:

1. When a new message of any length is given, the oracle creates and gives a fixed-length message digest that is a random string of 0s and 1s. The oracle records the message and the message digest.
2. When a message is given for which a digest exists, the oracle simply gives the digest in the record.

3. The digest for a new message needs to be chosen independently from all previous digests. This implies that the oracle cannot use a formula or an algorithm to calculate the digest.

Example 11.3

Assume an oracle with a table and a fair coin. The table has two columns. The left column shows the messages whose digests have been issued by the oracle. The second column lists the digests created for those messages. We assume that the digest is always 16 bits regardless of the size of the message. Table 11.1 shows an example of this table in which the message and the message digest are listed in hexadecimal. The oracle has already created three digests.

Table 11.1 *Oracle table after issuing the first three digests*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB45CD1048765412AAAB6662BE	A38B

Now assume that two events occur:

a. The message AB1234CD8765BDAD is given for digest calculation. The oracle checks its table. This message is not in the table, so the oracle flips its coin 16 times. Assume that result is HHTHHHTTHTHHTTTH, in which the letter H represents *heads* and the letter T represents *tails*. The oracle interprets H as a 1-bit and T as a 0-bit and gives 1101110010110001 in binary, or DCB1 in hexadecimal, as the message digest for this message and adds the note of the message and the digest in the table (Table 11.2).

Table 11.2 *Oracle table after issuing the fourth digest*

Message	*Message Digest*
4523AB1352CDEF45126	13AB
723BAE38F2AB3457AC	02CA
AB1234CD8765BDAD	DCB1
AB45CD1048765412AAAB6662BE	A38B

b. The message 4523AB1352CDEF45126 is given for digest calculation. The oracle checks its table and finds that there is a digest for this message in the table (first row). The oracle simply gives the corresponding digest (13AB).

Example 11.4

The oracle in Example 11.3 cannot use a formula or algorithm to create the digest for a message. For example, imagine the oracle uses the formula h(M) = M mod *n*. Now suppose that the oracle has already given $h(M_1)$ and $h(M_2)$. If a new message is presented as $M_3 = M_1 + M_2$, the oracle does not have to calculate the $h(M_3)$. The new digest is just $[h(M_1) + h(M_2)] \bmod n$ since

$$h(M_3) = (M_1 + M_2) \bmod n = M_1 \bmod n + M_2 \bmod n = [h(M_1) + h(M_2)] \bmod n$$

This violates the third requirement that each digest must be randomly chosen based on the message given to the oracle.

Pigeonhole Principle

The first thing we need to be familiar with to understand the analysis of the Random Oracle Model is the **pigeonhole principle:** if n pigeonholes are occupied by n +1 pigeons, then at least one pigeonhole is occupied by two pigeons.The generalized version of the pigeonhole principle is that if n pigeonholes are occupied by kn +1 pigeons, then at least one pigeonhole is occupied by $k + 1$ pigeons.

Because the whole idea of hashing dictates that the digest should be shorter than the message, according to the pigeonhole principle there can be collisions. In other words, there are some digests that correspond to more than one message; the relationship between the possible messages and possible digests is many-to-one.

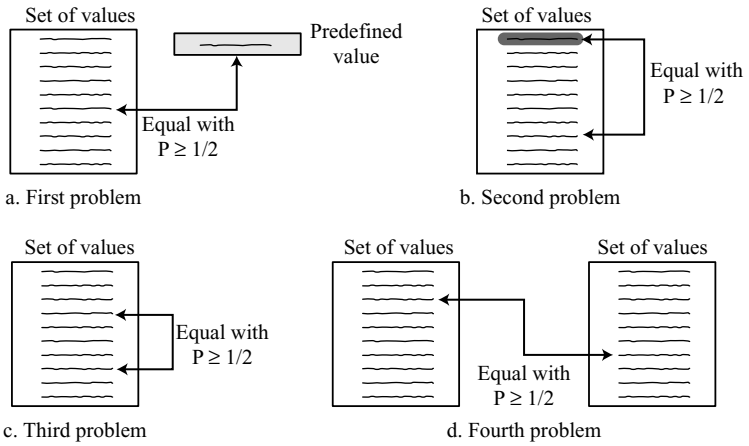
Example 11.5

Assume that the messages in a hash function are 6 bits long and the digests are only 4 bits long. Then the possible number of digests (pigeonholes) is $2^4 = 16$, and the possible number of messages (pigeons) is $2^6 = 64$. This means $n = 16$ and $kn + 1 = 64$, so k is larger than 3. The conclusion is that at least one digest corresponds to four $(k + 1)$ messages.

Birthday Problems

The second thing we need to know before analyzing the Random Oracle Model is the famous **birthday problems.** Four different birthday problems are usually encountered in the probability courses. The third problem, sometimes referred to as *birthday paradox,* is the most common one in the literature. Figure 11.7 shows the idea of each problem.

Figure 11.7 *Four birthday problems*



Description of Problems

Below the birthday problems are described in terms that can be applied to the security of hash functions. Note that the term *likely* in all cases means with the probability $P \geq 1/2$.

- **Problem 1:** What is the minimum number, k, of students in a classroom such that it is *likely* that at least one student has a predefined birthday? This problem can be generalized as follows. We have a uniformly distributed random variable with N possible values (between 0 and $N - 1$). What is the minimum number of instances, k, such that it is *likely* that at least one instance is equal to a predefined value?
- **Problem 2:** What is the minimum number, k, of students in a classroom such that it is *likely* that at least one student has the same birthday as the student selected by the professor? This problem can be generalized as follows. We have a uniformly distributed random variable with N possible values (between 0 and $N - 1$). What is the minimum number of instances, k, such that it is *likely* that at least one instance is equal to the selected one?
- **Problem 3:** What is the minimum number, k, of students in a classroom such that it is *likely* that at least two students have the same birthday? This problem can be generalized as follows. We have a uniformly distributed random variable with N possible values (between 0 and $N - 1$). What is the minimum number of instances, k, such that it is *likely* that at least two instances are equal?
- **Problem 4:** We have two classes, each with k students. What is the minimum value of k so that it is *likely* that at least one student from the first classroom has the same birthday as a student from the second classroom? This problem can be generalized as follows. We have a uniformly distributed random variable with N possible values (between 0 and $N - 1$). We generate two sets of random values each with k instances. What is the minimum number of, k, such that it is *likely* that at least one instance from the first set is equal to one instance in the second set?

Summary of Solutions

Solutions to these problems are given in Appendix E for interested readers; The results are summarized in Table 11.3.

Table 11.3 *Summarized solutions to four birthday problems*

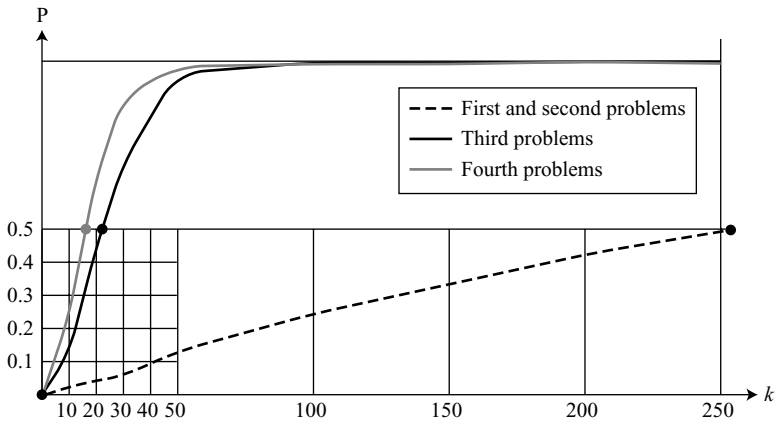
Problem	*Probability*	*General value for k*	*Value of k with* $P = 1/2$	*Number of students* ($N = 365$)
1	$P \approx 1 - e^{-k/N}$	$k \approx \ln[1/(1-P)] \times N$	$k \approx 0.69 \times N$	253
2	$P \approx 1 - e^{-(k-1)/N}$	$k \approx \ln[1/(1-P)] \times N + 1$	$k \approx 0.69 \times N + 1$	254
3	$P \approx 1 - e^{-k(k-1)/2N}$	$k \approx \{2 \ln [1/(1-P)]\}^{1/2} \times N^{1/2}$	$k \approx 1.18 \times N^{1/2}$	23
4	$P \approx 1 - e^{-k^2/2N}$	$k \approx \{\ln [1/(1-P)]\}^{1/2} \times N^{1/2}$	$k \approx 0.83 \times N^{1/2}$	16

The shaded value, 23, is the solution to the classical birthday paradox; if there are just 23 students in a classroom, it is likely (with $P \geq 1/2$) that two students have the same birthday (ignoring the year they have been born).

Comparison

The value of k in problems 1 or 2 is proportional to N; the value of k in problems 3 or 4 is proportional to $N^{1/2}$. As we will see shortly, the first two problems are related to preimage and second preimage attacks; the third and the fourth problems are related to the collision attack. The comparison shows it is much more difficult to launch a preimage or second preimage attack than to launch a collision attack. Figure 11.8 gives the graph of P versus k. For the first and second problem only one graph is shown (probabilities are very close). The graphs for the second and the third problems are more distinct.

Figure 11.8 *Graph of four birthday problems*



Attacks on Random Oracle Model

To better understand the nature of the hash functions and the importance of the Random Oracle Model, consider how Eve can attack a hash function created by the oracle. Suppose that the hash function creates digests of n bits. Then the digest can be thought of as a random variable uniformly distributed between 0 and $N - 1$ in which $N = 2^n$. In other words, there are 2^n possible values for the digest; each time the oracle randomly selects one of these values for a message. Note that this does not mean that the selection is exhaustive; some values may never be selected, but some may be selected several times. We assume that the hash function algorithm is public and Eve knows the size of the digest, n.

Preimage Attack

Eve has intercepted a digest D = h(M); she wants to find any message M′ such that D = h(M′). Eve can create a list of k messages and run Algorithm 11.1.

The algorithm can find a message for which D is the digest or it may fail. What is the probability of success of this algorithm? Obviously, it depends on the size of list, k, chosen by Eve. To find the probability, we use the first birthday problem. The digest created by the program defines the outcomes of a random variable. The probability of

Algorithm 11.1 *Preimage attack+*

```
Preimage_Attack (D)
{
   for (i = 1 to k)
   {
       create (M [i])
       T ← h(M [i])                    // T is a temporary digest
       if (T = D) return M [i]
   }
   return failure
}
```

success is $P \approx 1 - e^{-k/N}$. If Eve needs to be at least 50 percent successful, what should be the size of k? We also showed this value in Table 11.3 for the first birthday problem: $k \approx 0.69 \times N$, or $k \approx 0.69 \times 2^n$. In other words, for Eve to be successful more than 50 percent of the time, she needs to create a list of digest that is proportional to 2^n.

The difficulty of a preimage attack is proportional to 2^n.

Example 11.6

A cryptographic hash function uses a digest of 64 bits. How many digests does Eve need to create to find the original message with the probability more than 0.5?

Solution

The number of digests to be created is $k \approx 0.69 \times 2^n \approx 0.69 \times 2^{64}$. This is a large number. Even if Eve can create 2^{30} (almost one billion) messages per second, it takes 0.69×2^{34} seconds or more than 500 years. This means that a message digest of size 64 bits is secure with respect to preimage attack, but, as we will see shortly, is not secured to collision attack.

Second Preimage Attack

Eve has intercepted a digest D = h(M) and the corresponding message M; she wants to find another message M′ so that h(M′) = D. Eve can create a list of $k - 1$ messages and run Algorithm 11.2.

Algorithm 11.2 *Second preimage attack*

```
Second_Preimage_Attack (D, M)
{
   for (i = 1 to k –1)
   {
       create (M [i])
       T ← h (M [i])                   // T is a temporary digest
       if (T = D) return M [i]
   }
   return failure
}
```

The algorithm can find a second message for which D is also the digest or it may fail. What is the probability of success of this algorithm? Obviously, it depends on the size of list, k, chosen by Eve. To find the probability, we use the second birthday problem. The digest created by the program defines the outcomes of a random variable. The probability of success is $P \approx 1 - e^{-(k-1)/N}$. If Eve needs to be at least 50 percent successful, what should be the size of k? We also showed this value in Table 11.3 for the second birthday problem: $k \approx 0.69 \times N + 1$ or $k \approx 0.69 \times 2^n + 1$. In other words, for Eve to be successful more than 50 percent of the time, she needs to create a list of digest that is proportional to 2^n.

The difficulty of a second preimage attack is proportional to 2^n.

Collision Attack

Eve needs to find two messages, M and M′; such that h(M) = h(M′). Eve can create a list of k messages and run Algorithm 11.3.

Algorithm 11.3 *Collision attack*

```
Collision_Attack
{
   for (i = 1 to k )
   {
      create (M[i])
      D[i] ← h (M[i])                    // D [i] is a list of created digests
      for (j = 1 to i – 1)
      {
         if (D[i] = D[j]) return (M[i] and M[j])
      }
   }
   return failure
}
```

The algorithm can find two messages with the same digest. What is the probability of success of this algorithm? Obviously, it depends on the size of list, k, chosen by Eve. To find the probability, we use the third birthday problem. The digest created by program defines the outcomes of a random variable. The probability of success is $P \approx 1 - e^{-k(k-1)/2N}$. If Eve needs to be at least fifty percent successful, what should be the size of k? We also showed this value in Table 11.3 for the third birthday problem: $k \approx 1.18 \times N^{1/2}$, or $k \approx 1.18 \times 2^{n/2}$. In other words, for Eve to be successful more than 50 percent of the time, she needs to create a list of digests that is proportional to $2^{n/2}$.

The difficulty of a collision attack is proportional to $2^{n/2}$.

Example 11.7

A cryptographic hash function uses a digest of 64 bits. How many digests does Eve need to create to find two messages with the same digest with the probability more than 0.5?

Solution

The number of digests to be created is $k \approx 1.18 \times 2^{n/2} \approx 1.18 \times 2^{32}$. If Eve can test 2^{20} (almost one million) messages per second, it takes 1.18×2^{12} seconds, or less than two hours. This means that a message digest of size 64 bits is not secure against the collision attack.

Alternate Collision Attack

The previous collision attack may not be useful for Eve. The adversary needs to create two messages, one real and one bogus, that hash to the same value. Each message should be meaningful. The previous algorithm does not provide this type of collision. The solution is to create two meaningful messages, but add redundancies or modifications to the message to change the contents of the message without changing the meaning of each. For example, a number of messages can be made from the first message by adding spaces, or changing the words, or adding some redundant words, and so on. The second message can also create a number of messages. Let us call the original message M and the bogus message M′. Eve creates k different variants of M (M_1, M_2, …, M_k) and k different variants of M′ (M'_1, M'_2, …, M'_k). Eve then uses Algorithm 11.4 to launch the attack.

Algorithm 11.4 *Alternate collision attack*

```
Alternate_Collision_Attack (M [k], M′[k])
{
   for (i = 1 to k )
   {
      D[i] ← h (M[i])
      D′[i] ← h (M′[i])
      if (D [i] = D′[j]) return (M[i], M′[j])
   }
   return failure
}
```

What is the probability of success of this algorithm? Obviously, it depends on the size of the list, k, chosen by Eve. To find the probability, we use the fourth birthday problem. The two digest lists created by program defines the two outcomes of a random variable. The probability of success is $P \approx 1 - e^{-k^2/N}$. If Eve needs to be at least 50 percent successful, what should be the size of k? We also showed this value in Table 11.3 for the fourth birthday problem: $k \approx 0.83 \times N^{1/2}$ or $k \approx 0.83 \times 2^{n/2}$. In other words, for Eve to be successful more than 50% of the time, she needs to create a list of digests that is proportional to $2^{n/2}$.

The difficulty of an alternative collision attack is proportional to $2^{n/2}$.

Summary of Attacks

Table 11.4 shows the level of difficulty for each attack if the digest is n bits.

Table 11.4 *Levels of difficulties for each type of attack*

Attack	*Value of k with P=1/2*	*Order*
Preimage	$k \approx 0.69 \times 2^n$	2^n
Second preimage	$k \approx 0.69 \times 2^n + 1$	2^n
Collision	$k \approx 1.18 \times 2^{n/2}$	$2^{n/2}$
Alternate collision	$k \approx 0.83 \times 2^{n/2}$	$2^{n/2}$

Table 11.4 shows that the order, or the difficulty rate of the attack, is much less for collision attack than for preimage or second preimage attacks. If a hash algorithm is resistant to collision, we should not worry about preimage and second preimage attacks.

Example 11.8

Originally hash functions with a 64-bit digest were believed to be immune to collision attacks. But with the increase in the processing speed, today everyone agrees that these hash functions are no longer secure. Eve needs only $2^{64/2} = 2^{32}$ tests to launch an attack with probability 1/2 or more. Assume she can perform 2^{20} (one million) tests per second. She can launch an attack in $2^{32}/2^{20} = 2^{12}$ seconds (almost an hour).

Example 11.9

MD5 (see Chapter 12), which was one of the standard hash functions for a long time, creates digests of 128 bits. To launch a collision attack, the adversary needs to test 2^{64} ($2^{128/2}$) tests in the collision algorithm. Even if the adversary can perform 2^{30} (more than one billion) tests in a second, it takes 2^{34} seconds (more than 500 years) to launch an attack. This type of attack is based on the Random Oracle Model. It has been proved that MD5 can be attacked on less than 2^{64} tests because of the structure of the algorithm.

Example 11.10

SHA-1 (see Chapter 12), a standard hash function developed by NIST, creates digests of 160 bits. The function is attacks. To launch a collision attack, the adversary needs to test $2^{160/2} = 2^{80}$ tests in the collision algorithm. Even if the adversary can perform 2^{30} (more than one billion) tests in a second, it takes 2^{50} seconds (more than ten thousand years) to launch an attack. However, researchers have discovered some features of the function that allow it to be attacked in less time than calculated above.

Example 11.11

The new hash function, that is likely to become NIST standard, is SHA-512 (see Chapter 12), which has a 512-bit digest. This function is definitely resistant to collision attacks based on the Random Oracle Model. It needs $2^{512/2} = 2^{256}$ tests to find a collision with the probability of 1/2.

Attacks on the Structure

All discussions related to the attacks on hash functions have been based on an ideal cryptographic hash function that acts like an oracle; they were based on the Random Oracle Model. Although this type of analysis provides systematic evaluation of the algorithms, practical hash functions can have some internal structures that can make

them much weaker. It is not possible to make a hash function that creates digests that are completely random. The adversary may have other tools to attack hash function. One of these tools, for example, is the *meet-in-the-middle* attack that we discussed in Chapter 6 for double DES. We will see in the next chapters that some hash algorithms are subject to this type of attack. These types of hash function are far from the ideal model and should be avoided.

11.3 MESSAGE AUTHENTICATION

A message digest guarantees the integrity of a message. It guarantees that the message has not been changed. A message digest, however, does not authenticate the sender of the message. When Alice sends a message to Bob, Bob needs to know if the message is coming from Alice. To provide message authentication, Alice needs to provide proof that it is Alice sending the message and not an impostor. A message digest per se cannot provide such a proof. The digest created by a cryptographic hash function is normally called a modification detection code (MDC). The code can detect any modification in the message. What we need for message authentication (data origin authentication) is a message authentication code (MAC).

Modification Detection Code

A modification detection code (MDC) is a message digest that can prove the integrity of the message: that message has not been changed. If Alice needs to send a message to Bob and be sure that the message will not change during transmission, Alice can create a message digest, MDC, and send both the message and the MDC to Bob. Bob can create a new MDC from the message and compare the received MDC and the new MDC. If they are the same, the message has not been changed. Figure 11.9 shows the idea.

Figure 11.9 *Modification detection code (MDC)*

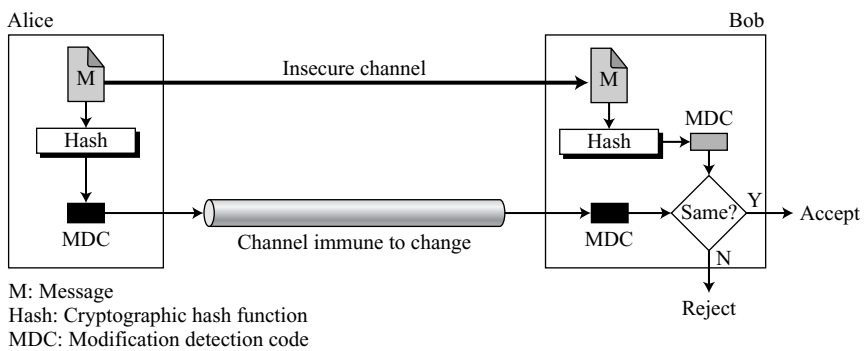


Figure 11.9 shows that the message can be transferred through an insecure channel. Eve can read or even modify the message. The MDC, however, needs to be transferred through a safe channel. The term *safe* here means immune to change. If both the

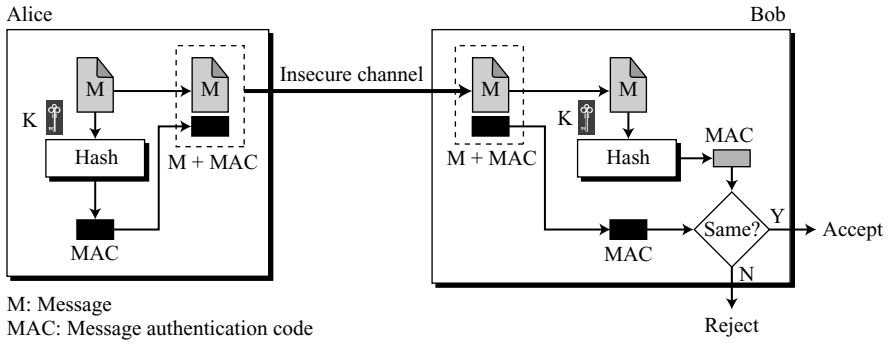
message and the MDC are sent through the insecure channel, Eve can intercept the message, change it, create a new MDC from the message, and send both to Bob. Bob never knows that the message has come from Eve. Note that the term *safe* can mean a trusted party; the term *channel* can mean the passage of time. For example, if Alice makes an MDC from her will and deposits it with her attorney, who keeps it locked away until her death, she has used a safe channel.

Alice writes her will and announces it publicly (insecure channel). Alice makes an MDC from the message and deposits it with her attorney, which is kept until her death (a secure channel). Although Eve may change the contents of the will, the attorney can create an MDC from the will and prove that Eve's version is a forgery. If the cryptography hash function used to create the MDC has the three properties described at the beginning of this chapter, Eve will lose.

Message Authentication Code (MAC)

To ensure the integrity of the message and the data origin authentication—that Alice is the originator of the message, not somebody else—we need to change a modification detection code (MDC) to a **message authentication code (MAC).** The difference between a MDC and a MAC is that the second includes a secret between Alice and Bob—for example, a secret key that Eve does not possess. Figure 11.10 shows the idea.

Figure 11.10 *Message authentication code*



Alice uses a hash function to create a MAC from the concatenation of the key and the message, h (K|M). She sends the message and the MAC to Bob over the insecure channel. Bob separates the message from the MAC. He then makes a new MAC from the concatenation of the message and the secret key. Bob then compares the newly created MAC with the one received. If the two MACs match, the message is authentic and has not been modified by an adversary.

Note that there is no need to use two channels in this case. Both message and the MAC can be sent on the same insecure channel. Eve can see the message, but she cannot forge a new message to replace it because Eve does not possess the secret key between Alice and Bob. She is unable to create the same MAC as Alice did.

The MAC we have described is referred to as a prefix MAC because the secret key is appended to the beginning of the message. We can have a postfix MAC, in which the key is appended to the end of the message. We can combine the prefix and postfix MAC, with the same key or two different keys. However, the resulting MACs are still insecure.

Security of a MAC

Suppose Eve has intercepted the message M and the digest h(K|M). How can Eve forge a message without knowing the secret key? There are three possible cases:

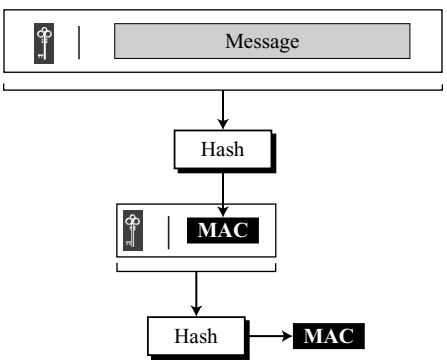
1. If the size of the key allows exhaustive search, Eve may prepend all possible keys at the beginning of the message and make a digest of the (K|M) to find the digest equal to the one intercepted. She then knows the key and can successfully replace the message with a forged message of her choosing.
2. The size of the key is normally very large in a MAC, but Eve can use another tool: the preimage attack discussed in Algorithm 11.1. She uses the algorithm until she finds X such that h(X) is equal to the MAC she has intercepted. She now can find the key and successfully replace the message with a forged one. Because the size of the key is normally very large for exhaustive search, Eve can only attack the MAC using the preimage algorithm.
3. Given some pairs of messages and their MACs, Eve can manipulate them to come up with a new message and its MAC.

The security of a MAC depends on the security of the underlying hash algorithm.

Nested MAC

To improve the security of a MAC, **nested MACs** were designed in which hashing is done in two steps. In the first step, the key is concatenated with the message and is hashed to create an intermediate digest. In the second step, the key is concatenated with the intermediate digest to create the final digest. Figure 11.12 shows the general idea.

Figure 11.11 *Nested MAC*

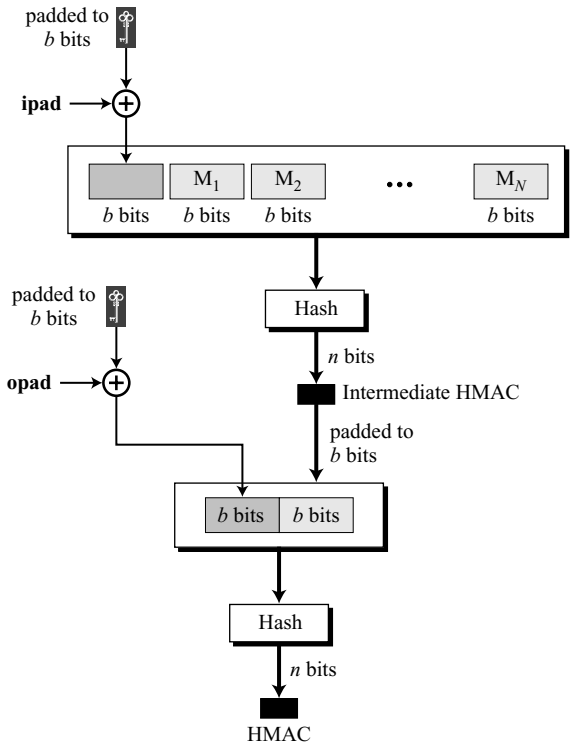


HMAC

NIST has issued a standard (FIPS 198) for a nested MAC that is often referred to as **HMAC** (hashed MAC, to distinguish it from CMAC, discussed in the next section). The implementation of HMAC is much more complex than the simplified nested MAC shown in Figure 11.11. There are additional features, such as padding. Figure 11.12 shows the details. We go through the steps:

1. The message is divided into N blocks, each of b bits.
2. The secret key is left-padded with 0's to create a b-bit key. Note that it is recommended that the secret key (before padding) be longer than n bits, where n is the size of the HMAC.
3. The result of step 2 is exclusive-ored with a constant called **ipad** (**input pad**) to create a b-bit block. The value of ipad is the $b/8$ repetition of the sequence 00110110 (36 in hexadecimal).
4. The resulting block is prepended to the N-block message. The result is $N + 1$ blocks.
5. The result of step 4 is hashed to create an n-bit digest. We call the digest the intermediate HMAC.

Figure 11.12 *Details of HMAC*

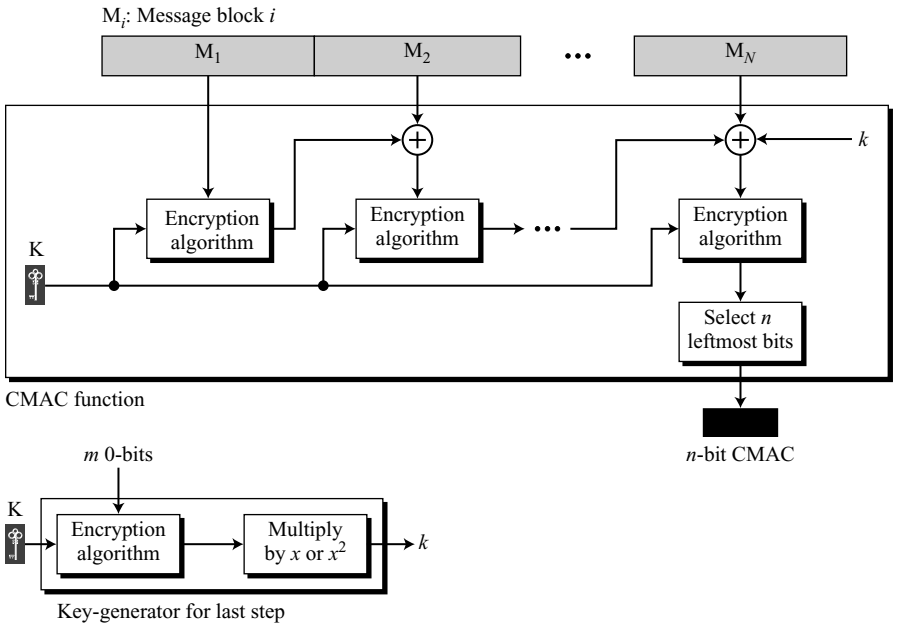


6. The intermediate *n*-bit HMAC is left padded with 0s to make a *b*-bit block.
7. Steps 2 and 3 are repeated by a different constant opad (**output pad**). The value of opad is the *b*/8 repetition of the sequence 01011100 (5C in hexadecimal).
8. The result of step 7 is prepended to the block of step 6.
9. The result of step 8 is hashed with the same hashing algorithm to create the final *n*-bit HMAC.

CMAC

NIST has also defined a standard (FIPS 113) called Data Authentication Algorithm, or **CMAC,** or **CBCMAC.** The method is similar to the cipher block chaining (CBC) mode discussed in Chapter 8 for symmetric-key encipherment. However, the idea here is not to create *N* blocks of ciphertext from *N* blocks of plaintext. The idea is to create one block of MAC from *N* blocks of plaintext using a symmetric-key cipher *N* times. Figure 11.13 shows the idea.

Figure 11.13 *CMAC*



The message is divided into *N* blocks, each *m* bits long. The size of the CMAC is *n* bits. If the last block is not *m* bits, it is padded with a 1-bit followed by enough 0-bits to make it *m* bits. The first block of the message is encrypted with the symmetric key to create an *m*-bit block of encrypted data. This block is XORed with the next block and the result is encrypted again to create a new *m*-bit block. The process continues until the last block of the message is encrypted. The *n* leftmost bit from the last block is the CMAC. In addition to the symmetric key, K, CMAC also uses another key, *k*,

which is applied only at the last step. This key is derived from the encryption algorithm with plaintext of m 0-bits using the cipher key, K. The result is then multiplied by x if no padding is applied and multiplied by x^2 if padding is applied. The multiplication is in GF(2^m) with the irreducible polynomial of degree m selected by the particular protocol used.

Note that this is different from the CBC used for confidentiality, in which the output of each encryption is sent as the ciphertext and at the same time XORed with the next plaintext block. Here the intermediate encrypted blocks are not sent as ciphertext; they are only used to be XORed with the next block.

11.4 RECOMMENDED READING

The following books and websites give more details about subjects discussed in this chapter. The items enclosed in brackets refer to the reference list at the end of the book.

Books

Several books that give a good coverage of cryptographic hash functions include [Sti06], [Sta06], [Sch99], [Mao04], [KPS02], [PHS03], and [MOV96].

WebSites

The following websites give more information about topics discussed in this chapter.

http://en.wikipedia.org/wiki/Preimage_attack
http://en.wikipedia.org/wiki/Collision_attack#In_cryptography
http://en.wikipedia.org/wiki/Pigeonhole_principle
csrc.nist.gov/ispab/2005-12/B_Burr-Dec2005-ISPAB.pdf
http://en.wikipedia.org/wiki/Message_authentication_code
http://en.wikipedia.org/wiki/HMAC
csrc.nist.gov/publications/fips/fips198/fips-198a.pdf
http://www.faqs.org/rfcs/rfc2104.html
http://en.wikipedia.org/wiki/Birthday_paradox

11.5 KEY TERMS

birthday problems
CBCMAC
CMAC
collision resistance
cryptographic hash function
hashed message authentication code (HMAC)
input pad (ipad)
message authentication code (MAC)
message digest
message digest domain
modification detection code (MDC)
nested MAC
output pad (opad)
pigeonhole principle
preimage resistance
Random Oracle Model
second preimage resistance

11.6 SUMMARY

- A fingerprint or a message digest can be used to ensure the integrity of a document or a message. To ensure the integrity of a document, both the document and the fingerprint are needed; to ensure the integrity of a message, both the message and the message digest are needed. The message digest needs to be kept safe from change.
- A cryptographic hash function creates a message digest out of a message.The function must meet three criteria: preimage resistance, second preimage resistance, and collision resistance.
- The first criterion, preimage resistance, means that it must be extremely hard for Eve to create any message from the digest. The second criterion, second preimage resistance, ensures that if Eve has a message and the corresponding digest, she should not be able to create a second message whose digest is the same as the first. The third criterion, collision resistance, ensures that Eve cannot find two messages that hash to the same digest.
- The Random Oracle Model, which was introduced in 1993 by Bellare and Rogaway, is an ideal mathematical model for a hash function.
- The pigeonhole principle states that if n pigeonholes are occupied by n +1 pigeons, then at least one pigeonhole is occupied by two pigeons.The generalized version of pigeonhole principle is that if n pigeonholes are occupied by kn + 1 pigeons, then at least one pigeonhole is occupied by k + 1 pigeons.
- The four birthday problems are used to analyze the Random Oracle Model. The first problem is used to analyze the preimage attack, the second problem is used to analyze the second preimage attack, and the third and the fourth problems are used to analyze the collision attack.
- A modification detection code (MDC) is a message digest that can prove the integrity of the message: that the message has not been changed. To prove the integrity of the message and the data origin authentication, we need to change a modification detection code (MDC) to a message authentication code (MAC). The difference between an MDC and a MAC is that the second includes a secret between the sender and the receiver.
- NIST has issued a standard (FIPS 198) for a nested MAC that is often referred to as HMAC (hashed MAC). NIST has also defined another standard (FIPS 113) called CMAC, or CBCMAC.

11.7 PRACTICE SET

Review Questions

1. Distinguish between message integrity and message authentication.
2. Define the first criterion for a cryptographic hash function.
3. Define the second criterion for a cryptographic hash function.

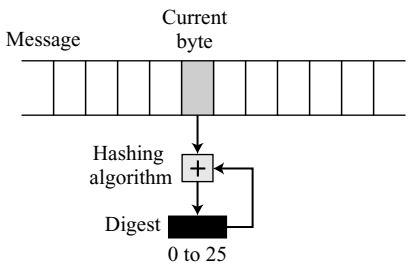
4. Define the third criterion for a cryptographic hash function.
5. Define the Random Oracle Model and describe its application in analyzing attacks on hash functions.
6. State the pigeonhole principle and describe its application in analyzing hash functions.
7. Define the four birthday problems discussed in this chapter.
8. Associate each birthday problem with one of the attacks on a hash function.
9. Distinguish between an MDC and a MAC.
10. Distinguish between HMAC and CMAC.

Exercises

11. In the Random Oracle Model, why does the oracle need to make a note of the digest created for a message and give the same digest for the same message?
12. Explain why private-public keys cannot be used in creating a MAC.
13. Ignoring the birth month, how many attempts, on average, are needed to find a person with the same birth date as yours? Assume that all months have 30 days.
14. Ignoring the birth month, how many attempts, on average, are needed to find two persons with the same birth date? Assume that all months have 30 days.
15. How many attempts, on average, are needed to find a person the same age as you, given a group of people born after 1950?
16. How many attempts, on average, are needed to find two people of the same age if we look for people born after 1950?
17. Answer the following questions about a family of six people, assuming that the birthdays are uniformly distributed through the days of a week, through the days of a month, through each month of a year, and through the 365 days of the year. Also assume that a year is exactly 365 days and each month is exactly 30 days.
    a. What is the probability that two of the family members have the same birthday? What is the probability that none of them have the same birthday?
    b. What is the probability that two of the family members are born in the same month? What is the probability that none of them were born in the same month?
    c. What is the probability that one of the family members is born on the first day of a month?
    d. What is the probability that three of the family members are born on the same day of the week?
18. What is the probability of birthday collision in two classes, one with k students and the other with l students?
19. In a class of 100 students, what is the probability that two or more students have Social Security Numbers with the same last four digits?
20. There are 100 students in a class and the professor assigns five grades (A, B, C, D, F) to a test. Show that at least 20 students have one of the grades.
21. Does the pigeonhole principle require the random distribution of pigeons to the pigeonholes?

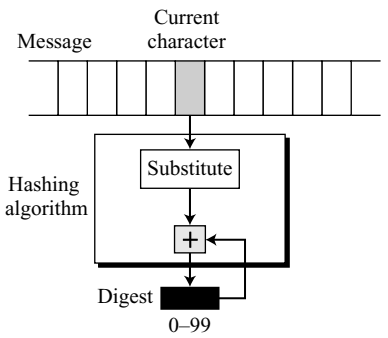
22. Assume that Eve is determined to find a preimage in Algorithm 11.1 What is the average number of times Eve needs to repeat the algorithm?
23. Assume Eve is determined to find a collision in Algorithm 11.3 What is the average number of times Eve needs to repeat the algorithm?
24. Assume we have a very simple message digest. Our unrealistic message digest is just one number between 0 and 25. The digest is initially set to 0. The cryptographic hash function adds the current value of the digest to the value of the current character (between 0 and 25). Addition is in modulo 26. Figure 11.14 shows the idea. What is the value of the digest if the message is "HELLO"? Why is this digest not secure?

Figure 11.14 *Exercise 24*



25. Let us increase the complexity of the previous exercise. We take the value of the current character, substitute it with another number, and then add it to the previous value of the digest in modulo 100 arithmetic. The digest is initially set to 0. Figure 11.15 shows the idea. What is the value of the digest if the message is "HELLO"? Why is this digest not secure?

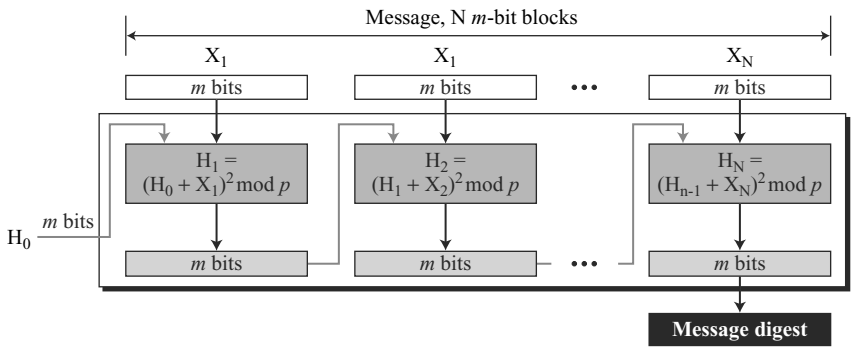
Figure 11.15 *Exercise 25*



26. Use modular arithmetic to find the digest of a message. Figure 11.16 shows the procedure. The steps are as follows:
    a. Let the length of the message digest be *m* bits.
    b. Choose a prime number, *p*, of *m* bits as the modulus.
    c. Represent the message as a binary number and pad the message with extra 0's to make it multiple of *m* bits.
    d. Divide the padded message into *N* blocks, each of *m* bits. Call the *i*th block X_i.
    e. Choose an initial digest of *m* bits, H_0.
    f. Repeat the following N times:
    $$H_i = (H_{i-1} + X_i)^2 \bmod p$$
    g. The digest is H_N.

    What is the value of the digest if the message is "HELLO"? Why is this digest not secure?

Figure 11.16 *Exercise 26*



27. A hash function, called Modular Arithmetic Secure Hash (**MASH**), is described below. Write an algorithm to calculate the digest, given the message. Find the digest of a message of your own.
    a. Let the length of the message digest be *N* bits.
    b. Choose two prime numbers, *p* and *q*. Calculate $M = pq$.
    c. Represent the message as a binary number and pad the message with extra 0s to make it a multiple of *N*/2 bits. N is chosen as a multiple of 16, less than the number of bits in M.
    d. Divide the padded message into *m* blocks, each of N/2 bits. Call each block X_i.
    e. Add the length of the message modulo N/2 as a binary number to the message. This makes the message $m + 1$ blocks of N/2 bits.
    f. Expand the message to obtain $m + 1$ blocks, each of N bits as shown below:
    Divide blocks X_1 to X_m into 4-bit groups. Insert 1111 before each group.
    Divide block X_{m+1} into 4-bit groups. Insert 1010 before each group.
    Call the expanded blocks $Y_1, Y_2, \ldots, Y_{m+1}$

g. Choose an initial digest of N bits, H_0.
h. Choose a constant K of N bits.
i. Repeat the following $m + 1$ times (T_i and G_i are intermediate values). The "||" symbol means to concatenate.

$$T_i = ((H_{i-1} \oplus Y_i) \,||\, K)^{257} \bmod M \qquad G_i = T_i \bmod 2^N \qquad H_i = H_{i-1} \oplus G_i$$

j. The digest is H_{m+1}.

28. Write an algorithm in pseudocode to solve the first birthday problem (in general form).
29. Write an algorithm in pseudocode to solve the second birthday problem (in general form).
30. Write an algorithm in pseudocode to solve the third birthday problem (in general form).
31. Write an algorithm in pseudocode to solve the fourth birthday problem (in general form).
32. Write an algorithm in pseudocode for HMAC.
33. Write an algorithm in pseudocode for CMAC.

PART

3

Integrity, Authentication, and Key Management

In Chapter 1, we saw that cryptography provides three techniques: symmetric-key ciphers, asymmetric-key ciphers, and hashing. Part Three discusses cryptographic hash functions and their applications. This part also explores other issues related to topics discussed in Parts One and Two, such as key management. Chapter 11 discusses the general idea behind message integrity and message authentication. Chapter 12 explores several cryptographic hash functions. Chapter 13 discusses digital signatures. Chapter 14 shows the ideas and methods of entity authentication. Finally, Chapter 15 discusses key management used for symmetric-key and asymmetric-key cryptography.

Chapter 11: Message Integrity and Message Authentication

Chapter 11 discusses general ideas related to cryptographic hash functions that are used to create a message digest from a message. Message digests guarantee the integrity of the message. The chapter then shows how simple message digests can be modified to authenticate the message.

Chapter 12: Cryptographic Hash Functions

Chapter 12 investigates several standard cryptographic hash function belonging to two broad categories: those with a compression function made from scratch and those with a block cipher as the compression function. The chapter then describes one hash function from each category, SHA-512 and Whirlpool.

Chapter 13: Digital Signatures

Chapter 13 discusses digital signatures. The chapter introduces several digital signature schemes, including RSA, ElGamal, Schnorr, DSS, and elliptic curve. The chapter also investigates some attacks on the above schemes and how they can be prevented.

Chapter 14: Entity Authentication

Chapter 14 first distinguishes between message authentication and entity authentication. The chapter then discusses some methods of entity authentication, including the use of a password, challenge-response methods, and zero-knowledge protocols. The chapter also includes some discussion on biometrics.

Chapter 15: Key Management

Chapter 15 first explains different approaches to key managements including the use of a a key-distribution center (KDC), certification authorities (CAs), and public-key infrastructure (PKI). This chapter shows how symmetric-key and asymmetric-key cryptography can complement each other to solve some problems such as key management.

CHAPTER 12

Cryptographic Hash Functions

Objectives

This chapter has several objectives:

- To introduce general ideas behind cryptographic hash functions
- To discuss the Merkle-Damgard scheme as the basis for iterated hash functions
- To distinguish between two categories of hash functions: those with a compression function made from scratch and those with a block cipher as the compression function
- To discuss the structure of SHA-512 as an example of a cryptographic hash function with a compression function made from scratch
- To discuss the structure of Whirlpool as an example of a cryptographic hash function with a block cipher as the compression function

12.1 INTRODUCTION

As discussed in Chapter 11, a cryptographic hash function takes a message of arbitrary length and creates a message digest of fixed length. The ultimate goal of this chapter is to discuss the details of the two most promising cryptographic hash algorithms—*SHA-512* and *Whirlpool*. However, we first need to discuss some general ideas that may be applied to any cryptographic hash function.

Iterated Hash Function

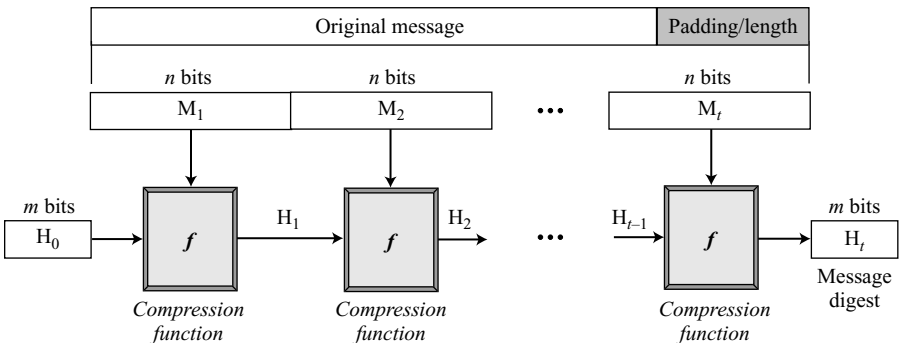
All cryptographic hash functions need to create a fixed-size digest out of a variable-size message. Creating such a function is best accomplished using iteration. Instead of using a hash function with variable-size input, a function with fixed-size input is created and is used a necessary number of times. The fixed-size input function is referred to as a

compression function. It compresses an n-bit string to create an m-bit string where n is normally greater than m. The scheme is referred to as an **iterated cryptographic hash function.**

Merkle-Damgard Scheme

The **Merkle-Damgard scheme** is an iterated hash function that is collision resistant if the compression function is collision resistant. This can be proved, but the proof is left as an exercise. The scheme is shown in Figure 12.1.

Figure 12.1 *Merkle-Damgard scheme*



The scheme uses the following steps:

1. The message length and padding are appended to the message to create an augmented message that can be evenly divided into blocks of n bits, where n is the size of the block to be processed by the compression function.
2. The message is then considered as t blocks, each of n bits. We call each block M_1, M_2,…, M_t. We call the digest created at t iterations H_1, H_2,…, H_t.
3. Before starting the iteration, the digest H_0 is set to a fixed value, normally called IV (initial value or initial vector).
4. The compression function at each iteration operates on H_{i-1} and M_i to create a new H_i. In other words, we have $H_i = f(H_{i-1}, M_i)$, where f is the compression function.
5. H_t is the cryptographic hash function of the original message, that is, h(M).

If the compression function in the Merkle-Damgard scheme is collision resistant, the hash function is also collision resistant.

Two Groups of Compression Functions

The Merkle-Damgard scheme is the basis for many cryptographic hash functions today. The only thing we need to do is design a compression function that is collision resistant

and insert it in the Merkle-Damgard scheme. There is a tendency to use two different approaches in designing a hash function. In the first approach, the compression function is made from scratch: it is particularly designed for this purpose. In the second approach, a symmetric-key block cipher serves as a compression function.

Hash Functions Made from Scratch

A set of cryptographic hash functions uses compression functions that are made from scratch. These compression functions are specifically designed for the purposes they serve.

Message Digest (MD) Several hash algorithms were designed by Ron Rivest. These are referred to as **MD2, MD4,** and **MD5,** where MD stands for Message Digest. The last version, MD5, is a strengthened version of MD4 that divides the message into blocks of 512 bits and creates a 128-bit digest. It turned out that a message digest of size 128 bits is too small to resist collision attack.

Secure Hash Algorithm (SHA) The **Secure Hash Algorithm (SHA)** is a standard that was developed by the National Institute of Standards and Technology (NIST) and published as a Federal Information Processing standard (FIP 180). It is sometimes referred to as **Secure Hash Standard (SHS).** The standard is mostly based on MD5. The standard was revised in 1995 under FIP 180-1, which includes **SHA-1**. It was revised later under FIP 180-2, which defines four new versions: **SHA-224, SHA-256, SHA-384,** and **SHA-512.** Table 12.1 lists some of the characteristics of these versions.

Table 12.1 *Characteristics of Secure Hash Algorithms (SHAs)*

Characteristics	*SHA-1*	*SHA-224*	*SHA-256*	*SHA-384*	*SHA-512*
Maximum Message size	$2^{64} - 1$	$2^{64} - 1$	$2^{64} - 1$	$2^{128} - 1$	$2^{128} - 1$
Block size	512	512	512	1024	1024
Message digest size	160	224	256	384	512
Number of rounds	80	64	64	80	80
Word size	32	32	32	64	64

All of these versions have the same structure. SHA-512 is discussed in detail later in this chapter.

Other Algorithms **RACE Integrity Primitives Evaluation Message Digest (RIPMED)** has several versions. **RIPEMD-160** is a hash algorithm with a 160-bit message digest. RIPEMD-160 uses the same structure as MD5 but uses two parallel lines of execution. **HAVAL** is a variable-length hashing algorithm with a message digest of size 128, 160, 192, 224, and 256.The block size is 1024 bits.

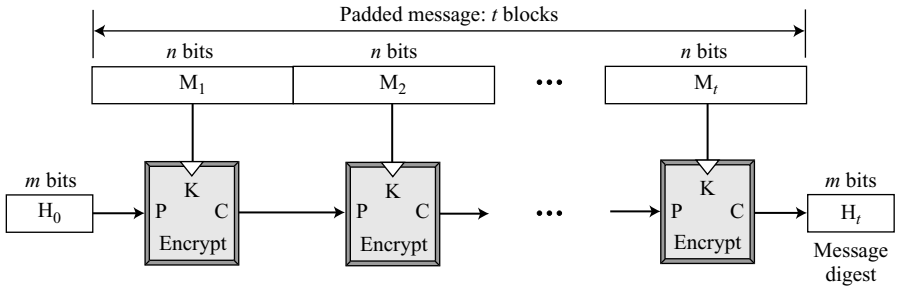
Hash Functions Based on Block Ciphers

An iterated cryptographic hash function can use a symmetric-key block cipher as a compression function. The whole idea is that there are several secure symmetric-key block ciphers, such as triple DES or AES, that can be used to make a one-way function instead of creating a new compression function. The block cipher in this case only

performs encryption. Several schemes have been proposed.We later describe one of the most promising, *Whirlpool*.

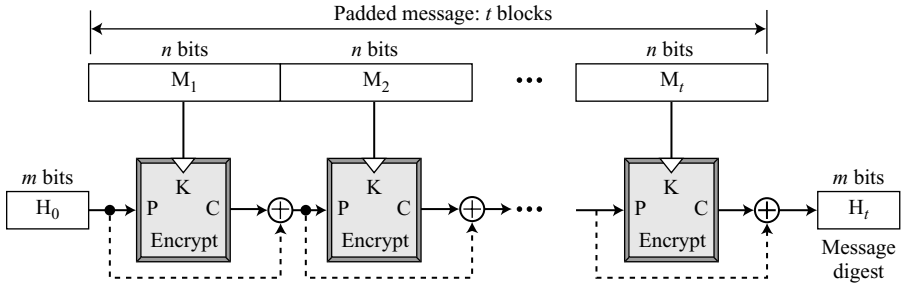
Rabin Scheme The iterated hash function proposed by Rabin is very simple. The **Rabin scheme** is based on the Merkle-Damgard scheme. The compression function is replaced by any encrypting cipher. The message block is used as the key; the previously created digest is used as the plaintext. The ciphertext is the new message digest. Note that the size of the digest is the size of data block cipher in the underlying cryptosystem. For example, if DES is used as the block cipher, the size of the digest is only 64 bits. Although the scheme is very simple, it is subject to a meet-in-the-middle attack discussed in Chapter 6, because the adversary can use the decryption algorithm of the cryptosystem. Figure 12.2 shows the Rabin scheme.

Figure 12.2 *Rabin scheme*



Davies-Meyer Scheme The **Davies-Meyer scheme** is basically the same as the Rabin scheme except that it uses forward feed to protect against meet-in-the-middle attack. Figure 12.3 shows the Davies-Meyer scheme.

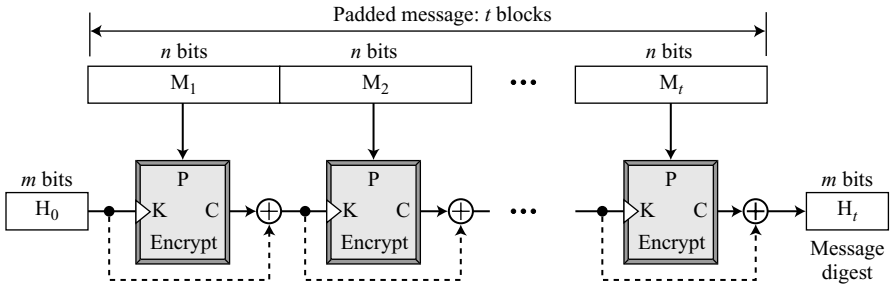
Figure 12.3 *Davies-Meyer scheme*



Matyas-Meyer-Oseas Scheme The **Matyas-Meyer-Oseas scheme** is a dual version of the Davies-Meyer scheme: the message block is used as the key to the cryptosystem. The scheme can be used if the data block and the cipher key are the same size. For

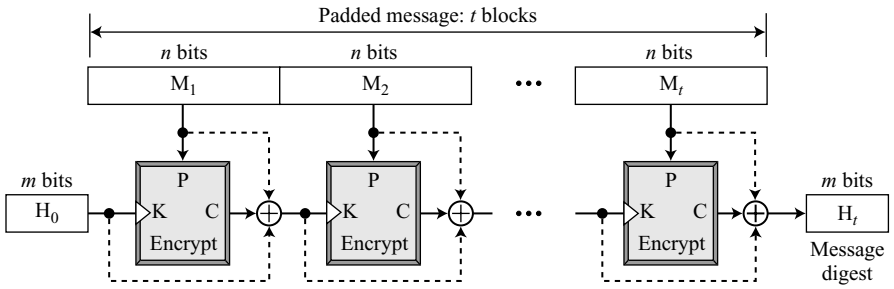
example, AES is a good candidate for this purpose. Figure 12.4 shows the Matyas-Meyer-Oseas scheme.

Figure 12.4 *Matyas-Meyer-Oseas scheme*



Miyaguchi-Preneel Scheme The **Miyaguchi-Preneel scheme** is an extended version of Matyas-Meyer-Oseas. To make the algorithm stronger against attack, the plaintext, the cipher key, and the ciphertext are all exclusive-ored together to create the new digest. This is the scheme used by the Whirlpool hash function. Figure 12.5 shows the Miyaguchi-Preneel scheme.

Figure 12.5 *Miyaguchi-Preneel scheme*



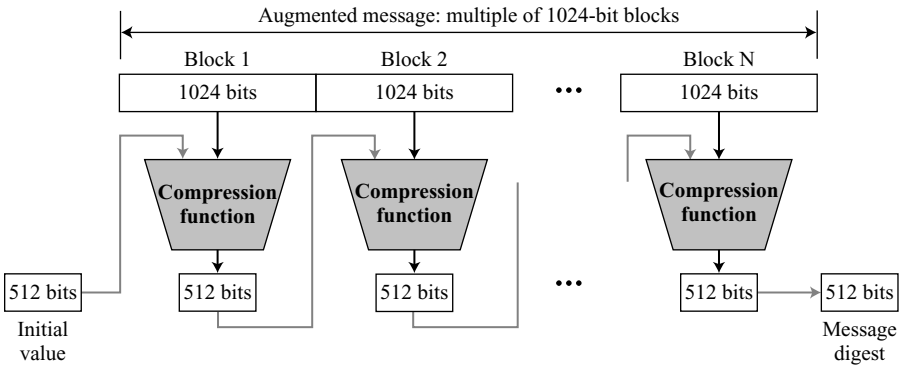
12.2 SHA-512

SHA-512 is the version of SHA with a 512-bit message digest. This version, like the others in the SHA family of algorithms, is based on the Merkle-Damgard scheme. We have chosen this particular version for discussion because it is the latest version, it has a more complex structure than the others, and its message digest is the longest. Once the structure of this version is understood, it should not be difficult to understand the structures of the other versions. For characteristics of SHA-512 see Table 12.1.

Introduction

SHA-512 creates a digest of 512 bits from a multiple-block message. Each block is 1024 bits in length, as shown in Figure 12.6.

Figure 12.6 *Message digest creation SHA-512*



The digest is initialized to a predetermined value of 512 bits. The algorithm mixes this initial value with the first block of the message to create the first intermediate message digest of 512 bits. This digest is then mixed with the second block to create the second intermediate digest. Finally, the $(N - 1)$th digest is mixed with the Nth block to create the Nth digest. When the last block is processed, the resulting digest is the message digest for the entire message.

Message Preparation

SHA-512 insists that the length of the original message be less than 2^{128} bits. This means that if the length of a message is equal to or greater than 2^{128}, it will not be processed by SHA-512. This is not usually a problem because 2^{128} bits is probably larger than the total storage capacity of any system.

SHA-512 creates a 512-bit message digest out of a message less than 2^{128}.

Example 12.1

This example shows that the message length limitation of SHA-512 is not a serious problem. Suppose we need to send a message that is 2^{128} bits in length. How long does it take for a communications network with a data rate of 2^{64} bits per second to send this message?

Solution

A communications network that can send 2^{64} bits per second is not yet available. Even if it were, it would take many years to send this message. This tells us that we do not need to worry about the SHA-512 message length restriction.

Example 12.2

This example also concerns the message length in SHA-512. How many pages are occupied by a message of 2^{128} bits?

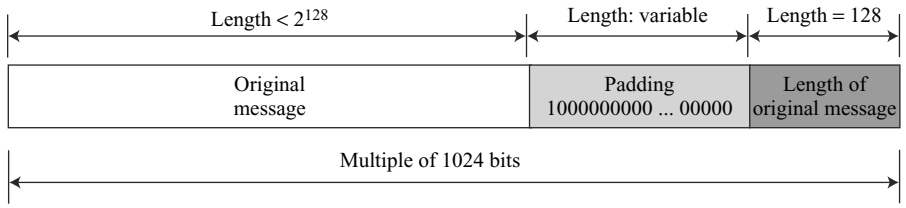
Solution

Suppose that a character is 64, or 2^6, bits. Each page is less than 2048, or approximately 2^{12}, characters. So 2^{128} bits need at least $2^{128} / 2^{18}$, or 2^{110}, pages. This again shows that we need not worry about the message length restriction.

Length Field and Padding

Before the message digest can be created, SHA-512 requires the addition of a 128-bit unsigned-integer length field to the message that defines the length of the message in bits. This is the length of the original message before padding. An unsigned integer field of 128 bits can define a number between 0 and $2^{128} - 1$, which is the maximum length of the message allowed in SHA-512. The length field defines the length of the original message before adding the length field or the padding (Figure 12.7).

Figure 12.7 *Padding and length field in SHA-512*



Before the addition of the length field, we need to pad the original message to make the length a multiple of 1024. We reserve 128 bits for the length field, as shown in Figure 12.7. The length of the padding field can be calculated as follows. Let |M| be the length of the original message and |P| be the length of the padding field.

$$(|M| + |P| + 128) = 0 \bmod 1024 \quad \rightarrow \quad |P| = (-|M| - 128) \bmod 1024$$

The format of the padding is one 1 followed by the necessary number of 0s.

Example 12.3

What is the number of padding bits if the length of the original message is 2590 bits?

Solution

We can calculate the number of padding bits as follows:

$$|P| = (-2590 - 128) \bmod 1024 = -2718 \bmod 1024 = 354$$

The padding consists of one 1 followed by 353 0's.

Example 12.4

Do we need padding if the length of the original message is already a multiple of 1024 bits?

Solution

Yes we do, because we need to add the length field. So padding is needed to make the new block a multiple of 1024 bits.

Example 12.5

What is the minimum and maximum number of padding bits that can be added to a message?

Solution

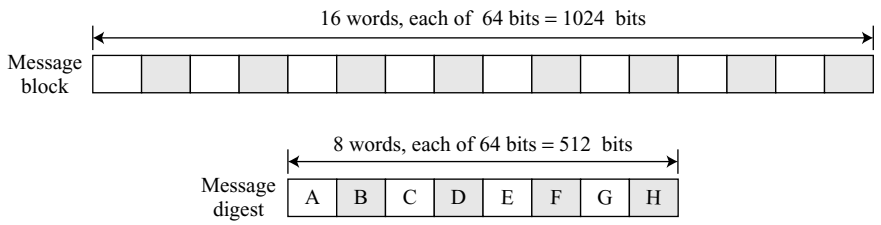
a. The minimum length of padding is 0 and it happens when $(-M - 128) \bmod 1024$ is 0. This means that $|M| = -128 \bmod 1024 = 896 \bmod 1024$ bits. In other words, the last block in the original message is 896 bits. We add a 128-bit length field to make the block complete.

b. The maximum length of padding is 1023 and it happens when $(-|M| - 128) = 1023 \bmod 1024$. This means that the length of the original message is $|M| = (-128 - 1023) \bmod 1024$ or the length is $|M| = 897 \bmod 1024$. In this case, we cannot just add the length field because the length of the last block exceeds one bit more than 1024. So we need to add 897 bits to complete this block and create a second block of 896 bits. Now the length can be added to make this block complete.

Words

SHA-512 operates on words; it is **word oriented.** A word is defined as 64 bits. This means that, after the padding and the length field are added to the message, each block of the message consists of sixteen 64-bit words. The message digest is also made of 64-bit words, but the message digest is only eight words and the words are named A, B, C, D, E, F, G, and H, as shown in Figure 12.8.

Figure 12.8 *A message block and the digest as words*

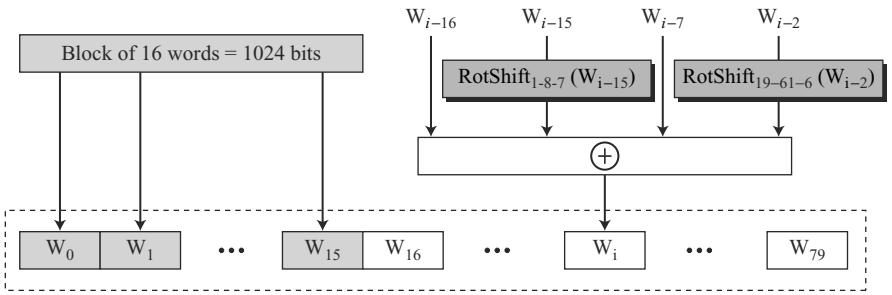


SHA-252 is word-oriented. Each block is 16 words; the digest is only 8 words.

Word Expansion

Before processing, each message block must be expanded. A block is made of 1024 bits, or sixteen 64-bit words. As we will see later, we need 80 words in the processing phase. So the 16-word block needs to be expanded to 80 words, from W_0 to W_{79}. Figure 12.9 shows the **word-expansion** process. The 1024-bit block becomes the first 16 words; the rest of the words come from already-made words according to the operation shown in the figure.

Figure 12.9 *Word expansion in SHA-512*



RotShift$_{1\text{-}m\text{-}n}$ (x): $\boldsymbol{RotR_l(x) \oplus RotR_m(x) \oplus ShL_n(x)}$

$\boldsymbol{RotR_i(x)}$: Right-rotation of the argument x by i bits
$\boldsymbol{ShL_i(x)}$: Shift-left of the argument x by i bits and padding the left by 0's.

Example 12.6

Show how W_{60} is made.

Solution

Each word in the range W_{16} to W_{79} is made from four previously-made words. W_{60} is made as

$$W_{60} = W_{44} \oplus \text{RotShift}_{1\text{-}8\text{-}7}(W_{45}) \oplus W_{53} \oplus \text{RotShift}_{19\text{-}61\text{-}6}(W_{58})$$

Message Digest Initialization

The algorithm uses eight constants for message digest initialization. We call these constants A_0 to H_0 to match with the word naming used for the digest. Table 12.2 shows the value of these constants.

Table 12.2 *Values of constants in message digest initialization of SHA-512*

Buffer	*Value (in hexadecimal)*	*Buffer*	*Value (in hexadecimal)*
A_0	`6A09E667F3BCC908`	E_0	`510E527FADE682D1`
B_0	`BB67AE8584CAA73B`	F_0	`9B05688C2B3E6C1F`
C_0	`3C6EF372EF94F82B`	G_0	`1F83D9ABFB41BD6B`
D_0	`A54FE53A5F1D36F1`	H_0	`5BE0CD19137E2179`

The reader may wonder where these values come from. The values are calculated from the first eight prime numbers (2, 3, 5, 7, 11, 13, 17, and 19). Each value is the fraction part of the square root of the corresponding prime number after converting to binary and keeping only the first 64 bits. For example, the eighth prime is 19, with the square root $(19)^{1/2} = 4.35889894354$. Converting the number to binary with only 64 bits in the fraction part, we get

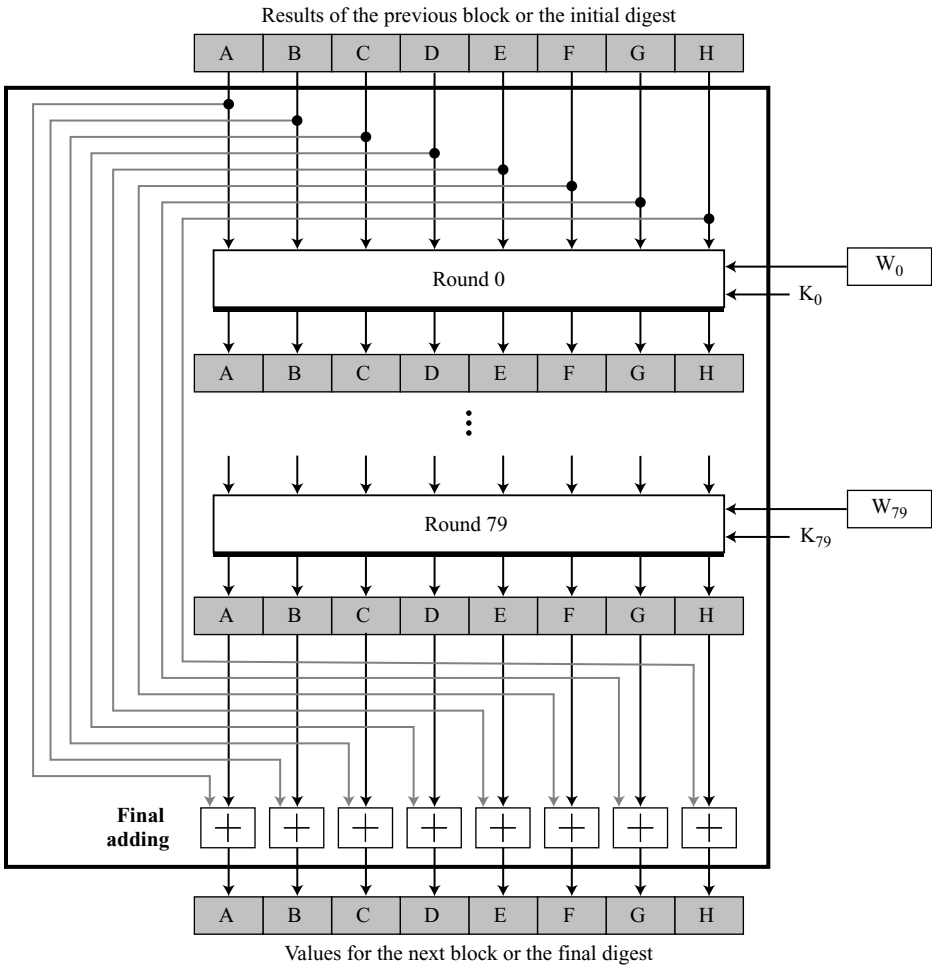
$$(100.0101\ 1011\ 1110 \ldots 1001)_2 \quad \rightarrow \quad (4.5\text{BE0CD19137E2179})_{16}$$

SHA-512 keeps the fraction part, $(5\text{BE0CD19137E2179})_{16}$, as an unsigned integer.

Compression Function

SHA-512 creates a 512-bit (eight 64-bit words) message digest from a multiple-block message where each block is 1024 bits. The processing of each block of data in SHA-512 involves 80 rounds. Figure 12.10 shows the general outline for the compression function. In each round, the contents of eight previous buffers, one word from the expanded block (W_i), and one 64-bit constant (K_i) are mixed together and then operated on to create a new set of eight buffers. At the beginning of processing, the values of the eight buffers are saved into eight temporary variables. At the end of the processing (after step 79), these values are added to the values created from step 79. We call this last operation the *final adding,* as shown in the figure.

Figure 12.10 *Compression function in SHA-512*



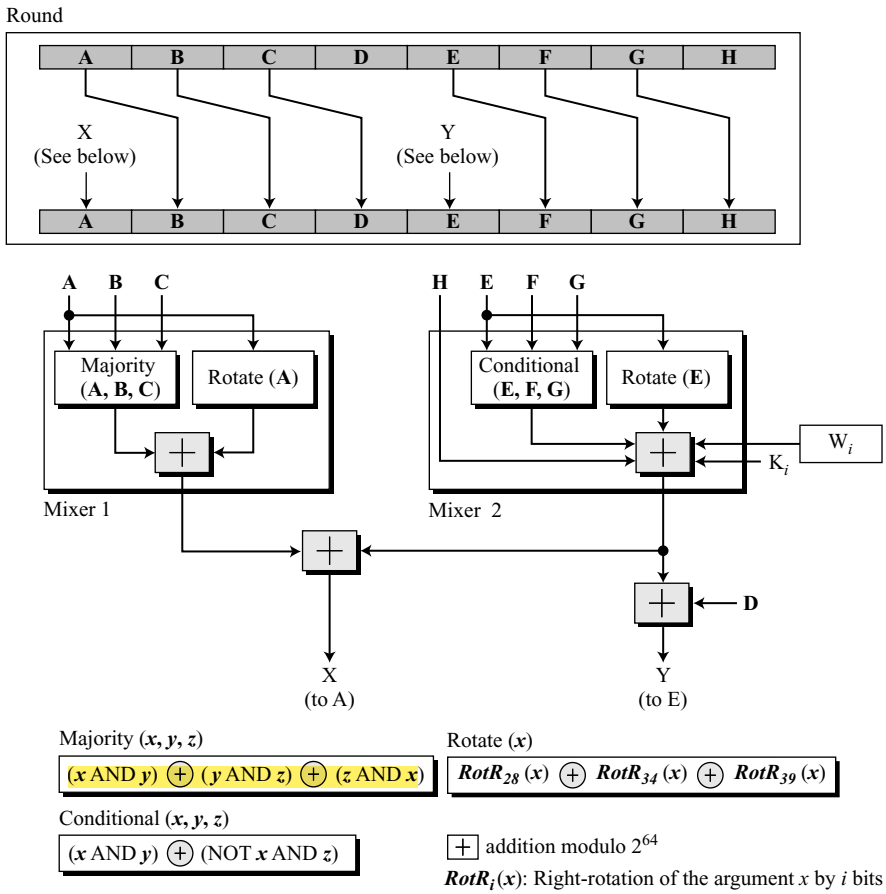
Structure of Each Round

In each round, eight new values for the 64-bit buffers are created from the values of the buffers in the previous round. As Figure 12.11 shows, six buffers are the exact copies of one of the buffers in the previous round as shown below:

$$A \rightarrow B \quad B \rightarrow C \quad C \rightarrow D \quad E \rightarrow F \quad F \rightarrow G \quad G \rightarrow H$$

Two of the new buffers, A and E, receive their inputs from some complex functions that involve some of the previous buffers, the corresponding word for this round (W_i), and the corresponding constant for this round (K_i). Figure 12.11 shows the structure of each round.

Figure 12.11 *Structure of each round in SHA-512*



There are two mixers, three functions, and several operators. Each mixer combines two functions. The description of the functions and operators follows:

1. The Majority function, as we call it, is a bitwise function. It takes three corresponding bits in three buffers (A, B, and C) and calculates

$$(A_j \text{ AND } B_j) \oplus (B_j \text{ AND } C_j) \oplus (C_j \text{ AND } A_j)$$

   The resulting bit is the majority of three bits. If two or three bits are 1's, the resulting bit is 1; otherwise it is 0.

2. The Conditional function, as we call it, is also a bitwise function. It takes three corresponding bits in three buffers (E, F, and G) and calculates

$$(E_j \text{ AND } F_j) \oplus (\text{NOT } E_j \text{ AND } G_j)$$

   The resulting bit is the logic "If E_j then F_j; else G_j".

3. The Rotate function, as we call it, right-rotates the three instances of the same buffer (A or E) and applies the exclusive-or operation on the results.

$$\text{Rotate (A): } \text{RotR}_{28}(A) \oplus \text{RotR}_{34}(A) \oplus \text{RotR}_{29}(A)$$

$$\text{Rotate (E): } \text{RotR}_{28}(E) \oplus \text{RotR}_{34}(E) \oplus \text{RotR}_{29}(E)$$

4. The right-rotation function, $\text{Rot}R_i(x)$, is the same as the one we used in the word-expansion process. It right-rotates its argument *i* bits; it is actually a circular shift-right operation.

5. The addition operator used in the process is addition modulo 2^{64}. This means that the result of adding two or more buffers is always a 64-bit word.

6. There are 80 constants, K_0 to K_{79}, each of 64 bits as shown in Table 12.3 in hexadecimal format (four in a row). Similar to the initial values for the eight digest buffers, these values are calculated from the first 80 prime numbers (2, 3,…, 409).

Table 12.3 *Eighty constants used for eighty rounds in SHA-512*

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	4597F299CFC657E2	5FCB6FAB3AD6FAEC	6C44198C4A475817

Each value is the fraction part of the cubic root of the corresponding prime number after converting it to binary and keeping only the first 64 bits. For example, the 80th prime is 409, with the cubic root $(409)^{1/3} = 7.42291412044$. Converting this number to binary with only 64 bits in the fraction part, we get

$$(111.0110\ 1100\ 0100\ 0100\ \ldots\ 0111)_2 \rightarrow (7.6C44198C4A475817)_{16}$$

SHA-512 keeps the fraction part, $(6C44198C4A475817)_{16}$, as an unsigned integer.

Example 12.7

We apply the Majority function on buffers A, B, and C. If the leftmost hexadecimal digits of these buffers are 0x7, 0xA, and 0xE, respectively, what is the leftmost digit of the result?

Solution

The digits in binary are 0111, 1010, and 1110.

a. The first bits are 0, 1, and 1. The majority is 1. We can also prove it using the definition of the Majority function:

$$(0 \text{ AND } 1) \oplus (1 \text{ AND } 1) \oplus (1 \text{ AND } 0) = 0 \oplus 1 \oplus 0 = 1$$

b. The second bits are 1, 0, and 1. The majority is 1.

c. The third bits are 1, 1, and 1. The majority is 1.

d. The fourth bits are 1, 0, and 0. The majority is 0.

The result is 1110, or 0xE in hexadecimal.

Example 12.8

We apply the Conditional function on E, F, and G buffers. If the leftmost hexadecimal digits of these buffers are 0x9, 0xA, and 0xF respectively, what is the leftmost digit of the result?

Solution

The digits in binary are 1001, 1010, and 1111.

a. The first bits are 1, 1, and 1. Since $E_1 = 1$, the result is F_1, which is 1. We can also use the definition of the Condition function to prove the result:

$$(1 \text{ AND } 1) \oplus (\text{NOT } 1 \text{ AND } 1) = 1 \oplus 0 = 1$$

b. The second bits are 0, 0, and 1. Since E_2 is 0, the result is G_2, which is 1.

c. The third bits are 0, 1, and 1. Since E_3 is 0, the result is G_3, which is 1.

d. The fourth bits are 1, 0, and 1. Since E_4 is 1, the result is F_4, which is 0.

The result is 1110, or 0xE in hexadecimal.

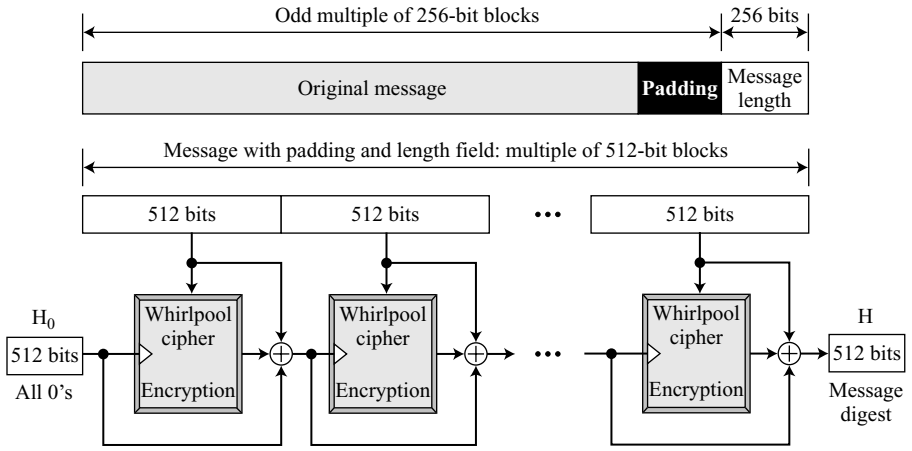
Analysis

With a message digest of 512 bits, SHA-512 expected to be resistant to all attacks, including collision attacks. It has been claimed that this version's improved design makes it more efficient and more secure than the previous versions. However, more research and testing are needed to confirm this claim.

12.3 WHIRLPOOL

Whirlpool is designed by Vincent Rijmen and Paulo S. L. M. Barreto. It is endorsed by the **New European Schemes for Signatures, Integrity, and Encryption (NESSIE).** Whirlpool is an iterated cryptographic hash function, based on the Miyaguchi-Preneel scheme, that uses a symmetric-key block cipher in place of the compression function. The block cipher is a modified AES cipher that has been tailored for this purpose. Figure 12.12 shows the Whirlpool hash function.

Figure 12.12 *Whirlpool hash function*



Preparation

Before starting the hash algorithm, the message needs to be prepared for processing. Whirlpool requires that the length of the original message be less than 2^{256} bits. A message needs to be padded before being processed. The padding is a single 1-bit followed by the necessary numbers of 0-bits to make the length of the padding an odd multiple of 256 bits. After padding, a block of 256 bits is added to define the length of the original message. This block is treated as an unsigned number.

After padding and adding the length field, the augmented message size is an even multiple of 256 bits or a multiple of 512 bits. Whirlpool creates a digest of 512 bits from a multiple 512-bit block message. The 512-bit digest, H_0, is initialized to all 0's. This value becomes the cipher key for encrypting the first block. The ciphertext resulting from encrypting each block becomes the cipher key for the next block after being exclusive-ored with the previous cipher key and the plaintext block. The message digest is the final 512-bit ciphertext after the last exclusive-or operation.

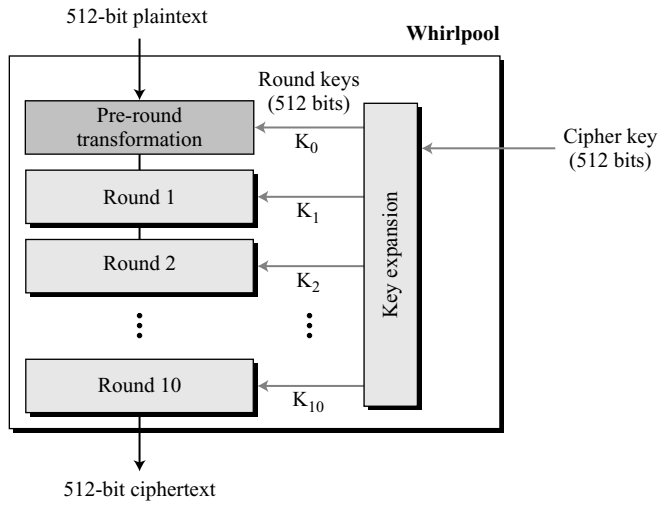
Whirlpool Cipher

The **Whirlpool cipher** is a non-Feistel cipher like AES that was mainly designed as a block cipher to be used in a hash algorithm. Instead of giving the whole description of this cipher, we just assume that the reader is familiar with AES from Chapter 7. Here the Whirlpool cipher is compared with the AES cipher and their differences are mentioned.

Rounds

Whirlpool is a round cipher that uses 10 rounds. The block size and key size are 512 bits. The cipher uses 11 round keys, K_0 to K_{10}, each of 512 bits. Figure 12.13 shows the general design of the Whirlpool cipher.

Figure 12.13 *General idea of the Whirlpool cipher*



States and Blocks

Like the AES cipher, the Whirlpool cipher uses states and blocks. However, the size of the block or state is 512 bits. A block is considered as a row matrix of 64 bytes; a state is considered as a square matrix of 8×8 bytes. Unlike AES, the block-to-state or state-to-block transformation is done row by row. Figure 12.14 shows the block, the state, and the transformation in the Whirlpool cipher.

Structure of Each Round

Figure 12.15 shows the structure of each round. Each round uses four transformations.

SubBytes Like in AES, **SubBytes** provide a nonlinear transformation. A byte is represented as two hexadecimal digits. The left digit defines the row and the right digit defines the column of the substitution table. The two hexadecimal digits at the junction of the row and the column are the new byte. Figure 12.16 shows the idea.

Figure 12.14 *Block and state in the Whirlpool cipher*

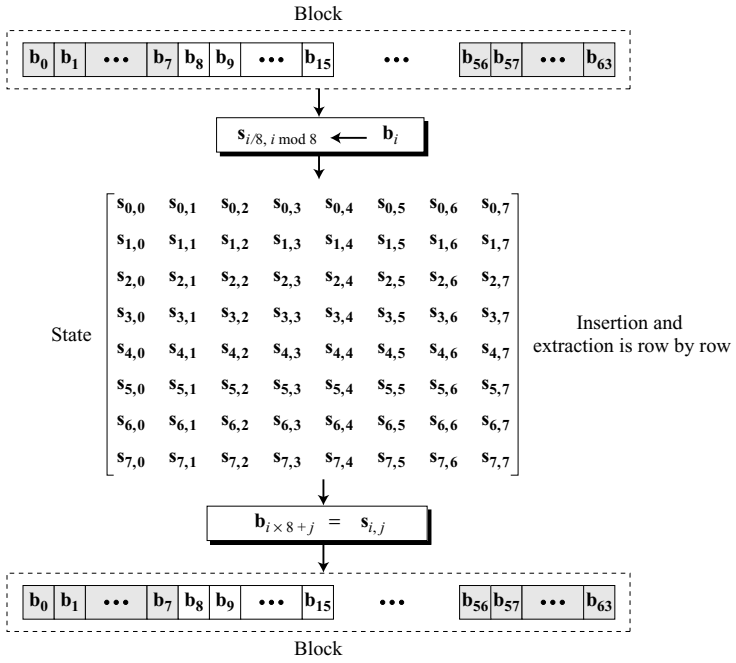


Figure 12.15 *Structure of each round in the Whirlpool cipher*

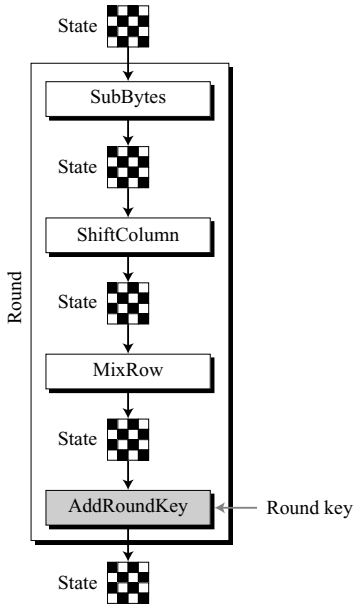
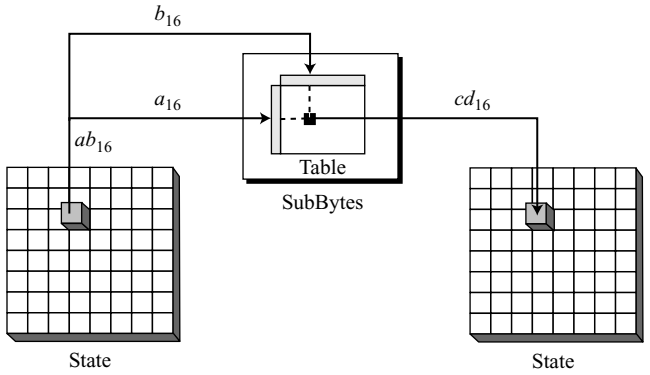


Figure 12.16 *SubBytes transformations in the Whirlpool cipher*



In the SubBytes transformation, the state is treated as an 8×8 matrix of bytes. Transformation is done one byte at a time. The contents of each byte are changed, but the arrangement of the bytes in the matrix remains the same. In the process, each byte is transformed independently; we have 64 distinct byte-to-byte transformations.

Table 12.4 shows the substitution table (S-Box) for SubBytes transformation. The transformation definitely provides confusion effect. For example, two bytes, $5A_{16}$ and $5B_{16}$, which differ only in one bit (the rightmost bit), are transformed to $5B_{16}$ and 88_{16}, which differ in five bits.

Table 12.4 *SubBytes transformation table (S-Box)*

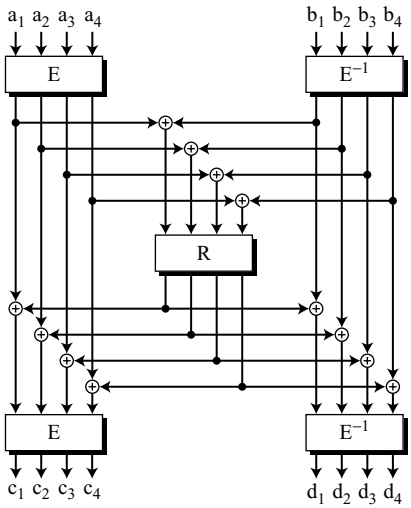
	0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***A***	***B***	***C***	***D***	***E***	***F***
0	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
1	16	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
2	15	77	37	E5	9F	F0	4A	CA	58	C9	29	0A	B1	A0	6B	85
3	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	C8
4	FB	EF	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
5	63	02	AA	71	C8	19	49	C9	F2	E3	5B	88	9A	26	32	B0
6	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
7	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
8	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
9	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B	5E	78	38	8C	C1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D	3C	81	94	F7	9B	13	2C	D3	E7	6E	C4	03	56	44	7E	A9
E	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F	16	3A	69	09	70	B6	C0	ED	CC	42	98	A4	28	5C	F8	86

The entries in Table 12.4 can be calculated algebraically using the GF(2^4) field with the irreducible polynomials ($x^4 + x + 1$) as shown in Figure 12.17. Each hexadecimal digit in a byte is the input to a minibox (E and E^{-1}). The results are fed into another minibox, R. The E boxes calculate the exponential of input hexadecimal; the R box uses a pseudorandom number generator.

$$\mathbf{E}(\text{input}) = (x^3 + x + 1)^{\text{input}} \bmod (x^4 + x + 1) \text{ if input} \neq 0\text{xF}$$
$$\mathbf{E}(0\text{xF}) = 0$$

The E^{-1} box is just the inverse of the E box where the roles of input and output are changed. The input/output values for boxes are also tabulated in Figure 12.17.

Figure 12.17 *SubBytes in the Whirlpool cipher*



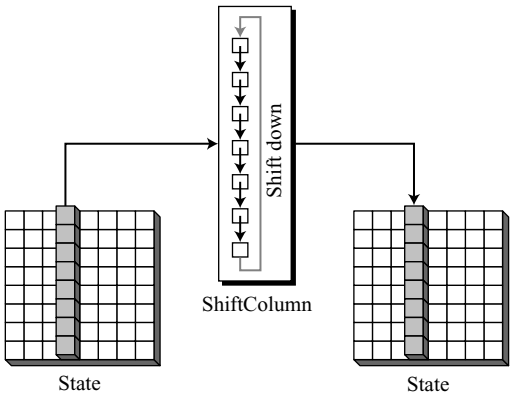
Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	E box
Output	1	B	9	C	D	6	F	3	E	8	7	4	A	2	5	0	

Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	E^{-1} box
Output	F	0	D	7	B	E	5	A	9	2	C	1	3	4	8	6	

Input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	R box
Output	7	C	B	D	E	4	9	F	6	3	8	A	2	5	1	0	

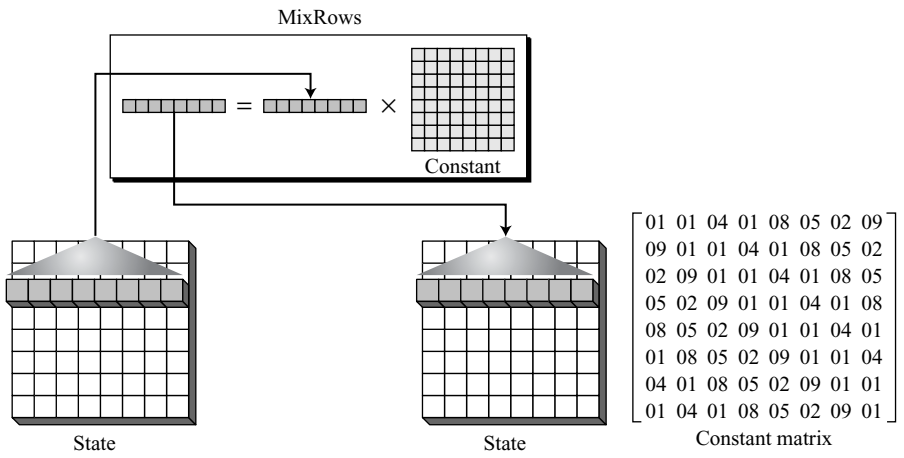
ShiftColumns To provide permutation, Whirlpool uses the **ShiftColumns** transformation, which is similar to the *ShiftRows* transformation in AES, except that the columns instead of rows are shifted. Shifting depends on the position of the column. Column 0 goes through 0-byte shifting (no shifting), while column 7 goes through 7-byte shifting. Figure 12.18 shows the shifting transformation.

Figure 12.18 *ShiftColumns transformation in the Whirlpool cipher*



MixRows The **MixRows** transformation has the same effect as the MixColumns transformation in AES: it diffuses the bits. The MixRows transformation is a matrix transformation where bytes are interpreted as 8-bit words (or polynomials) with coefficients in GF(2). Multiplication of bytes is done in GF(2^8), but the modulus is different from the one used in AES. The Whirlpool cipher uses (0x11D) or ($x^8 + x^4 + x^3 + x^2 + 1$) as the modulus. Addition is the same as XORing of 8-bit words. Figure 12.19 shows the MixRows transformation.

Figure 12.19 *MixRows transformation in the Whirlpool cipher*

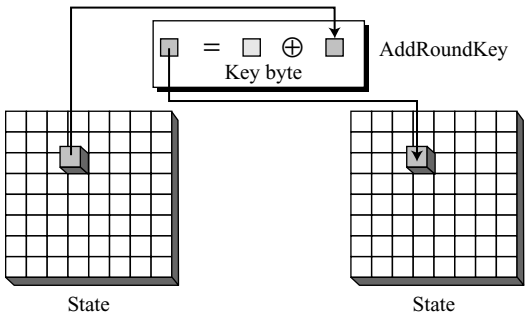


The figure shows multiplication of a single row by the constant matrix; the multiplication can actually be done by multiplying the whole state by the constant

matrix. Note that in the constant matrix, each row is the circular right shift of the previous row.

AddRoundKey The **AddRoundKey** transformation in the Whirlpool cipher is done byte by byte, because each round key is also a state of an 8×8 matrix. Figure 12.20 shows the process. A byte from the data state is added, in $GF(2^8)$ field, to the corresponding byte in the round-key state. The result is the new byte in the new state.

Figure 12.20 *AddRoundKey transformation in the Whirlpool cipher*



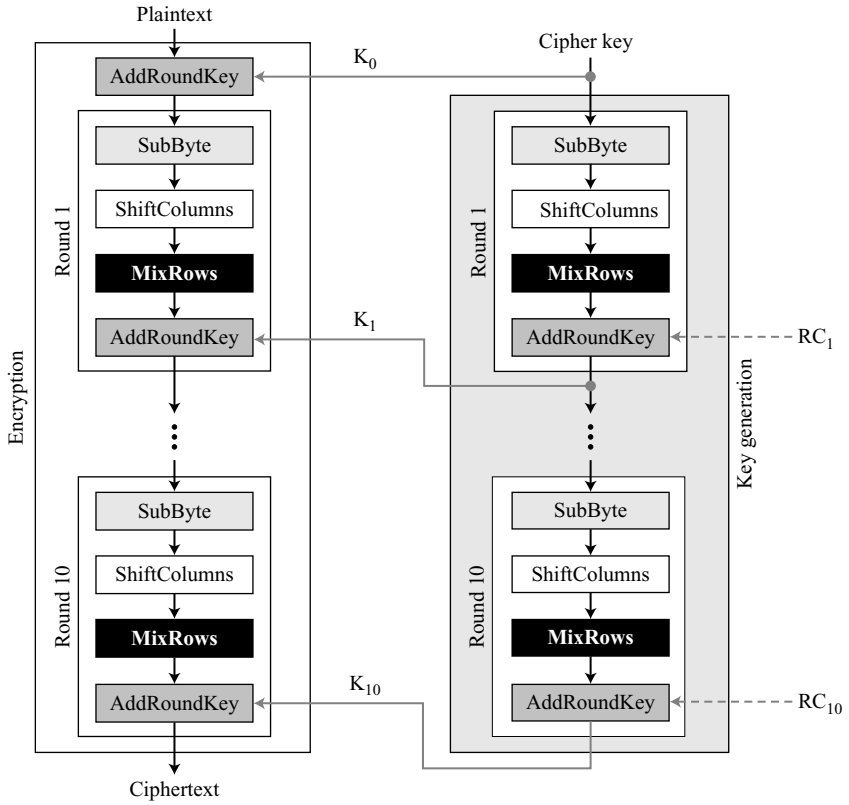
Key Expansion

As Figure 12.21 shows, the key-expansion algorithm in Whirlpool is totally different from the algorithm in AES. Instead of using a new algorithm for creating round keys, Whirlpool uses a copy of the encryption algorithm (without the pre-round) to create the round keys. The output of each round in the encryption algorithm is the round key for that round. At first glance, this looks like a circular definition; where do the round keys for the key expansion algorithm come from? Whirlpool has elegantly solved this problem by using ten round constants (RCs) as the virtual round keys for the key-expansion algorithm. In other words, the key-expansion algorithm uses constants as the round keys and the encryption algorithm uses the output of each round of the key-expansion algorithm as the round keys. The key-generation algorithm treats the cipher key as the *plaintext* and encrypts it. Note that the cipher key is also K_0 for the encryption algorithm.

Round Constants Each round constant, RC_r is an 8×8 matrix where only the first row has non-zero values. The rest of the entries are all 0's. The values for the first row in each constant matrix can be calculated using the SubBytes transformation (Table 12.4).

$$RC_{round}[row, column] = SubBytes\,(8(round - 1) + column) \qquad \text{if } row = 0$$
$$RC_{round}[row, column] = 0 \qquad \text{if } row \neq 0$$

Figure 12.21 *Key expansion in the Whirlpool cipher*



In other words, RC_1 uses the first eight entries in the SubBytes transformation table [Table 12.4]; RC_2 uses the second eight entries, and so on. For example, Figure 12.22 shows RC_3, where the first row is the third eight entries in the SubBytes table.

Figure 12.22 *Round constant for the third round*

$$RC_3 = \begin{bmatrix} \mathbf{1D} & \mathbf{E0} & \mathbf{D7} & \mathbf{C2} & \mathbf{2E} & \mathbf{4B} & \mathbf{FE} & \mathbf{57} \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

Summary

Table 12.5 summarizes some characteristics of the Whirlpool cipher.

Table 12.5 *Main characteristics of the Whirlpool cipher*

Block size: 512 bits
Cipher key size: 512 bits
Number of rounds: 10
Key expansion: using the cipher itself with round constants as round keys
Substitution: SubBytes transformation
Permutation: ShiftColumns transformation
Mixing: MixRows transformation
Round Constant: cubic roots of the first eighty prime numbers

Analysis

Although Whirlpool has not been extensively studied or tested, it is based on a robust scheme (Miyaguchi-Preneel), and for a compression function uses a cipher that is based on AES, a cryptosystem that has been proved very resistant to attacks. In addition, the size of the message digest is the same as for SHA-512. Therefore it is expected to be a very strong cryptographic hash function. However, more testing and researches are needed to confirm this. The only concern is that Whirlpool, which is based on a cipher as the compression function, may not be as efficient as SHA-512, particularly when it is implemented in hardware.

12.4 RECOMMENDED READING

For more details about subjects discussed in this chapter, we recommend the following books and websites. The items enclosed in brackets refer to the reference list at the end of the book.

Books

Several books give a good coverage of cryptographic hash functions, including [Sti06], [Sta06], [Sch99], [Mao04], [KPS02], [PHS03], and [MOV97].

WebSites

The following websites give more information about topics discussed in this chapter.

http://www.unixwiz.net/techtips/iguide-crypto-hashes.html
http://www.faqs.org/rfcs/rfc4231.html
http://www.itl.nist.gov/fipspubs/fip180-1.htm
http://www.ietf.org/rfc/rfc3174.txt
http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html

12.5 KEY TERMS

AddRoundKey
compression function
Davies-Meyer scheme
HAVAL
iterated cryptographic hash function
Matyas-Meyer-Oseas scheme
MD2
MD4
MD5
Merkle-Damgard scheme
Message Digest (MD)
MixRows
Miyaguchi-Preneel scheme
New European Schemes for Signatures, Integrity, and Encryption (NESSIE)
Rabin scheme
RACE Integrity Primitives Evaluation Message Digest (RIPMED)
RIPEMD-160
Secure Hash Algorithm (SHA)
Secure Hash Standard (SHS)
SHA-1
SHA-224
SHA-256
SHA-384
SHA-512
ShiftColumns
SubBytes
Whirlpool cipher
Whirlpool cryptographic hash function
word expansion

12.6 SUMMARY

- All cryptographic hash functions must create a fixed-size digest out of a variable-size message. Creating such a function is best accomplished using iteration. A compression function is repeatedly used to create the digest. The scheme is referred to as an iterated hash function.
- The Merkle-Damgard scheme is an iterated cryptographic hash function that is collision resistant if the compression function is collision resistant. The Merkle-Damgard scheme is the basis for many cryptographic hash functions today.
- There is a tendency to use two different approaches in designing the compression function. In the first approach, the compression function is made from scratch: it is particularly designed for this purpose. In the second approach, a symmetric-key block cipher serves instead of a compression function.
- A set of cryptographic hash functions uses compression functions that are made from scratch. These compression functions are specifically designed for the purpose they serve. Some examples are the Message Digest (MD) group, the Secure Hash Algorithm (SHA) group, RIPEMD, and HAVAL.
- An iterated cryptographic hash function can use a symmetric-key block cipher instead of a compression function. Several schemes for this approach have been proposed, including the Rabin scheme, Davies-Meyer scheme, Matyas-Meyer-Oseas scheme, and Miyaguchi-Preneel scheme.

- One of the promising cryptographic hash functions is SHA-512 with a 512-bit message digest based on the Merkle-Damgard scheme. It is made from scratch for this purpose.
- Another promising cryptographic hash function is Whirlpool, which is endorsed by NESSIE. Whirlpool is an iterated cryptographic hash function, based on the Miyaguchi-Preneel scheme, that uses a symmetric-key block cipher in place of the compression function. The block cipher is a modified AES cipher tailored for this purpose.

12.7 PRACTICE SET

Review Questions

1. Define a cryptographic hash function.
2. Define an iterated cryptographic hash function.
3. Describe the idea of the Merkle-Damgard scheme and why this idea is so important for the design of a cryptographic hash function.
4. List some family of hash functions that do not use a cipher as the compression function.
5. List some schemes that have been designed to use a block cipher as the compression function.
6. List the main features of the SHA-512 cryptographic hash function. What kind of compression function is used in SHA-512?
7. List some features of the Whirlpool cryptographic hash function. What kind of compression function is used in Whirlpool?
8. Compare and contrast features of SHA-512 and Whirlpool cryptographic hash functions.

Exercises

9. In SHA-512, show the value of the length field in hexadecimal for the following message lengths:
   a. 1000 bits
   b. 10,000 bits
   c. 1000,000 bits
10. In Whirlpool, show the value of the length field in hexadecimal for the following message lengths:
    a. 1000 bits
    b. 10,000 bits
    c. 1000,000 bits
11. What is the padding for SHA-512 if the length of the message is:
    a. 5120 bits
    b. 5121 bits
    c. 6143 bits

12. What is the padding for Whirlpool if the length of the message is:
    a. 5120 bits
    b. 5121 bits
    c. 6143 bits
13. In each of the following cases, show that if two messages are the same, their last blocks are also the same (after padding and adding the length field):
    a. The hash function is SHA-512.
    b. The hash function is Whirlpool.
14. Calculate G_0 in Table 12.2 using the seventh prime (17).
15. Compare the compression function of SHA-512 without the last operation (final adding) with a Feistel cipher of 80 rounds. Show the similarities and differences.
16. The compression function used in SHA-512 (Figure 12.10) can be thought of as an encrypting cipher with 80 rounds. If the words, W_0 to W_{79}, are thought of as round keys, which one of the schemes described in this chapter (Rabin, Davies-Meyer, Matyas-Meyer Oseas, or Miyaguchi-Preneel) does it resemble? Hint: Think about the effect of the *final adding* operation.
17. Show that SHA-512 is subject to meet-in-the middle attack if the *final adding* operation is removed from the compression function.
18. Make a table similar to Table 12.5 to compare AES and Whirlpool.
19. Show that the third operation does not need to be removed from the tenth round in Whirlpool cipher, but it must be removed in the AES cipher.
20. Find the result of $\text{RotR}_{12}(x)$ if
    x = 1234 5678 ABCD 2345 3456 5678 ABCD 2468
21. Find the result of $\text{ShL}_{12}(x)$ if
    x = 1234 5678 ABCD 2345 3456 5678 ABCD 2468
22. Find the result of Rotate(x) if
    x = 1234 5678 ABCD 2345 3456 5678 ABCD 2468
23. Find the result of Conditional (x, y, z) if
    x = 1234 5678 ABCD 2345 3456 5678 ABCD 2468
    y = 2234 5678 ABCD 2345 3456 5678 ABCD 2468
    x = 3234 5678 ABCD 2345 3456 5678 ABCD 2468
24. Find the result of Majority (x, y, z) if
    x = 1234 5678 ABCD 2345 3456 5678 ABCD 2468
    y = 2234 5678 ABCD 2345 3456 5678 ABCD 2468
    x = 3234 5678 ABCD 2345 3456 5678 ABCD 2468
25. Write a routine (in pseudocode) to calculate $\text{RotR}_i(x)$ in SHA-512 (Figure 12.9).
26. Write a routine (in pseudocode) to calculate $\text{ShL}_i(x)$ in SHA-512 (Figure 12.9).

27. Write a routine (in pseudocode) for the Conditional function in SHA-512 (Figure 12.11).
28. Write a routine (in pseudocode) for the Majority function in SHA-512 (Figure 12.11).
29. Write a routine (in pseudocode) for the Rotate function in SHA-512 (Figure 12.11).
30. Write a routine (in pseudocode) to calculate the initial digest (values of A_0 to H_0) in SHA-512 (Table 12.2).
31. Write a routine (in pseudocode) to calculate the eighty constants in SHA-512 (Table 12.3).
32. Write a routine (in pseudocode) for word-expansion algorithm in SHA-512 as shown in Figure 12.9. Use an array of 80 elements to hold all words.
33. Write a routine (in pseudocode) for the compression function in SHA-512.
34. Write a routine (in pseudocode) to change a block of 512 bits to an 8×8 state matrix (Figure 12.14).
35. Write a routine (in pseudocode) to change an 8×8 state matrix to a block of 512 bits (Figure 12.14).
36. Write a routine (in pseudocode) for the SubBytes transformation in the Whirlpool cipher (Figure 12.16).
37. Write a routine (in pseudocode) for the ShiftColumns transformation in the Whirlpool cipher (Figure 12.18).
38. Write a routine (in pseudocode) for the MixRows transformation in the Whirlpool cipher (Figure 12.19).
39. Write a routine (in pseudocode) for the AddRoundKey transformation in the Whirlpool cipher (Figure 12.20).
40. Write a routine (in pseudocode) for key expansion in Whirlpool cipher (Figure 12.21).
41. Write a routine (in pseudocode) to create the round constants in the Whirlpool cipher (Figure 12.20).
42. Write a routine (in pseudocode) for the Whirlpool cipher.
43. Write a routine (in pseudocode) for the Whirlpool cryptographic hash function.
44. Use the Internet (or other available resources) to find information about SHA-1. Then compare the compression function in SHA-1 with that in SHA-512. What are the similarities? What are the differences?
45. Use the Internet (or other available resources) to find information about the following compression functions, and compare them with SHA-512.
    a. SHA-224
    b. SHA-256
    c. SHA-384
46. Use the Internet (or other available resources) to find information about RIPEMD, and compare it with SHA-512.
47. Use the Internet (or other available resources) to find information about HAVAL, and compare it with SHA-512.