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.

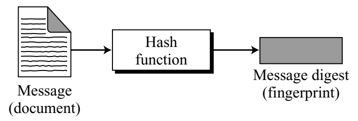
Document and Fingerprint

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*

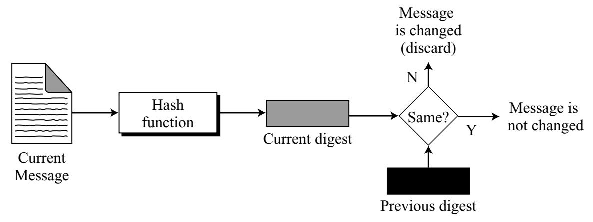
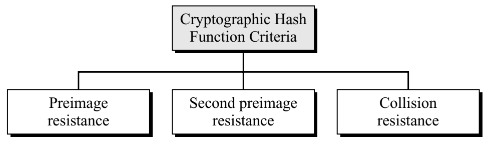


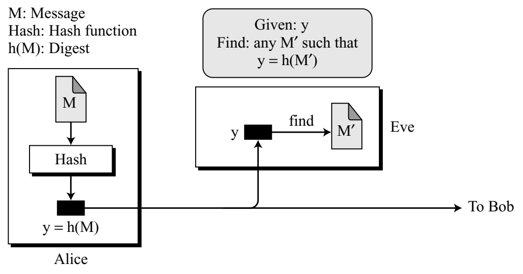
Figure 11.3 *Criteria of a cryptographic hash function*



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*

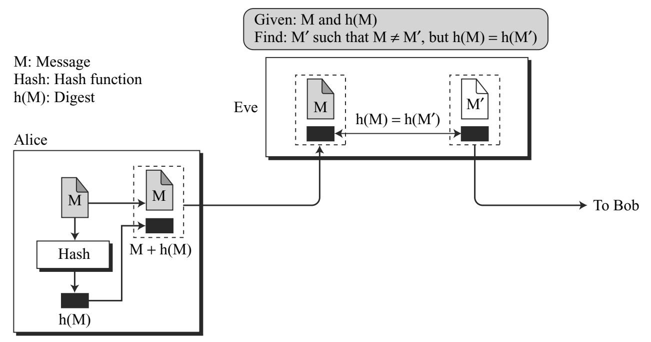
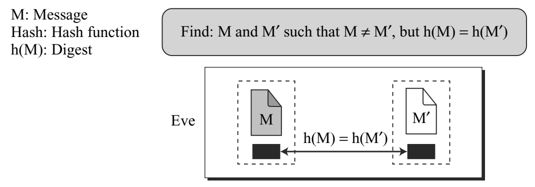


Figure 11.6 *Collision*



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 HHTHHHHTTHTHHTTTH, 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)] mod 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.

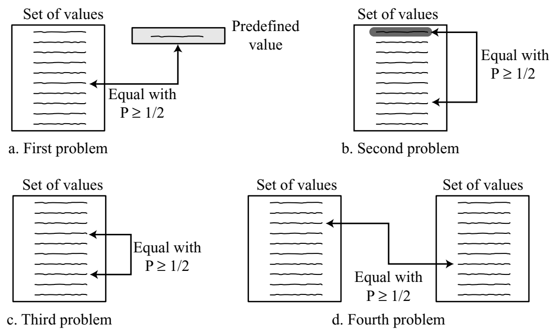
Example 11.5

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*

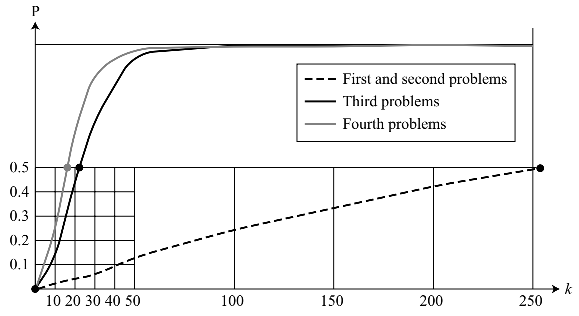
Problem	*Probability*	*General value for k*	*Value of k with P = 1/2*	*Number of students ($N = 365$)*
1	$P \approx 1 - e^{-k/N}$	$k \approx \ln[1/(1 - P)] \times N$	$k \approx 0.69 \times N$	253
2	$P \approx 1 - e^{-(k-1)/N}$	$k \approx \ln[1/(1 - P)] \times N + 1$	$k \approx 0.69 \times N + 1$	254
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 ($M_1, M_2, \ldots, M_k$) and k different variants of M′ ($M'_1, M'_2, \ldots, M'_k$). 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)*

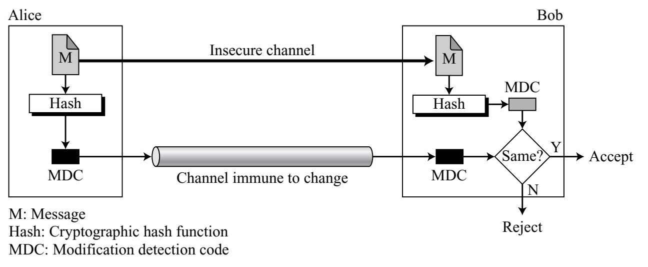


Figure 11.9 shows that the message can be transferred through an insecure channel. Eve can read or even modify the message. The MDC, however, needs to be transferred through a safe channel. The term *safe* here means immune to change. If both the

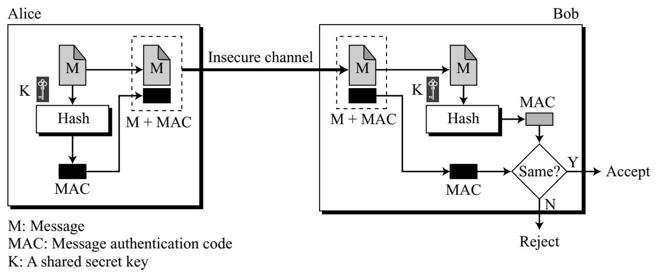
message and the MDC are sent through the insecure channel, Eve can intercept the message, change it, create a new MDC from the message, and send both to Bob. Bob never knows that the message has come from Eve. Note that the term *safe* can mean a trusted party; the term *channel* can mean the passage 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:

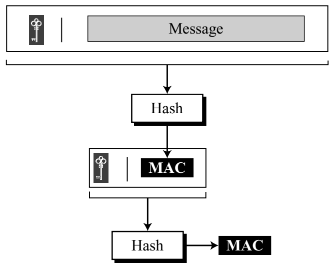
1. If the size of the key allows exhaustive search, Eve may prepend all possible keys at the beginning of the message and make a digest of the (K|M) to find the digest equal to the one intercepted. She then knows the key and can successfully replace the message with a forged message of her choosing.
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*

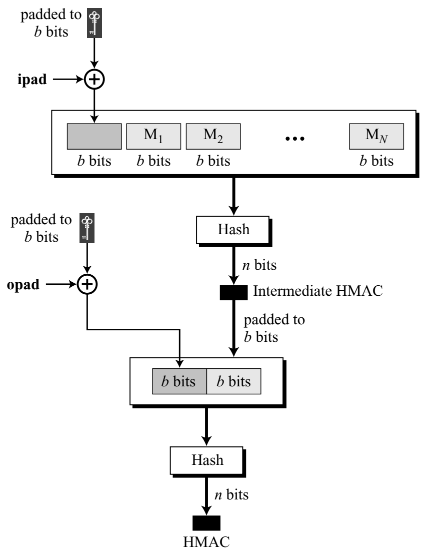


HMAC

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*

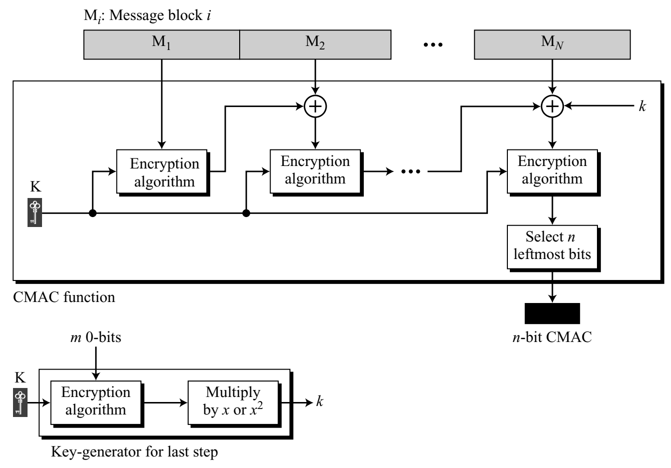


6. The intermediate n-bit HMAC is left padded with 0s to make a b-bit block.
7. Steps 2 and 3 are repeated by a different constant opad (**output pad**). The value of opad is the $b/8$ repetition of the sequence 01011100 (5C in hexadecimal).
8. The result of step 7 is prepended to the block of step 6.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