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.

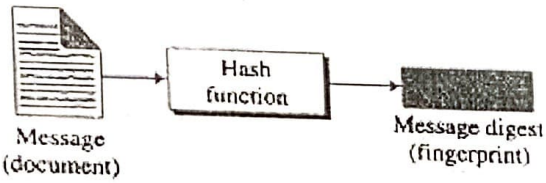
Document and Fingerprint

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.

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)*

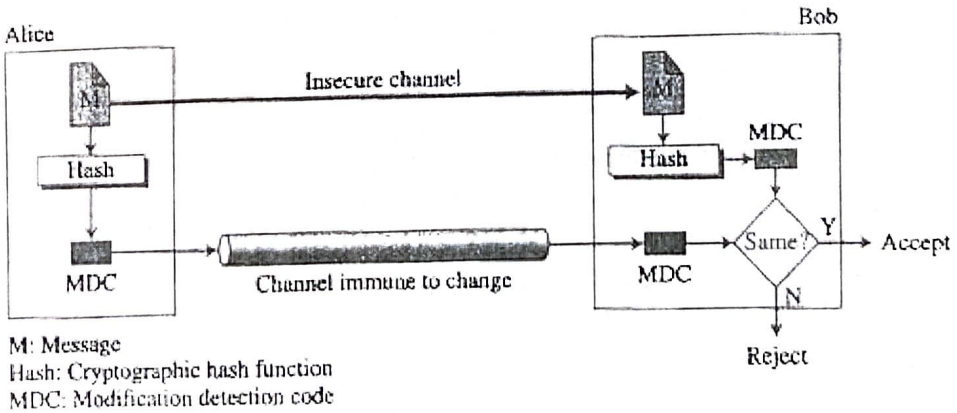


Figure 11.9 shows that the message can be transferred through an insecure channel. Eve can read or even modify the message. The MDC, however, needs to be transferred through a safe channel. The term *safe* here means immune to change. If both the

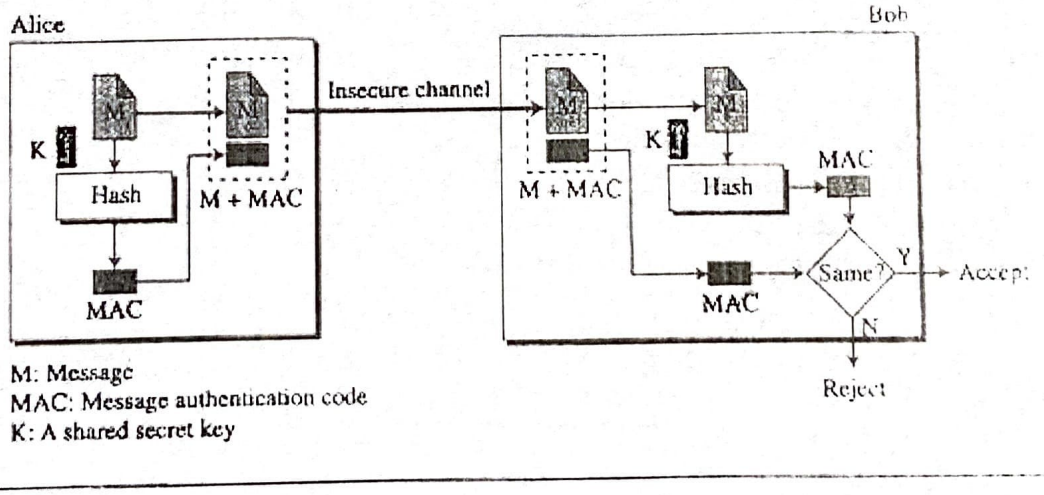
message and the MDC are sent through the insecure channel. Eve can intercept the message, change it, create a new MDC from the message, and send both to Bob. Bob never knows that the message has come from Eve. Note that the term *safe* can mean a trusted party; the term *channel* can mean the passage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

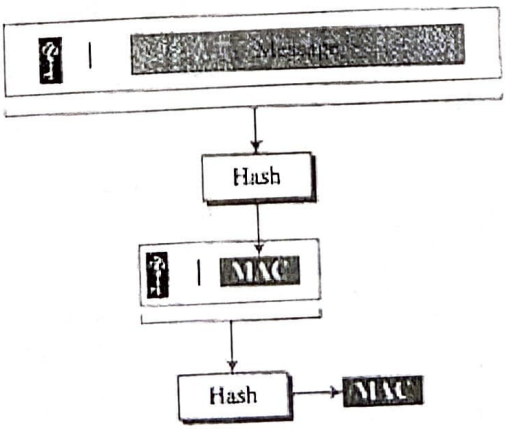
1. If the size of the key allows exhaustive search, Eve may prepend all possible keys at the beginning of the message and make a digest of the (K|M) to find the digest equal to the one intercepted. She then knows the key and can successfully replace the message with a forged message of her choosing.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

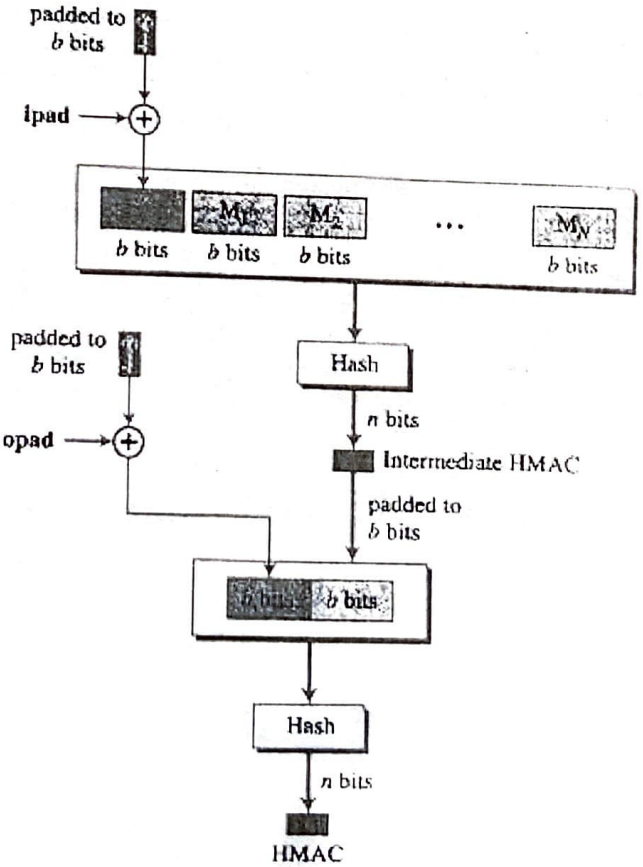


HMAC

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*

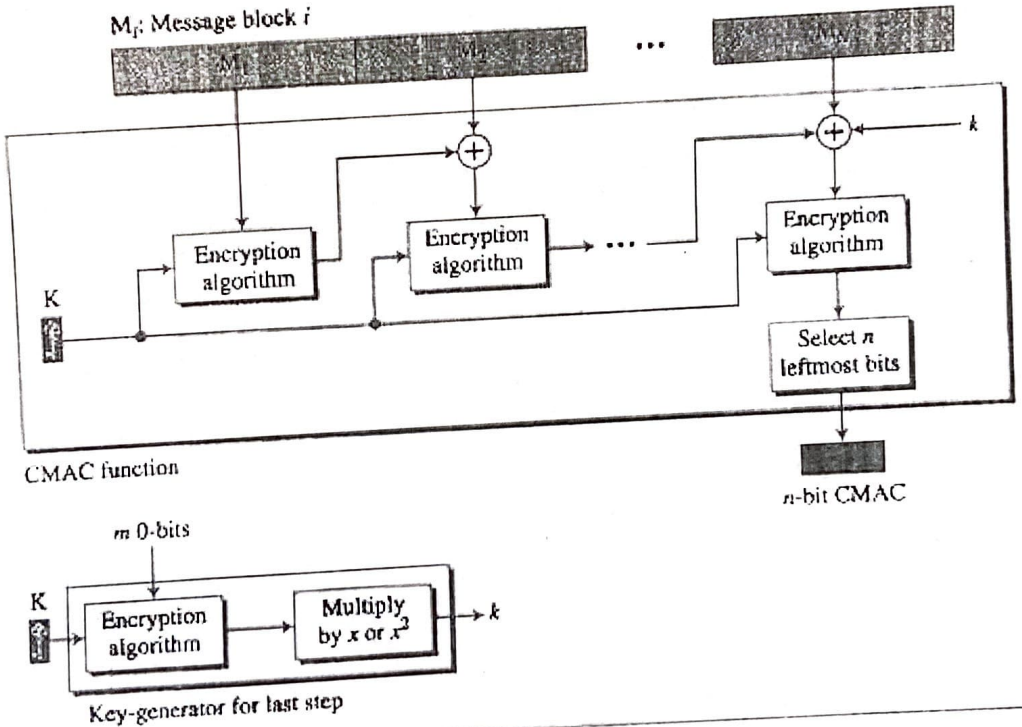


6. The intermediate n-bit HMAC is left padded with 0s to make a b-bit block.
7. Steps 2 and 3 are repeated by a different constant opad (**output pad**). The value of opad is the $b/8$ repetition of the sequence 01011100 (5C in hexadecimal).
8. The result of step 7 is prepended to the block of step 6.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