CHAPTER 12

Message Authentication Codes

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- List and explain the possible attacks that are relevant to message authentication.
- Define the term *message authentication code.*
- List and explain the requirements for a message authentication code.
- Present an overview of HMAC.
- Present an overview of CMAC.
- Explain the concept of authenticated encryption.
- Present an overview of CCM.
- Present an overview of GCM.
- Discuss the concept of key wrapping and explain its use.
- Understand how a hash function or a message authentication code can be used for pseudorandom number generation.

One of the most fascinating and complex areas of cryptography is that of message authentication and the related area of digital signatures. It would be impossible, in anything less than book length, to exhaust all the cryptographic functions and protocols that have been proposed or implemented for message authentication and digital signatures. Instead, the purpose of this chapter and the next is to provide a broad overview of the subject and to develop a systematic means of describing the various approaches.

This chapter begins with an introduction to the requirements for authentication and digital signature and the types of attacks to be countered. Then the basic approaches are surveyed. The remainder of the chapter deals with the fundamental approach to message authentication known as the message authentication code (MAC). Following an overview of this topic, the chapter looks at security considerations for MACs. This is followed by a discussion of specific MACs in two categories: those built from cryptographic hash functions and those built using a block cipher mode of operation. Next, we look at a relatively recent approach known as authenticated encryption. Finally, we look at the use of cryptographic hash functions and MACs for pseudorandom number generation.

12.1 MESSAGE AUTHENTICATION REQUIREMENTS

In the context of communications across a network, the following attacks can be identified.

1. **Disclosure:** Release of message contents to any person or process not possessing the appropriate cryptographic key.

2. **Traffic analysis:** Discovery of the pattern of traffic between parties. In a connection-oriented application, the frequency and duration of connections could be determined. In either a connection-oriented or connectionless environment, the number and length of messages between parties could be determined.
3. **Masquerade:** Insertion of messages into the network from a fraudulent source. This includes the creation of messages by an opponent that are purported to come from an authorized entity. Also included are fraudulent acknowledgments of message receipt or nonreceipt by someone other than the message recipient.
4. **Content modification:** Changes to the contents of a message, including insertion, deletion, transposition, and modification.
5. **Sequence modification:** Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
6. **Timing modification:** Delay or replay of messages. In a connection-oriented application, an entire session or sequence of messages could be a replay of some previous valid session, or individual messages in the sequence could be delayed or replayed. In a connectionless application, an individual message (e.g., datagram) could be delayed or replayed.
7. **Source repudiation:** Denial of transmission of message by source.
8. **Destination repudiation:** Denial of receipt of message by destination.

Measures to deal with the first two attacks are in the realm of message confidentiality and are dealt with in Part One. Measures to deal with items (3) through (6) in the foregoing list are generally regarded as message authentication. Mechanisms for dealing specifically with item (7) come under the heading of digital signatures. Generally, a digital signature technique will also counter some or all of the attacks listed under items (3) through (6). Dealing with item (8) may require a combination of the use of digital signatures and a protocol designed to counter this attack.

In summary, **message authentication** is a procedure to verify that received messages come from the alleged source and have not been altered. Message authentication may also verify sequencing and timeliness. A digital signature is an authentication technique that also includes measures to counter repudiation by the source.

12.2 MESSAGE AUTHENTICATION FUNCTIONS

Any message authentication or digital signature mechanism has two levels of functionality. At the lower level, there must be some sort of function that produces an **authenticator:** a value to be used to authenticate a message. This lower-level function is then used as a primitive in a higher-level authentication protocol that enables a receiver to verify the authenticity of a message.

This section is concerned with the types of functions that may be used to produce an authenticator. These may be grouped into three classes.

- **Hash function:** A function that maps a message of any length into a fixed-length hash value, which serves as the authenticator
- **Message encryption:** The ciphertext of the entire message serves as its authenticator
- **Message authentication code (MAC):** A function of the message and a secret key that produces a fixed-length value that serves as the authenticator

Hash functions, and how they may serve for message authentication, are discussed in Chapter 11. The remainder of this section briefly examines the remaining two topics. The remainder of the chapter elaborates on the topic of MACs.

Message Encryption

Message encryption by itself can provide a measure of authentication. The analysis differs for symmetric and public-key encryption schemes.

Symmetric Encryption Consider the straightforward use of symmetric encryption (Figure 12.1a). A message M transmitted from source A to destination B is encrypted using a secret key K shared by A and B. If no other party knows the key, then confidentiality is provided: No other party can recover the plaintext of the message.

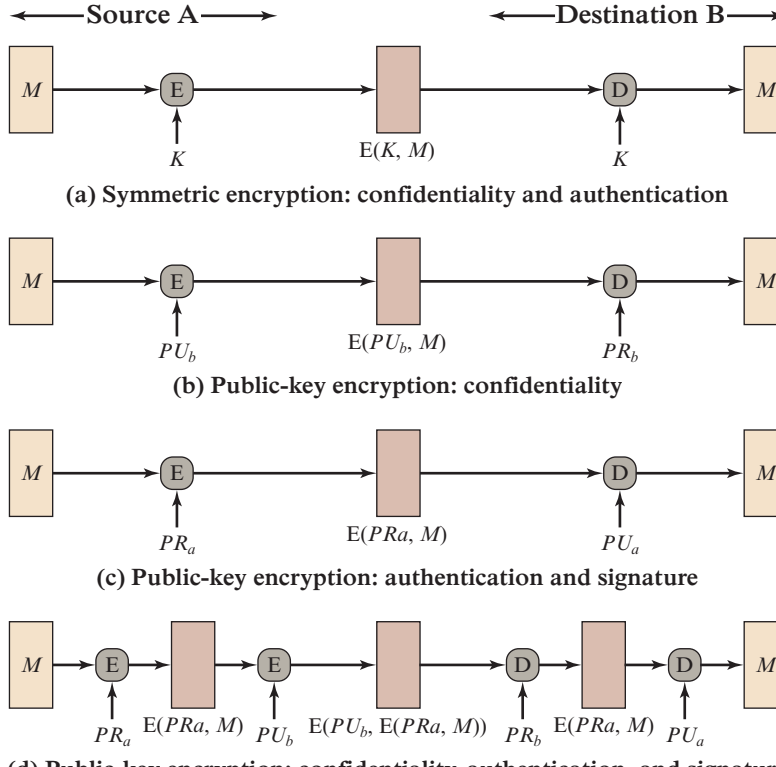


Figure 12.1 Basic Uses of Message Encryption

In addition, B is assured that the message was generated by A. Why? The message must have come from A, because A is the only other party that possesses K and therefore the only other party with the information necessary to construct ciphertext that can be decrypted with K. Furthermore, if M is recovered, B knows that none of the bits of M have been altered, because an opponent that does not know K would not know how to alter bits in the ciphertext to produce the desired changes in the plaintext.

So we may say that symmetric encryption provides authentication as well as confidentiality. However, this flat statement needs to be qualified. Consider exactly what is happening at B. Given a decryption function D and a secret key K, the destination will accept *any* input X and produce output $Y = D(K, X)$. If X is the ciphertext of a legitimate message M produced by the corresponding encryption function, then Y is some plaintext message M. Otherwise, Y will likely be a meaningless sequence of bits. There may need to be some automated means of determining at B whether Y is legitimate plaintext and therefore must have come from A.

The implications of the line of reasoning in the preceding paragraph are profound from the point of view of authentication. Suppose the message M can be any arbitrary bit pattern. In that case, there is no way to determine automatically, at the destination, whether an incoming message is the ciphertext of a legitimate message. This conclusion is incontrovertible: If M can be any bit pattern, then regardless of the value of X, the value $Y = D(K, X)$ is *some* bit pattern and therefore must be accepted as authentic plaintext.

Thus, in general, we require that only a small subset of all possible bit patterns be considered legitimate plaintext. In that case, any spurious ciphertext is unlikely to produce legitimate plaintext. For example, suppose that only one bit pattern in 10^6 is legitimate plaintext. Then the probability that any randomly chosen bit pattern, treated as ciphertext, will produce a legitimate plaintext message is only 10^{-6}.

For a number of applications and encryption schemes, the desired conditions prevail as a matter of course. For example, suppose that we are transmitting English-language messages using a Caesar cipher with a shift of one ($K = 1$). A sends the following legitimate ciphertext:

```
nbsftfbupbutboeepftfbupbutboemjuumfmbnctfbujwz
```

B decrypts to produce the following plaintext:

```
mareseatoatsanddoeseatoatsandlittlelambseativy
```

A simple frequency analysis confirms that this message has the profile of ordinary English. On the other hand, if an opponent generates the following random sequence of letters:

```
zuvrsoevgqxlzwigamdvnmhpmccxiuureosfbcebtqxsxq
```

this decrypts to

```
ytuqrndufpwkyvhfzlcumlgolbbwhttqdnreabdaspwrwp
```

which does not fit the profile of ordinary English.

It may be difficult to determine *automatically* if incoming ciphertext decrypts to intelligible plaintext. If the plaintext is, say, a binary object file or digitized X-rays, determination of properly formed and therefore authentic plaintext may be difficult. Thus, an opponent could achieve a certain level of disruption simply by issuing messages with random content purporting to come from a legitimate user.

One solution to this problem is to force the plaintext to have some structure that is easily recognized but that cannot be replicated without recourse to the encryption function. We could, for example, append an error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption, as illustrated in Figure 12.2a. A prepares a plaintext message M and then provides this as input to a function F that produces an FCS. The FCS is appended to M and the entire block is then encrypted. At the destination, B decrypts the incoming block and treats the results as a message with an appended FCS. B applies the same function F to attempt to reproduce the FCS. If the calculated FCS is equal to the incoming FCS, then the message is considered authentic. It is unlikely that any random sequence of bits would exhibit the desired relationship.

Note that the order in which the FCS and encryption functions are performed is critical. The sequence illustrated in Figure 12.2a is referred to in [DIFF79] as **internal error control**, which the authors contrast with **external error control** (Figure 12.2b). With internal error control, authentication is provided because an opponent would have difficulty generating ciphertext that, when decrypted, would have valid error control bits. If instead the FCS is the outer code, an opponent can construct messages with valid error-control codes. Although the opponent cannot know what the decrypted plaintext will be, he or she can still hope to create confusion and disrupt operations.

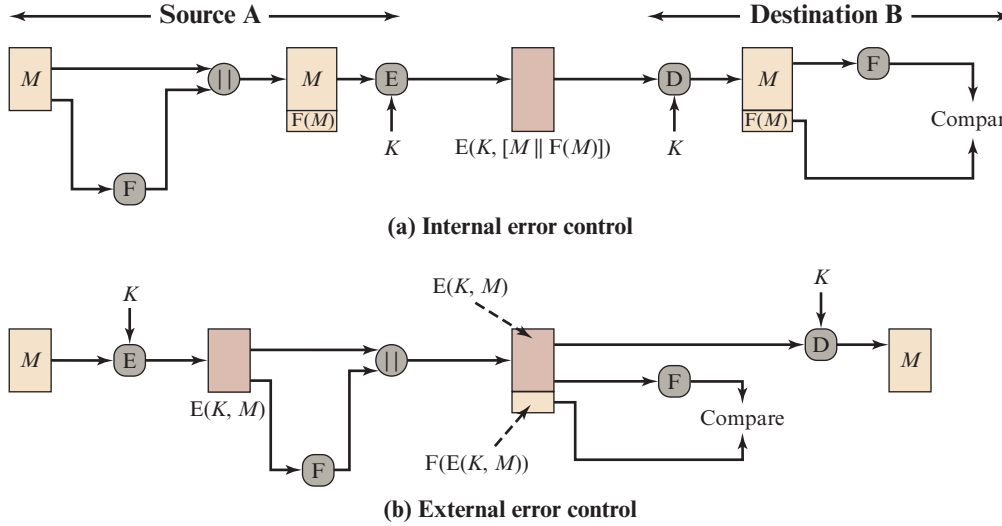


Figure 12.2 Internal and External Error Control

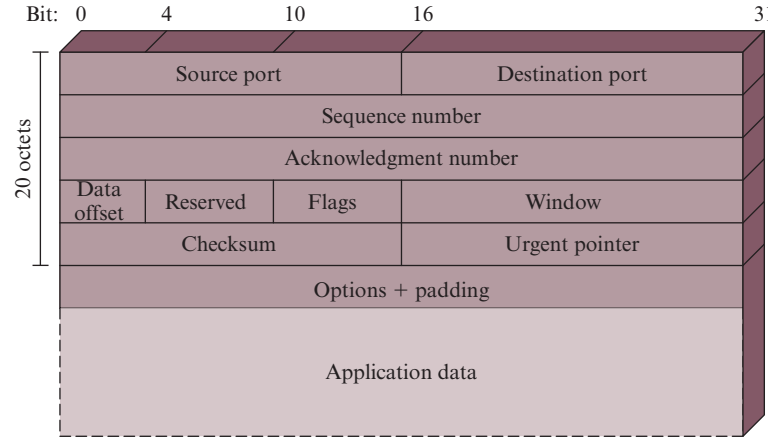


Figure 12.3 TCP Segment

An error-control code is just one example; in fact, any sort of structuring added to the transmitted message serves to strengthen the authentication capability. Such structure is provided by the use of a communications architecture consisting of layered protocols. As an example, consider the structure of messages transmitted using the TCP/IP protocol architecture. Figure 12.3 shows the format of a TCP segment, illustrating the TCP header. Now suppose that each pair of hosts shared a unique secret key, so that all exchanges between a pair of hosts used the same key, regardless of application. Then we could simply encrypt all of the datagram except the IP header. Again, if an opponent substituted some arbitrary bit pattern for the encrypted TCP segment, the resulting plaintext would not include a meaningful header. In this case, the header includes not only a checksum (which covers the header) but also other useful information, such as the sequence number. Because successive TCP segments on a given connection are numbered sequentially, encryption assures that an opponent does not delay, misorder, or delete any segments.

Public-Key Encryption The straightforward use of public-key encryption (Figure 12.1b) provides confidentiality but not authentication. The source (A) uses the public key PU_b of the destination (B) to encrypt M. Because only B has the corresponding private key PR_b, only B can decrypt the message. This scheme provides no authentication, because any opponent could also use B's public key to encrypt a message and claim to be A.

To provide authentication, A uses its private key to encrypt the message, and B uses A's public key to decrypt (Figure 12.1c). This provides authentication using the same type of reasoning as in the symmetric encryption case: The message must have come from A because A is the only party that possesses PR_a and therefore the only party with the information necessary to construct ciphertext that can be decrypted with PU_a. Again, the same reasoning as before applies: There must be some internal structure to the plaintext so that the receiver can distinguish between well-formed plaintext and random bits.

Assuming there is such structure, then the scheme of Figure 12.1c does provide authentication. It also provides what is known as digital signature.[1] Only A could have constructed the ciphertext because only A possesses PR_a. Not even B, the recipient, could have constructed the ciphertext. Therefore, if B is in possession of the ciphertext, B has the means to prove that the message must have come from A. In effect, A has "signed" the message by using its private key to encrypt. Note that this scheme does not provide confidentiality. Anyone in possession of A's public key can decrypt the ciphertext.

To provide both confidentiality and authentication, A can encrypt M first using its private key, which provides the digital signature, and then using B's public key, which provides confidentiality (Figure 12.1d). The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.

Message Authentication Code

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as a **cryptographic checksum** or MAC, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key K. When A has a message to send to B, it calculates the MAC as a function of the message and the key:

$$\text{MAC} = \text{C}(K, M)$$

where

M = input message
C = MAC function
K = shared secret key
MAC = message authentication code

The message plus MAC are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new MAC. The received MAC is compared to the calculated MAC (Figure 12.4a). If we assume that only the receiver and the sender know the identity of the secret key, and if the received MAC matches the calculated MAC, then

1. The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the MAC, then the receiver's calculation of the MAC will differ from the received MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to correspond to the alterations in the message.
2. The receiver is assured that the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper MAC.

[1]This is not the way in which digital signatures are constructed, as we shall see, but the principle is the same.

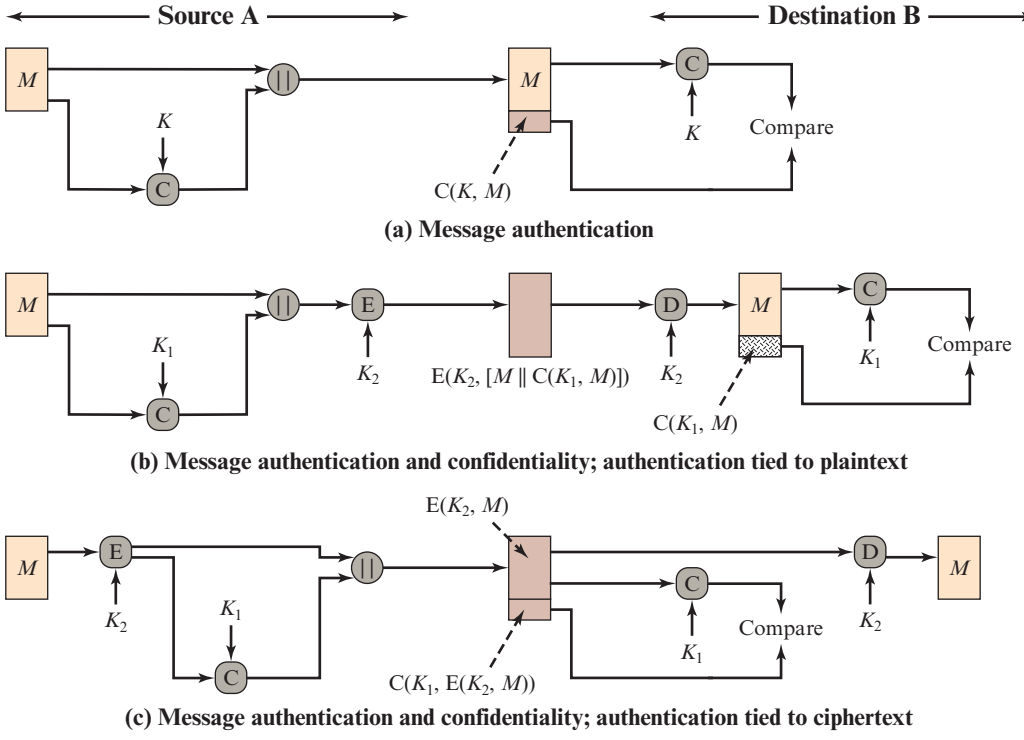


Figure 12.4 Basic Uses of Message Authentication code (MAC)

3. If the message includes a sequence number (such as is used with HDLC, X.25, and TCP), then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

A MAC function is similar to encryption. One difference is that the MAC algorithm need not be reversible, as it must be for decryption. In general, the MAC function is a many-to-one function. The domain of the function consists of messages of some arbitrary length, whereas the range consists of all possible MACs and all possible keys. If an n-bit MAC is used, then there are 2^n possible MACs, whereas there are N possible messages with $N \gg 2^n$. Furthermore, with a k-bit key, there are 2^k possible keys.

For example, suppose that we are using 100-bit messages and a 10-bit MAC. Then, there are a total of 2^{100} different messages but only 2^{10} different MACs. So, on average, each MAC value is generated by a total of $2^{100}/2^{10} = 2^{90}$ different messages. If a 5-bit key is used, then there are $2^5 = 32$ different mappings from the set of messages to the set of MAC values.

It turns out that, because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption.

The process depicted in Figure 12.4a provides authentication but not confidentiality, because the message as a whole is transmitted in the clear. Confidentiality can be provided by performing message encryption either after (Figure 12.4b) or before (Figure 12.4c) the MAC algorithm. In both these cases, two separate keys are needed,

each of which is shared by the sender and the receiver. In the first case, the MAC is calculated with the message as input and is then concatenated to the message. The entire block is then encrypted. In the second case, the message is encrypted first. Then the MAC is calculated using the resulting ciphertext and is concatenated to the ciphertext to form the transmitted block. Typically, it is preferable to tie the authentication directly to the plaintext, so the method of Figure 12.4b is used.

Because symmetric encryption will provide authentication and because it is widely used with readily available products, why not simply use this instead of a separate message authentication code? [DAVI89] suggests three situations in which a message authentication code is used.

1. There are a number of applications in which the same message is broadcast to a number of destinations. Examples are notification to users that the network is now unavailable or an alarm signal in a military control center. It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication code. The responsible system has the secret key and performs authentication. If a violation occurs, the other destination systems are alerted by a general alarm.
2. Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages. Authentication is carried out on a selective basis, messages being chosen at random for checking.
3. Authentication of a computer program in plaintext is an attractive service. The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources. However, if a message authentication code were attached to the program, it could be checked whenever assurance was required of the integrity of the program.

Three other rationales may be added.

4. For some applications, it may not be of concern to keep messages secret, but it is important to authenticate messages. An example is the Simple Network Management Protocol Version 3 (SNMPv3), which separates the functions of confidentiality and authentication. For this application, it is usually important for a managed system to authenticate incoming SNMP messages, particularly if the message contains a command to change parameters at the managed system. On the other hand, it may not be necessary to conceal the SNMP traffic.
5. Separation of authentication and confidentiality functions affords architectural flexibility. For example, it may be desired to perform authentication at the application level but to provide confidentiality at a lower level, such as the transport layer.
6. A user may wish to prolong the period of protection beyond the time of reception and yet allow processing of message contents. With message encryption, the protection is lost when the message is decrypted, so the message is protected against fraudulent modifications only in transit but not within the target system.

Finally, note that the MAC does not provide a digital signature, because both sender and receiver share the same key.

12.3 REQUIREMENTS FOR MESSAGE AUTHENTICATION CODES

A MAC, also known as a cryptographic checksum, is generated by a function MAC of the form

$$T = \text{MAC}(K, M)$$

where M is a variable-length message, K is a secret key shared only by sender and receiver, and MAC(K, M) is the fixed-length authenticator, sometimes called a **tag**. The tag is appended to the message at the source at a time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the tag.

When an entire message is encrypted for confidentiality, using either symmetric or asymmetric encryption, the security of the scheme generally depends on the bit length of the key. Barring some weakness in the algorithm, the opponent must resort to a brute-force attack using all possible keys. On average, such an attack will require $2^{(k-1)}$ attempts for a k-bit key. In particular, for a ciphertext-only attack, the opponent, given ciphertext C, performs $P_i = \text{D}(K_i, C)$ for all possible key values K_i until a P_i is produced that matches the form of acceptable plaintext.

In the case of a MAC, the considerations are entirely different. In general, the MAC function is a many-to-one function, due to the many-to-one nature of the function. Using brute-force methods, how would an opponent attempt to discover a key? If confidentiality is not employed, the opponent has access to plaintext messages and their associated MACs. Suppose $k > n$; that is, suppose that the key size is greater than the MAC size. Then, given a known M_1 and T_1, with $T_1 = \text{MAC}(K, M_1)$, the cryptanalyst can perform $T_i = \text{MAC}(K_i, M_1)$ for all possible key values k_i. At least one key is guaranteed to produce a match of $T_i = T_1$. Note that a total of 2^k tags will be produced, but there are only $2^n < 2^k$ different tag values. Thus, a number of keys will produce the correct tag and the opponent has no way of knowing which is the correct key. On average, a total of $2^k/2^n = 2^{(k-n)}$ keys will produce a match. Thus, the opponent must iterate the attack.

- **Round 1**

  Given: M_1, $T_1 = \text{MAC}(K, M_1)$
  Compute $T_i = \text{MAC}(K_i, M_1)$ for all 2^k keys
  Number of matches $\approx 2^{(k-n)}$

- **Round 2**

  Given: M_2, $T_2 = \text{MAC}(K, M_2)$
  Compute $T_i = \text{MAC}(K_i, M_2)$ for the $2^{(k-n)}$ keys resulting from Round 1
  Number of matches $\approx 2^{(k-2\times n)}$

And so on. On average, α rounds will be needed $k = \alpha \times n$. For example, if an 80-bit key is used and the tag is 32 bits, then the first round will produce about 2^{48} possible keys. The second round will narrow the possible keys to about 2^{16} possibilities. The third round should produce only a single key, which must be the one used by the sender.

If the key length is less than or equal to the tag length, then it is likely that a first round will produce a single match. It is possible that more than one key will produce such a match, in which case the opponent would need to perform the same test on a new (message, tag) pair.

Thus, a brute-force attempt to discover the authentication key is no less effort and may be more effort than that required to discover a decryption key of the same length. However, other attacks that do not require the discovery of the key are possible.

Consider the following MAC algorithm. Let $M = (X_1 \| X_2 \| \ldots \| X_m)$ be a message that is treated as a concatenation of 64-bit blocks X_i. Then define

$$\Delta(M) = X_1 \oplus X_2 \oplus \cdots \oplus X_m$$
$$\text{MAC}(K, M) = \text{E}(K, \Delta(M))$$

where $\oplus$ is the exclusive-OR (XOR) operation and the encryption algorithm is DES in electronic codebook mode. Thus, the key length is 56 bits, and the tag length is 64 bits. If an opponent observes $\{M \| \text{MAC}(K, M)\}$, a brute-force attempt to determine K will require at least 2^{56} encryptions. But the opponent can attack the system by replacing X_1 through X_{m-1} with any desired values Y_1 through Y_{m-1} and replacing X_m with Y_m, where Y_m is calculated as

$$Y_m = Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{m-1} \oplus \Delta(M)$$

The opponent can now concatenate the new message, which consists of Y_1 through Y_m, using the original tag to form a message that will be accepted as authentic by the receiver. With this tactic, any message of length $64 \times (m - 1)$ bits can be fraudulently inserted.

Thus, in assessing the security of a MAC function, we need to consider the types of attacks that may be mounted against it. With that in mind, let us state the requirements for the function. Assume that an opponent knows the MAC function but does not know K. Then the MAC function should satisfy the following requirements.

1. If an opponent observes M and MAC(K, M), it should be computationally infeasible for the opponent to construct a message M' such that
$$\text{MAC}(K, M') = \text{MAC}(K, M)$$
2. MAC(K, M) should be uniformly distributed in the sense that for randomly chosen messages, M and M', the probability that $\text{MAC}(K, M) = \text{MAC}(K, M')$ is 2^{-n}, where n is the number of bits in the tag.
3. Let M' be equal to some known transformation on M. That is, $M' = \text{f}(M)$. For example, f may involve inverting one or more specific bits. In that case,
$$\Pr[\text{MAC}(K, M) = \text{MAC}(K, M')] = 2^{-n}$$

The first requirement speaks to the earlier example, in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key. The second requirement deals with the need to thwart a brute-force attack based on chosen plaintext. That is, if we assume

that the opponent does not know K but does have access to the MAC function and can present messages for MAC generation, then the opponent could try various messages until finding one that matches a given tag. If the MAC function exhibits uniform distribution, then a brute-force method would require, on average, $2^{(n-1)}$ attempts before finding a message that fits a given tag.

The final requirement dictates that the authentication algorithm should not be weaker with respect to certain parts or bits of the message than others. If this were not the case, then an opponent who had M and MAC(K, M) could attempt variations on M at the known "weak spots" with a likelihood of early success at producing a new message that matched the old tags.

12.4 SECURITY OF MACs

Just as with encryption algorithms and hash functions, we can group attacks on MACs into two categories: brute-force attacks and cryptanalysis.

Brute-Force Attacks

A brute-force attack on a MAC is a more difficult undertaking than a brute-force attack on a hash function because it requires known message-tag pairs. Let us see why this is so. To attack a hash code, we can proceed in the following way. Given a fixed message x with n-bit hash code $h = \mathrm{H}(x)$, a brute-force method of finding a collision is to pick a random bit string y and check if $\mathrm{H}(y) = \mathrm{H}(x)$. The attacker can do this repeatedly off line. Whether an off-line attack can be used on a MAC algorithm depends on the relative size of the key and the tag.

To proceed, we need to state the desired security property of a MAC algorithm, which can be expressed as follows.

- **Computation resistance:** Given one or more text-MAC pairs $[x_i, \mathrm{MAC}(K, x_i)]$, it is computationally infeasible to compute any text-MAC pair $[x, \mathrm{MAC}(K, x)]$ for any new input $x \neq x_i$.

In other words, the attacker would like to come up with the valid MAC code for a given message x. There are two lines of attack possible: attack the key space and attack the MAC value. We examine each of these in turn.

If an attacker can determine the MAC key, then it is possible to generate a valid MAC value for any input x. Suppose the key size is k bits and that the attacker has one known text-tag pair. Then the attacker can compute the n-bit tag on the known text for all possible keys. At least one key is guaranteed to produce the correct tag, namely, the valid key that was initially used to produce the known text-tag pair. This phase of the attack takes a level of effort proportional to 2^k (that is, one operation for each of the 2^k possible key values). However, as was described earlier, because the MAC is a many-to-one mapping, there may be other keys that produce the correct value. Thus, if more than one key is found to produce the correct value, additional text-tag pairs must be tested. It can be shown that the level of effort drops off rapidly with each additional text-MAC pair and that the overall level of effort is roughly 2^k [MENE97].

An attacker can also work on the tag without attempting to recover the key. Here, the objective is to generate a valid tag for a given message or to find a message that matches a given tag. In either case, the level of effort is comparable to that for attacking the one-way or weak collision-resistant property of a hash code, or 2^n. In the case of the MAC, the attack cannot be conducted off line without further input; the attacker will require chosen text-tag pairs or knowledge of the key.

To summarize, the level of effort for brute-force attack on a MAC algorithm can be expressed as $\min(2^k, 2^n)$. The assessment of strength is similar to that for symmetric encryption algorithms. It would appear reasonable to require that the key length and tag length satisfy a relationship such as $\min(k, n) \geq N$, where N is perhaps in the range of 128 bits.

Cryptanalysis

There is much more variety in the structure of MACs than in hash functions, so it is difficult to generalize about the cryptanalysis of MACs. As with encryption algorithms and hash functions, cryptanalytic attacks on MAC algorithms seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a MAC algorithm to cryptanalysis is to compare its strength to the effort required for a brute-force attack. That is, an ideal MAC algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.

12.5 MACs BASED ON HASH FUNCTIONS: HMAC

Later in this chapter, we look at examples of a MAC based on the use of a symmetric block cipher. This has traditionally been the most common approach to constructing a MAC. In recent years, there has been increased interest in developing a MAC derived from a **cryptographic hash function**. The motivations for this interest are

1. Cryptographic hash functions such as MD5 and SHA generally execute faster in software than symmetric block ciphers such as DES.
2. Library code for cryptographic hash functions is widely available.

With the development of AES and the more widespread availability of code for encryption algorithms, these considerations are less significant, but hash-based MACs continue to be widely used.

A hash function such as SHA was not designed for use as a MAC and cannot be used directly for that purpose, because it does not rely on a secret key. There have been a number of proposals for the incorporation of a secret key into an existing hash algorithm. The approach that has received the most support is HMAC [BELL96a, BELL96b]. HMAC has been issued as RFC 2104, has been chosen as the mandatory-to-implement MAC for IP security, and is used in other Internet protocols, such as SSL. HMAC has also been issued as a NIST standard (FIPS 198).

HMAC Design Objectives

RFC 2104 lists the following design objectives for HMAC.

- To use, without modifications, available hash functions. In particular, to use hash functions that perform well in software and for which code is freely and widely available.
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required.
- To preserve the original performance of the hash function without incurring a significant degradation.
- To use and handle keys in a simple way.
- To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions about the embedded hash function.

The first two objectives are important to the acceptability of HMAC. HMAC treats the hash function as a "black box." This has two benefits. First, an existing implementation of a hash function can be used as a module in implementing HMAC. In this way, the bulk of the HMAC code is prepackaged and ready to use without modification. Second, if it is ever desired to replace a given hash function in an HMAC implementation, all that is required is to remove the existing hash function module and drop in the new module. This could be done if a faster hash function were desired. More important, if the security of the embedded hash function were compromised, the security of HMAC could be retained simply by replacing the embedded hash function with a more secure one (e.g., replacing SHA-2 with SHA-3).

The last design objective in the preceding list is, in fact, the main advantage of HMAC over other proposed hash-based schemes. HMAC can be proven secure provided that the embedded hash function has some reasonable cryptographic strengths. We return to this point later in this section, but first we examine the structure of HMAC.

HMAC Algorithm

Figure 12.5 illustrates the overall operation of HMAC. Define the following terms.

H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)
IV = initial value input to hash function
M = message input to HMAC (including the padding specified in the embedded hash function)
Y_i = i th block of M, $0 \leq i \leq (L - 1)$
L = number of blocks in M
b = number of bits in a block
n = length of hash code produced by embedded hash function
K = secret key; recommended length is $\geq n$; if key length is greater than b, the key is input to the hash function to produce an n-bit key
K^+ = K padded with zeros on the right so that the result is b bits in length

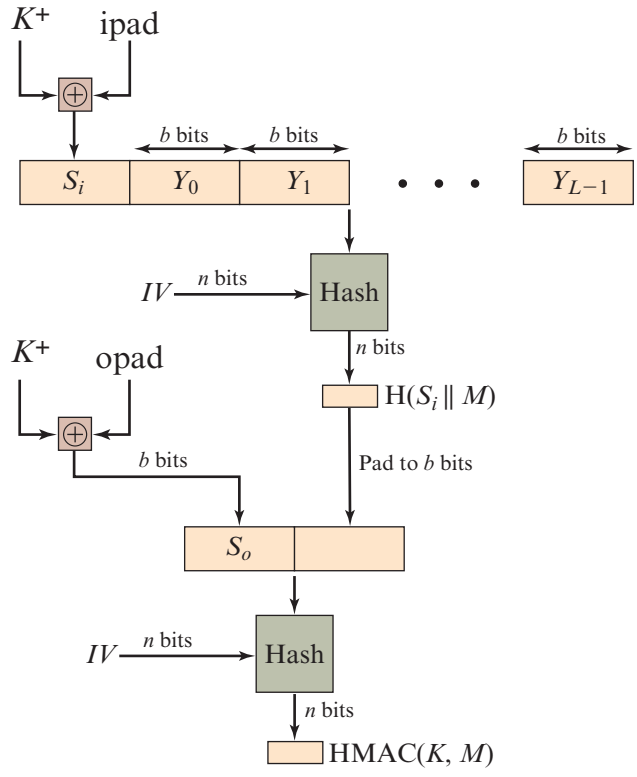


Figure 12.5 HMAC Structure

ipad = 00110110 (36 in hexadecimal) repeated $b/8$ times
opad = 01011100 (5C in hexadecimal) repeated $b/8$ times

Then HMAC can be expressed as

$$\text{HMAC}(K, M) = \text{H}[(K^+ \oplus \text{opad}) \| \text{H}[(K^+ \oplus \text{ipad}) \| M]]$$

We can describe the algorithm as follows.

1. Append zeros to the left end of K to create a b-bit string K^+ (e.g., if K is of length 160 bits and $b = 512$, then K will be appended with 44 zeroes).
2. XOR (bitwise exclusive-OR) K^+ with ipad to produce the b-bit block S_i.
3. Append M to S_i.
4. Apply H to the stream generated in step 3.
5. XOR K^+ with opad to produce the b-bit block S_o.
6. Append the hash result from step 4 to S_o.
7. Apply H to the stream generated in step 6 and output the result.

Note that the XOR with ipad results in flipping one-half of the bits of K. Similarly, the XOR with opad results in flipping one-half of the bits of K, using a

different set of bits. In effect, by passing S_i and S_o through the compression function of the hash algorithm, we have pseudorandomly generated two keys from K.

HMAC should execute in approximately the same time as the embedded hash function for long messages. HMAC adds three executions of the hash compression function (for S_i, S_o, and the block produced from the inner hash).

A more efficient implementation is possible, as shown in Figure 12.6. Two quantities are precomputed:

$$\text{f}(IV, (K^+ \oplus \text{ipad}))$$
$$\text{f}(IV, (K^+ \oplus \text{opad}))$$

where f(cv, block) is the compression function for the hash function, which takes as arguments a chaining variable of n bits and a block of b bits and produces a chaining variable of n bits. These quantities only need to be computed initially and every time the key changes. In effect, the precomputed quantities substitute for the initial value (IV) in the hash function. With this implementation, only one additional instance of the compression function is added to the processing normally produced by the hash

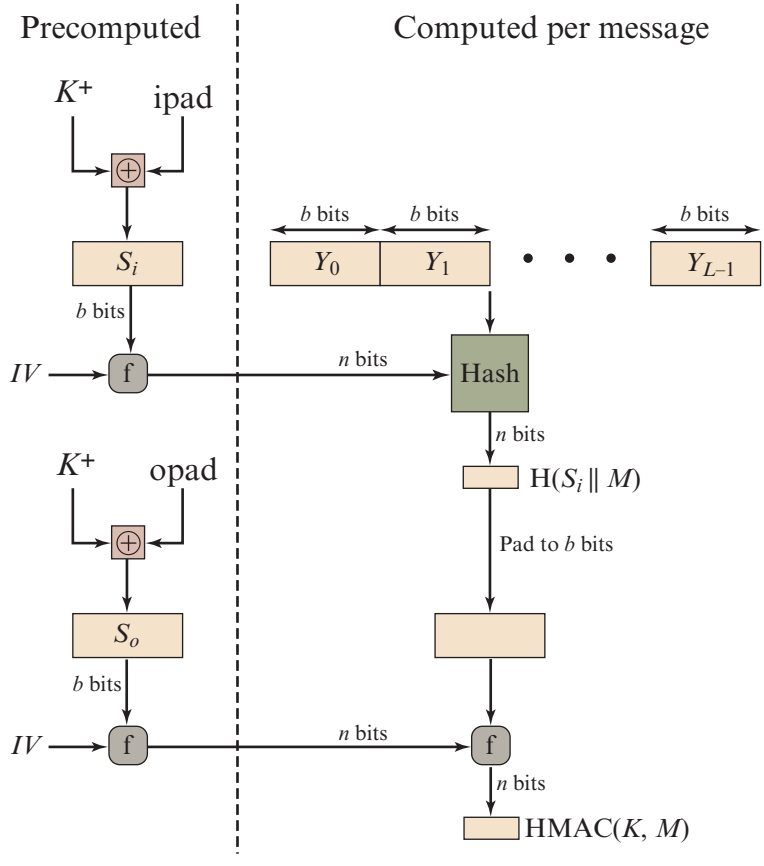


Figure 12.6 Efficient Implementation of HMAC

function. This more efficient implementation is especially worthwhile if most of the messages for which a MAC is computed are short.

Security of HMAC

The security of any MAC function based on an embedded hash function depends in some way on the cryptographic strength of the underlying hash function. The appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC.

The security of a MAC function is generally expressed in terms of the probability of successful forgery with a given amount of time spent by the forger and a given number of message-tag pairs created with the same key. In essence, it is proved in [BELL96a] that for a given level of effort (time, message–tag pairs) on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function.

1. The attacker is able to compute an output of the compression function even with an *IV* that is random, secret, and unknown to the attacker.
2. The attacker finds collisions in the hash function even when the *IV* is random and secret.

In the first attack, we can view the compression function as equivalent to the hash function applied to a message consisting of a single b-bit block. For this attack, the *IV* of the hash function is replaced by a secret, random value of n bits. An attack on this hash function requires either a brute-force attack on the key, which is a level of effort on the order of 2^n, or a birthday attack, which is a special case of the second attack, discussed next.

In the second attack, the attacker is looking for two messages M and M' that produce the same hash: $\mathrm{H}(M) = \mathrm{H}(M')$. This is the birthday attack discussed in Chapter 11. We have shown that this requires a level of effort of $2^{n/2}$ for a hash length of n. On this basis, the security of MD5 is called into question, because a level of effort of 2^{64} looks feasible with today's technology. Does this mean that a 128-bit hash function such as MD5 is unsuitable for HMAC? The answer is no, because of the following argument. To attack MD5, the attacker can choose any set of messages and work on these off line on a dedicated computing facility to find a collision. Because the attacker knows the hash algorithm and the default *IV*, the attacker can generate the hash code for each of the messages that the attacker generates. However, when attacking HMAC, the attacker cannot generate message/code pairs off line because the attacker does not know K. Therefore, the attacker must observe a sequence of messages generated by HMAC under the same key and perform the attack on these known messages. For a hash code length of 128 bits, this requires 2^{64} observed blocks (2^{72} bits) generated using the same key. On a 1-Gbps link, one would need to observe a continuous stream of messages with no change in key for about 250,000 years in order to succeed. Thus, if speed is a concern, it is fully acceptable to use MD5 rather than SHA-1 as the embedded hash function for HMAC.

12.6 MACs BASED ON BLOCK CIPHERS: DAA AND CMAC

In this section, we look at two MACs that are based on the use of a block cipher mode of operation. We begin with an older algorithm, the Data Authentication Algorithm (DAA), which is now obsolete. Then we examine CMAC, which is designed to overcome the deficiencies of DAA.

Data Authentication Algorithm

The Data Authentication Algorithm (DAA), based on DES, has been one of the most widely used MACs for a number of years. The algorithm is both a FIPS publication (FIPS PUB 113) and an ANSI standard (X9.17). However, as we discuss subsequently, security weaknesses in this algorithm have been discovered, and it is being replaced by newer and stronger algorithms.

The algorithm can be defined as using the cipher block chaining (CBC) mode of operation of DES (Figure 6.4) with an initialization vector of zero. The data (e.g., message, record, file, or program) to be authenticated are grouped into contiguous 64-bit blocks: $D_1, D_2, \ldots, D_N$. If necessary, the final block is padded on the right with zeroes to form a full 64-bit block. Using the DES encryption algorithm E and a secret key K, a data authentication code (DAC) is calculated as follows (Figure 12.7).

$$
\begin{aligned}
O_1 &= \mathrm{E}(K, D) \\
O_2 &= \mathrm{E}(K, [D_2 \oplus O_1]) \\
O_3 &= \mathrm{E}(K, [D_3 \oplus O_2]) \\
&\cdot \\
&\cdot \\
&\cdot \\
O_N &= \mathrm{E}(K, [D_N \oplus O_{N-1}])
\end{aligned}
$$

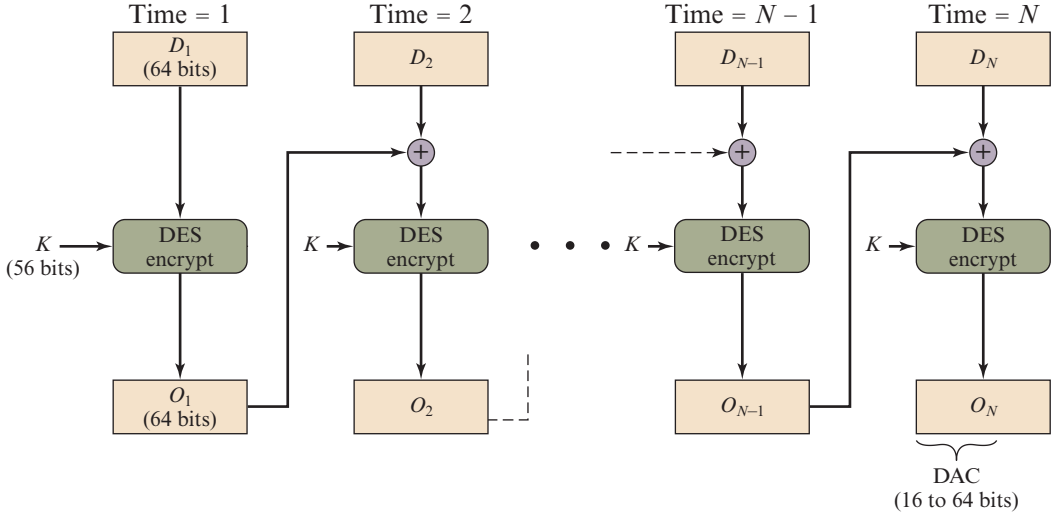


Figure 12.7 Data Authentication Algorithm (FIPS PUB 113)

The DAC consists of either the entire block O_N or the leftmost M bits of the block, with $16 \leq M \leq 64$.

Cipher-Based Message Authentication Code (CMAC)

As was mentioned, DAA has been widely adopted in government and industry. [BELL00] demonstrated that this MAC is secure under a reasonable set of security criteria, with the following restriction. Only messages of one fixed length of mn bits are processed, where n is the cipher block size and m is a fixed positive integer. As a simple example, notice that given the CBC MAC of a one-block message X, say $T = \text{MAC}(K, X)$, the adversary immediately knows the CBC MAC for the two-block message $X \| (X \oplus T)$ since this is once again T.

Black and Rogaway [BLAC00] demonstrated that this limitation could be overcome using three keys: one key K of length k to be used at each step of the cipher block chaining and two keys of length b, where b is the cipher block length. This proposed construction was refined by Iwata and Kurosawa so that the two n-bit keys could be derived from the encryption key, rather than being provided separately [IWAT03]. This refinement, adopted by NIST, is the Cipher-based Message Authentication Code **(CMAC)** mode of operation for use with AES and triple DES. It is specified in NIST Special Publication 800-38B.

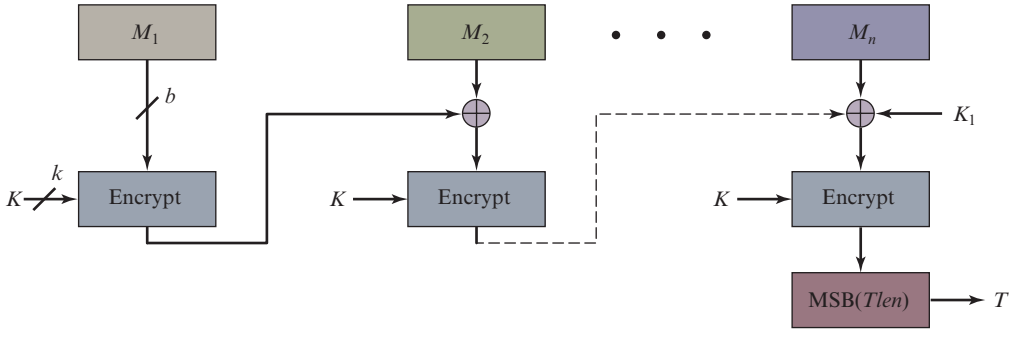
First, let us define the operation of CMAC when the message is an integer multiple n of the cipher block length b. For AES, $b = 128$, and for triple DES, $b = 64$. The message is divided into n blocks $(M_1, M_2, \ldots, M_n)$. The algorithm makes use of a k-bit encryption key K and a b-bit constant, K_1. For AES, the key size k is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits. CMAC is calculated as follows (Figure 12.8).

$$
\begin{aligned}
C_1 &= \text{E}(K, M_1) \\
C_2 &= \text{E}(K, [M_2 \oplus C_1]) \\
C_3 &= \text{E}(K, [M_3 \oplus C_2]) \\
&\cdot \\
&\cdot \\
&\cdot \\
C_n &= \text{E}(K, [M_n \oplus C_{n-1} \oplus K_1]) \\
T &= \text{MSB}_{Tlen}(C_n)
\end{aligned}
$$

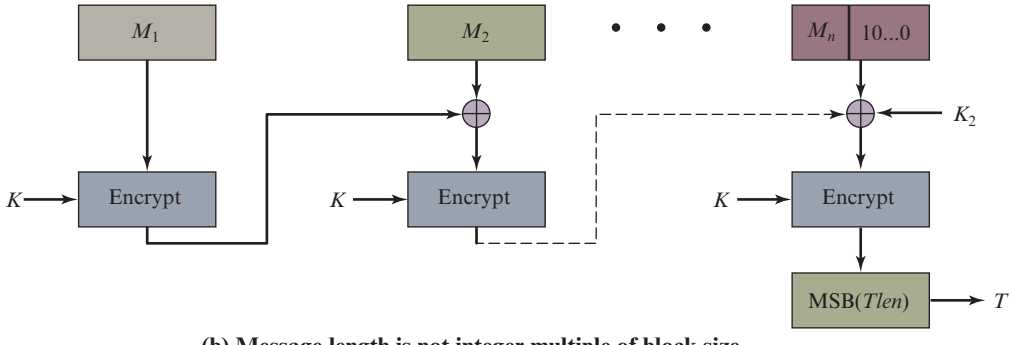
where

$$
\begin{aligned}
T &= \text{message authentication code, also referred to as the tag} \\
Tlen &= \text{bit length of T} \\
\text{MSB}_s(X) &= \text{the } s \text{ leftmost bits of the bit string } X
\end{aligned}
$$

If the message is not an integer multiple of the cipher block length, then the final block is padded to the right (least significant bits) with a 1 and as many 0s as necessary so that the final block is also of length b. The CMAC operation then proceeds as before, except that a different b-bit key K_2 is used instead of K_1.



(a) Message length is integer multiple of block size



(b) Message length is not integer multiple of block size

Figure 12.8 Cipher-based Message Authentication Code (CMAC)

The two b-bit keys are derived from the k-bit encryption key as follows.

$$
\begin{aligned}
L &= \mathrm{E}(K, 0^b) \\
K_1 &= L \cdot x \\
K_2 &= L \cdot x^2 = (L \cdot x) \cdot x
\end{aligned}
$$

where multiplication ($\cdot$) is done in the finite field GF(2^b) and x and x^2 are first- and second-order polynomials that are elements of GF(2^b). Thus, the binary representation of x consists of $b - 2$ zeros followed by 10; the binary representation of x^2 consists of $b - 3$ zeros followed by 100. The finite field is defined with respect to an irreducible polynomial that is lexicographically first among all such polynomials with the minimum possible number of nonzero terms. For the two approved block sizes, the polynomials are $x^{64} + x^4 + x^3 + x + 1$ and $x^{128} + x^7 + x^2 + x + 1$.

To generate K_1 and K_2, the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting ciphertext by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. This property of finite fields of the form GF(2^b) was explained in the discussion of MixColumns in Chapter 6.

12.7 AUTHENTICATED ENCRYPTION: CCM AND GCM

Authenticated encryption (AE) is a term used to describe encryption systems that simultaneously protect confidentiality and authenticity (integrity) of communications. Many applications and protocols require both forms of security, but until recently the two services have been designed separately.

There are four common approaches to providing both confidentiality and encryption for a message M.

- **Hashing followed by encryption (H → E):** First compute the cryptographic hash function over M as $h = H(M)$. Then encrypt the message plus hash function: $E(K, (M \| h))$.
- **Authentication followed by encryption (A → E):** Use two keys. First authenticate the plaintext by computing the MAC value as $T = MAC(K_1, M)$. Then encrypt the message plus tag: $E(K_2, [M \| T])$. This approach is taken by the SSL/TLS protocols (Chapter 19).
- **Encryption followed by authentication (E → A):** Use two keys. First encrypt the message to yield the ciphertext $C = E(K_2, M)$. Then authenticate the ciphertext with $T = MAC(K_1, C)$ to yield the pair (C, T). This approach is used in the IPSec protocol (Chapter 22).
- **Independently encrypt and authenticate (E + A).** Use two keys. Encrypt the message to yield the ciphertext $C = E(K_2, M)$. Authenticate the plaintext with $T = MAC(K_1, M)$ to yield the pair (C, T). These operations can be performed in either order. This approach is used by the SSH protocol (Chapter 19).

Both decryption and verification are straightforward for each approach. For H → E, A → E, and E + A, decrypt first, then verify. For E → A, verify first, then decrypt. There are security vulnerabilities with all of these approaches. The H → E approach is used in the Wired Equivalent Privacy (WEP) protocol to protect WiFi networks. This approach had fundamental weaknesses and led to the replacement of the WEP protocol. [BLAC05] and [BELL00] point out that there are security concerns in each of the three encryption/MAC approaches listed above. Nevertheless, with proper design, any of these approaches can provide a high level of security. This is the goal of the two approaches discussed in this section, both of which have been standardized by NIST.

Counter with Cipher Block Chaining-Message Authentication Code

The **CCM** mode of operation was standardized by NIST specifically to support the security requirements of IEEE 802.11 WiFi wireless local area networks (Chapter 20), but can be used in any networking application requiring authenticated encryption. CCM is a variation of the encrypt-and-MAC approach to authenticated encryption. It is defined in NIST SP 800-38C.

The key algorithmic ingredients of CCM are the AES encryption algorithm (Chapter 6), the CTR mode of operation (Chapter 7), and the CMAC authentication

algorithm (Section 12.6). A single key K is used for both encryption and MAC algorithms. The input to the CCM encryption process consists of three elements.

1. Data that will be both authenticated and encrypted. This is the plaintext message P of data block.
2. Associated data A that will be authenticated but not encrypted. An example is a protocol header that must be transmitted in the clear for proper protocol operation but which needs to be authenticated.
3. A nonce N that is assigned to the payload and the associated data. This is a unique value that is different for every instance during the lifetime of a protocol association and is intended to prevent replay attacks and certain other types of attacks.

Figure 12.9 illustrates the operation of CCM. For authentication, the input includes the nonce, the associated data, and the plaintext. This input is formatted as a sequence of blocks B_0 through B_r. The first block contains the nonce plus some formatting bits that indicate the lengths of the N, A, and P elements. This is followed by zero or more blocks that contain A, followed by zero of more blocks that contain P. The resulting sequence of blocks serves as input to the CMAC algorithm, which produces a MAC value with length *Tlen*, which is less than or equal to the block length (Figure 12.9a).

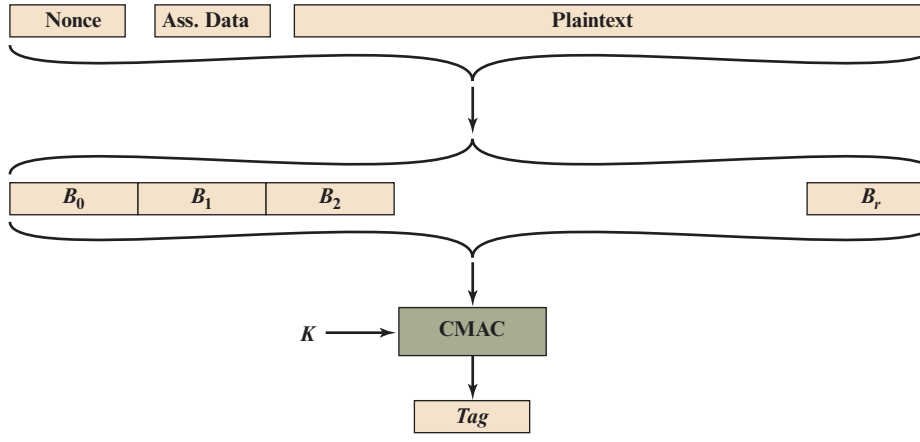
For encryption, a sequence of counters is generated that must be independent of the nonce. The authentication tag is encrypted in CTR mode using the single counter Ctr_0. The *Tlen* most significant bits of the output are XORed with the tag to produce an encrypted tag. The remaining counters are used for the CTR mode encryption of the plaintext (Figure 7.7). The encrypted plaintext is concatenated with the encrypted tag to form the ciphertext output (Figure 12.9b).

SP 800-38C defines the authentication/encryption process as follows.

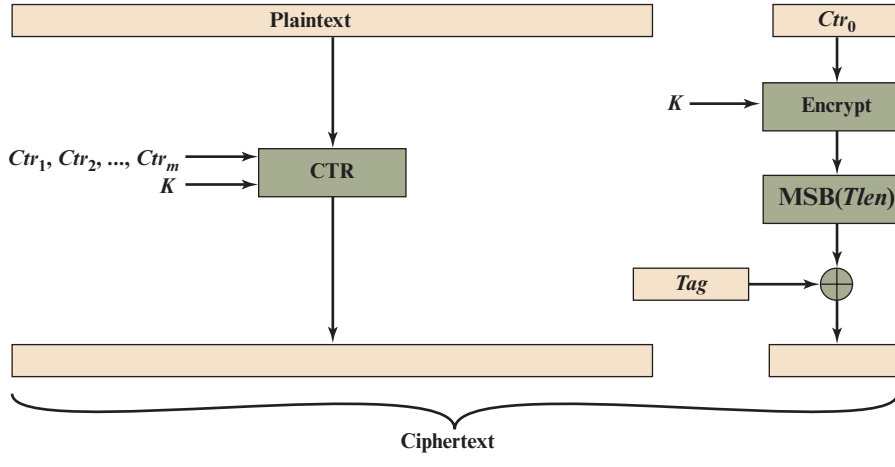
1. Apply the formatting function to (N, A, P) to produce the blocks $B_0, B_1, \ldots, B_r$.
2. Set $Y_0 = \mathrm{E}(K, B_0)$.
3. For $i = 1$ to r, do $Y_i = \mathrm{E}(K, (B_i \oplus Y_{i-1}))$.
4. Set $T = \mathrm{MSB}_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lceil Plen/128 \rceil$.
6. For $j = 0$ to m, do $S_j = \mathrm{E}(K, Ctr_j)$.
7. Set $S = S_1 \| S_2 \| \cdots \| S_m$.
8. Return $C = (P \oplus \mathrm{MSB}_{Plen}(S)) \| (T \oplus \mathrm{MSB}_{Tlen}(S_0))$.

For decryption and verification, the recipient requires the following input: the ciphertext C, the nonce N, the associated data A, the key K, and the initial counter Ctr_0. The steps are as follows.

1. If $Clen \leq Tlen$, then return INVALID.
2. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lceil Clen/128 \rceil$.
3. For $j = 0$ to m, do $S_j = \mathrm{E}(K, Ctr_j)$.



(a) Authentication



(b) Encryption

Figure 12.9 Counter with Cipher Block Chaining-Message Authentication Code (CCM)

4. Set $S = S_1 \| S_2 \| \cdots \| S_m$.
5. Set $P = \text{MSB}_{Clen-Tlen}(C) \oplus \text{MSB}_{Clen-Tlen}(S)$.
6. Set $T = \text{LSB}_{Tlen}(C) \oplus \text{MSB}_{Tlen}(S_0)$.
7. Apply the formatting function to N, A, P) to produce the blocks $B_0, B_1, \ldots, B_r$.
8. Set $Y_0 = \text{E}(K, B_0)$.
9. For $i = 1$ to r do $Y_i = \text{E}(K, (B_i \oplus Y_{i-1}))$.
10. If $T \neq \text{MSB}_{Tlen}(Y_r)$, then return INVALID, else return P.

CCM is a relatively complex algorithm. Note that it requires two complete passes through the plaintext, once to generate the MAC value, and once for encryption. Further, the details of the specification require a tradeoff between the length of the nonce and the length of the tag, which is an unnecessary restriction. Also note that the encryption key is used twice with the CTR encryption mode: once to generate the tag and once to encrypt the plaintext plus tag. Whether these complexities add to the security of the algorithm is not clear. In any case, two analyses of the algorithm ([JONS02] and [ROGA03]) conclude that CCM provides a high level of security.

Galois/Counter Mode

The GCM mode of operation, standardized by NIST in NIST SP 800-38D, is designed to be parallelizable so that it can provide high throughput with low cost and low latency. In essence, the message is encrypted in variant of CTR mode. The resulting ciphertext is multiplied with key material and message length information over GF(2^{128}) to generate the authenticator tag. The standard also specifies a mode of operation that supplies the MAC only, known as GMAC.

The GCM mode makes use of two functions: GHASH, which is a keyed hash function, and GCTR, which is essentially the CTR mode with the counters determined by a simple increment by one operation.

$\text{GHASH}_H(X)$ takes a input the hash key H and a bit string X such that $\text{len}(X) = 128m$ bits for some positive integer m and produces a 128-bit MAC value. The function may be specified as follows (Figure 12.10a).

1. Let $X_1, X_2, \ldots, X_{m-1}, X_m$ denote the unique sequence of blocks such that $X = X_1 \| X_2 \| \cdots \| X_{m-1} \| X_m$.
2. Let Y_0 be a block of 128 zeros, designated as 0^{128}.
3. For $i = 1, \ldots, m$, let $Y_i = (Y_{i-1} \oplus X_i) \cdot H$, where $\cdot$ designates multiplication in GF(2^{128}).
4. Return Y_m.

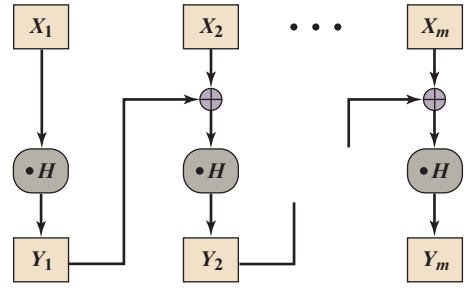
The $\text{GHASH}_H(X)$ function can be expressed as

$$(X_1 \cdot H^m) \oplus (X_2 \cdot H^{m-1}) \oplus \cdots \oplus (X_{m-1} \cdot H^2) \oplus (X_m \cdot H)$$

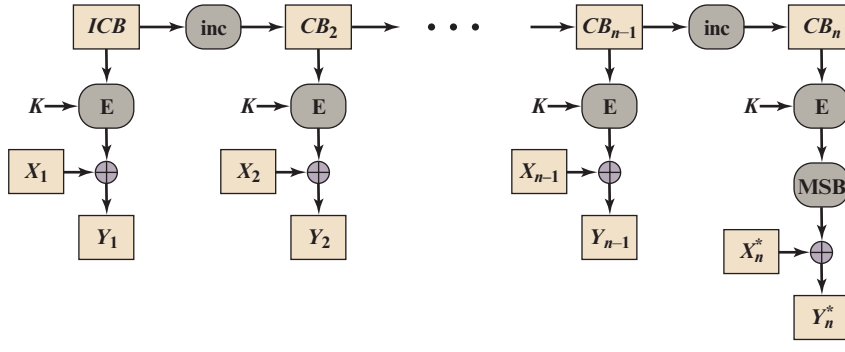
This formulation has desirable performance implications. If the same hash key is to be used to authenticate multiple messages, then the values $H^2, H^3, \ldots$ can be precalculated one time for use with each message to be authenticated. Then, the blocks of the data to be authenticated $(X_1, X_2, \ldots, X_m)$ can be processed in parallel, because the computations are independent of one another.

$\text{GCTR}_K(ICB, X)$ takes a input a secret key K and a bit string X arbitrary length and returns a ciphertext Y of bit length (X). The function may be specified as follows (Figure 12.10b).

1. If X is the empty string, then return the empty string as Y.
2. Let $n = \lceil (\text{len}(X)/128) \rceil$. That is, n is the smallest integer greater than or equal to $(X)/128$.



(a) $\text{GHASH}_H(X_1 \| X_2 \| \ldots \| X_m) = Y_m$



(b) $\text{GCTR}_K(ICB, X_1 \| X_2 \| \ldots \| X_n^*) = Y_1 \| Y_2 \| \ldots \| Y_n^*$

Figure 12.10 GCM Authentication and Encryption Functions

3. Let $X_1, X_2, \ldots, X_{n-1}, X_n^*$ denote the unique sequence of bit strings such that

$$X = X_1 \| X_2 \| \cdots \| X_{n-1} \| X_n^*;$$
$$X_1, X_2, \ldots, X_{n-1} \quad \text{are complete 128-bit blocks.}$$

4. Let $CB_1 = ICB$.
5. For, $i = 2$ to n let $CB_i = \text{inc}_{32}(CB_{i-1})$, where the $\text{inc}_{32}(S)$ function increments the rightmost 32 bits of S by 1 mod 2^{32}, and the remaining bits are unchanged.
6. For $i = 1$ to $n - 1$, do $Y_i = X_i \oplus \text{E}(K, CB_i)$.
7. Let $Y_n^* = X_n^* \oplus \text{MSB}_{\text{len}(X_n^*)}(\text{E}(K, CB_n))$.
8. Let $Y = Y_1 \| Y_2 \| \ldots \| Y_{n-1} \| Y_n^*$
9. Return Y.

Note that the counter values can be quickly generated and that the encryption operations can be performed in parallel.

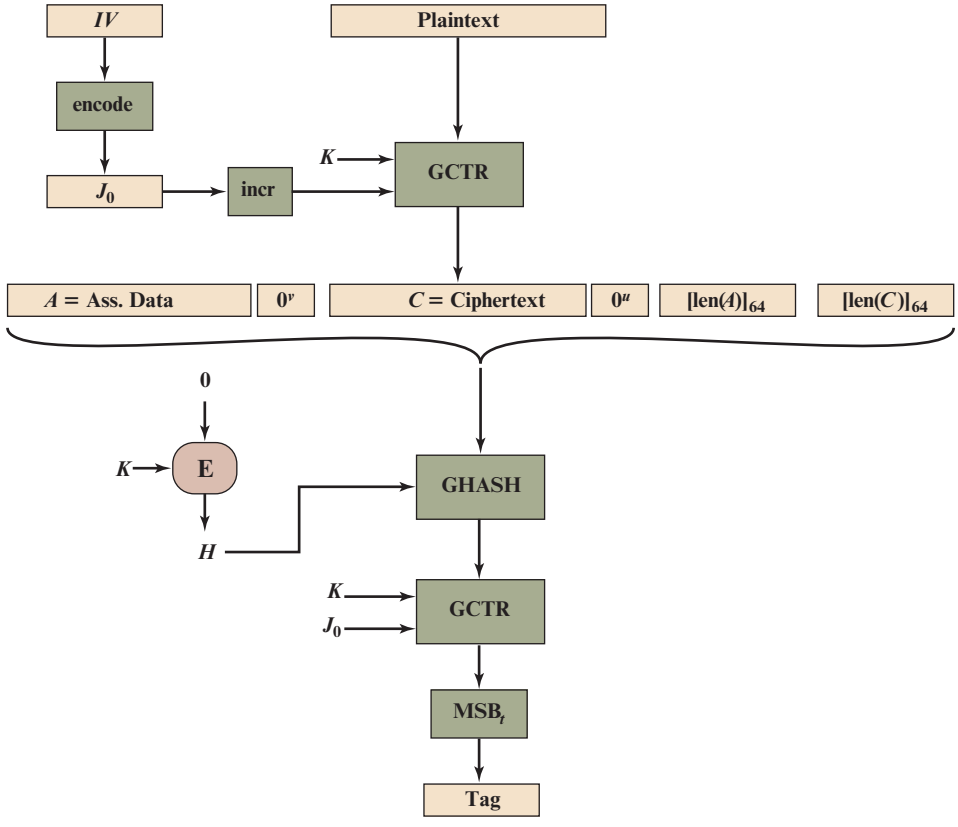


Figure 12.11 Galois Counter—Message Authentication Code (GCM)

We can now define the overall authenticated encryption function (Figure 12.11). The input consists of a secret key K, an initialization vector IV, a plaintext P, and additional authenticated data A. The notation $[x]_s$ means the s-bit binary representation of the nonnegative integer x. The steps are as follows.

1. Let $H = E(K, 0^{128})$.
2. Define a block, J_0, as
   If $\text{len}(IV) = 96$, then let $J_0 = IV \| 0^{31} \| 1$.
   If $\text{len}(IV) \neq 96$, then let $s = 128\lceil \text{len}(IV)/128 \rceil - \text{len}(IV)$, and let
   $J_0 = \text{GHASH}_H(IV \| 0^{s+64} \| [\text{len}(IV)]_{64})$.
3. Let $C = \text{GCTR}_K(\text{inc}_{32}(J_0), P)$.
4. Let $u = 128\lceil \text{len}(C)/128 \rceil - \text{len}(C)$ and let $v = 128\lceil \text{len}(A)/128 \rceil - \text{len}(A)$.
5. Define a block, S, as
$$S = \text{GHASH}_H(A \| 0^v \| C \| 0^u \| [\text{len}(A)]_{64} \| [\text{len}(C)]_{64})$$
6. Let $T = MSB_t(GCTR_K(J_0, S))$, where t is the supported tag length.
7. Return (C, T).

In step 1, the hash key is generated by encrypting a block of all zeros with the secret key K. In step 2, the pre-counter block (J_0) is generated from the IV. In particular, when the length of the IV is 96 bits, then the padding string $0^{31} \| 1$ is appended to the IV to form the pre-counter block. Otherwise, the IV is padded with the minimum number of 0 bits, possibly none, so that the length of the resulting string is a multiple of 128 bits (the block size); this string in turn is appended with 64 additional 0 bits, followed by the 64-bit representation of the length of the IV, and the GHASH function is applied to the resulting string to form the pre-counter block.

Thus, GCM is based on the CTR mode of operation and adds a MAC that authenticates both the message and additional data that requires only authentication. The function that computes the hash uses only multiplication in a Galois field. This choice was made because the operation of multiplication is easy to perform within a Galois field and is easily implemented in hardware [MCGR03].

[MCGR04] examines the available block cipher modes of operation and shows that a CTR-based authenticated encryption approach is the most efficient mode of operation for high-speed packet networks. The paper further demonstrates that GCM meets a high level of security requirements.

12.8 KEY WRAPPING

Background

The most recent block cipher mode of operation defined by NIST is the **Key Wrap (KW) mode** of operation (SP 800-38F), which uses AES or triple DEA as the underlying encryption algorithm. The AES version is also documented in RFC 3394.

The purpose of **key wrapping** is to securely exchange a symmetric key to be shared by two parties, using a symmetric key already shared by those parties. The latter key is called a **key encryption key (KEK)**.

Two questions need to be addressed at this point. First, why do we need to use a symmetric key already known to two parties to encrypt a new symmetric key? Such a requirement is found in a number of protocols described in this book, such as the key management portion of IEEE 802.11 and IPsec. This question is explored in Chapter 14.

The second question is, why do we need a new mode? The intent of the new mode is to operate on keys whose length is greater than the block size of the encryption algorithm. For example, AES uses a block size of 128 bits but can use a key size of 128, 192, or 256 bits. In the latter two cases, encryption of the key involves multiple blocks. We consider the value of key data to be greater than the value of other data, because the key will be used multiple times, and compromise of the key compromises all of the data encrypted with the key. Therefore, NIST desired a robust encryption mode. KW is robust in the sense that each bit of output can be expected

to depend in a nontrivial fashion on each bit of input. This is not the case for any of the other modes of operation that we have described. For example, in all of the modes so far described, the last block of plaintext only influences the last block of ciphertext. Similarly, the first block of ciphertext is derived only from the first block of plaintext.

To achieve this robust operation, KW achieves a considerably lower throughput than the other modes, but the tradeoff may be appropriate for some key management applications. Also, KW is only used for small amounts of plaintext compared to, say, the encryption of a message or a file.

The Key Wrapping Algorithm

The key wrapping algorithm operates on blocks of 64 bits. The input to the algorithm consists of a 64-bit constant, discussed subsequently, and a plaintext key that is divided into blocks of 64 bits. We use the following notation:

$\text{MSB}_{64}(W)$	most significant 64 bits of W
$\text{LSB}_{64}(W)$	least significant 64 bits of W
W	temporary value; output of encryption function
$\oplus$	bitwise exclusive-OR
$\|$	concatenation
K	key encryption key
n	number of 64-bit key data blocks
s	number of stages in the wrapping process; $s = 6n$
P_i	ith plaintext key data block; $1 \leq i \leq n$
C_i	ith ciphertext data block; $0 \leq i \leq n$
$A(t)$	64-bit integrity check register after encryption stage t; $1 \leq t \leq s$
$A(0)$	initial integrity check value (ICV); in hexadecimal: A6A6A6A6A6A6A6A6
$R(t, i)$	64-bit register i after encryption stage t; $1 \leq t \leq s$; $1 \leq i \leq n$

We now describe the key wrapping algorithm:

Inputs: Plaintext, n 64-bit values $(P_1, P_2, \ldots, P_n)$
Key encryption key, K

Outputs: Ciphertext, $(n + 1)$ 64-bit values $(C_0, C_1, \ldots, C_n)$

1. **Initialize variables.**

```
A(0) = A6A6A6A6A6A6A6A6
  for i = 1 to n
    R(0, i) = Pi
```

2. **Calculate intermediate values.**

```
for t = 1 to s
W = E(K, [A(t-1) || R(t-1, 1)])
A(t) = t ⊕ MSB64(W)
R(t, n) = LSB64(W)
for i = 1 to n-1
    R(t, i) = R(t-1, i+1)
```

3. **Output results.**

```
C0 = A(s)
for i = 1 to n
    Ci = R(s, i)
```

Note that the ciphertext is one block longer than the plaintext key, to accommodate the ICV. Upon unwrapping (decryption), both the 64-bit ICV and the plaintext key are recovered. If the recovered ICV differs from the input value of hexadecimal A6A6A6A6A6A6A6A6, then an error or alteration has been detected and the plaintext key is rejected. Thus, the key wrap algorithm provides not only confidentiality but also data integrity.

Figure 12.12 illustrated the key wrapping algorithm for encrypting a 256-bit key. Each box represents one encryption stage (one value of t). Note that the A output is fed as input to the next stage ($t + 1$), whereas the R output skips forward n stages ($t + n$), which in this example is $n = 4$. This arrangement further increases the avalanche effect and the mixing of bits. To achieve this skipping of stages, a sliding buffer is used, so that the R output from stage t is shifted in the buffer one position for each stage, until it becomes the input for stage $t + n$. This might be clearer if we expand the inner **for** loop for a 256-bit key ($n = 4$). Then the assignments are as follows:

$$R(t, 1) = R(t - 1, 2)$$
$$R(t, 2) = R(t - 1, 3)$$
$$R(t, 3) = R(t - 1, 4)$$

For example, consider that at stage 5, the R output has a value of $R(5, 4) = x$. At stage 6, we execute $R(6, 3) = R(5, 4) = x$. At stage 7, we execute $R(7, 2) = R(6, 3) = x$. At stage 8, we execute $R(8, 1) = R(7, 2) = x$. So, at stage 9, the input value of $R(t - 1, 1)$ is $R(8, 1) = x$.

Figure 12.13 depicts the operation of stage t for a 256-bit key. The dashed feedback lines indicate the assignment of new values to the stage variables.

Key Unwrapping

The key unwrapping algorithm can be defined as follows:

Inputs: Ciphertext, $(n + 1)$ 64-bit values $(C_0, C_1, \ldots, C_n)$
Key encryption key, K

Outputs: Plaintext, n 64-bit values $(P_1, P_2, \ldots, P_n)$, ICV

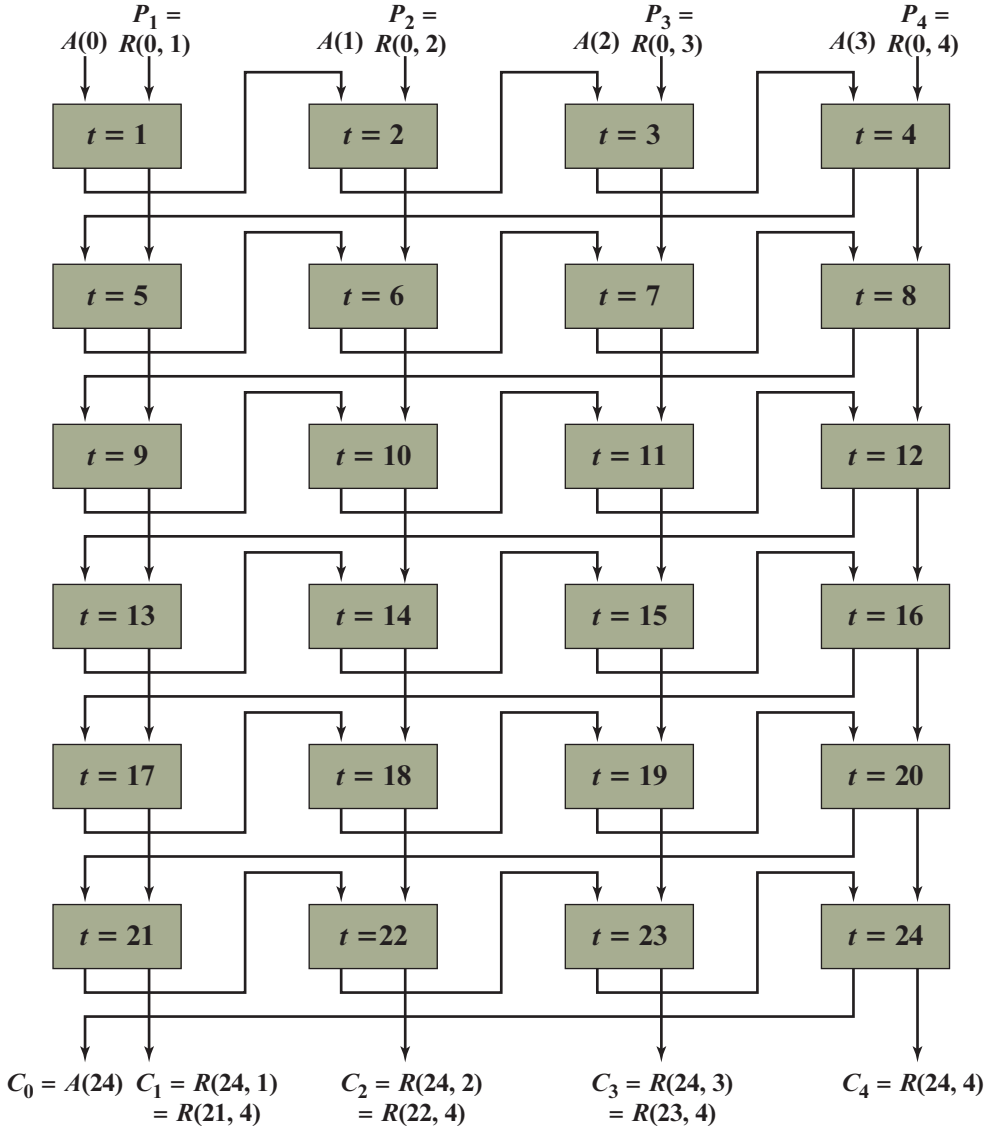


Figure 12.12 Key Wrapping Operation for 256-Bit Key

1. **Initialize variables.**

   $\mathbf{A}(s) = C_0$

   for $i = 1$ **to** n

   $\mathbf{R}(s, i) = C_i$

2. **Calculate intermediate values.**

   for $t = s$ **to** 1

   $W = \mathrm{D}(K, [(\mathbf{A}(t) \oplus t) \| \mathbf{R}(t, n)])$

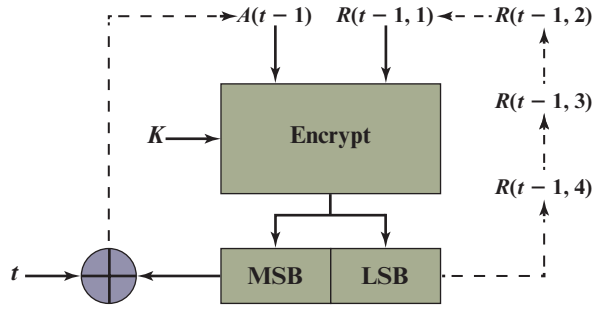


Figure 12.13 Key Wrapping Operation for 256-Bit Key: Stage t

```
A(t-1) = MSB64(W)
R(t-1, 1) = LSB64(W)
for i = 2 to n
     R(t-1, i) = R(t, i-1)
```

3. **Output results.**

```
if A(0) = A6A6A6A6A6A6A6A6
then
  for i = 1 to n
    P(i) = R(0, i)
else
  return error
```

Note that the decryption function is used in the unwrapping algorithm.

We now demonstrate that the unwrap function is the inverse of the wrap function, that is, that the unwrap function recovers the plaintext key and the ICV. First, note that because the index variable t is counted down from s to 1 for unwrapping, stage t of the unwrap algorithm corresponds to stage t of the wrap algorithm. The input variables to stage t of the wrap algorithm are indexed at $t - 1$ and the output variables of stage t of the unwrap algorithm are indexed at $t - 1$. Thus, to demonstrate that the two algorithms are inverses of each other, we need only demonstrate that the output variables of stage t of the unwrap algorithm are equal to the input variables to stage t of the wrap algorithm.

This demonstration is in two parts. First we demonstrate that the calculation of A and R variables prior to the **for** loop are inverses. To do this, let us simplify the notation a bit. Define the 128-bit value T to be the 64-bit value t followed by 64 zeros. Then, the first three lines of step 2 of the wrap algorithm can be written as the following single line:

$$A(t) \| R(t, n) = T \oplus \mathrm{E}(K, [A(t-1) \| R(t-1, 1)]) \quad \textbf{(12.1)}$$

The first three lines of step 2 of the unwrap algorithm can be written as:

$$A(t-1) \| R(t-1, 1) = \mathrm{D}(K, ([A(t) \| R(t, n)] \oplus T)) \quad \textbf{(12.2)}$$

Expanding the right-hand side by substituting from Equation 12.1,

$$\mathrm{D}(K, ([A(t) \| R(t, n)] \oplus T)) = \mathrm{D}(K, ([T \oplus \mathrm{E}(K, [A(t-1) \| R(t-1, 1)])] \oplus T))$$

Now we recognize that $T \oplus T = 0$ and that for any x, $x \oplus 0 = x$. So,

$$\begin{aligned}\mathrm{D}(K, ([A(t) \| R(t, n)] \oplus T)) &= \mathrm{D}(K, ([E(K, [A(t-1) \| R(t-1, 1)])) \\ &= A(t-1) \| R(t-1, 1)\end{aligned}$$

The second part of the demonstration is to show that the **for** loops in step 2 of the wrap and unwrap algorithms are inverses. For stage k of the wrap algorithm, the variables $R(t-1, 1)$ through $R(t-1, n)$ are input. $R(t-1, 1)$ is used in the encryption calculation. $R(t-1, 2)$ through $R(t-1, n)$ are mapped, respectively into $R(t, 1)$ through $R(t, n-1)$, and $R(t, n)$ is output from the encryption function. For stage k of the unwrap algorithm, the variables $R(t, 1)$ through $R(t, n)$ are input. $R(t, n)$ is input to the decryption function to produce $R(t-1, 1)$. The remaining variables $R(t-1, 2)$ through $R(t-1, n)$ are generated by the **for** loop, such that they are mapped, respectively, from $R(t, 1)$ through $R(t, n-1)$.

Thus, we have shown that the output variables of stage k of the unwrap algorithm equal the input variables of stage k of the wrap algorithm.

12.9 PSEUDORANDOM NUMBER GENERATION USING HASH FUNCTIONS AND MACs

The essential elements of any pseudorandom number generator (PRNG) are a seed value and a deterministic algorithm for generating a stream of pseudorandom bits. If the algorithm is used as a pseudorandom function (PRF) to produce a required value, such as a session key, then the seed should only be known to the user of the PRF. If the algorithm is used to produce a stream encryption function, then the seed has the role of a secret key that must be known to the sender and the receiver.

We noted in Chapters 8 and 10 that, because an encryption algorithm produces an apparently random output, it can serve as the basis of a (PRNG). Similarly, a hash function or MAC produces apparently random output and can be used to build a PRNG. Both ISO standard 18031 (*Random Bit Generation*) and NIST SP 800-90 (*Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) define an approach for random number generation using a cryptographic hash function. SP 800-90 also defines a random number generator based on HMAC. We look at these two approaches in turn.

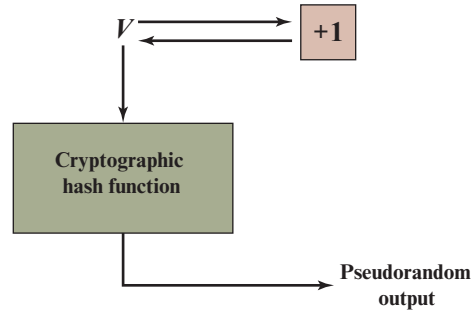
PRNG Based on Hash Function

Figure 12.14a shows the basic strategy for a hash-based PRNG specified in SP 800-90 and ISO 18031. The algorithm takes as input:

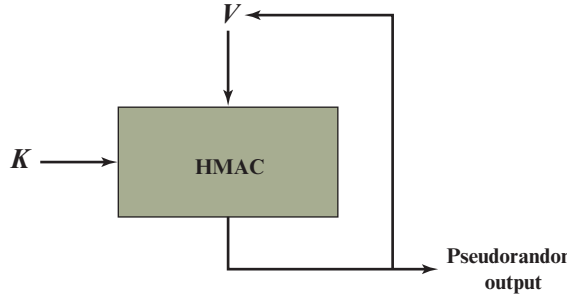
V = seed

$seedlen$ = bit length of V $\geq K + 64$, where k is a desired security level expressed in bits

n = desired number of output bits



(a) PRNG using cryptographic hash function



(b) PRNG using HMAC

Figure 12.14 Basic Structure of Hash-Based PRNGs (SP 800-90)

The algorithm uses the cryptographic hash function H with an hash value output of *outlen* bits. The basic operation of the algorithm is

$m = \lceil n/\text{outlen} \rceil$
data = V
W = the null string
For i = 1 to m
    w_i = H (data)
    W = $\|$ w_i
 data = (data + 1) mod $2^{seedlen}$
Return leftmost n bits of W

Thus, the pseudorandom bit stream is $w_1 \| w_2 \| \ldots \| w_m$ with the final block truncated if required.

The SP 800-90 specification also provides for periodically updating V to enhance security. The specification also indicates that there are no known or suspected weaknesses in the hash-based approach for a strong cryptographic hash algorithm, such as SHA-2.

PRNG Based on MAC Function

Although there are no known or suspected weaknesses in the use of a cryptographic hash function for a PRNG in the manner of Figure 12.14a, a higher degree of confidence can be achieved by using a MAC. Almost invariably, HMAC is used for constructing a MAC-based PRNG. This is because HMAC is a widely used standardized MAC function and is implemented in many protocols and applications. As SP 800-90 points out, the disadvantage of this approach compared to the hash-based approach is that the execution time is twice as long, because HMAC involves two executions of the underlying hash function for each output block. The advantage of the HMAC approach is that it provides a greater degree of confidence in its security, compared to a pure hash-based approach.

For the MAC-based approach, there are two inputs: a key K and a seed V. In effect, the combination of K and V form the overall seed for the PRNG specified in SP 800-90. Figure 12.14b shows the basic structure of the PRNG mechanism, and the leftmost column of Figure 12.15 shows the logic. Note that the key remains the same for each block of output, and the data input for each block is equal to the tag output of the previous block. The SP 800-90 specification also provides for periodically updating K and V to enhance security.

It is instructive to compare the SP 800-90 recommendation with the use of HMAC for a PRNG in some applications, and this is shown in Figure 12.15. For the IEEE 802.11i wireless LAN security standard (Chapter 20), the data input consists of the seed concatenated with a counter. The counter is incremented for each block w_i of output. This approach would seem to offer enhanced security compared to the SP 800-90 approach. Consider that for SP 800-90, the data input for output block w_i is just the output w_{i-1} of the previous execution of HMAC. Thus, an opponent who is able to observe the pseudorandom output knows both the input and output of HMAC. Even so, with the assumption that HMAC is secure, knowledge of the input and output should not be sufficient to recover K and hence not sufficient to predict future pseudorandom bits.

The approach taken by the Transport Layer Security protocol (Chapter 19) and the Wireless Transport Layer Security Protocol (Chapter 20) involves invoking HMAC twice for each block of output w_i. As with IEEE 802.11, this is done in such a way that the output does not yield direct information about the input. The double use of HMAC doubles the execution burden and would seem to be security overkill.

$m = \lceil n/\text{outlen} \rceil$ $w_0 = V$ $W =$ the null string For $i = 1$ to m $\quad w_i = \text{MAC}(K, w_{i-1})$ $\quad W = W \| w_i$ Return leftmost n bits of W	$m = \lceil n/outlen \rceil$ $W =$ the null string For $i = 1$ to m $\quad w_i = \text{MAC}(K, (V \| i))$ $\quad W = W \| w_i$ Return leftmost n bits of W	$m = \lceil n/outlen \rceil$ $A(0) = V$ $W =$ the null string For $i = 1$ to m $A(i) = \text{MAC}(K, A(i-1))$ $w_i = \text{MAC}(K, (A(i) \| V)$ $W = W \| w_i$ Return leftmost n bits of W
NIST SP 800-90	**IEEE 802.11i**	**TLS/WTLS**

Figure 12.15 Three PRNGs Based on HMAC

12.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

authenticator cryptographic checksum	cryptographic hash function key wrapping	message authentication message authentication code (MAC)

Review Questions

12.1 What types of attacks are addressed by message authentication?

12.2 In general, a MAC function is a many-to-one function. Justify this statement. State one point of difference between a MAC function and encryption.

12.3 What are some approaches to producing message authentication?

12.4 When sending a message to B, A can use A's private key and B's public key to achieve both secrecy and authentication. Which key is used to achieve which goal?

12.5 What is a message authentication code?

12.6 What is the difference between a message authentication code and a one-way hash function?

12.7 In what ways can a hash value be secured so as to provide message authentication?

12.8 Is it necessary to recover the secret key in order to attack a MAC algorithm?

12.9 What is the advantage and disadvantage of the HMAC approach used for PRNG, compared to a pure hash-based approach?

Problems

12.1 An error-detection function can be used to compute a frame check sequence (FCS) or checksum (Figure 12.2). The FCS can provide error-detection capability to detect whether any bit of the transmitted message is altered. We could append the FCS to each message before encryption (Figure 12.2a), which is referred to as internal error control. Alternatively, we could append the FCS to each message after encryption (Figure 12.2b), which is referred to as external error control. With internal error control, authentication is provided because an opponent would have difficulty generating ciphertext that would have valid error control bits when it is decrypted. Will the FCS still provide authentication to the message if external error control is used?

12.2 The data authentication algorithm (DAA) based on DES with an initialization vector *IV* of zero (Figure 12.7) has been discovered to have security weaknesses. It is being replaced by newer and stronger algorithms. Show that the DAA cannot be trusted.

12.3 Section 12.6 mentions that a refined CMAC using two additional b-bit keys, K_1 and K_2, is derived from the k-bit encryption key. Describe how you would generate K_1 and K_2.

12.4 In this problem, we demonstrate that for CMAC, a variant that XORs the second key after applying the final encryption doesn't work. Let us consider this for the case of the message being an integer multiple of the block size. Then, the variant can be expressed as $\text{VMAC}(K, M) = \text{CBC}(K, M) \oplus K_1$. Now suppose an adversary is able to ask for the MACs of three messages: the message $\mathbf{0} = 0^n$, where n is the cipher block size; the message $\mathbf{1} = 1^n$; and the message $\mathbf{1} \| \mathbf{0}$. As a result of these three queries, the adversary gets $T_0 = \text{CBC}(K, \mathbf{0}) \oplus K_1$; $T_1 = \text{CBC}(K, \mathbf{1}) \oplus K_1$ and $T_2 = \text{CBC}(K, [CBC(K, 1)]) \oplus K_1$. Show that the adversary can compute the correct MAC for the (unqueried) message $\mathbf{0} \| (T_0 \oplus T_1)$.

12.5 In the discussion of subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey.

a. What constants are needed for block sizes of 192 bits and 256 bits?

b. Explain how the left shift and XOR accomplishes the desired result.

12.6 Section 12.7 listed four general approaches to provide confidentiality and message encryption: H → E, A → E, E → A, and E + A.

a. Which of the above performs decryption before verification?

b. Which of the above performs verification before decryption?

12.7 Show that the GHASH function calculates

$$(X_1 \cdot H^m) \oplus (X_2 \cdot H^{m-1}) \oplus \cdots \oplus (X_{m-1} \cdot H^2) \oplus (X_m \cdot H)$$

12.8 Draw a figure similar to Figure 12.11 that shows authenticated decryption.

12.9 Alice wants to send a single bit of information (a yes or a no) to Bob by means of a word of length 2. Alice and Bob have four possible keys available to perform message authentication. The following matrix shows the 2-bit word sent for each message under each key:

	Message	
Key	0	1
1	00	11
2	01	10
3	10	01
4	11	00

a. The preceding matrix is in a useful form for Alice. Construct a matrix with the same information that would be more useful for Bob.

b. What is the probability that someone else can successfully impersonate Alice?

c. What is the probability that someone can replace an intercepted message with another message successfully?

12.10 Draw figures similar to Figures 12.12 and 12.13 for the unwrap algorithm.

12.11 Consider the following key wrapping algorithm:

1. Initialize variables.

$\boldsymbol{A} = \text{A6A6A6A6A6A6A6A6}$
for $\boldsymbol{i} = 1$ **to** $\boldsymbol{n}$
  $\boldsymbol{R}(\boldsymbol{i}) = \boldsymbol{P}_{\text{i}}$

2. Calculate intermediate values.

for $\boldsymbol{j} = 0$ **to** 5
  for $\boldsymbol{i} = 1$ **to** $\boldsymbol{n}$
    $\boldsymbol{B} = \text{E}(\boldsymbol{K}, [\boldsymbol{A} \parallel \boldsymbol{R}(\boldsymbol{i})])$
    $\boldsymbol{t} = (\boldsymbol{n} \times \boldsymbol{j}) + \boldsymbol{i}$
    $\boldsymbol{A} = \boldsymbol{t} \oplus \text{MSB}_{64}(\boldsymbol{B})$
    $\boldsymbol{R}(\boldsymbol{i}) = \text{LSB}_{64}(\boldsymbol{B})$

3. Output results.

$\boldsymbol{C}_0 = \boldsymbol{A}$
for $\boldsymbol{i} = 1$ **to** $\boldsymbol{n}$
  $\boldsymbol{C}_{\text{i}} = \boldsymbol{R}(\boldsymbol{i})$

a. Compare this algorithm, functionally, with the algorithm specified in SP 800-38F and described in Section 12.8.
b. Write the corresponding unwrap algorithm.