Module - 5 (Authentication)

Contents

5.1. Authentication requirements 2

5.2. MESSAGE AUTHENTICATION FUNCTIONS 2

MESSAGE ENCRYPTION 3

5.3 Message Authentication Code 7

REQUIREMENTS FOR MESSAGE AUTHENTICATION CODES 10

5.4 HASH FUNCTION 12

5.5 SECURITY OF MACs 16

5.6 MD5 17

Description of MD5 17

Security of MD5 21

5.7 Secure Hash Algorithm (SHA) 22

Description of SHA 22

SHA 512 24

5.8.1 MACs BASED ON HASH FUNCTIONS: HMAC 29

5.8.2 MACs BASED ON BLOCK CIPHERS: DAA AND CMAC 34

Cipher-Based Message Authentication Code (CMAC) 35

5.9 X.509 Authentication Service 37

5.1. 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.

5.2. 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

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.

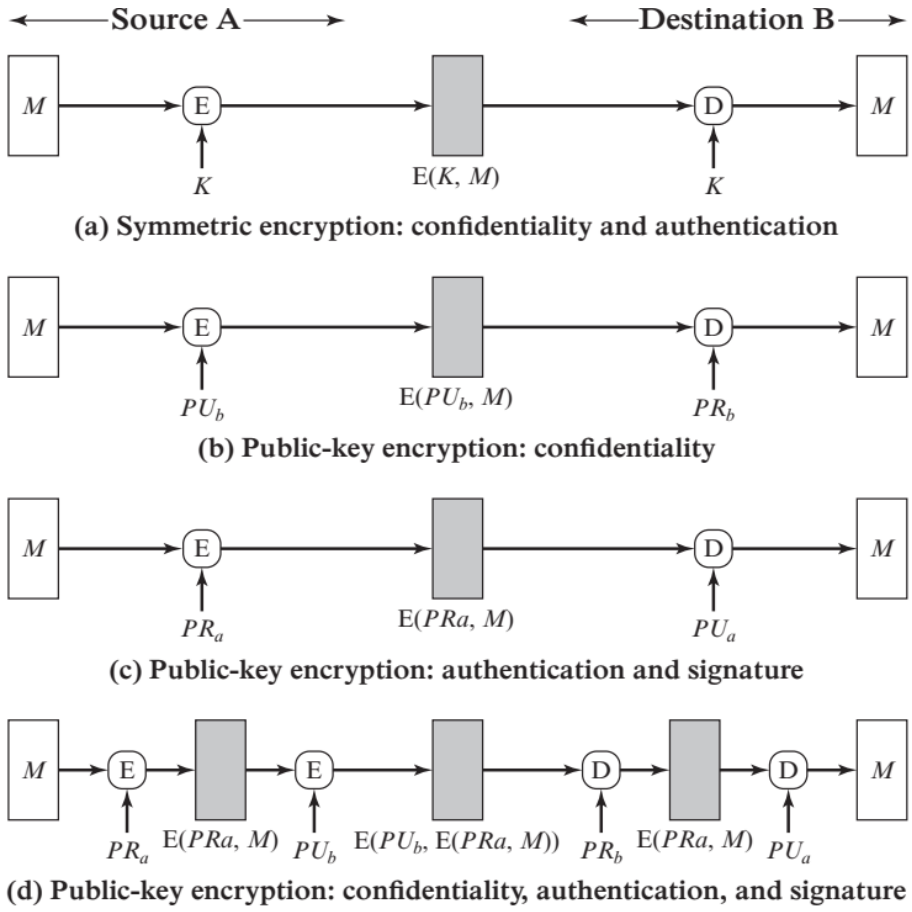


Figure 12.1 Basic Uses of Message Encryption

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.

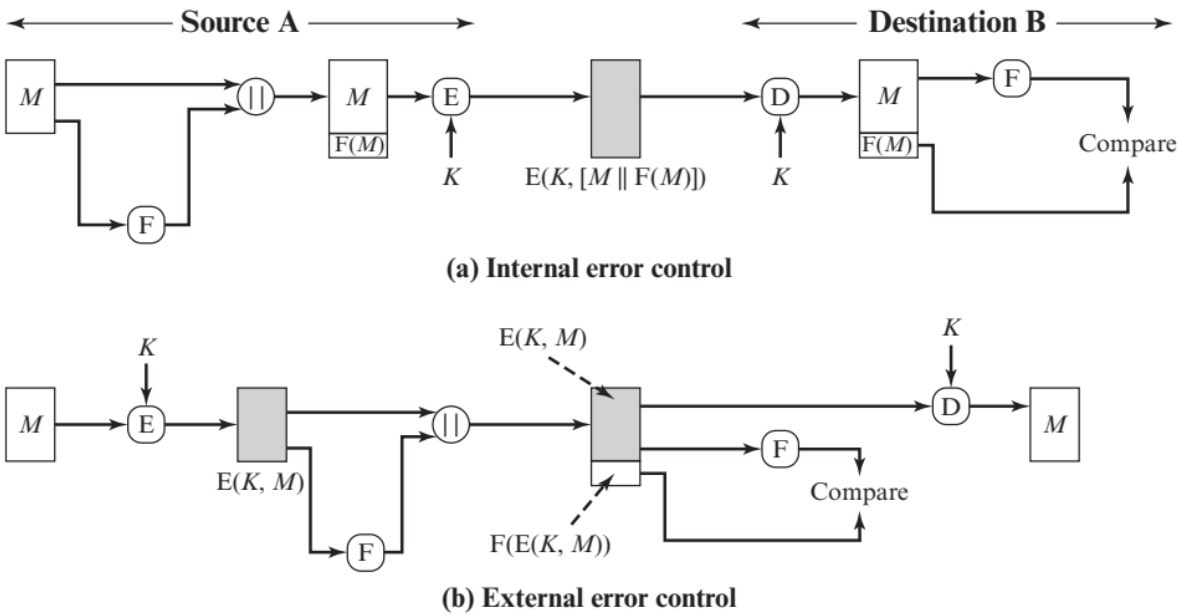


Figure 12.2 Internal and External Error Control

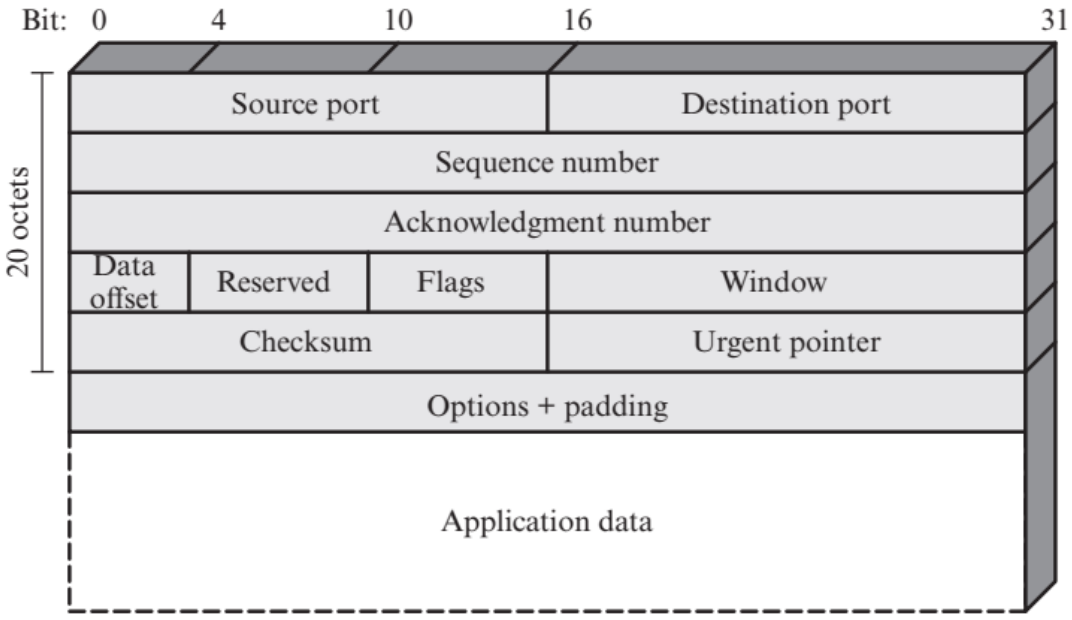


Figure 12.3 TCP Segment

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 *PRa* and therefore the only party with the information necessary to construct ciphertext that can be decrypted with *PUa*. 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 *PRa*. 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.

5.3 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: MAC = 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.

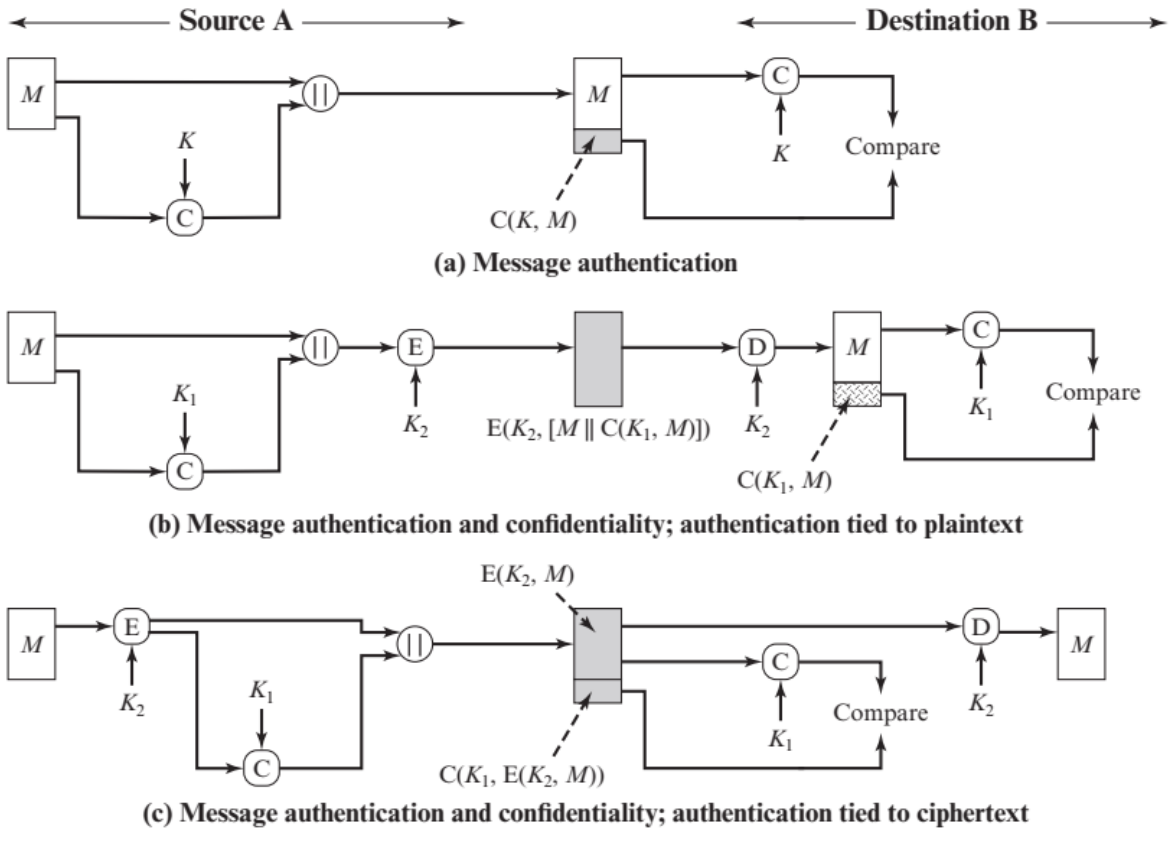


Figure 12.4 Basic Uses of Message Authentication code (MAC)

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* W 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 2100 different messages but only 210 different MACs. So, on average, each MAC value is generated by a total of 2100/210 = 290 different messages. If a 5-bit key is used, then there are 25 = 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.

REQUIREMENTS FOR MESSAGE AUTHENTICATION CODES

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

T = 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 Pi = D(Ki, C) for all possible key values Ki until a Pi 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 7 n; that is, suppose that the key size is greater than the MAC size. Then, given a known M1 and T1, with T1 = MAC(K, M1), the cryptanalyst can perform Ti = MAC(Ki, M1) for all possible key values ki. At least one key is guaranteed to produce a match of Ti = T1. Note that a total of 2k tags will be produced, but there are only 2n 6 2k 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 2k/2n = 2(k - n) keys will produce a match. Thus, the opponent must iterate the attack.

■ Round 1
Given: M1, T1 = MAC(K, M1)
Compute Ti = MAC(Ki, M1) for all 2k keys
Number of matches L 2(k - n)

■ Round 2
Given: M2, T2 = MAC(K, M2)
Compute Ti = MAC(Ki, M2) for the 2(k - n) keys resulting from Round 1
Number of matches L 2(k - 2 * n)
And so on. On average, a rounds will be needed k = a * n.

For example, if an 80-bit key is used and the tag is 32 bits, then the first round will produce about 248 possible keys. The second round will narrow the possible keys to about 216 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 $M = (X1 \| X2 \| c \| Xm)$ be a message that is treated as a concatenation of 64-bit blocks Xi. Then define $\Delta(M) = X1 \oplus X2 \oplus c \oplus Xm$

$MAC(K, M) = E(K, \Delta(M))$

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 $\{M \| MAC(K, M)\}$, a brute-force attempt to determine K will require at least 256 encryptions. But the opponent can attack the system by replacing $X1$ through $Xm - 1$ with any desired values $Y1$ through $Ym - 1$ and replacing Xm with Ym, where Ym is calculated as

$Ym = Y1 \oplus Y2 \oplus g \oplus Ym - 1 \oplus \Delta(M)$

The opponent can now concatenate the new message, which consists of $Y1$ through Ym, using the original tag to form a message that will be accepted as authentic by the receiver. With this tactic, any message of length $64 * (m - 1)$ 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 $MAC(K, M') = MAC(K, M)$

2. MAC(*K*, *M)* should be uniformly distributed in the sense that for randomly chosen messages,*M* and *M′*, the probability that MAC(*K*, *M*) = 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′* = f(*M*). For example, f may involve inverting one or more specific bits. In that case,

Pr [MAC(*K*, *M*) = MAC(*K*, *M′*)] = 2-*n*

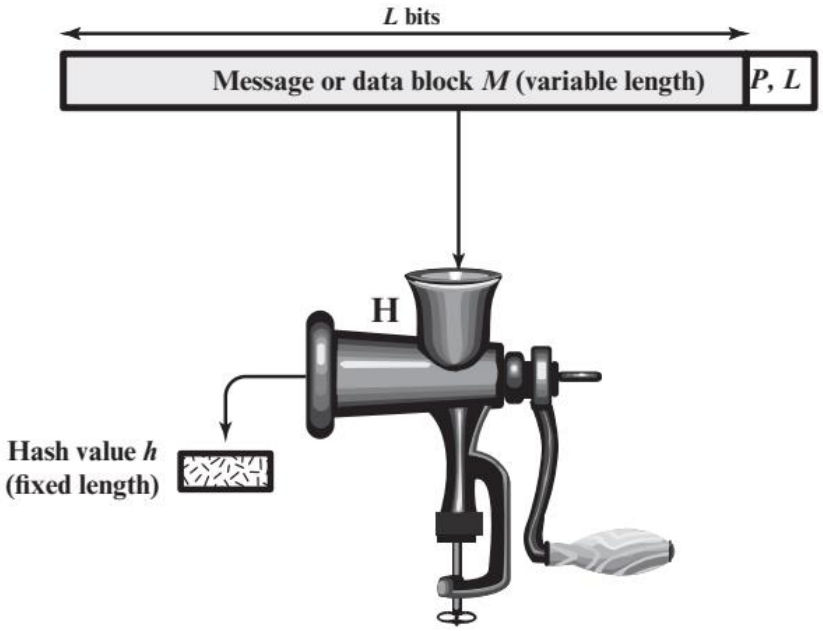
The first requirement speaks to the earlier example, in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key. The second requirement deals with the need to thwart a brute-force attack based on chosen plaintext. That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

5.4 HASH FUNCTION

A hash function H accepts a variable-length block of data M as input and produces a fixed-size hash value h = H(M). A "good" hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random. In general terms, the principal object of a hash function is data integrity. A change to any bit or bits in M results, with high probability, in a change to the hash value.

The kind of hash function needed for security applications is referred to as a cryptographic hash function. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly more efficient than brute force) to find either (a) a data object that maps to a pre-specified hash result (the one-way property) or (b) two data objects that map to the same hash result (the collision-free property). Because of these characteristics, hash functions are often used to determine whether or not data has changed.

Figure 11.1 depicts the general operation of a cryptographic hash function. Typically, the input is padded out to an integer multiple of some fixed length (e.g., 1024 bits), and the padding includes the value of the length of the original message in bits. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value, as explained subsequently.



P, L = padding plus length field

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

APPLICATIONS OF CRYPTOGRAPHIC HASH FUNCTIONS

Perhaps the most versatile cryptographic algorithm is the cryptographic hash function. It is used in a wide variety of security applications and Internet protocols. To better understand some of the requirements and security implications for cryptographic hash functions, it is useful to look at the range of applications in which it is employed.

Message Authentication

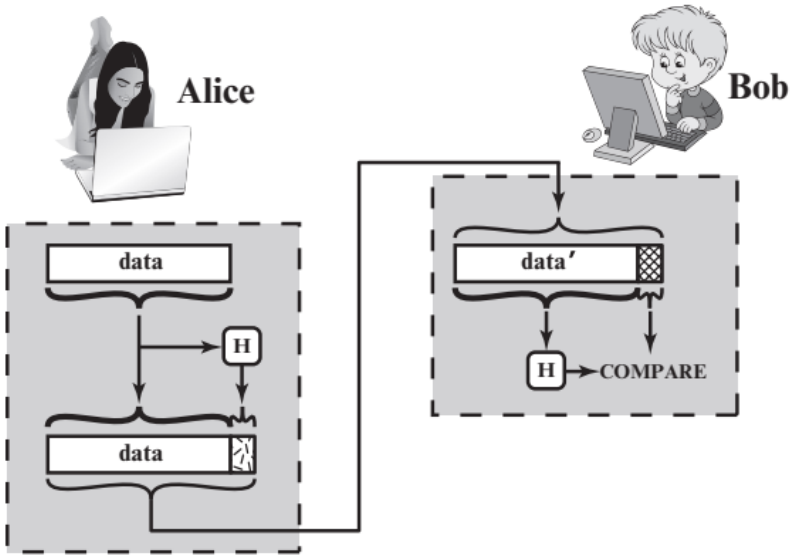
Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., there is no modification, insertion, deletion, or replay). In many cases, there is a requirement that the authentication mechanism assures that purported identity of the sender is valid. When a hash function is used to provide message authentication, the hash function value is often referred to as a **message digest**.

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

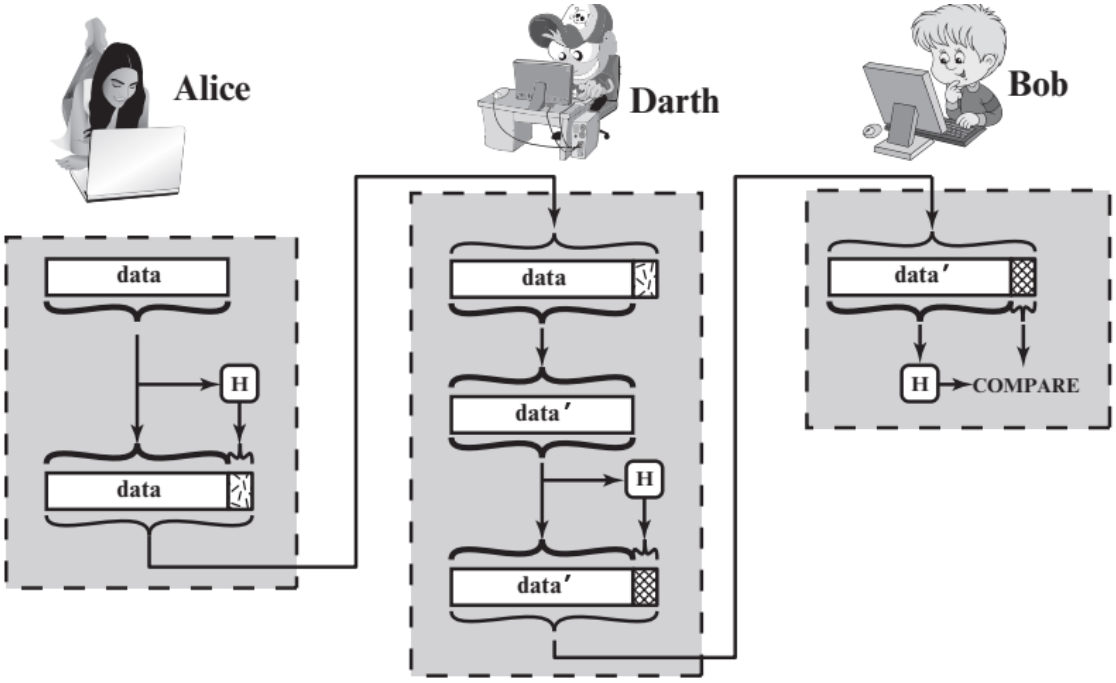
If there is a mismatch, the receiver knows that the message (or possibly the hash value) has been altered (Figure 11.2a).

The hash value must be transmitted in a secure fashion. That is, the hash value must be protected so that if an adversary alters or replaces the message, it is not feasible for adversary to also alter the hash value to fool the receiver. This type

of attack is shown in Figure 11.2b. In this example, Alice transmits a data block and attaches a hash value. Darth intercepts the message, alters or replaces the data block, and calculates and attaches a new hash value. Bob receives the altered data with the new hash value and does not detect the change. To prevent this attack, the hash value generated by Alice must be protected.



(a) Use of hash function to check data integrity



(b) Man-in-the-middle attack

Figure 11.2 Attack Against Hash Function

Figure 11.3 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows.

- **a.** The message plus concatenated hash code is encrypted using symmetric encryption. Because only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication. Because encryption is applied to the entire message plus hash code, confidentiality is also provided.
- **b.** Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality.
- **c.** It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value *S*. A computes the hash value over the concatenation of *M* and *S* and appends the resulting hash value to *M*. Because B possesses *S*, it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.
- **d.** Confidentiality can be added to the approach of method (c) by encrypting the entire message plus the hash code.

When confidentiality is not required, method (b) has an advantage over methods (a) and (d), which encrypts the entire message, in that less computation is required. Nevertheless, there has been growing interest in techniques that avoid encryption (Figure 11.3c). Several reasons for this interest are pointed out in [TSUD92].

■ Encryption software is relatively slow. Even though the amount of data to be encrypted per message is small, there may be a steady stream of messages into and out of a system.

■ Encryption hardware costs are not negligible. Low-cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have this capability.

■ Encryption hardware is optimized toward large data sizes. For small blocks of data, a high proportion of the time is spent in initialization/invocation overhead.

■ Encryption algorithms may be covered by patents, and there is a cost associated with licensing their use.

More commonly, message authentication is achieved using a message authentication code (MAC), also known as a keyed hash function. Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC, which is associated with the protected message. If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the associated MAC value. An attacker who alters the message will be unable to alter the associated MAC value without knowledge of the secret key. Note that the verifying party also knows who the sending party is because no one else knows the secret key.

Note that the combination of hashing and encryption results in an overall function that is, in fact, a MAC (Figure 11.3b). That is, E(K, H(M)) is a function of a variable-length message M and a secret key K, and it produces a fixed-size output that is secure against an opponent who

does not know the secret key. In practice, specific MAC algorithms are designed that are generally more efficient than an encryption algorithm.

5.5 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 $h = H(x)$, a brute-force method of finding a collision is to pick a random bit string y and check if $H(y) = H(x)$. 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 $[xi, MAC(K, xi)]$, it is computationally infeasible to compute any text-MAC pair $[x, MAC(K, x)]$ for any new input $x \neq xi$.

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) Ú N, where N is perhaps in the range of 128 bits.

Cryptanalysis

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 bruteforce attack. That is, an ideal MAC algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.

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. Furthermore, far less work has been done on developing such attacks. A useful survey of some methods for specific MACs is [PREN96].

5.6 MD5

MD5 is an improved version of MD4 [1386, 1322]. Although more complex than MD4, it is similar in design and also produces a 128-bit hash.

Description of MD5

After some initial processing, MD5 processes the input text in 512-bit blocks, divided into 16 32-bit sub-blocks. The output of the algorithm is a set of four 32-bit blocks, which concatenate to form a single 128-bit hash value.

First, the message is padded so that its length is just 64 bits short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros as are required. Then, a 64-bit representation of the message's length (before padding bits were added) is appended to the result. These two steps serve to make the message length an exact multiple of 512 bits in length (required for the rest of the algorithm), while ensuring that different messages will not look the same after padding.

Four 32-bit variables are initialized:

A = 0x01234567

B = 0x89abcdef

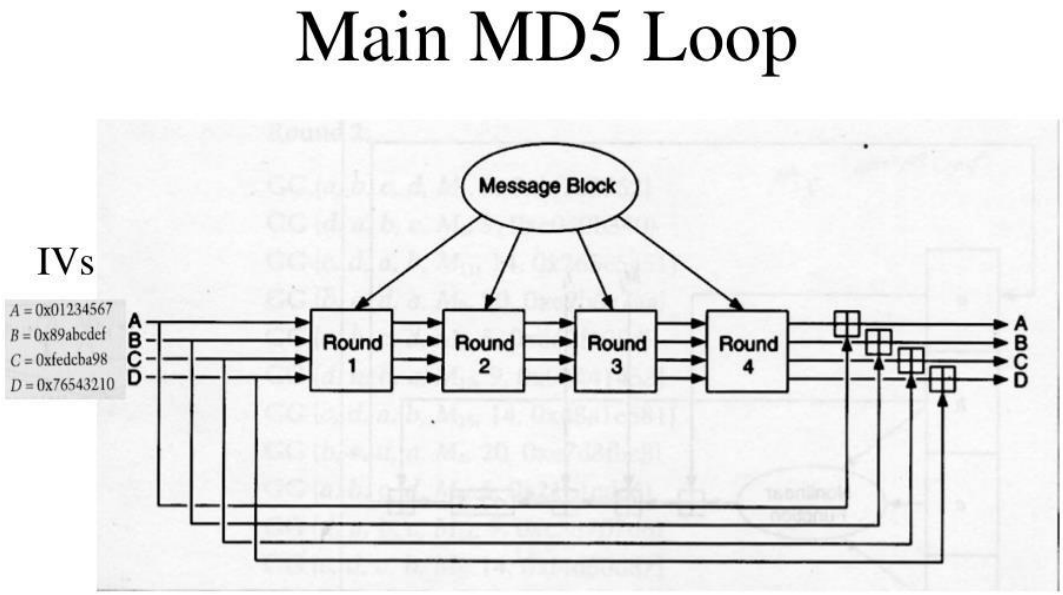
C = 0xfedcba98

D = 0x76543210

These are called chaining variables. Now, the main loop of the algorithm begins. This loop continues for as many 512-bit blocks as are in the message.

The four variables are copied into different variables: a gets A, b gets B, c gets C, and d gets D. The main loop has four rounds (MD4 had only three rounds), all very similar. Each round uses a different operation 16 times. Each operation performs a nonlinear function on three of a, b, c, and d. Then it adds that result to the fourth variable, a sub-block of the text and a constant. Then it rotates that result to the right a variable number of bits and adds the result to one of a, b, c, or d. Finally the result replaces one of a, b, c, or d. See Figures 18.5 and 18.6.

Figure shows MD5 main loop.



There are four nonlinear functions, one used in each operation (a different one for each round).

F(X,Y,Z) = (X ¥ Y) ¦ ((¬ X) ¥ Z)

G(X,Y,Z) = (X ¥ Z) ¬ (Y (¬ Z))

H(X,Y,Z) = X • Y • Z

I(X,Y,Z) = Y • (X ¦ (¬ Z))

(• is XOR,¥ is AND, ¦ is OR, and ¬ is NOT.)

These functions are designed so that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of the result will also be independent and unbiased. The function F is the bit-wise conditional: If X then Y else Z. The function H is the bit-wise parity operator.

If M j represents the j th sub-block of the message (from 0 to 15), and <<<s represents a left circular shift of s bits, the four operations are:

FF(a,b,c,d,Mj,s,ti) denotes a = b + ((a + F(b,c,d) + Mj + ti) <<< s)

GG(a,b,c,d,Mj,s,ti) denotes a = b + ((a + G(b,c,d) + Mj + ti) <<< s)

HH(a,b,c,d,Mj,s,ti) denotes a = b + ((a + H(b,c,d) + Mj + ti) <<< s)

II(a,b,c,d,Mj,s,ti) denotes a = b + ((a + I(b,c,d) + Mj + ti) <<< s)

The four rounds (64 steps) look like:

Round 1:

FF (a, b, c, d, M0, 7, 0xd76aa478)

FF (d, a, b, c, M1, 12, 0xe8c7b756)

FF (c, d, a, b, M2, 17, 0x242070db)

FF (b, c, d, a, M3, 22, 0xc1bdceee)

FF (a, b, c, d, M4, 7, 0xf57c0faf)

FF (d, a, b, c, M5, 12, 0x4787c62a)

FF (c, d, a, b, M6, 17, 0xa8304613)

FF (b, c, d, a, M7, 22, 0xfd469501)

FF (a, b, c, d, M8, 7, 0x698098d8)

FF (d, a, b, c, M9, 12, 0x8b44f7af)

FF (c, d, a, b, M10, 17, 0xffff5bb1)

FF (b, c, d, a, M11, 22, 0x895cd7be)

FF (a, b, c, d, M12, 7, 0x6b901122)

FF (d, a, b, c, M13, 12, 0xfd987193)

FF (c, d, a, b, M14, 17, 0xa679438e)

FF (b, c, d, a, M15, 22, 0x49b40821)

Round 2:

GG (a, b, c, d, M1, 5, 0xf61e2562)

GG (d, a, b, c, M6, 9, 0xc040b340)

GG (c, d, a, b, M11, 14, 0x265e5a51)

GG (b, c, d, a, M0, 20, 0xe9b6c7aa)

GG (a, b, c, d, M5, 5, 0xd62f105d)

GG (d, a, b, c, M10, 9, 0x02441453)

GG (c, d, a, b, M15, 14, 0xd8a1e681)

GG (b, c, d, a, M4, 20, 0xe7d3fbc8)

GG (a, b, c, d, M9, 5, 0x21e1cde6)

GG (d, a, b, c, M14, 9, 0xc33707d6)

GG (c, d, a, b, M3, 14, 0xf4d50d87)

GG (b, c, d, a, M8, 20, 0x455a14ed)

GG (a, b, c, d, M13, 5, 0xa9e3e905)

GG (d, a, b, c, M2, 9, 0xfcefa3f8)

GG (c, d, a, b, M7, 14, 0x676f02d9)

GG (b, c, d, a, M12, 20, 0x8d2a4c8a)

Round 3:

HH (a, b, c, d, M5, 4, 0xfffa3942)

HH (d, a, b, c, M8, 11, 0x8771f681)

HH (c, d, a, b, M11, 16, 0x6d9d6122)

HH (b, c, d, a, M14, 23, 0xfde5380c)

HH (a, b, c, d, M1, 4, 0xa4beea44)

HH (d, a, b, c, M4, 11, 0x4bdecfa9)

HH (c, d, a, b, M7, 16, 0xf6bb4b60)

HH (b, c, d, a, M10, 23, 0xbebfbc70)

HH (a, b, c, d, M13, 4, 0x289b7ec6)

HH (d, a, b, c, M0, 11, 0xeaa127fa)HH (c, d, a, b, M3, 16, 0xd4ef3085)

HH (b, c, d, a, M6, 23, 0x04881d05)

HH (a, b, c, d, M9, 4, 0xd9d4d039)

HH (d, a, b, c, M12, 11, 0xe6db99e5)

HH (c, d, a, b, M15, 16, 0x1fa27cf8)

HH (b, c, d, a, M2, 23, 0xc4ac5665)

Round 4:

II (a, b, c, d, M0, 6, 0xf4292244)

II (d, a, b, c, M7, 10, 0x432aff97)

II (c, d, a, b, M14, 15, 0xab9423a7)

II (b, c, d, a, M5, 21, 0xfc93a039)

II (a, b, c, d, M12, 6, 0x655b59c3)

II (d, a, b, c, M3, 10, 0x8f0ccc92)

II (c, d, a, b, M10, 15, 0xffeff47d)

II (b, c, d, a, M1, 21, 0x85845dd1)

II (a, b, c, d, M8, 6, 0x6fa87e4f)

II (d, a, b, c, M15, 10, 0xfe2ce6e0)

II (c, d, a, b, M6, 15, 0xa3014314)

II (b, c, d, a, M13, 21, 0x4e0811a1)

II (a, b, c, d, M4, 6, 0xf7537e82)

II (d, a, b, c, M11, 10, 0xbd3af235)

II (c, d, a, b, M2, 15, 0x2ad7d2bb)

II (b, c, d, a, M9, 21, 0xeb86d391)

Those constants, t_i, were chosen as follows:

In step i, ti is the integer part of 232*abs(sin(i)), where i is in radians. After all of this, a, b, c, and d are added to A, B, C, D, respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, and D.

Security of MD5

Ron Rivest outlined the improvements of MD5 over MD4 [1322]:

1. A fourth round has been added.

2. Each step now has a unique additive constant.

3. The function G in round 2 was changed from ((X¥ Y) ¦ (X¥ Z) ¦ (Y¥ Z)) to ((X¥ Z) ¦ (Y¥ ¬ Z)) to make G less symmetric.

4. Each step now adds in the result of the previous step. This promotes a faster avalanche effect.

5. The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike.

6. The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds.

Tom Berson attempted to use differential cryptanalysis against a single round of MD5 [144], but his attack is ineffective against all four rounds. A more successful attack by den Boer and Bosselaers produces collisions using the compression function in MD5 [203, 1331, 1336]. This does not lend itself to attacks against MD5 in practical applications, and it does not affect the use of MD5 in Luby-Rackoff-like encryption algorithms (see Section 14.11). It does mean that one of the basic design principles of MD5—to design a collision-resistant compression function—has been violated. Although it is true that "there seems to be a weakness in the compression function, but it has no practical impact on the security of the hash function" [1336], I am wary of using MD5.

5.7 Secure Hash Algorithm (SHA)

NIST, along with the NSA, designed the Secure Hash Algorithm (SHA) for use with the Digital Signature Standard (see Section 20.2) [1154]. (The standard is the Secure Hash Standard (SHS); SHA is the algorithm used in the standard.) According to the Federal Register [539]:

A Federal Information Processing Standard (FIPS) for Secure Hash Standard (SHS) is being proposed. This proposed standard specified a Secure Hash Algorithm (SHA) for use with the proposed Digital Signature Standard .... Additionally, for applications not requiring a digital signature, the SHA is to be used whenever a secure hash algorithm is required for Federal applications And This Standard specifies a Secure Hash Algorithm (SHA), which is necessary to ensure the security of the Digital Signature Algorithm (DSA). When a message of any length < 264 bits is input, the SHA produces a 160-bit output called a message digest. The message digest is then input to the DSA, which computes the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process, because the message digest is usually much smaller than the message. The same message digest should be obtained by the verifier of the signature when the received version of the message is used as input to SHA. The SHA is called secure because it is designed to be computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with a very high probability, result in a different message digest, and the signature will fail to verify. The SHA is based on principles similar to those used by Professor Ronald L. Rivest of MIT when designing the MD4 message digest algorithm [1319], and is closely modelled after that algorithm.

SHA produces a 160-bit hash, longer than MD5.

Description of SHA

First, the message is padded to make it a multiple of 512 bits long. Padding is exactly the same as in MD5: First append a one, then as many zeros as necessary to make it 64 bits short of a multiple of 512, and finally a 64-bit representation of the length of the message before padding. Five 32-bit variables (MD5 has four variables, but this algorithm needs to produce a 160-bit hash) are initialized as follows:

A = 0x67452301

B = 0xefcdab89

C = 0x98badcfe

D = 0x10325476

E = 0xc3d2e1f0

The main loop of the algorithm then begins. It processes the message 512 bits at a time and continues for as many 512-bit blocks as are in the message. First the five variables are copied into different variables: a gets A, b gets B, c gets C, d gets D, and e gets E.

The main loop has four rounds of 20 operations each (MD5 has four rounds of 16 operations each). Each operation performs a nonlinear function on three of a, b, c, d, and e, and then does shifting and adding similar to MD5.

SHA's set of nonlinear functions is:

ft(X,Y,Z) = (X¥ Y) ¦ ((¬ X)¥ Z), for t = 0 to 19.

ft (X,Y,Z) = X • Y • Z, for t = 20 to 39.

ft (X,Y,Z) = (X¥ Y) ¦ (X¥ Z) ¦ (Y¥ Z), for t = 40 to 59.

ft (X,Y,Z) = X • Y • Z, for t = 60 to 79.

Four constants are used in the algorithm:

Kt= 0x5a827999, for t = 0 to 19.

Kt= 0x6ed9eba1, for t = 20 to 39.

Kt= 0x8f1bbcdc, for t = 40 to 59.

Kt= 0xca62c1d6, for t = 60 to 79.

(If you wonder where those numbers came from: 0x5a827999 = 21/2 /4, 0x6ed9eba1 = 31/2 /4, 0x8f1bbcdc = 51/2 /4, and 0xca62c1d6 = 101/2 /4; all times 232.)The message block is transformed from 16 32-bit words (M0 to M15) to 80 32-bit words (W0 to W79) using the following algorithm:

Wt= Mt, for t = 0 to 15

Wt= (Wt- 3 • Wt - 8 • Wt - 14 • Wt - 16) <<< 1, for t = 16 to 79.

(As an interesting aside, the original SHA specification did not have the left circular shift. The change "corrects a technical flaw that made the standard less secure than had been thought" [543]. The NSA has refused to elaborate on the exact nature of the flaw.)

If t is the operation number (from 0 to 79), Wt represents the t th sub-block of the expanded message, and <<< s represents a left circular shift of s bits, then the main loop looks like:

FOR t = 0 to 79

TEMP = (a <<< 5) + ft (b,c,d) + e + Wt + Kt

e = d

d = c

c = b <<< 30

b = a

a = TEMP

Figure shows one operation. Shifting the variables accomplishes the same thing as MD5 does by using different variables in different locations. After all of this, a, b, c, d, and e are added to A, B, C, D, and E respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, D, and E.

Security of SHA

SHA is very similar to MD4, but has a 160-bit hash value. The main changesare the addition of an expand transformation and the addition of the previous step's output into the next step for a faster avalanche effect. Ron Rivest made public the design decisions behind MD5, but SHA's designers did not. Here are Rivest's MD5 improvements to MD4 and how they compare with SHA's:

1. "A fourth round has been added." SHA does this, too. However, in SHA the fourth round uses the same f function as the second round.

2. "Each step now has a unique additive constant." SHA keeps the MD4 scheme where it reuses the constants for each group of 20 rounds.3. "The function G in round 2 was changed from ((X¥ Y) ¦ (X¥ Z) ¦ (Y¥Z)) to ((X¥ Z) ¦ (Y¥ ¬ (Z))) to make G less symmetric." SHA uses the MD4 version: ((X¥ Y) ¦ (X¥ Z) ¦ (Y¥ Z)).

4. "Each step now adds in the result of the previous step. This promotes a faster avalanche effect." This change has been made in SHA as well.

The difference in SHA is that a fifth variable is added, and not b, c, or d, which is already used in ft. This subtle change makes the den Boer-Bosselaers attack against MD5 impossible against SHA.

5. "The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike." SHA is completely different, since it uses a cyclic error-correcting code.

6. "The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds." SHA uses a constant shift amount in each round. This shift amount is relatively prime to the word size, as in MD4.

This leads to the following comparison: SHA is MD4 with the addition of an expand transformation, an extra round, and better avalanche effect; MD5 is MD4 with improved bit hashing, an extra round, and better avalanche effect. There are no known cryptographic attacks against SHA. Because it produces a 160-bit hash, it is more resistant to brute-force attacks (including birthday attacks) than 128-bit hash functions .

SHA 512

SHA-512 is a hashing algorithm that performs a hashing function on some data given to it.

Hashing algorithms are used in many things such as internet security, digital certificates and even blockchains. Since hashing algorithms play such a vital role in digital security and cryptography, this is an easy-to-understand walkthrough, with some basic and simple maths along with some diagrams, for a hashing algorithm called SHA-512. It's part of a group of hashing algorithms called SHA-2 which includes SHA-256 as well which is used in the bitcoin blockchain for hashing.

Before starting with an explanation of SHA-512, I think it would be useful to have a basic idea of what a hashing function's features are.

Hashing Functions

Hashing functions take some data as input and produce an output (called hash digest) of fixed length for that input data. This output should, however, satisfy some conditions to be useful.

1. Uniform distribution: Since the length of the output hash digest is of a fixed length and the input size may vary, it is apparent that there are going to be some output values that can be obtained for different input values. Even though this is the case, the hash function should be such that for any input value, each possible output value should be equally likely. That is to say that every possible output has the same likelihood to be produced for any given input value.
2. Fixed Length: This is should be quite self-explanatory. The output values should all be of a fixed length. So, for example, a hashing function could have an output size of 20 characters or 12 characters, etc. SHA-512 has an output size of 512 bits.
3. Collision resistance: Simply speaking, this means that there aren't any or rather it is not feasible to find two distinct inputs to the hash function that result in the same output (hash digest).

That's a simple introduction about hash functions. Now let's look at SHA-512.

Hashing Algorithm — SHA-512

So, SHA-512 does its work in a few stages. These stages go as follows:

1. Input formatting
2. Hash buffer initialization
3. Message Processing
4. Output

Let's look at these one-by-one.

Input Formatting:

SHA-512 can't actually hash a message input of any size, i.e. it has an input size limit. This limit is imposed by its very structure as you may see further on. The entire formatted mesage has basically three parts: the original message, padding bits, size of original message. And this should all have a combined size of a whole multiple of 1024 bits. This is because the formatted message will be processed as blocks of 1024 bits each, so each bock should have 1024 bits to work with.

<pic: original message>

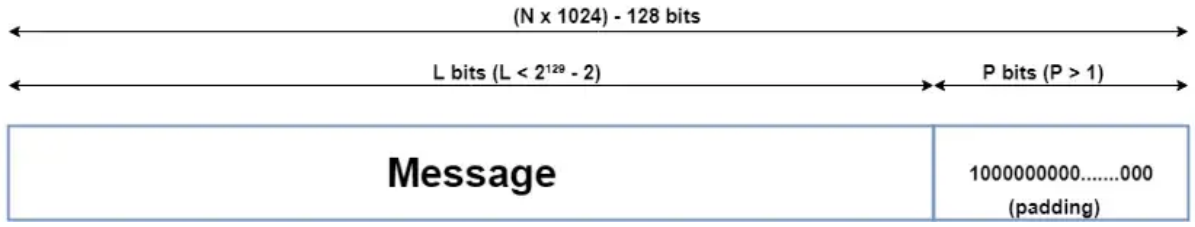
Original message

Padding bits

The input message is taken and some padding bits are appended to it in order to get it to the desired length. The bits that are used for padding are simply '0' bits with a leading '1' (100000…000). Also, according to the algorithm, padding *needs* to be done, even if it is by one bit. So a single padding bit would only be a '1'.

The total size should be equal to 128 bits short of a multiple of 1024 since the goal is to have the formatted message size as a multiple of 1024 bits (N x 1024).

<pic: msg + pad>



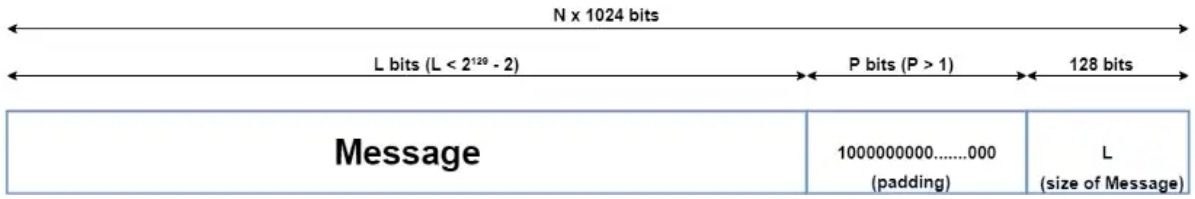
Message with padding

Padding size

After this, the size of the original message given to the algorithm is appended. This size value needs to be represented in 128 bits and is the only reason that the SHA-512 has a limitation for its input message.

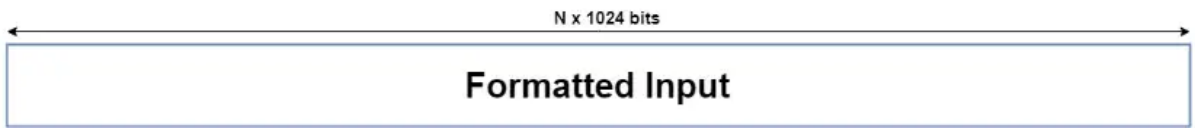
Since the size of the original message needs to be represented in 128 bits and the largest number that can be represented using 128 bits is (2^{128}-1), the message size can be at most (2^{128}-1) bits; and also taking into consideration the necessary single padding bit, the maximum size for the original message would then be (2^{128}-2). Even though this limit exists, it doesn't actually cause a problem since the actual limit is so high (2^{128}-2 = 340,282,366,920,938,463,463,374,607,431,768,211,454 bits).

<pic: msg + pad +size>



Message with padding and size

Now that the padding bits and the size of the message have been appended, we are left with the completely formatted input for the SHA-512 algorithm.



Formatted Message

2. Hash buffer initialization:

The algorithm works in a way where it processes each block of 1024 bits from the message using the result from the previous block. Now, this poses a problem for the first 1024 bit block which can't use the result from any previous processing. This problem can be solved by using a default value to be used for the first block in order to start off the process. (Have a look at the second-last diagram).

Since each intermediate result needs to be used in processing the next block, it needs to be stored somewhere for later use. This would be done by the *hash buffer*, this would also then hold the final hash digest of the entire processing phase of SHA-512 as the last of these 'intermediate' results.

Initialization Vector

a = 0x6A09E667F3BCC908	b = 0xBB67AE8584CAA73B
c = 0x3C6EF372FE94F82B	d = 0xA54FF53A5F1D36F1
e = 0x510E527FADE682D1	f = 0x9B05688C2B3E6C1F
g = 0x1F83D9ABFB41BD6B	h = 0x5BE0CD19137E2179

<pic: IV> Hash buffer and Initialization Vector values

So, the default values used for starting off the chain processing of each 1024 bit block are also stored into the hash buffer at the start of processing. The actual value used is of little consequence, but for those interested, the values used are obtained by taking the first 64 bits of the fractional parts of the square roots of the first 8 prime numbers (2,3,5,7,11,13,17,19). These values are called the Initial Vectors (IV).

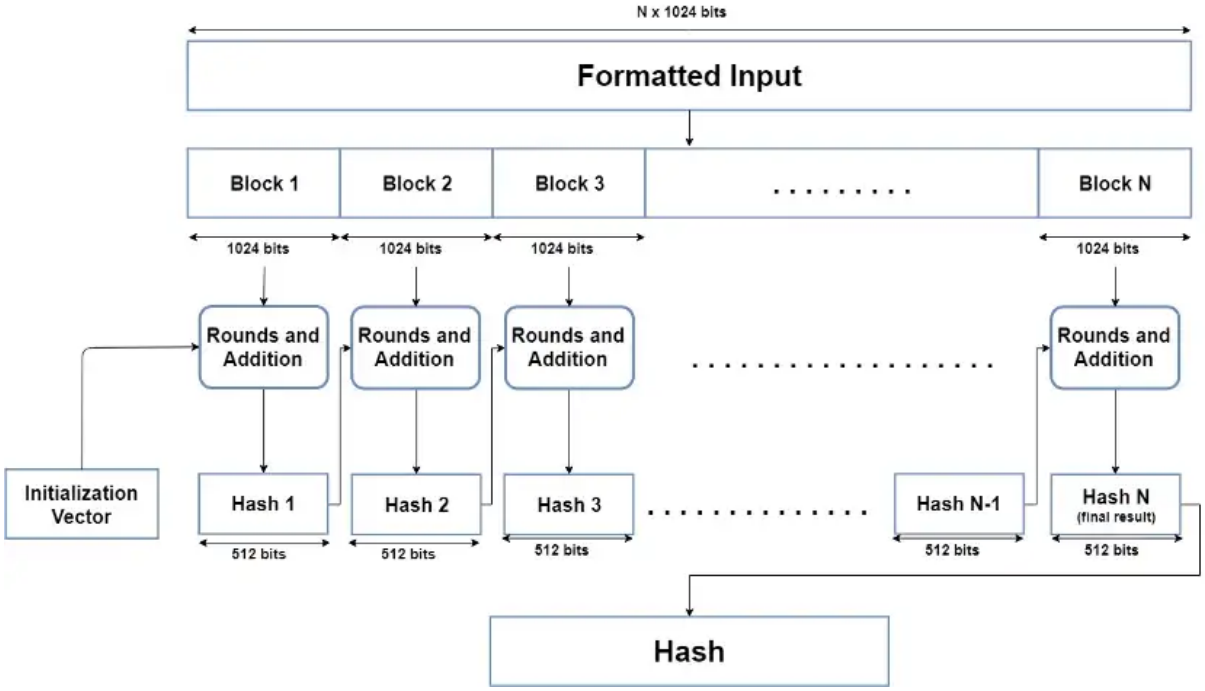
Why 8 prime numbers instead of 9? Because the hash buffer actually consists of 8 subparts (registers) for storing them.

3. Message Processing:

Message processing is done upon the formatted input by taking one block of 1024 bits at a time. The actual processing takes place by using two things: The 1024 bit block, and the result from the previous processing.

This part of the SHA-512 algorithm consists of several 'Rounds' and an addition operation.

<pic: Formatted input 1024 bit blocks;F(M.n ,H.n-1)=H.n>



William Stallings, Cryptography and Network Security — Principles and Practise (Seventh Edition) referred for diagram

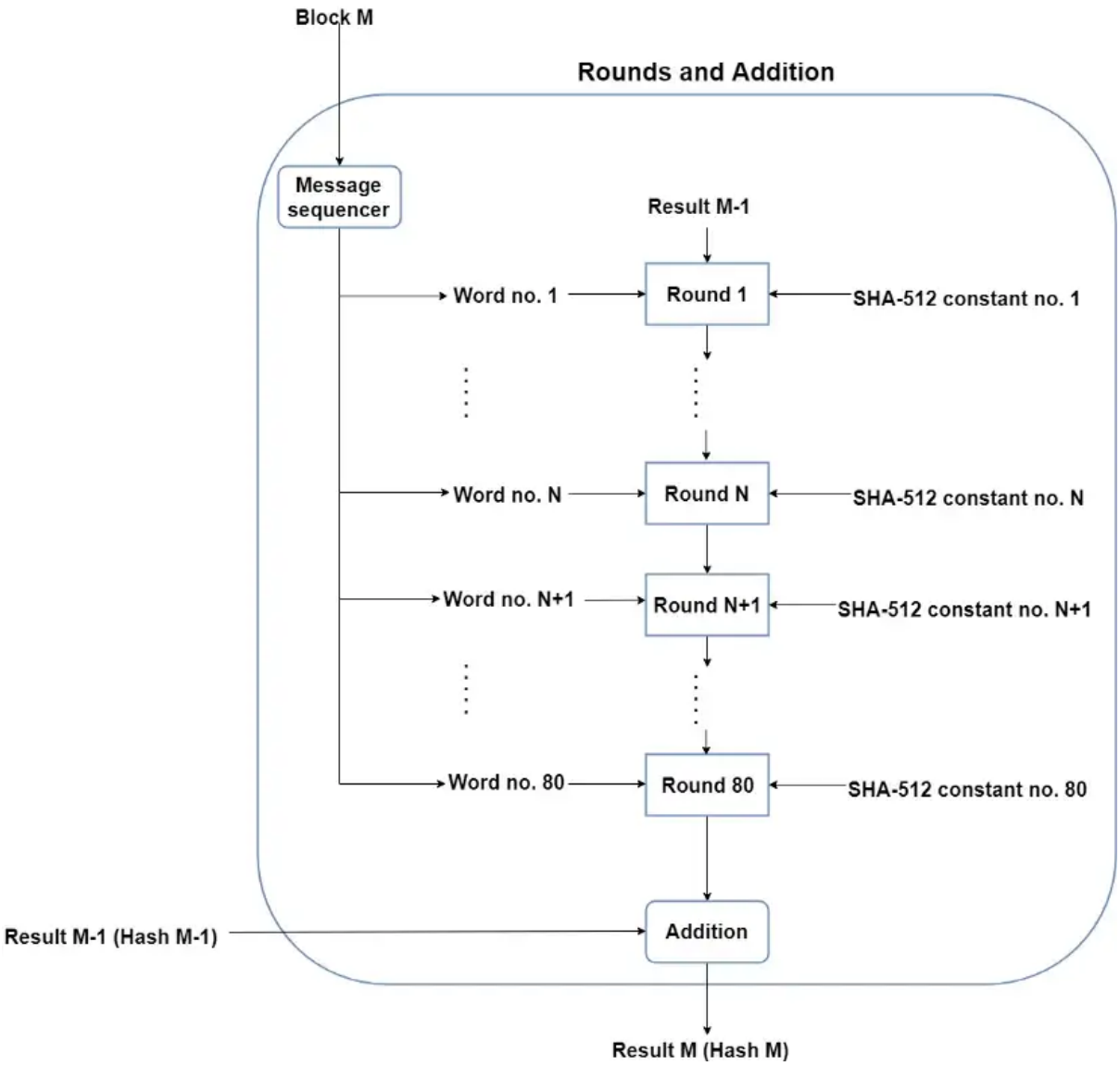
So, the Message block (1024 bit) is expanded out into ‘Words’ using a ‘message sequencer’. Eighty Words to be precise, each of them having a size of 64 bits.

Rounds

The main part of the message processing phase may be considered to be the Rounds. Each round takes 3 things: one Word, the output of the previous Round, and a SHA-512 constant. The first Round doesn’t have a previous Round whose output it can use, so it uses the final output from the previous message processing phase for the previous block of 1024 bits. For the first Round of the first block (1024 bits) of the formatted input, the Initial Vector (IV) is used.

SHA-512 constants are predetermined values, each of whom is used for each Round in the message processing phase. Again, these aren’t very important, but for those interested, they are the first 64 bits from the fractional part of the cube roots of the first 80 prime numbers.Why 80? Because there are 80 Rounds and each of them needs one of these constants.

Once the Round function takes these 3 things, it processes them and gives an output of 512 bits. This is repeated for 80 Rounds. After the 80th Round, its output is simply added to the result of the previous message processing phase to get the final result for this iteration of message processing.



4. Output:

After every block of 1024 bits goes through the message processing phase, i.e. the last iteration of the phase, we get the final 512 bit Hash value of our original message. So, the intermediate results are all used from each block for processing the next block. And when the final 1024 bit block has finished being processed, we have with us the final result of the SHA-512 algorithm for our original message.

Thus, we obtain the final hash value from our original message. The SHA-512 is part of a group of hashing algorithms that are very similar in how they work, called SHA-2. Algorithms such as SHA-256 and SHA-384 are a part of this group alongside SHA-512. SHA-256 is also used in the Bitcoin blockchain as the designated hash function.

5.8.1 MACs BASED ON HASH FUNCTIONS: HMAC

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 pre packaged 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)

Yi = *i* th block of M, 0 … *i* … (*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 Ú *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 left so that the result is *b* bits in length

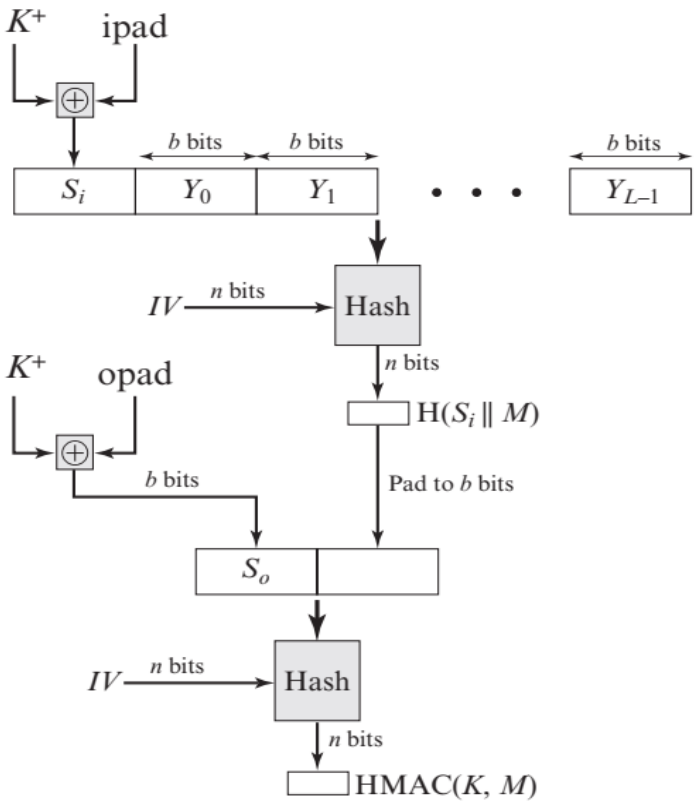


Figure 12.5 HMAC Structure

ipad = 00110110 (36 in hexadecimal) repeated *b*/8 times

opad = 01011100 (5C in hexadecimal) repeated *b*/8 times

Then HMAC can be expressed as

HMAC(K, M) = H[(K+ $\oplus$ opad) } H[(K+ $\oplus$ 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 Si.
3. Append M to Si.
4. apply H to the stream generated in step 3.
5. XOR K+ with opad to produce the b-bit block So.
6. Append the hash result from step 4 to So.
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 Si and So 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 Si, So, and the block produced from the inner hash). A more efficient implementation is possible, as shown in Figure 12.6. Two quantities are precomputed:

f(IV, (K+ $\oplus$ ipad))

f(IV, (K+ $\oplus$ 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 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: $H(M) = H(M')$. 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 264 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 264 observed blocks (272 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 150,000 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.

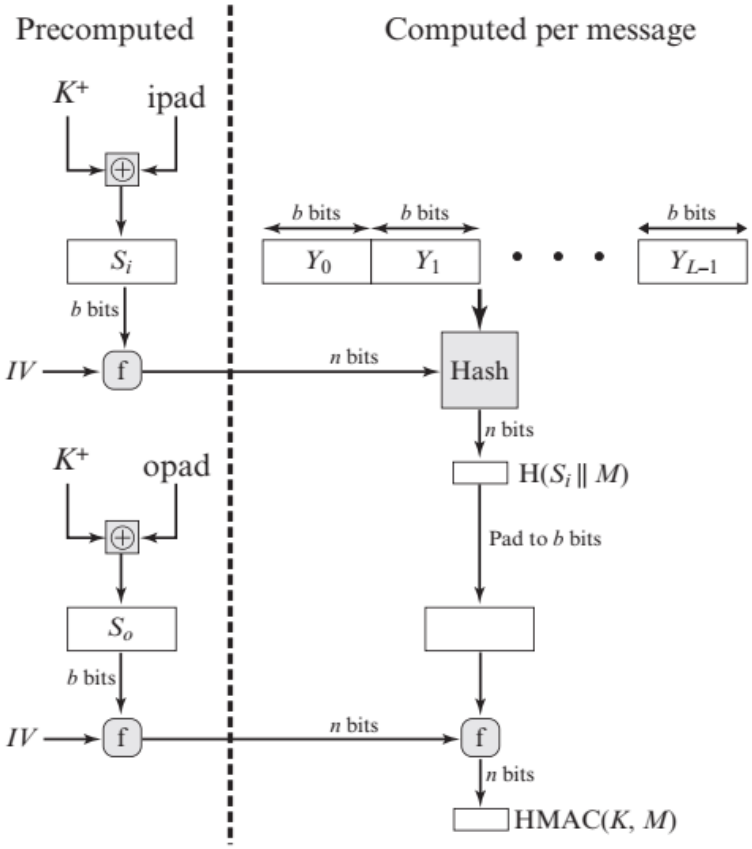


Figure 12.6 Efficient Implementation of HMAC

5.8.2 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, c , *DN*. 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).
$O1 = E(K, D)$

$O2 = E(K, [D2 \oplus O1])$

$O3 = E(K, [D3 \oplus O2])$

.

.

$O_N = E(K, [DN \oplus ON - 1])$

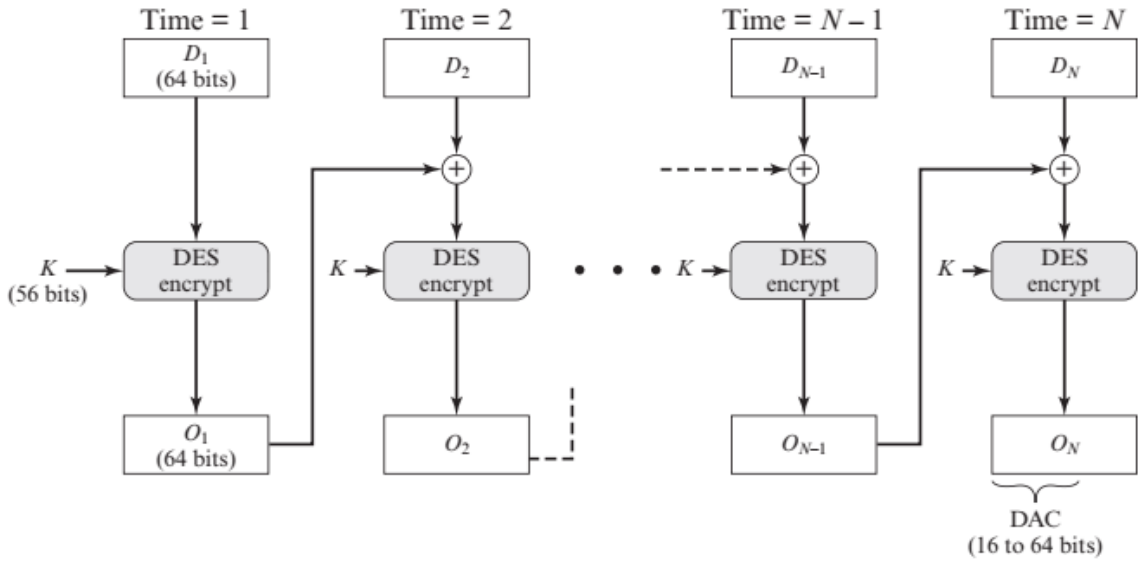


Figure 12.7 Data Authentication Algorithm (FIPS PUB 113)

The DAC consists of either the entire block ON or the leftmost M bits of the block, with 16 … M … 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 $T = MAC(K, X)$, the adversary immediately knows the CBC MAC for the twoblock 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 ($M1$, $M2$, c , Mn). The algorithm makes use of a k-bit encryption key K and a b-bit constant, $K1$. 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).

$C1 = E(K, M1)$

$C2 = E(K, [M2 \oplus C1])$

$C3 = E(K, [M3 \oplus C2])$

.

.

$Cn = E(K, [Mn \oplus Cn - 1 \oplus K1])$

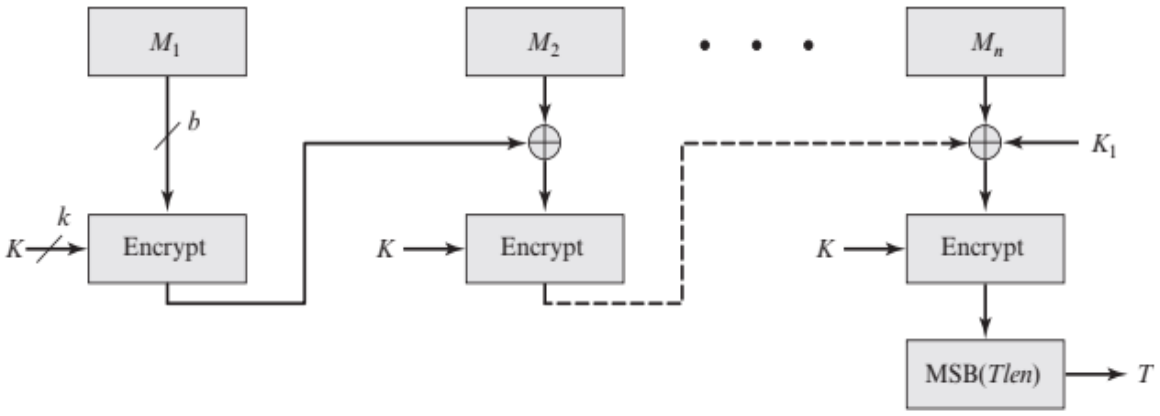
$T = MSB_{Tlen}(Cn)$

where

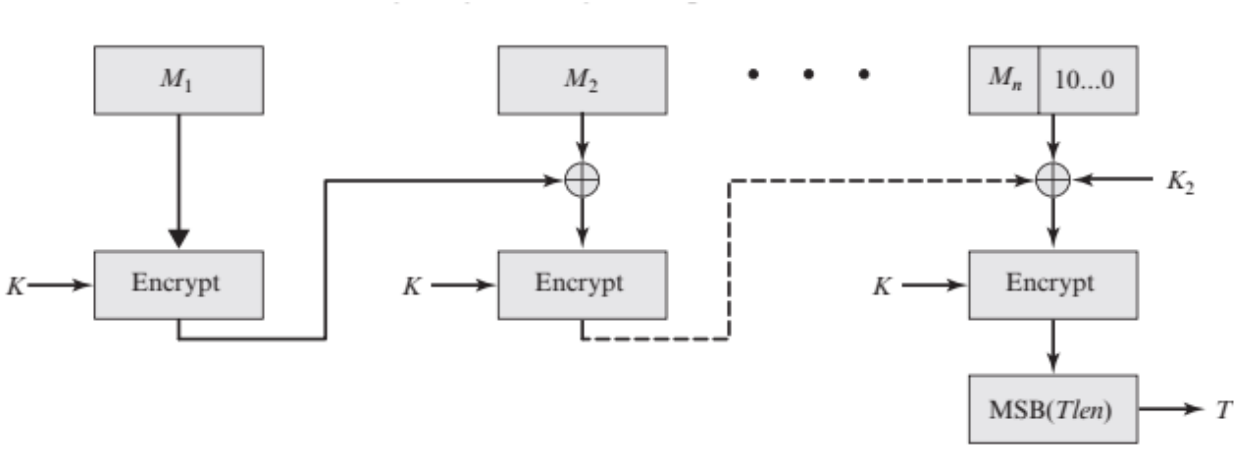
T	= message authentication code, also referred to as the tag
$Tlen$	= bit length of T

$MSB_s(X)$ = the s leftmost bits of the bit string X

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 $K2$ is used instead of $K1$.



(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.
$L = \mathrm{E}(K, 0b)$

$K1 = L \# x$

$K2 = L \# x2 = (L \# x) \# x$

where multiplication (#) is done in the finite field GF(2b) and x and $x2$ are first- and second-order polynomials that are elements of GF(2b). Thus, the binary representation of x consists of b - 2 zeros followed by 10; the binary representation of $x2$ 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 $x64 + x4 + x3 + x + 1$ and $x128 + x7 + x2 + x + 1$. To generate $K1$ and $K2$, 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(2b)

5.9 X.509 Authentication Service

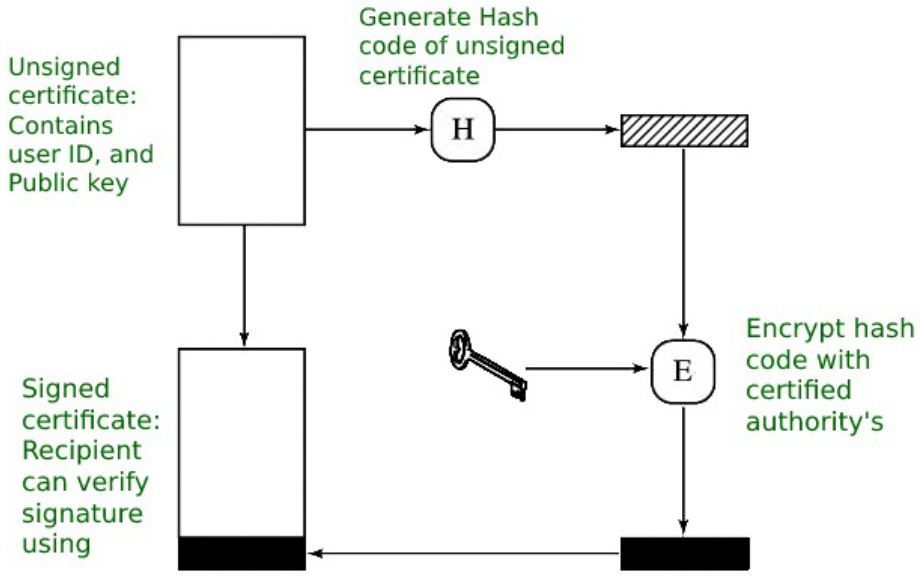
X.509 is a digital certificate that is built on top of a widely trusted standard known as ITU or International Telecommunication Union X.509 standard, in which the format of PKI certificates is defined. X.509 digital certificate is a certificate-based authentication security framework that can be used for providing secure transaction processing and private information. These are primarily used for handling the security and identity in computer networking and internet-based communications.

Working of X.509 Authentication Service Certificate:

The core of the X.509 authentication service is the public key certificate connected to each user. These user certificates are assumed to be produced by some trusted certification authority and positioned in the directory by the user or the certified authority. These directory servers are only used for providing an effortless reachable location for all users so that they can acquire

certificates. X.509 standard is built on an IDL known as ASN.1. With the help of Abstract Syntax Notation, the X.509 certificate format uses an associated public and private key pair for encrypting and decrypting a message.

Once an X.509 certificate is provided to a user by the certified authority, that certificate is attached to it like an identity card. The chances of someone stealing it or losing it are less, unlike other unsecured passwords. With the help of this analogy, it is easier to imagine how this authentication works: the certificate is basically presented like an identity at the resource that requires authentication.

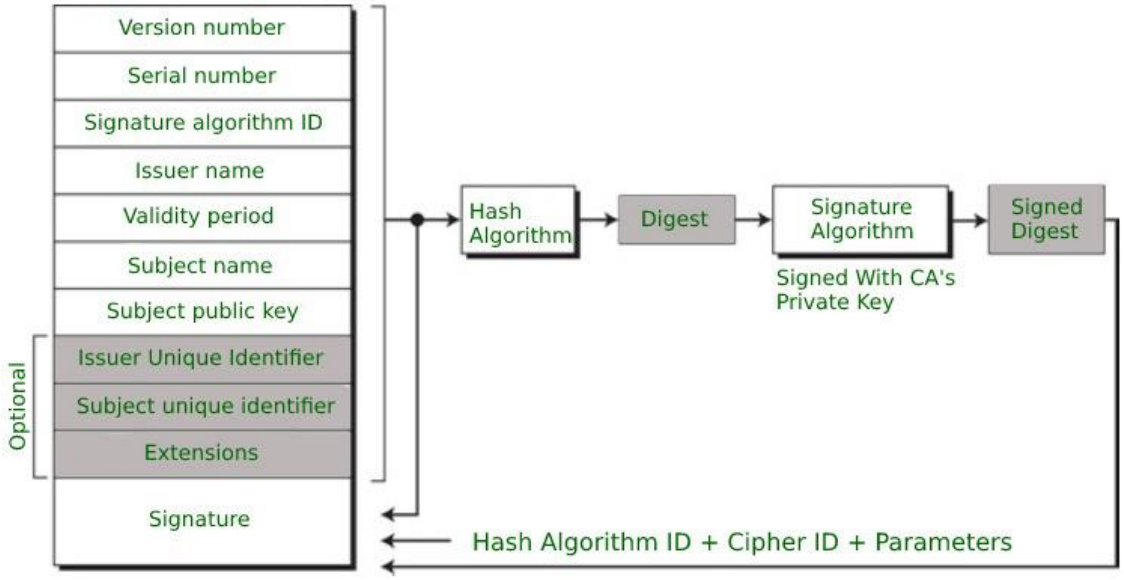


Format of X.509 Authentication Service Certificate:

Generally, the certificate includes the elements given below:

- **Version number:** It defines the X.509 version that concerns the certificate.
- **Serial number:** It is the unique number that the certified authority issues.
- **Signature Algorithm Identifier:** This is the algorithm that is used for signing the certificate.
- **Issuer name:** Tells about the X.500 name of the certified authority which signed and created the certificate.
- **Period of Validity:** It defines the period for which the certificate is valid.
- **Subject Name:** Tells about the name of the user to whom this certificate has been issued.
- **Subject's public key information:** It defines the subject's public key along with an identifier of the algorithm for which this key is supposed to be used.

- **Extension block:** This field contains additional standard information.
- **Signature:** This field contains the hash code of all other fields which is encrypted by the certified authority private key.



Applications of X.509 Authentication Service Certificate:

Many protocols depend on X.509 and it has many applications, some of them are given below:

- Document signing and Digital signature
- Web server security with the help of Transport Layer Security (TLS)/Secure Sockets Layer (SSL) certificates
- Email certificates
- Code signing
- Secure Shell Protocol (SSH) keys
- Digital Identities