# Module 3

Message Authentication-Requirements- Authentication functions- Message authentication codes- Hash functions- Secure Hash Algorithm, MD5, Digital signatures- protocols- Digital signature standards, Digital Certificates.
Application Level Authentications- Kerberos, X.509 Authentication Service, X.509 certificates.

## 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 by (i.e., contain no modification, insertion, deletion, or replay) and that the purported identity of the sender is valid.

## Authentication Requirements

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

- **Disclosure:** Release of message contents to any person or process not possessing the appropriate cryptographic key.
- **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.
- **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 non receipt by someone other than the message recipient.
- **Content modification:** Changes to the contents of a message, including insertion, deletion, transposition, and modification.
- **Sequence modification:** Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.

- **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.

- **Source repudiation:** Denial of transmission of message by source.

- **Destination repudiation:** Denial of receipt of message by destination.

Measures to deal with the first two attacks are in the realm of message confidentiality. 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.

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.

**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 1ow 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. Three types of functions can be used as an authenticator.

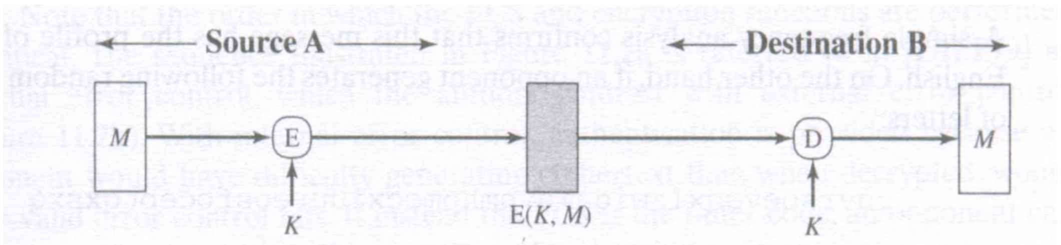
- **Message encryption:** The cipher text of the entire message serves as authenticator

- **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 function:** A function that maps a message of any length into a fixed length hash value, which serves as the authenticator

## Message Encryption

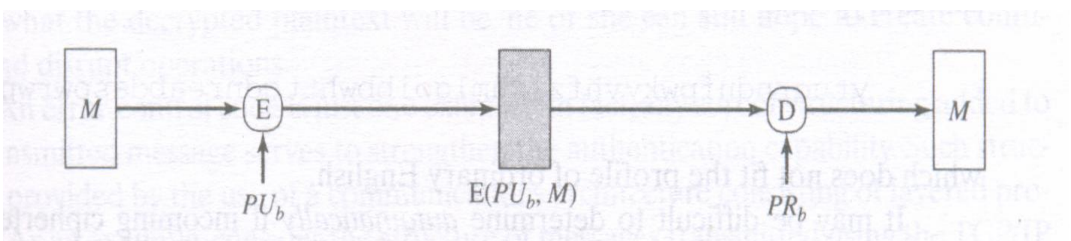
Consider a Message M is transmitted from source A to destination B.it is encrypted using key K, which is shared by A and B. if no one else knows the key confidentiality is provided.



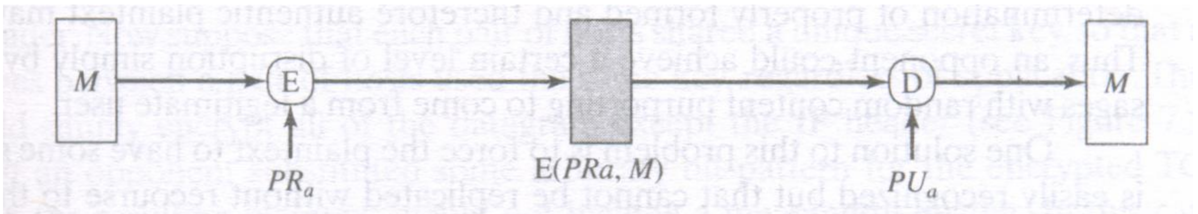
(a) Symmetric encryption: confidentiality and authentication

### Public Key encryption

The use of public key provides only confidentiality. The source A uses the public key $PU_b$ of the destination B to encrypt the message 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, claiming to be A.



(b) Public-key encryption: confidentiality



(c) Public-key encryption: authentication and signature

To provide authentication, A uses its private key to encrypt the message and B uses' A's public key to decrypt the message. It ensures that the message has come from A because A is the only party that possess $PR_a$ and therefore the only party with the information necessary to construct cipher text that can be decrypted with $PU_a$.

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.

**Message Authentication Codes**

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)

*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. 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

- 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 MAC will differ from the received

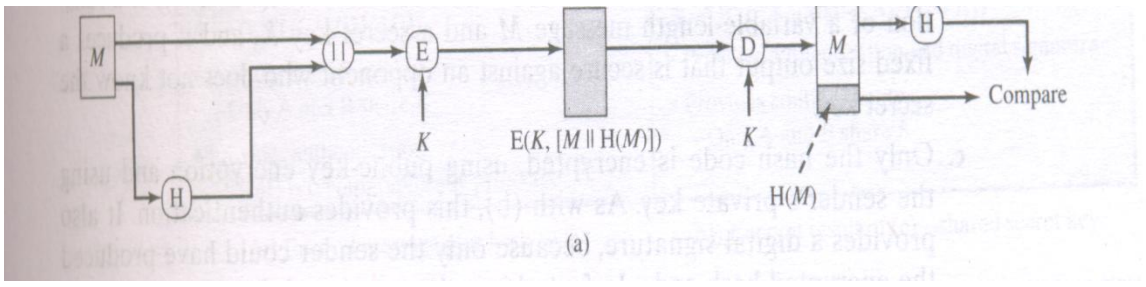
MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to correspond to the alterations in the message.

- 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.
- If the message includes a sequence number, then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

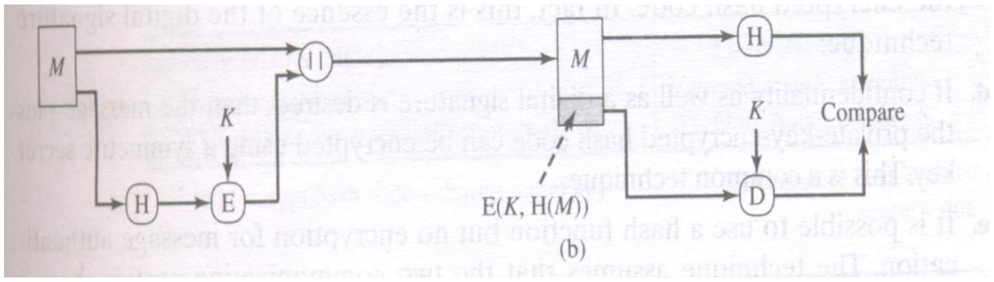
**Hash Functions**

As with MAC, a hash function accepts a variable sized message M as input and produces a fixed sized output referred as hash code H(M). it is a function to the input message. It is also referred as hash value or message digests. The hash code is a function of all the bits of the message and provides an error detection capability. A change to any bits or bits in the message results in a change to the hash code.

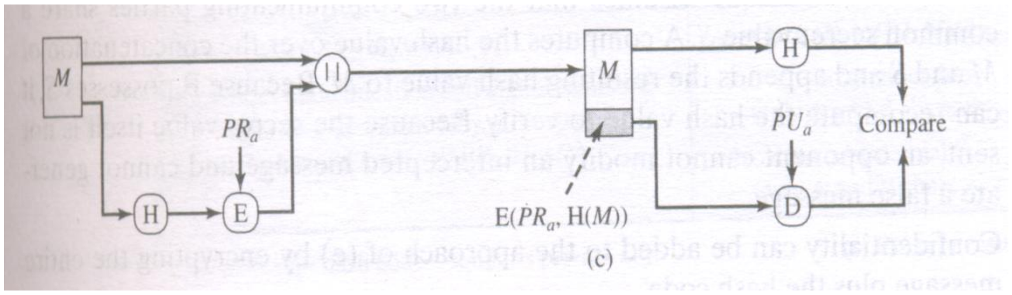
Figure illustrates a variety of ways in which a hash code can be used provide message authentication.



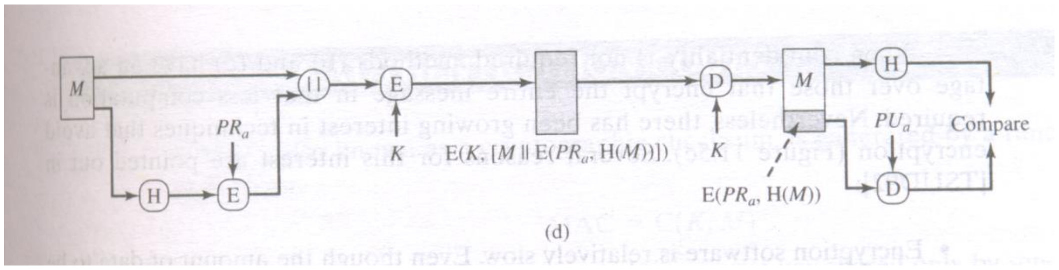
The message plus concatenated hash code is encrypted using symmetric encryption. Only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication. Because encryption is applied to the entire message plus hash code, confidentiality is also provided.



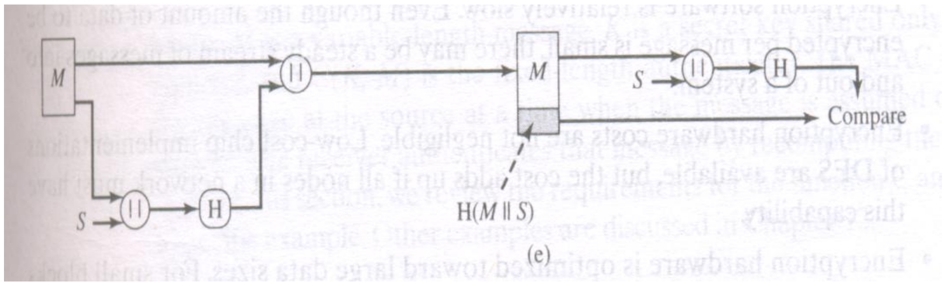
Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality. Note that the combination of hashing and encryption results in an overall function that is, in fact, a MAC. That is, E (K, H(M)) is a function of 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.



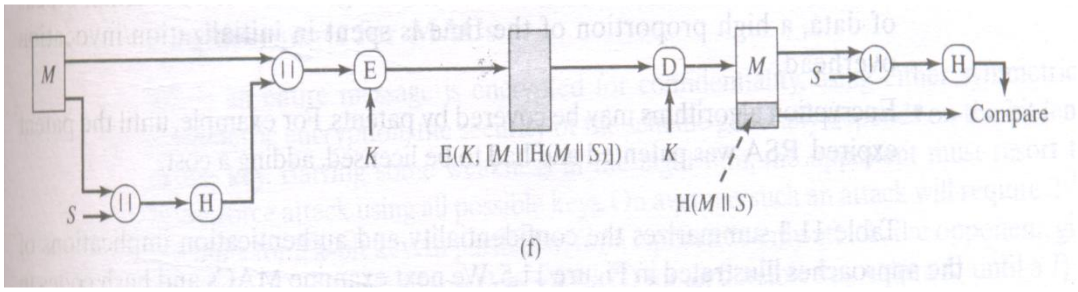
Only the hash code is encrypted, using public-key encryption and using the sender’s private key. It provides authentication and a digital signature, because only the sender could have p the encrypted hash code. In fact, this is the essence of the digital s technique.



If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.



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 recomputed 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.



Confidentiality can be added to the approach of (e) by encrypting the entire message plus the hash code.

## Secure Hash Algorithm

The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard. SHA-1 produces a hash value of 160 bits. NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256 , SHA-384 , and SHA-512. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. Comparison of different versions of SHA algorithm is shown in the table below.

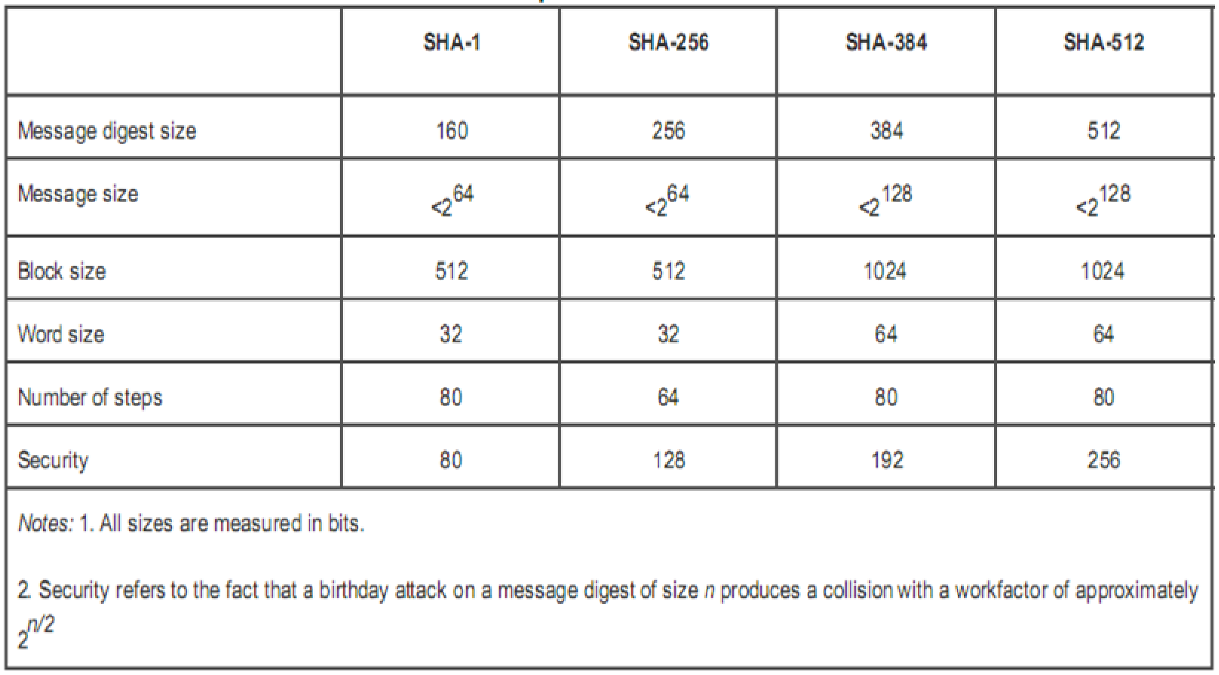


Table . Comparison of SHA Parameters

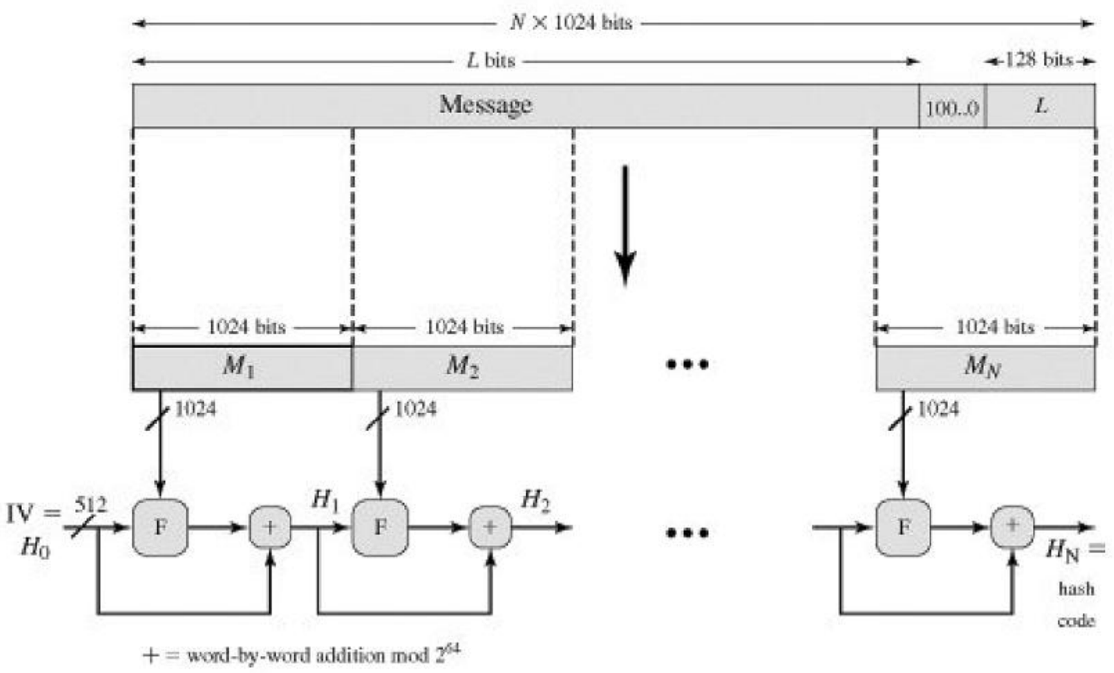
| | SHA-1 | SHA-256 | SHA-384 | SHA-512 |
|---|---|---|---|---|
| Message digest size | 160 | 256 | 384 | 512 |
| Message size | $<2^{64}$ | $<2^{64}$ | $<2^{128}$ | $<2^{128}$ |
| Block size | 512 | 512 | 1024 | 1024 |
| Word size | 32 | 32 | 64 | 64 |
| Number of steps | 80 | 64 | 80 | 80 |
| Security | 80 | 128 | 192 | 256 |

*Notes:* 1. All sizes are measured in bits.

2. Security refers to the fact that a birthday attack on a message digest of size *n* produces a collision with a workfactor of approximately $2^{n/2}$

SHA-512 algorithm takes as input a message with a maximum length of less than $2^{128}$ bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks depicts the overall processing of a message to produce a digest. Below figure shows the processing of input blocks and produces the output message digest.

**Figure 1. Message Digest Generation Using SHA-512**

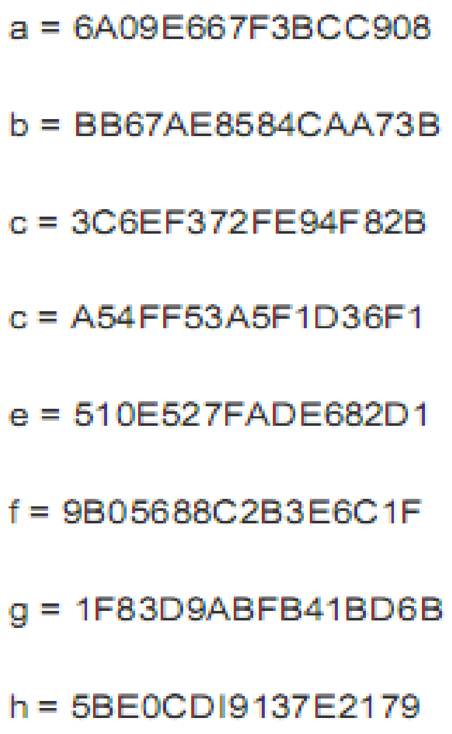


The processing consists of the following steps:-

Step 1: Append padding bits. The message is padded so that its length is congruent to 896 modulo 1024 [length ≡ 896 (mod 1024)]. Padding is always added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

Step 2: Append length. A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding). The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. Above figure, the expanded message is represented as the sequence of 1024-bit blocks $M_1$, $M_2$... $M_N$, so that the total length of the expanded message is N x 1024 bits.

Step 3: Initialize hash buffer. A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following 64-bit integers (hexadecimal values):



a = 6A09E667F3BCC908

b = BB67AE8584CAA73B

c = 3C6EF372FE94F82B

c = A54FF53A5F1D36F1

e = 510E527FADE682D1

f = 9B05688C2B3E6C1F

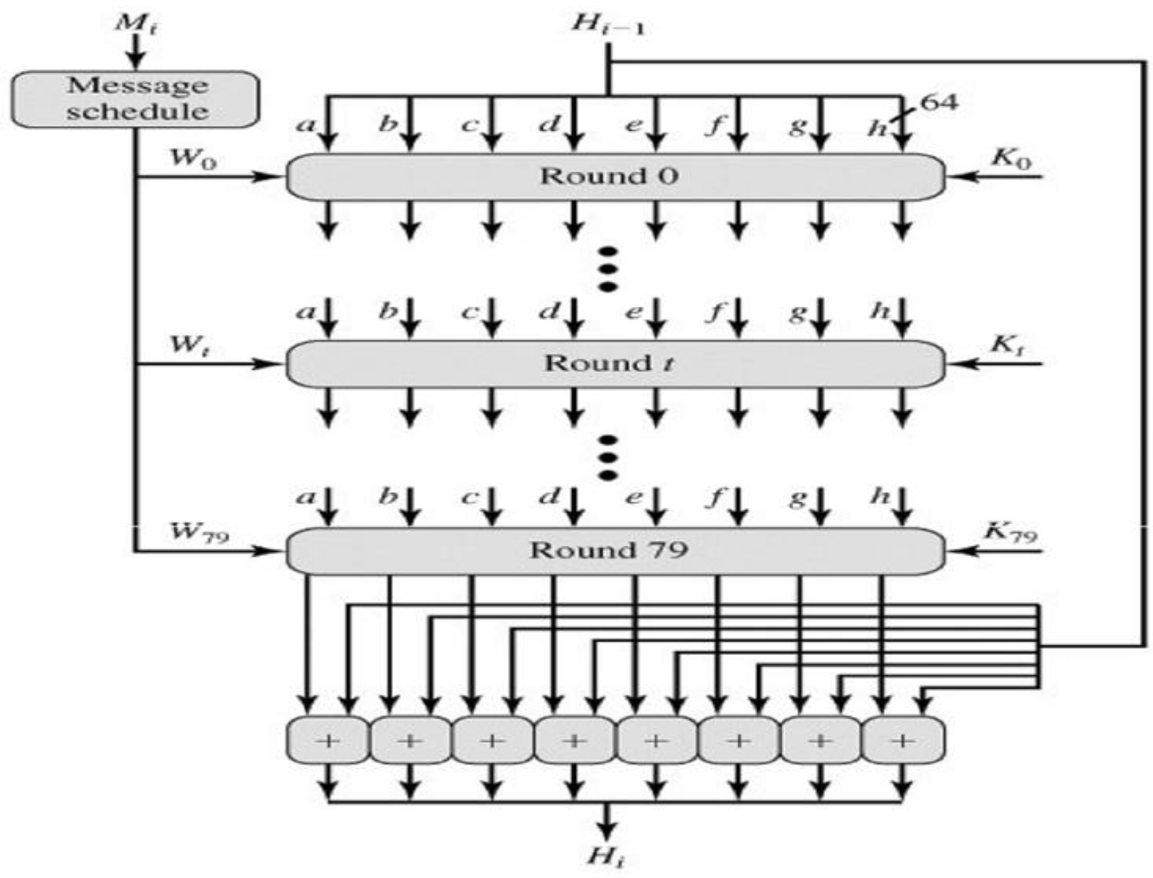
g = 1F83D9ABFB41BD6B

h = 5BE0CDI9137E2179

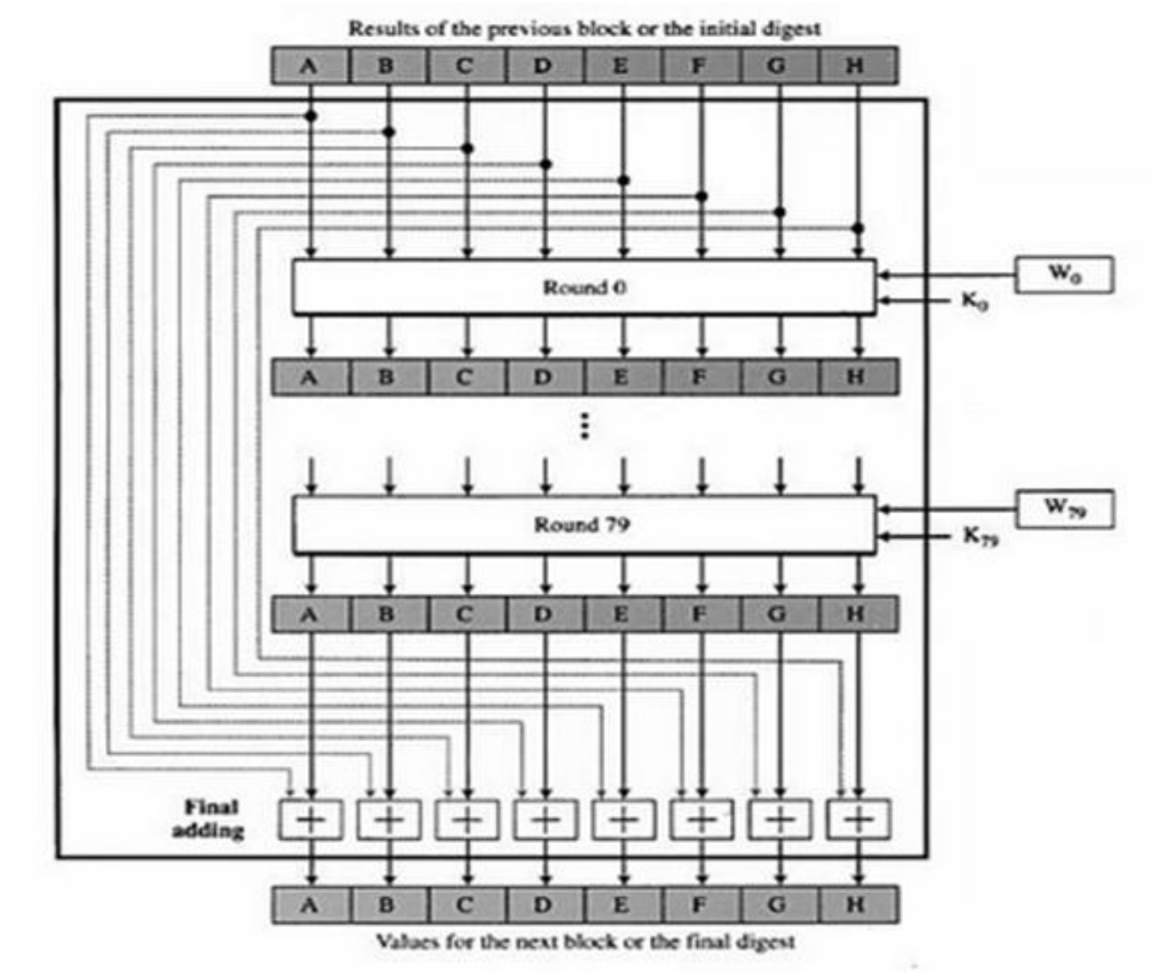
These values are stored in big-endian format, which is the most significant byte of a word in the low-address (leftmost) byte position. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

Step 4: Process message in 1024-bit (128-word) blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in above figure. The logic is illustrated in below figure.

Figure .2. SHA-512 Processing of a Single 1024-Bit Block



Each round takes as input the 512-bit buffer value abcdefgh, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, $H_{i-1}$. Each round t makes use of a 64-bit value $W_t$ derived from the current 1024-bit block being processed ($M_i$). These values are derived using a message schedule described subsequently. Each round also makes use of an additive constant $K_t$ where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularity in the input data. The output of the eightieth round is added to the input to the first round ($H_{i-1}$) to produce $H_i$. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in $H_{i-1}$ using addition modulo $2^{64}$.



Results of the previous block or the initial digest
A B C D E F G H
Round 0
$W_0$
$K_0$
Round 79
$W_{79}$
$K_{79}$
Final adding
Values for the next block or the final digest

Step 5: Output. After all N 1024-bit blocks have been processed; the output from the N-th stage is the 512-bit message digest.

We can summarize the behavior of SHA-512 as follows:

$$H_0 = IV$$

$$H_i = \mathrm{SUM}_{64}(H_{i-1}, abcdefgh_i)$$

$$MD = H_N$$

where

| | |
|---|---|
| IV | = initial value of the abcdefgh buffer, defined in step 3 |
| $abcdefgh_i$ | = the output of the last round of processing of the $i$th message block |
| N | = the number of blocks in the message (including padding and length fields) |
| $SUM_{64}$ | = Addition modulo $2^{64}$ performed separately on each word of the pair of inputs |
| *MD* | = final message digest value |

SHA-512 Round Function

The logic in each of the 80 steps of the processing of one 512-bit block is shown in the below figure. Each round is defined by the following set of equations:

$$
\begin{aligned}
T_1 &= h + \mathrm{Ch}(e, f, g) + \left(\textstyle\sum_1^{512} e\right) + W_t + K_t \\
T_2 &= \left(\textstyle\sum_0^{512} a\right) + \mathrm{Maj}(a, b, c) \\
a &= T_1 + T_2 \\
b &= a \\
c &= b \\
d &= c \\
e &= d + T_1 \\
f &= e \\
g &= f \\
h &= g
\end{aligned}
$$

where

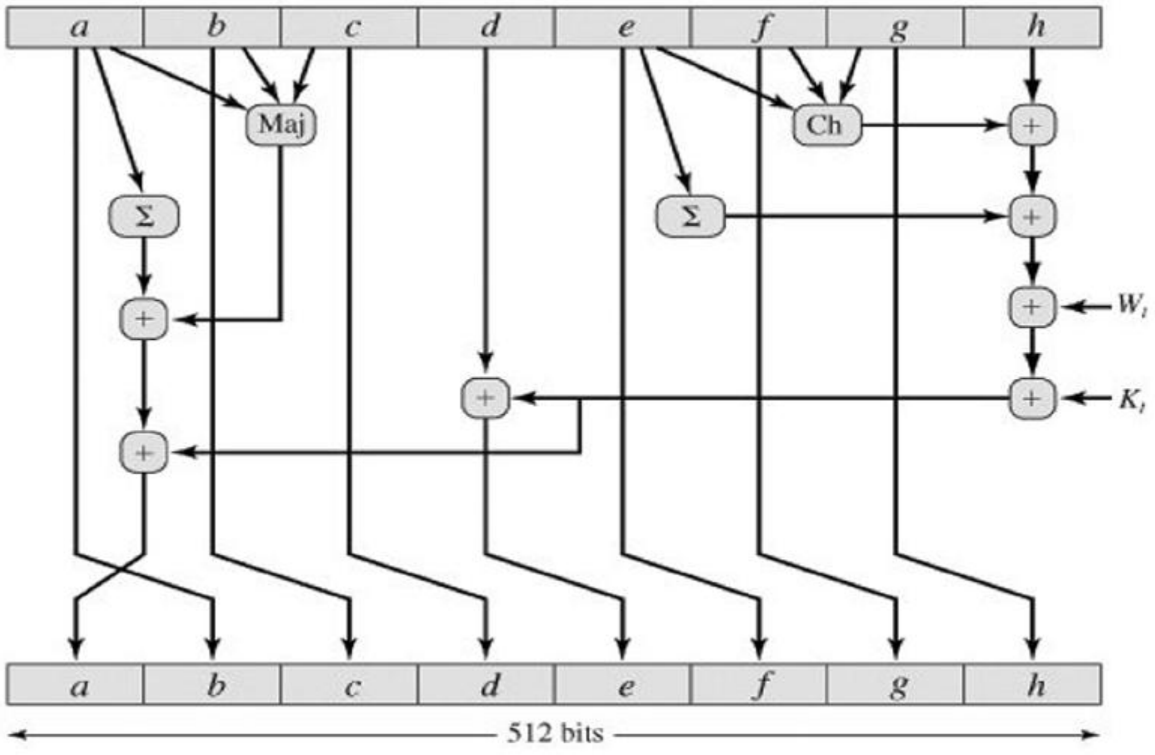
| | |
|---|---|
| $t$ | = step number; $0 \leq t \leq 79$ |
| Ch($e$, $f$, $g$) | = ($e$ AND $f$) $\oplus$ (NOT $e$ AND $g$) *the conditional function: If* $e$ *then* $f$ *else* $g$ |
| Maj($a$, $b$, $c$) | = ($a$ AND $b$) $\oplus$ ($a$ AND $c$) $\oplus$ ($b$ AND $c$) *the function is true only of the majority (two or three) of the arguments are true.* |

$$\left(\sum_0^{512} a\right) = \mathrm{ROTR}^{28}(a) \oplus \mathrm{ROTR}^{34}(a) \oplus \mathrm{ROTR}^{39}(a)$$

$$\left(\sum_1^{512} e\right) = \mathrm{ROTR}^{14}(e) \oplus \mathrm{ROTR}^{18}(e) \oplus \mathrm{ROTR}^{41}(e)$$

| | |
|---|---|
| $\mathrm{ROTR}^n(x)$ | = circular right shift (rotation) of the 64-bit argument *x* by *n* bits |
| $W_t$ | = a 64-bit word derived from the current 512-bit input block |
| $K_t$ | = a 64-bit additive constant |
| + | = addition modulo $2^{64}$ |

Figure 3. Elementary SHA-512 Operation (single round)



$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

where

$$\sigma_0^{512}(x) = \mathrm{ROTR}^1(x) \oplus \mathrm{ROTR}^8(x) \oplus \mathrm{SHR}^7(x)$$
$$\sigma_1^{512}(x) = \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^6(x)$$

| | |
|---|---|
| $\mathrm{ROTR}^n(x)$ | = circular right shift (rotation) of the 64-bit argument *x* by *n* bits |
| $\mathrm{SHR}^n(x)$ | = left shift of the 64-bit argument *x* by *n* bits with padding *by zeros on the right* |

**Figure .4. Creation of 80-word Input Sequence for SHA-512 Processing of Single Block**

[View full size image]

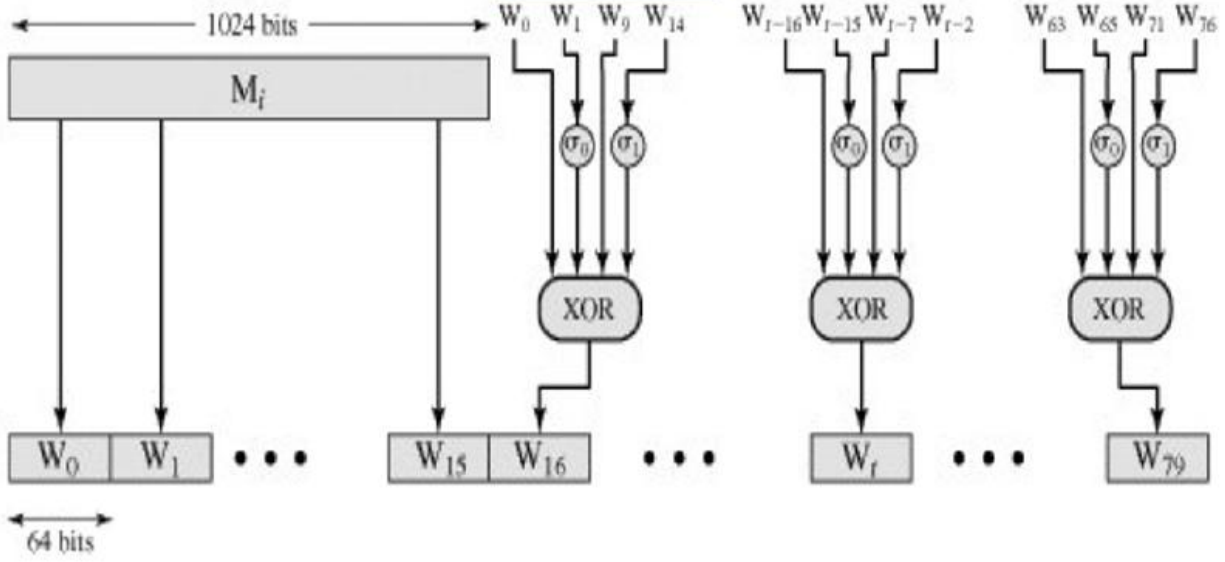
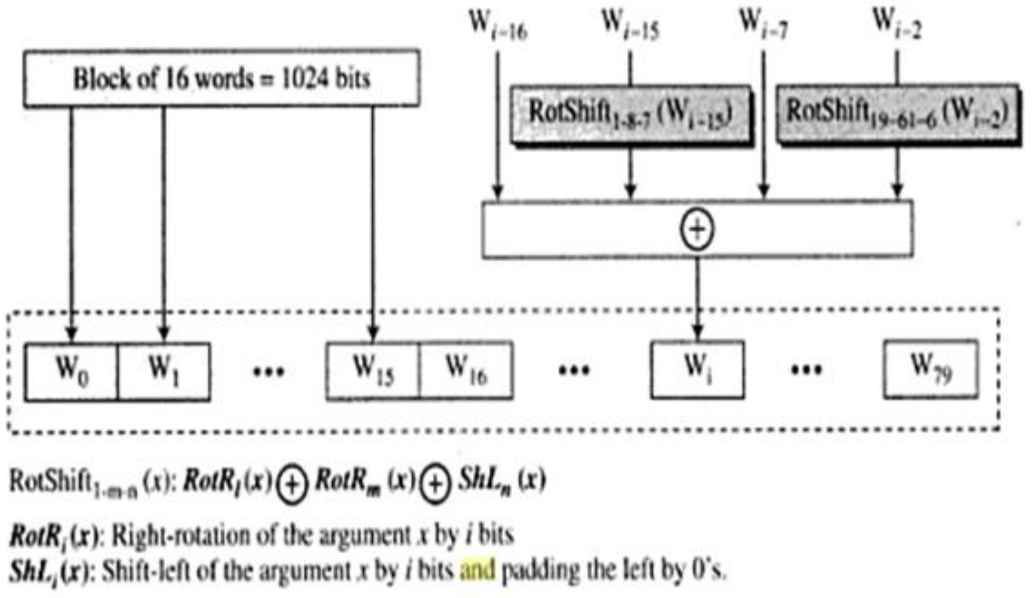


Figure *Word expansion in SHA-512*


$W_{i-16}$
$W_{i-15}$
$W_{i-7}$
$W_{i-2}$
Block of 16 words = 1024 bits
$\text{RotShift}_{1\text{-}8\text{-}7}(W_{i-15})$
$\text{RotShift}_{19\text{-}61\text{-}6}(W_{i-2})$
$\oplus$
$W_0$ $W_1$ ... $W_{15}$ $W_{16}$ ... $W_i$ ... $W_{79}$
$\text{RotShift}_{l\text{-}m\text{-}n}(x)$: $RotR_l(x) \oplus RotR_m(x) \oplus ShL_n(x)$
$RotR_i(x)$: Right-rotation of the argument $x$ by $i$ bits
$ShL_i(x)$: Shift-left of the argument $x$ by $i$ bits and padding the left by 0's.

In the first 16 steps of processing, the value of Wt is equal to the corresponding word in the message block. For the remaining 64 steps, the value of Wt consists of the circular left shift by one bit of the XOR of four of the preceding values of Wt, with two of those values subjected to shift and rotate operations. This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.

## MD5

MD5 is a message digest algorithm developed by Ron Rivest. MD5 is quite fast and produces 128-bit message digests. After some initial processing, the input text is processed in 512-bit blocks, which are further divided into 16 32-bit sub blocks. The output of the algorithm is a set of four 32-bit blocks, which make up the 128-bit message digest.

Step 1: Padding

The First step in MD5 is to add padding bits to the original message. The aim of this step is to make the length of the original message equal to a value, which is 64-bit less than an exact multiple of 512

The padding consists of a single 1-bit followed by as many 0-bits as required. Padding is always added even if the message length is already 64-bits less than a multiple of 512. The padding process is shown in the figure below.

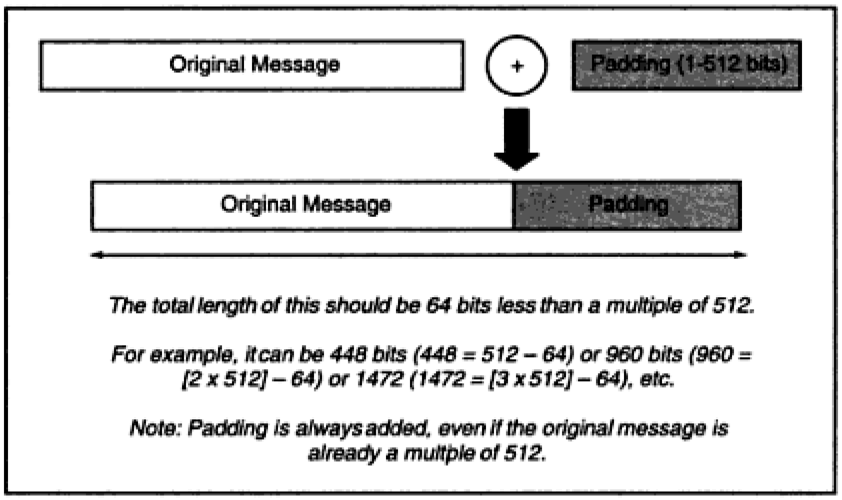

Original Message
+
Padding (1-512 bits)
Original Message
Padding
The total length of this should be 64 bits less than a multiple of 512.
For example, it can be 448 bits (448 = 512 – 64) or 960 bits (960 = [2 x 512] – 64) or 1472 (1472 = [3 x 512] – 64), etc.
Note: Padding is always added, even if the original message is already a multiple of 512.

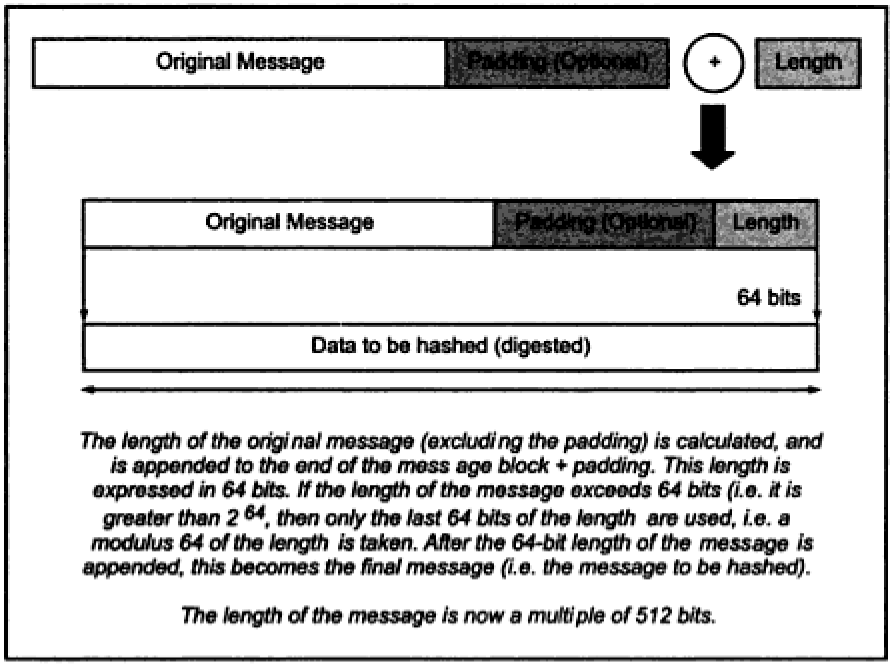
**Fig. 4.23** *Padding process*

Step 2: Append Length

After padding bits are added, the next step is to calculate the original length of the message and add it to the end of the message, after padding. The length of the message is calculated, excluding the padding bits. This length of the original message is expressed as a 64-bit value and these 64-bits are appended to the end of the original message along with padding.

If the length of the message exceeds $2^{64}$ bits, that is 64-bits are not enough to represent the length, only the low-order 64-bits of the length are used. So length is calculated as the actual length mod $2^{64}$

So length of the message is now and exact multiple of 512



*The length of the original message (excluding the padding) is calculated, and is appended to the end of the mess age block + padding. This length is expressed in 64 bits. If the length of the message exceeds 64 bits (i.e. it is greater than 2 $^{64}$, then only the last 64 bits of the length are used, i.e. a modulus 64 of the length is taken. After the 64-bit length of the message is appended, this becomes the final message (i.e. the message to be hashed).*

*The length of the message is now a multiple of 512 bits.*

**Fig. 4.24** *Append length*

Step 3: Divide the input into 512-bit blocks

Divide the input message into blocks, each of length 512 bits.

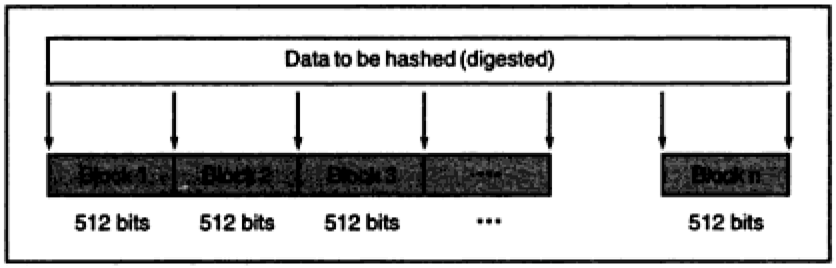
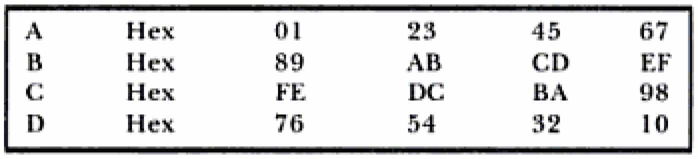


**Fig. 4.25** *Data is divided into 512-bit blocks*

Step 4: Initialize chaining variables

Four chaining variables called as A, B, C and D is initialized. Each of these is a 32-bit number. The initial hexadecimal values of these chaining variables are shown below.

Table 4.3 *Chaining Variables*



| | | | | | |
|---|---|---|---|---|---|
| A | Hex | 01 | 23 | 45 | 67 |
| B | Hex | 89 | AB | CD | EF |
| C | Hex | FE | DC | BA | 98 |
| D | Hex | 76 | 54 | 32 | 10 |

Step 5: Process Blocks

After all initializations, there is a loop that runs for as many 512 bit blocks as are in the message.

Step 5.1: Copy the four chaining variables into four corresponding variables a, b, c and d. So we have a=A, b=B, c=C and d=D.

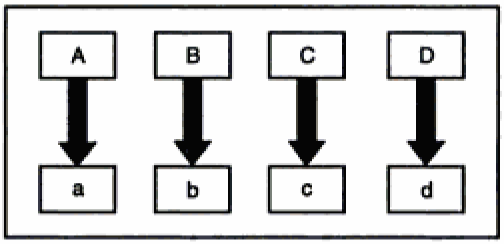


**Fig. 4.26** *Copying chaining variables into temporary variables*

The algorithm considers the combination of a, b, c and d as a 128-bit single register. This register abcd is useful in the actual algorithm for holding intermediate as well as final results.

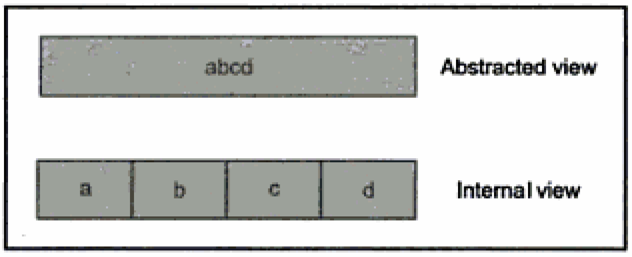


**Fig. 4.27** *Abstracted view of the chaining variables*

Step 5.2: Divide the current 512-bit block into 16 sub blocks. Thus each sub blocks contains 32 bits as shown below.

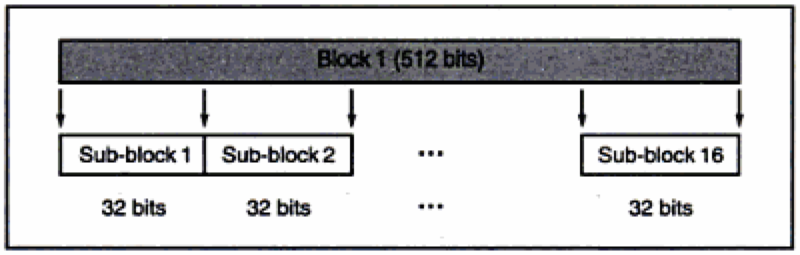


**Fig. 4.28** *Sub-blocks within a block*

Step 5.3: Here four rounds are there. In each round, we process all the 16 sub blocks belonging to a block. The inputs to each round are

1) All the 16 sub blocks
2) Variables a, b, c and d
3) Constants, t

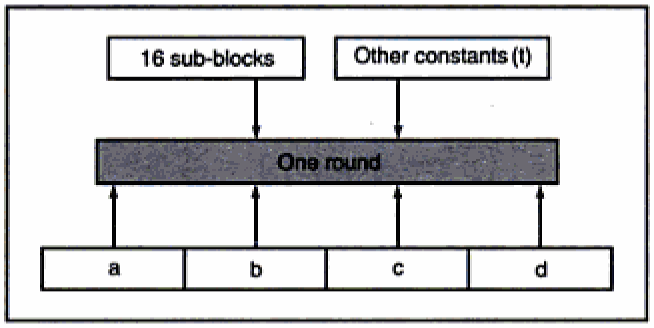


**Fig. 4.29** *Conceptual process within a round*

For each rounds we have the 16 input sub blocks, M[i] where $0 \leq i \leq 15$ and each sub-blocks consists of 32 bits. 't' is an array of constants and contains 64 elements with each element consisting of 32 bits. The elements of this array are represented as t[k] where k varies from 1 to 64. So each round uses 16 out of 64 values of t. In each case the output of the intermediate as well as the final iteration is copied into the register abcd. Totally there are 16 iterations in each round.

All the four rounds vary in one major way that in step 1 of the four rounds has different Processing. The other steps in all the four rounds are the same.

The operations in a round is summarized as

1) A process P is first performed on b, c and d. This process is different in all the four rounds.
2) The variable a is added to the output of the process P (ie to the registers abcd).
3) The message sub-block M[i] is added to the output of step 2 (ie to the registers abcd).
4) The constants t[k] is added to the output of step 3 (ie to the registers abcd).
5) The output of step 4 (ie to the registers abcd) is circular-left shifted by s bits. The value of s keeps changing.
6) The variable b is added to the output of step 5 (ie to the registers abcd).
7) The output of step 6 becomes the new abcd for the next step

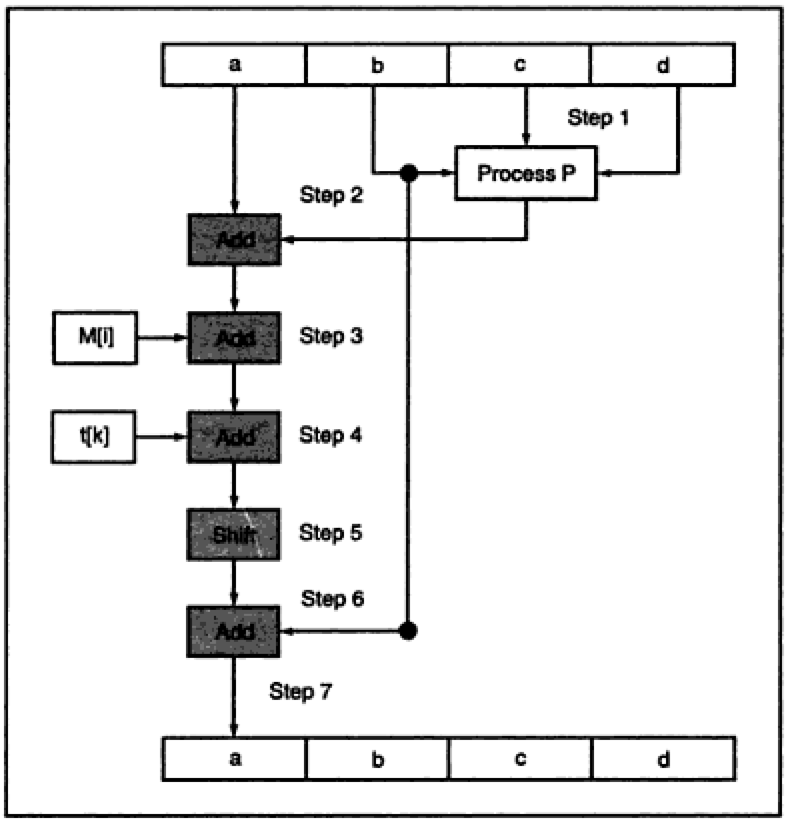


**Fig. 4.30** *One MD5 operation*

So we can express a single MD5 operation as

**a = b + ((a + Process P (b, c, d) + M[i] + T[k]) <<< s)**
**where,**

| | |
|---|---|
| **a, b, c, d** | **= Chaining variables, as described earlier** |
| **Process P** | **= A non-linear operation, as described subsequently** |
| **M[i]** | **= M[q x 16 + i], which is the ith 32-bit word in the qth 512-bit block of the message** |
| **t[k]** | **= A constant, as discussed subsequently** |
| **<<<s** | **= Circular-left shift by s bits** |

The Process P is different in four rounds. The process P is simply some basic Boolean operations on b, c and d. Below show the process P in each round. So we can simply substitute details of P in each of the round and keep everything else constant.

**Table 4.4** *Process P in each round*

| *Round* | *Process P* |
|---|---|
| 1 | (b AND c) OR ((NOT b) AND (d)) |
| 2 | (b AND d) OR (c AND (NOT d)) |
| 3 | B XOR c XOR d |
| 4 | C XOR (b OR (NOT d)) |

**Strength of MD5**

1) MD5 has a property that every bit of the message digest is some function of every bit in the input
2) The possibility that two messages produce the same message digest using MD5 is in the order of $2^{64}$ operations.
3) Given a message digest, working backwards to find the original message can lead upto $2^{128}$ operations.

**Weakness of MD5**

1) Two messages can produce the same message digest for each of the four individual rounds. But two messages that produce the same message digest for all the four rounds taken together are not possible.

2) Execution of MD5 on a single block of 512 bits will produce the same output for two different values in the chaining variable register abcd. This is called pseudocollision. But this is not extended to full MD5 consisting of four rounds, each containing 16 steps.
3) The operation of MD5 on two different 512 bit blocks produces the same 128-bit output. But this has not been generalized to a full message block.

## Digital Signature

Message authentication protects two parties who exchanges message from any third party. But it does not protect the third party from each other. In situations where there is not complete trust between the sender and receiver digital signature is needed. It is similar like handwritten signature.

It must have the following properties.

- It must verify the author and the date and time of the signature
- It must authenticate the contents at the time of the signature
- It must be verifiable by third parties, to resolve disputes

So digital signature function includes authentication function

The requirements of digital signature

- The signature must be a bit pattern that depends on the message being signed.
- The signature must use some information unique to the sender, to prevent both forgery and denial.
- It must be relatively easy to produce the digital signature.
- It must be relatively easy to recognize and verify the digital signature

- It must be computationally infeasible to forge a digital signature
- It must be practical to retain a copy of the digital signature in storage

There are two approaches for digital signature function

- Direct
- Arbitrated

**Direct Digital Signature**

The direct digital signature involves only the communicating parties. It is assumed that the destination knows the public key of the source. A digital signature may be formed by encrypting the entire message with the sender's private key or by encrypting hash code of the message with the sender's private key.

Confidentiality can be provided by further encrypting the entire message plus signature with either the receiver's public key or a shared secret key. The validity of the scheme depends on the security of the sender's private key.

**Arbitrated Digital Signature**

The problems associated with direct digital signatures can be addressed by using an arbiter.

As with direct signature schemes, there is a variety of arbitrated signature schemes. In general terms, they all operate as follows. Every signed message from a sender X to a receiver Y goes first to an arbiter A, who subjects the message and its signature to a number of tests to check its origin and content. The message is then dated and sent to Y with an indication that it has been verified to the satisfaction of the arbiter. The presence of A solves the problem faced by direct signature schemes that X might disown the message.

The arbiter plays a sensitive and crucial role in this sort of scheme, and all parties must have a great deal of trust that the arbitration mechanism is working properly.

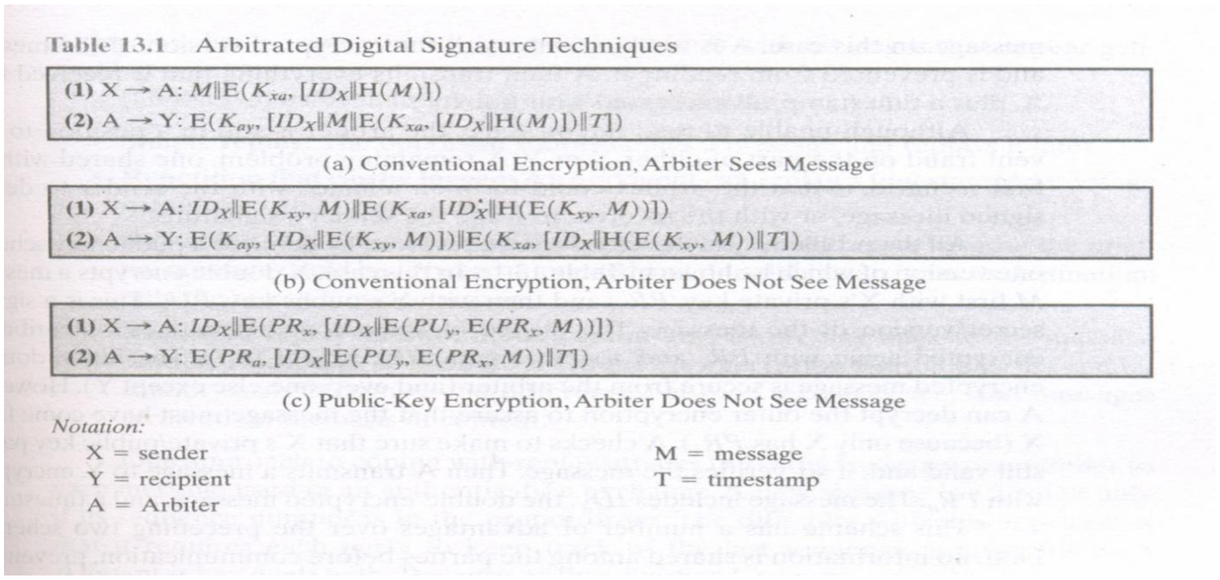


Table 13.1 Arbitrated Digital Signature Techniques

(1) $X \rightarrow A: M \| E(K_{xa}, [ID_X \| H(M)])$

(2) $A \rightarrow Y: E(K_{ay}, [ID_X \| M \| E(K_{xa}, [ID_X \| H(M)]) \| T])$

(a) Conventional Encryption, Arbiter Sees Message

(1) $X \rightarrow A: ID_X \| E(K_{xy}, M) \| E(K_{xa}, [ID_X \| H(E(K_{xy}, M))])$

(2) $A \rightarrow Y: E(K_{ay}, [ID_X \| E(K_{xy}, M)]) \| E(K_{xa}, [ID_X \| H(E(K_{xy}, M)) \| T])$

(b) Conventional Encryption, Arbiter Does Not See Message

(1) $X \rightarrow A: ID_X \| E(PR_x, [ID_X \| E(PU_y, E(PR_x, M))])$

(2) $A \rightarrow Y: E(PR_a, [ID_X \| E(PU_y, E(PR_x, M)) \| T])$

(c) Public-Key Encryption, Arbiter Does Not See Message

*Notation:*

| | |
|---|---|
| X = sender | M = message |
| Y = recipient | T = timestamp |
| A = Arbiter | |

An example is given in the picture.

In the first, symmetric encryption is used. It is assumed that the sender X and the arbiter A share a secret key $K_{xa}$, and that A and Y share secret key $K_{ay}$. X constructs a message M and computes its hash value X(M), Then X transmits the message plus a signature to A. The signature consists of an identifier $ID_x$ of X plus the hash value, all encrypted using Kx(. A decrypts the signature and checks the hash value to validate the message. Then A transmits a message to Y, encrypted with $K_{ay}$. The message includes $lD_x$, the original message from X, the signature, and a timestamp. Y can decrypt this to recover the message and the signature. The timestamp informs Y that this message is timely and not a replay. Y can store M and the signature. In case of dispute, Y, who claims to have received M from X, sends the following message to A

$E(K_{ay}, [ID||M||E(K_{xa}, [ID_X||H(M)])])$

The arbiter uses $K_{ay}$ to recover $ID_x$. M, and the signature, and then uses $K_{xa}$ to decrypt the signature and verify the hash code. In this scheme, Y cannot directly check X’s signature; the signature is there solely to settle disputes. Y considers the message from X authentic because it comes through A. In this scenario, both sides must have a high degree of trust in A

X must trust A not to reveal $K_{xa}$ and not to generate false signatures of the form E($K_{xa}$, [$ID_x$ ||H(M)])

Y must trust A to send E($K_{ay}$, [$ID_x$ ||M||E($K_{xa}$, [$ID_x$ ||H(M)||T]) only if the hash value is correct and the signature was generated by X.

Both sides must trust A to resolve disputes fairly.

If the arbiter does live up to this trust, then X is assured that no one can forge his signature and Y is assured that X cannot disavow his signature.

The preceding scenario also implies that A is able to read messages from X to Y and, indeed, that any eavesdropper is able to do so. Table 13.lb shows a scenario that provides the arbitration as before but also assures confidentiality. In this case it is assumed that X and Y share the secret key Now, X transmits an identifier, a copy of the message encrypted with and a signature to A. The signature consists of the identifier plus the hash value of the encrypted message, all encrypted using K0. As before, A decrypts the signature and checks the hash value to validate the message. In this case, A is working only with the encrypted version of the message and is prevented from reading it. A then transmits everything that it received from X, plus a timestamp, all encrypted with Kay, to Y.

Although unable to read the message, the arbiter is still in a position to prevent fraud on the part of either X or Y. A remaining problem, one shared with the first scenario, is that the arbiter could form an alliance with the sender to deny a signed message, or with the receiver to forge the sender's signature.

In third case, X double encrypts a message M first with X's private key, **PR,** and then with Y' public key, $PU_y$. This is a signed, secret version of the message. This signed message, together with X's identifier, is encrypted again with **$PR_x$** and, together with IDx, is sent to A. The inner, double- encrypted message is secure from the arbiter**.** However, A can decrypt the outer encryption to assure that the message must have come from X**.** A checks to make sure that X's private/public key pair is still valid and, if so, verifies the message. Then A transmits a message to Y, encrypted with **$PR_a$.** The message includes **$ID_x$,** the double-encrypted message, and a timestamp.

## Authentication Protocols

## Mutual Authentication

Authentication protocols enable communicating parties to satisfy themselves mutually about each other's identity and to exchange session keys. Central to the problem of authenticated key exchange are two issues: confidentiality and timeliness. To prevent masquerade and to prevent compromise of session keys, essential identification and session key information must be communicated in encrypted form. This requires the prior existence of secret or public keys that can be used for this purpose. The second issue, timeliness, is important because of the threat of message replays. Such replays, at worst, could allow an opponent to compromise a session key or successfully impersonate another party. At minimum, a successful replay can disrupt operations by presenting parties with messages that appear genuine but are not.

Following are the lists of examples of replay attacks :

Simple replay:  The opponent simply copies a message and replays it later.

Repetition that can be logged:  An opponent can replay a timestamped message within the valid time window.

Repetition that cannot be detected:  This situation could arise because the original message could have been suppressed and thus did not arrive at its destination; only the replay message arrives.

Backward replay without modification:  This is a replay back to the message sender. This attack is possible if symmetric encryption is used and the sender cannot easily recognize the difference between messages sent and messages received on the basis of content.

One approach to avoid replay attacks is to attach a sequence number to each message used in an authentication exchange. A new message is accepted only if its sequence number is in the proper order. The difficulty with this approach is that it requires each party to keep track of the last sequence number for each claimant it has dealt with. Because of this overhead, sequence numbers are generally not used for authentication and key exchange. Instead, one of the following two general approaches is used:

Timestamps: Party A accepts a message as fresh only if the message contains a timestamp that, in A's judgment, is close enough to A's knowledge of current time. This approach requires that clocks among the various participants be synchronized.

Challenge/response: Party A, expecting a fresh message from B, first sends B a nonce (challenge) and requires that the subsequent message (response) received from B contain the correct nonce value.

The timestamp approach should not be used for connection-oriented applications because of the inherent difficulties with this technique. First, some sort of protocol is needed to maintain synchronization among the various processor clocks. This protocol must be both fault tolerant, to cope with network errors, and secure, to cope with hostile attacks. Second, the opportunity for a successful attack will arise if there is a temporary loss of synchronization resulting from a fault in the clock mechanism of one of the parties. Finally, because of the variable and unpredictable nature of network delays, distributed clocks cannot be expected to maintain precise synchronization. Therefore, any timestamp-based procedure must allow for a window of time sufficiently large to accommodate network delays yet sufficiently small to minimize the opportunity for attack.

The challenge-response approach is unsuitable for a connectionless type of application because it requires the overhead of a handshake before any connectionless transmission, effectively negating the chief characteristic of a connectionless transaction. For such applications, reliance on some sort of secure time server and a consistent attempt by each party to keep its clocks in synchronization may be the best approach.

A symmetric encryption approach was proposed by Needham and Schroeder for secret key distribution using a Key Distribution Center (KDC) which includes authentication features. The protocol can be summarized as follows:

1. $A \longrightarrow KDC$: $ID_A \| ID_B \| N_1$
2. $KDC \longrightarrow A$: $E(K_a, [K_s \| ID_B \| N_1 \| E(K_b, [K_s \| ID_A])])$
3. $A \longrightarrow B$: $E(K_b, [K_s \| ID_A])$
4. $A \longrightarrow A$: $E(K_s, N_2)$
5. $A \longrightarrow B$: $E(K_s, f(N_2))$

Secret keys Ka and Kb are shared between A and the KDC and B and the KDC, respectively. The purpose of the protocol is to distribute securely a session key Ks to A and B. A securely acquires a new session key in step 2. The message in step 3 can be decrypted, and hence understood, only by B. Step 4 reflects B's knowledge of Ks, and step 5 assures B of A's knowledge of Ks and assures B that this is a fresh message because of the use of the nonce N2. The step 4 and 5 is to prevent a certain type of replay attack. In particular, if an opponent is able to capture the message in step 3 and replay it, this might in some fashion disrupt operations at B.

Despite the handshake of steps 4 and 5, the protocol is still vulnerable to a form of replay attack. Suppose that an opponent, X, has been able to compromise an old session key. Admittedly, this is a much more unlikely occurrence than that an opponent has simply observed and recorded step 3. Nevertheless, it is a potential security risk. X can impersonate A and trick B into using the old key by simply replaying step 3. Unless B remembers indefinitely all previous session keys used with A, B will be unable to determine that this is a replay. If X can intercept the handshake message, step 4, then it can impersonate A's response, step 5. From this point on, X can send bogus messages to B that appear to B to come from A using an authenticated session key.

Denning proposes to overcome this weakness by a modification to the Needham/Schroeder protocol that includes the addition of a timestamp to steps 2 and 3. Her proposal assumes that the master keys, Ka and Kb are secure, and it consists of the following steps:

1. $A \longrightarrow KDC$: $ID_A \| ID_B$
2. $KDC \longrightarrow A$: $E(K_a, [K_s \| ID_B \| T \| E(K_b, [K_s \| ID_A \| T])])$
3. $A \longrightarrow B$: $E(K_b, [K_s \| ID_A \| T])$
4. $B \longrightarrow A$: $E(K_s, N_1)$
5. $A \longrightarrow B$: $E(K_s, f(N_1))$

T is a timestamp that assures A and B that the session key has only just been generated. Thus, both A and B know that the key distribution is a fresh exchange. A and B can verify timeliness by checking that

$$|Clock - T| < \Delta t_1 + \Delta t_2$$

$D_{t1}$ is the estimated normal discrepancy between the KDC's clock and the local clock (at A or B) and $D_{t2}$ is the expected network delay time. Each node can set its clock against some standard reference source. Because the timestamp T is encrypted using the secure master keys, an opponent, even with knowledge of an old session key, cannot succeed because a replay of step 3 will be detected by B as untimely.

The Denning protocol seems to provide an increased degree of security compared to the Needham/Schroeder protocol. However, a new concern is raised: namely, that this new scheme requires reliance on clocks that are synchronized throughout the network. The distributed clocks can become unsynchronized as a result of sabotage on or faults in the clocks or the synchronization mechanism. The problem occurs when a sender's clock is ahead of the intended recipient's clock. In this case, an opponent can intercept a message from the sender and replay it later when the timestamp in the message becomes current at the recipient's site. This replay could cause unexpected results. Gong refers to such attacks as suppress-replay attacks.

One way to counter suppress-replay attacks is to enforce the requirement that parties regularly check their clocks against the KDC's clock. The other alternative, which avoids the need for clock synchronization, is to rely on handshaking protocols using nonces. This latter alternative is not vulnerable to a suppress-replay attack because the nonce's the recipient will choose in the future are unpredictable to the sender. The Needham/Schroeder protocol relies on nonce's only but, as we have seen, has other vulnerabilities.

**One-Way Authentication**

Using symmetric encryption, the decentralized key distribution scenario is impractical. This scheme requires the sender to issue a request to the intended recipient, await a response that includes a session key, and only then send the message. With some refinement, the KDC strategy illustrated is a candidate for encrypted electronic mail. Because we wish to avoid requiring that the recipient (B) be on line at the same time as the sender (A), steps 4 and 5 must be eliminated. For a message with content M, the sequence is as follows:

1. $A \longrightarrow KDC$: $ID_A||ID_B||N_1$
2. $KDC \longrightarrow A$: $E(K_a, [K_s||ID_B||N_1||E(K_b, [K_s||ID_A])])$
3. $A \longrightarrow B$: $E(K_b, [K_s||ID_A])||E(K_s, M)$

This approach guarantees that only the intended recipient of a message will be able to read it. It also provides a level of authentication that the sender is A. As specified, the protocol does not protect against replays. Some measure of defense could be provided by including a timestamp with the message. However, because of the potential delays in the e-mail process, such timestamps may have limited usefulness.

By using public-key encryptions approaches require that either the sender know the recipient's public key (confidentiality) or the recipient know the sender's public key (authentication) or both (confidentiality plus authentication). In addition, the public-key algorithm must be applied once or twice to what may be a long message. If confidentiality is the primary concern, then the following may be more efficient:

$$A \longrightarrow B: E(PU_b, K_s)||E(K_s, M)$$

In this case, the message is encrypted with a one-time secret key. A also encrypts this one-time key with B's public key. Only B will be able to use the corresponding private key to recover the one-time key and then use that key to decrypt the message. This scheme is more efficient than simply encrypting the entire message with B's public key. If authentication is the primary concern, then a digital signature may suffice:

$$A \longrightarrow B: M||E(PR_a, H(M))$$

This method guarantees that A cannot later deny having sent the message. However, this technique is open to another kind of fraud. Bob composes a message to his boss Alice that contains an idea that will save the company money. He appends his digital signature and sends it into the e-mail system. Eventually, the message will get delivered to Alice's mailbox. But suppose that Max has heard of Bob's idea and gains access to the mail queue before delivery. He finds Bob's message, strips off his signature, appends his, and requires the message to be delivered to Alice. Max gets credit for Bob's idea. To counter such a scheme, both the message and signature can be encrypted with the recipient's public key:

$$A \longrightarrow B: E(PU_b, [M||E(PR_a, H(M))])$$

The latter two schemes require that B know A's public key and be convinced that it is timely. An effective way to provide this assurance is the digital certificate:

$$A \longrightarrow B: M||E(PR_a, H(M))||E(PR_{as}, [T||ID_A||PU_a])$$

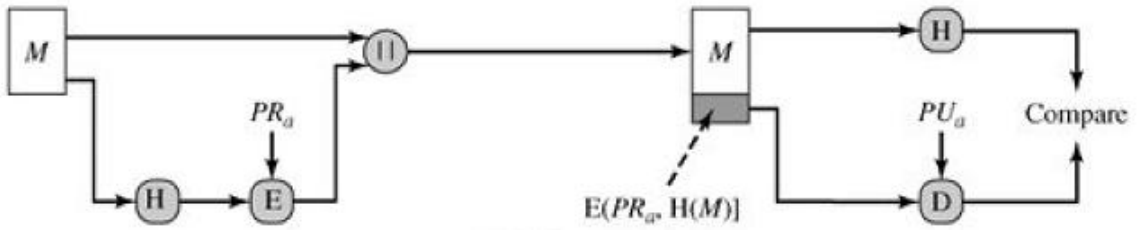
## Digital Signature Standard

The National Institute of Standards and Technology (NIST) have published Federal Information Processing Standard FIPS, known as the Digital Signature Standard (DSS). The DSS makes use of the Secure Hash Algorithm (SHA) and presents a new digital signature technique, the Digital

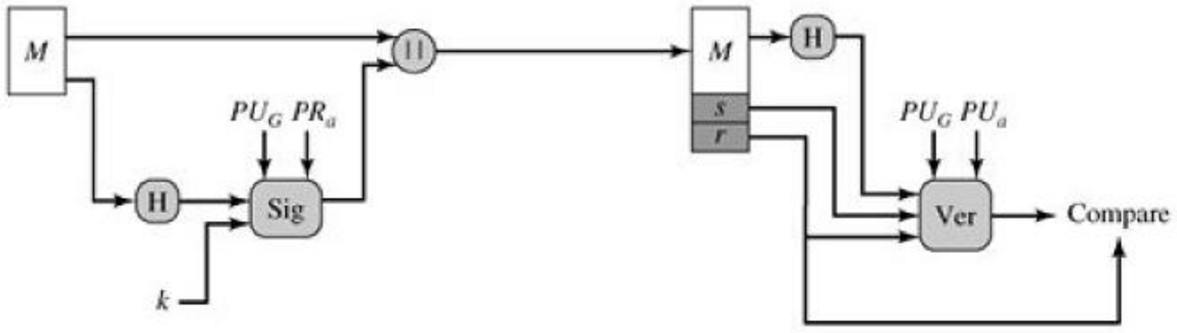
Signature Algorithm (DSA). The latter version of DSS incorporates digital signature algorithms based on RSA and on elliptic curve cryptography.

The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique. Figure below shows the DSS approach for generating digital signatures to that used with RSA. In the RSA approach, the message to be signed is input to a hash function that produces a secure hash code of fixed length. This hash code is then encrypted using the sender's private key to form the signature. Both the message and the signature are then transmitted. The recipient takes the message and produces a hash code. The recipient also decrypts the signature using the sender's public key. If the calculated hash code matches the decrypted signature, the signature is accepted as valid. Because only the sender knows the private key, only the sender could have produced a valid signature.

The DSS approach also makes use of a hash function. The hash code is provided as input to a signature function along with a random number k generated for this particular signature. The signature function also depends on the sender's private key ($PR_a$) and a set of parameters known to a group of communicating principals. This set to constitute a global public key ( $PU_G$ ). The result is a signature consisting of two components, labeled s and r. At the receiving end, the hash code of the incoming message is generated. This plus the signature is input to a verification function. The verification function also depends on the global public key as well as the sender's public key ( $PU_a$ ), which is paired with the sender's private key. The output of the verification function is a value that is equal to the signature component r if the signature is valid. The signature function is such that only the sender, with knowledge of the private key, could have produced the valid signature.



(a) RSA approach



(b) DSS approach

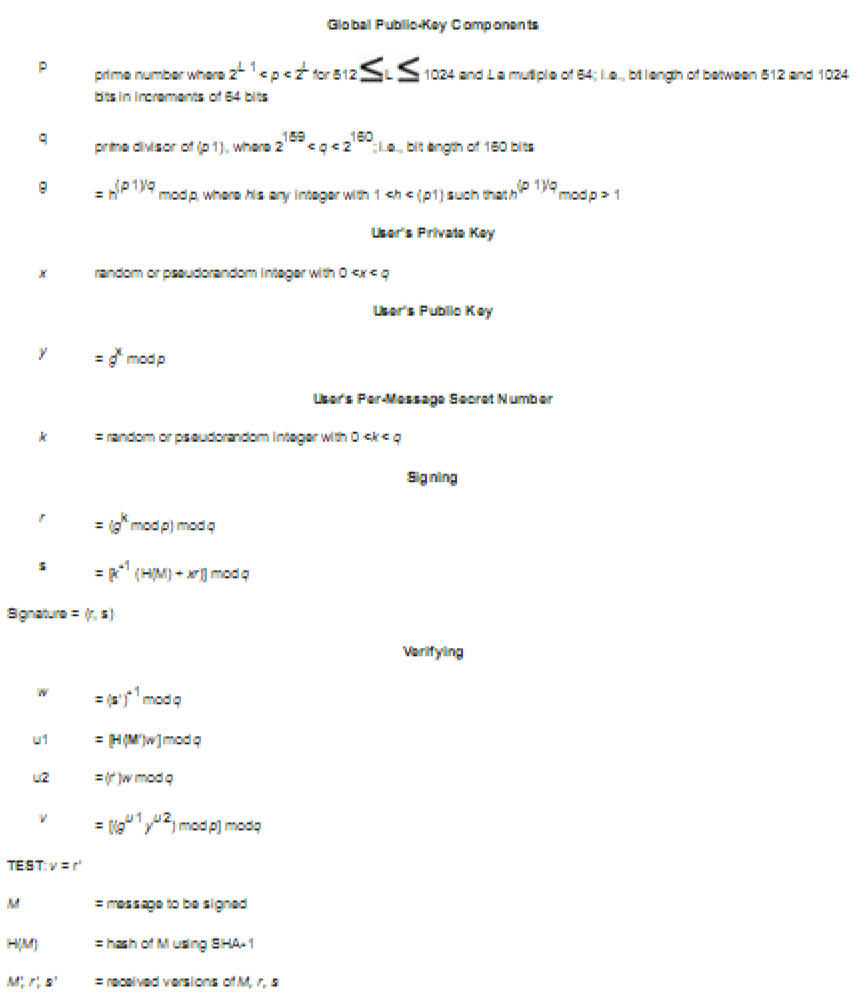
## The Digital Signature Algorithm

Below figure summarizes the algorithm. There are three parameters that are public and can be common to a group of users. A 160-bit prime number q is chosen. Next, a prime number p is selected with a length between 512 and 1024 bits such that q divides (p-1). Finally, g is chosen to be of the form $h^{(p-1)/q}$ mod p where h is an integer between 1 and ( p-1) with the restriction that g must be greater than 1.

Each user selects a private key and generates a public key. The private key x must be a number from 1 to (q-1) and should be chosen randomly or pseudo randomly. The public key is calculated from the private key as $y = g^x$ mod p. Given the public key y, it is believed to be computationally infeasible to determine x, which is the discrete logarithm of y to the base g mod p.

To create a signature, a user calculates two quantities, r and s, that are functions of the public key components ( p, q, g ), the user's private key ( x ), the hash code of the message, H( M ), and an additional integer k that should be generated randomly or pseudo randomly and be unique for each signing. At the receiving end, verification is performed using the formulas shown in below figure. The receiver generates a quantity v that is a function of the public key components, the sender's public key, and the hash code of the incoming message. If this quantity matches the r component of the signature, then the signature is validated.

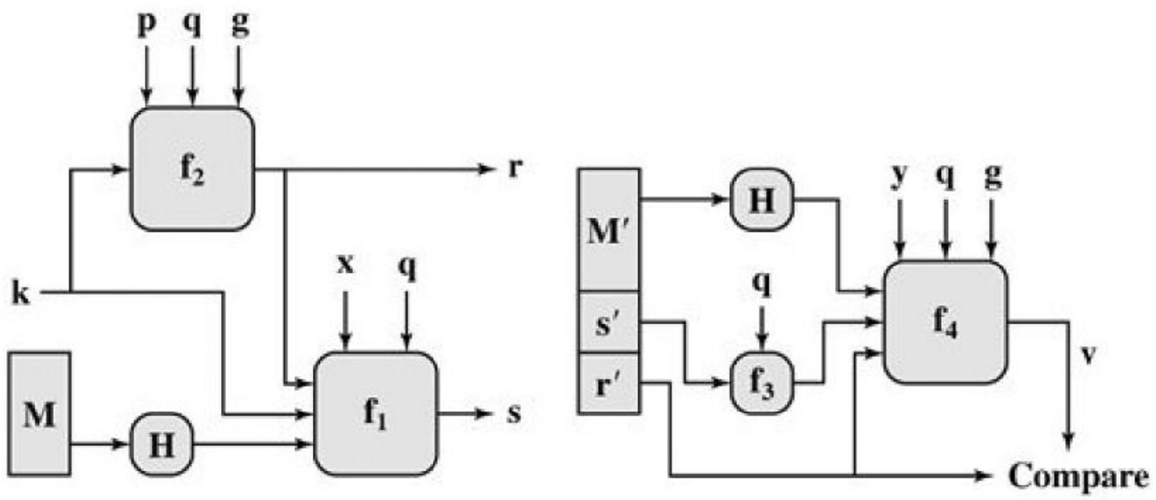
**Figure 2. The Digital Signature Algorithm (DSA)**



| | |
|---|---|
| | **Global Public-Key Components** |
| p | prime number where $2^{L 1} < p < 2^{L}$ for $512 \leq L \leq 1024$ and $L$ a mutiple of 64; i.e., bit length of between 512 and 1024 bits in increments of 64 bits |
| q | prime divisor of $(p 1)$, where $2^{159} < q < 2^{160}$; i.e., bit length of 160 bits |
| g | $= h^{(p 1)/q} \bmod p$, where $h$ is any integer with $1 < h < (p1)$ such that $h^{(p 1)/q} \bmod p > 1$ |
| | **User's Private Key** |
| $x$ | random or pseudorandom integer with $0 < x < q$ |
| | **User's Public Key** |
| $y$ | $= g^{x} \bmod p$ |
| | **User's Per-Message Secret Number** |
| $k$ | = random or pseudorandom integer with $0 < k < q$ |
| | **Signing** |
| $r$ | $= (g^{k} \bmod p) \bmod q$ |
| s | $= [k^{-1} (H(M) + xr)] \bmod q$ |
| Signature = $(r, s)$ | |
| | **Verifying** |
| $w$ | $= (s')^{-1} \bmod q$ |
| u1 | $= [H(M')w] \bmod q$ |
| u2 | $= (r')w \bmod q$ |
| $v$ | $= [(g^{u1} y^{u2}) \bmod p] \bmod q$ |
| TEST: $v = r'$ | |
| $M$ | = message to be signed |
| H(M) | = hash of M using SHA-1 |
| $M', r', s'$ | = received versions of $M, r, s$ |

The functions of signing and verifying are shown below. The test at the end is on the value r, which does not depend on the message at all. Instead, r is a function of k and the three global public-key components. The multiplicative inverse of k (mod q ) is passed to a function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover r using the incoming message and signature, the public key of the user, and the global public key.

Figure .3. DSS Signing and Verifying



$s = f_1(H(M), k, x, r, q) = (k^{-1}(H(M) + xr)) \bmod q$

$r = f_2(k, p, q, g) = (g^k \bmod p) \bmod q$

(a) Signing

$w = f_3(s', q) = (s')^{-1} \bmod q$

$v = f_4(y, q, g, H(M'), w, r')$

$= ((g^{(H(M')w) \bmod q} y^{r'w \bmod q}) \bmod p) \bmod q$

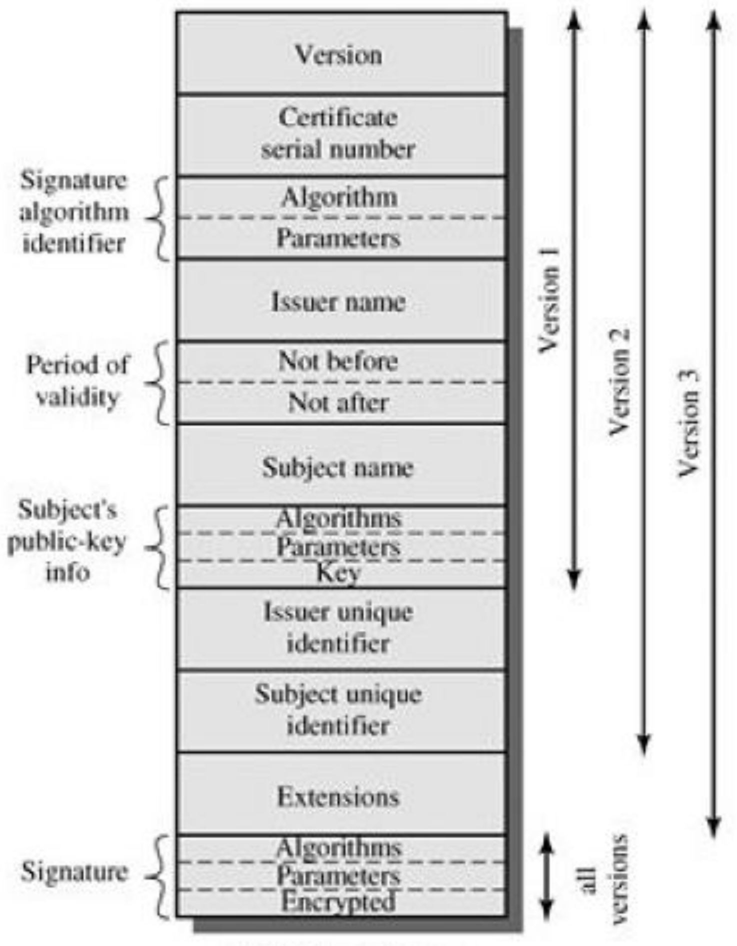
(b) Verifying

## Digital Certificates

A digital certificate is simply a small computer file. A digital certificate establishes the relation between a user and her public key. A digital certificate must contain the user name and the user’s public key. This will prove that a particular public key belongs to a particular user. Other information’s in the digital certificate includes serial number, validity date range for the certificate and who issued the certificate.

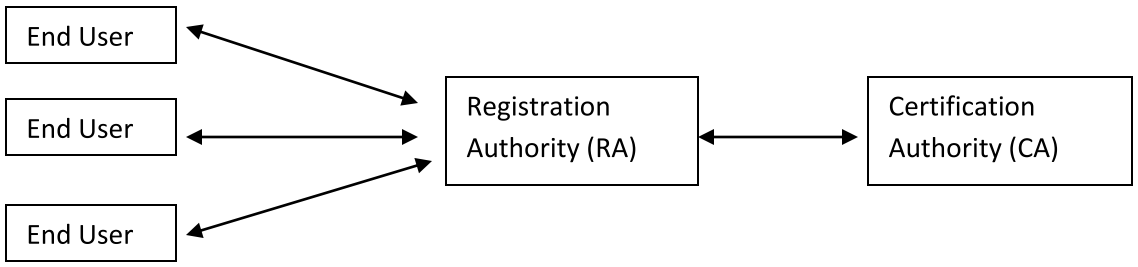
A certificate authority (CA) is a trusted agency that can issue digital certificates. CA is the one who everybody must trust. CA is usually a reputed organization such as a post office, financial institution, software company etc. Two of the worlds famous CA’s are VeriSign and Entrust. CA has the authority to issue digital certificates to individuals and organizations, which want to use those certificates in asymmetric cryptographic applications.

A standard called as X.509 defines the structure of a digital certificate. The figure shows the structure of X.509 version 3 certificates.



(a) X.509 certificate

Apart from the subject (end user) and the issuer (CA) a third party can also be involved in the certificate creation and management. Since a CA can be overloaded with a variety of tasks such as issuing new certificates, maintaining the old ones, revoking the ones that have become invalid for whatever reason etc, the CA can delegate some of its tasks to this third party, called as Registration Authority (RA). RA is an intermediate entity between the end users and the CA, which assists the CA in its day-to-day activities.

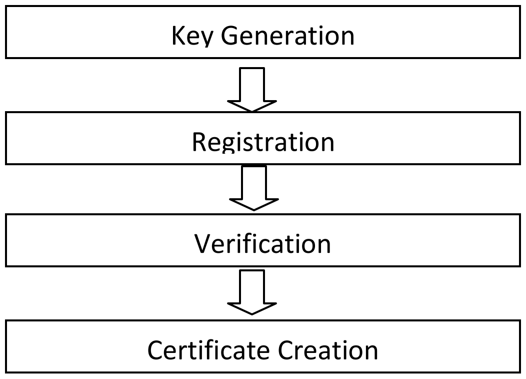


The RA provides following services:

- Accepting and verifying registration information about new users.
- Generating keys on behalf of the end users.
- Accepting and authorizing requests for key backups and recovery.
- Accepting and authorizing the requests for certificate revocation.

RA is mainly set up for facilitating the interaction between the end users and the CA. The RA cannot issue digital certificates. CA must issue the digital certificate, after a certificate is issued; the CA is responsible for all the certificate management aspects such as tracking its status, issuing revocation notices if the certificate needs to be invalidated for some reasons etc.

The creation of a digital certificate consists of following steps:



Step 1: Key generation

The action begins with the subject who wants to obtain a certificate. There are two different approaches for this purpose:

a) The subject can create a private key and public key pair using some software. This software is usually a part of the web browser or web server. The subject must keep the private key thus generated a secret. The subject then sends the public key along with other information and evidences about herself to the RA.
b) The RA can generate a key pair on the subjects or users behalf. This can happen in cases where either the user is not aware of the technicalities involved in the generation of a key pair or if a particular requirement demands that all the keys must be centrally generated

and distributed by the RA for the case of enforcing security policies and key management.

Step 2: Registration

This step is required only if the user generates the key pair in the first step. If the RA generates the key pair on the user's behalf this step will also be a part of the first step itself. If the user has generated the key pair, the user now sends the public key and the associated registration information and all the evidence about herself to the RA. For this the software provides a wizard in which the user enters data and when all data is correct, submits it. This data then travels over the network/Internet to the RA. The format for the certificate requests has been standardized and is called as Certificate Signing Request (CSR).

Step 3: Verification

After the registration process is complete, the RA has to verify the user's credentials. This verification is in two respects

a) Firstly, the RA needs to verify the user's credentials such as the evidences provided are correct and that they are acceptable.
b) Secondly, to ensure that the user who is requesting for the certificate does indeed possess the private key corresponding to the public key that is sent as a part of the certificate request to the RA. This check is called as checking the Proof of Possession (POP) of the private key.

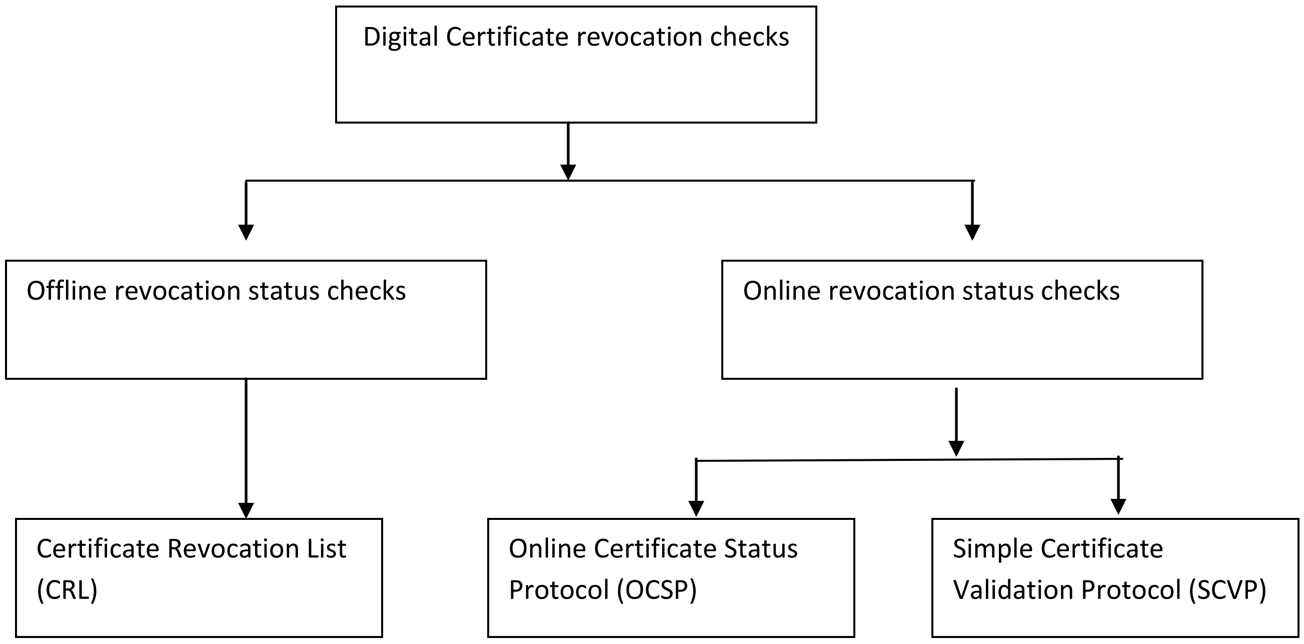
Step 4: Certificate creation

The RA passes on all the details of the user to the CA. The CA does its own verification and creates a digital certificate for the user. The CA sends the certificate to the user and also retains a copy of certificate for its own record. The CA's copy of the certificate is maintained in a certificate directory. The directory clients can request for and access information from this central repository using a directory access protocol such as Lightweight Directory Access Protocol (LDAP). The CA then sends the certificate to the user. This can be attached to an email or the CA can send an email to the user, informing that the certificate is ready and that the user should download it from the CA's site.

The Verification of a digital certificate consists of the following steps:

1) The user passes all fields expect the last one of the received digital certificate to a message digest algorithm. This algorithm should be same as the one used by the CA while signing the certificate.
2) The message digest algorithm calculates a message digest (hash) of all the fields of the certificate, except for the last one, called as MD1.
3) The user now extracts the digital signature of the CA from the certificate
4) The user de-signs the CA's signature
5) This produces another message digest, called as MD2.
6) The user compares the message digest MD1 with the result of de-signing the CA's signature MD2. If two match the user convinced that the digital certificate was indeed signed by the CA with its private key. If this comparison fails, the user will not trust the certificate and reject it.

Certification revocation status mechanisms


Digital Certificate revocation checks
Offline revocation status checks
Online revocation status checks
Certificate Revocation List (CRL)
Online Certificate Status Protocol (OCSP)
Simple Certificate Validation Protocol (SCVP)

## Kerberos

It is an authentication service designed for use in a distributed environment. It makes use of a trusted third party authentication service that enables clients and servers to establish authenticated communication.

Three types of threat exist in a network

- A user may gain access to a particular workstation and pretend to be another user operating from that workstation

- A user may alter the network address of a workstation so that the requests se from the altered workstation appear to come from the impersonated workstation.

- A user may eavesdrop on exchanges and use a replay attack to gain entrance to a server or to disrupt operations.

In any of these cases, an unauthorized user may be able to gain access to services and data that he or she is not authorized to access. Rather than building in elaborate authentication protocols at each server, Kerberos provides a centralized authentication server whose function is to authenticate users to servers and servers to users.

Two versions of Kerberos exists

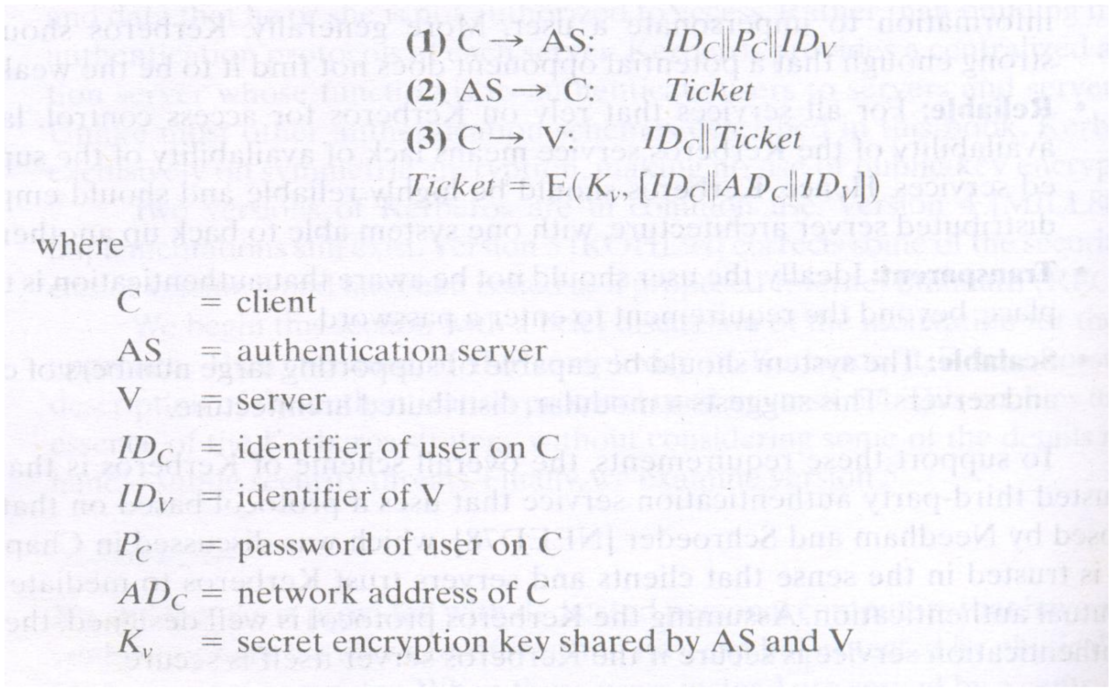
- Version 4
- Version 5

**Kerberos Version 4**

Version 4 of Kerberos makes use of DES to provide the authentication service. Each successive dialogue adds additional complexity to counter security vulnerabilities revealed in the preceding dialogue.

A Simple Authentication dialogue in an unprotected network environment, any client can apply to any server for service. The obvious security risk is that of impersonation. An opponent can pretend to another client and obtain unauthorized privileges on server machines. To counter this threat, servers must he able to confirm the identities of clients who request service. Each server can be required to undertake this task for each client/server interaction, but in an open environment, this places a substantial burden on each server.

An alternative is to use an authentication server (AS) that knows the passwords of all users and stores these in a centralized database. In addition, the AS shares a unique secret key with each server. These keys have been distributed physically or in some other secure manner. Consider the following hypothetical dialogue:



(1) $C \rightarrow AS$: $ID_C \| P_C \| ID_V$

(2) $AS \rightarrow C$: $Ticket$

(3) $C \rightarrow V$: $ID_C \| Ticket$

$$Ticket = E(K_v, [ID_C \| AD_C \| ID_V])$$

where

| | |
|---|---|
| C | = client |
| AS | = authentication server |
| V | = server |
| $ID_C$ | = identifier of user on C |
| $ID_V$ | = identifier of V |
| $P_C$ | = password of user on C |
| $AD_C$ | = network address of C |
| $K_v$ | = secret encryption key shared by AS and V |

In this scenario, the user logs on to a workstation and requests access to server V. The client module C in the user's workstation requests the user's password and then sends a message to the AS that includes the user's ID, the server's ID, and the user's password. The AS checks its database to see if the user has supplied the proper password for this user ID and whether this user is permitted access to server V. If both tests are passed, the AS accepts the user as authentic and must now convince the server that this user is authentic. To do so, the AS creates a ticket that contains the users ID and network address and the server's ID. This ticket is encrypted using the secret key shared by the AS and this server. This ticket is then sent back to C. Because the ticket is encrypted, it cannot be altered by C or by an opponent. With this ticket, C can now apply to V for service. C sends a message to V containing C's ID and the ticket. V decrypts the ticket and verifies that the user ID in the ticket is the same as the unencrypted user ID in the message. If these two match, the server considers the user authenticated and grants the requested service.

Each of the ingredients of message (3) is significant. The ticket is encrypted to prevent alteration or forgery. The server's ID (ID) is included in the ticket so that the server can verify that it has decrypted the ticket properly. 1D is included in the ticket to indicate that this ticket has been issued on behalf of C. Finally. AD serves II counter the following threat. An opponent could capture the ticket transmitted in message (2), then use the name JD and transmit a message of form (3) from another workstation.  The server would receive a valid ticket that matches the user ID and grant access to the user on that other workstation. To prevent this attack, the AS includes the ticket in the network address from which the original request came. The AS the ticket is valid only if it is transmitted from the same workstation that initially requested the ticket.

**The Version 4 Authentication Dialogue**

The scenario-taking place in Version 4 authentication dialog is given below

The client sends a message to the AS requesting access to TGS. TGS is ticket granting Server, which grants a ticket to users who has been authenticated to AS. The AS responds with a message, encrypted with a key derived from the users password $K_c$, which contains the ticket. The encrypted message also contains a copy of the session key for C and TGS. Because this session key is inside the message encrypted with $K_c$ only the user's client can read it. The same session key is included in the ticket, which can he read only by the TGS. Thus, the session key has been securely delivered to both C and the TGS.

Message (1) includes a tirnestamp, so that the AS knows that the message is timely.

Message (2) includes several elements of the ticket form accessible to C. This enables C to confirm that this ticket is for the TGS and learn its expiration time.

Armed with the ticket and the session key, C is ready to approach the TGS. As before. C sends the TGS a message that includes the ticket plus the ID of requested service. In addition, C transmits an authenticator, which includes the ID and address of C's user and a timestamp. Unlike ticket, which is reusable, the authenticator is intended for use only once and has a very short lifetime.

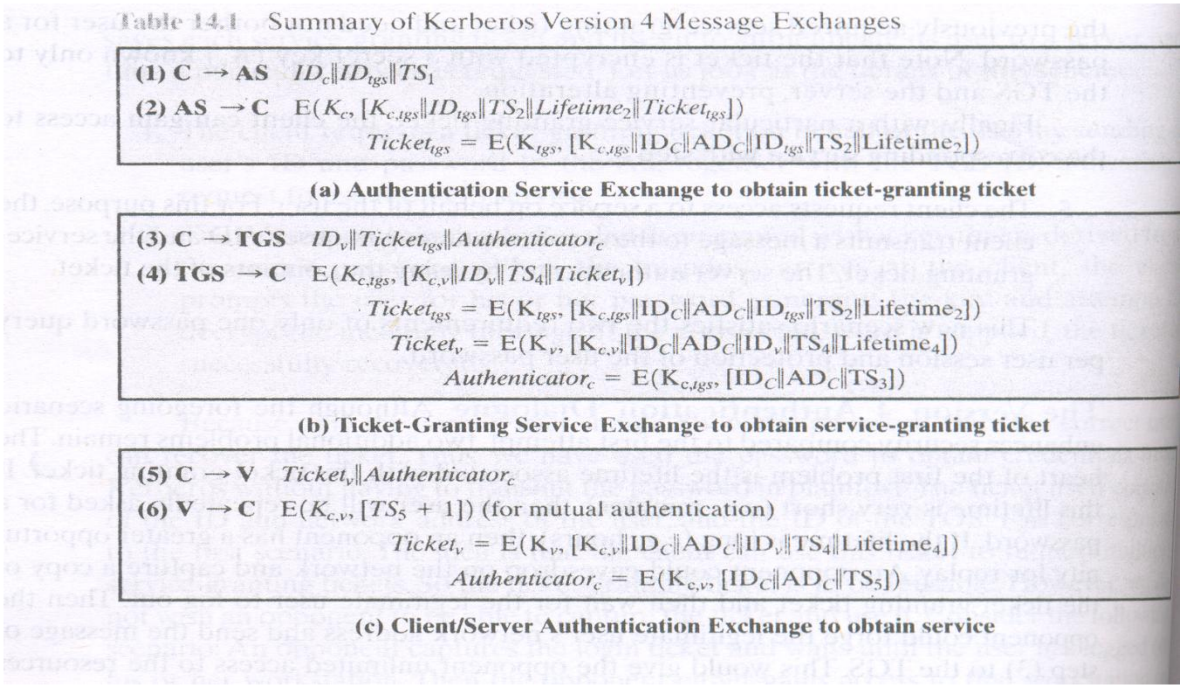


**Table 14.1** Summary of Kerberos Version 4 Message Exchanges

**(1) C → AS** $ID_c \| ID_{tgs} \| TS_1$

**(2) AS → C** $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

**(a) Authentication Service Exchange to obtain ticket-granting ticket**

**(3) C → TGS** $ID_v \| Ticket_{tgs} \| Authenticator_c$

**(4) TGS → C** $E(K_{c,tgs}, [K_{c,v} \| ID_v \| TS_4 \| Ticket_v])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$

$$Authenticator_c = E(K_{c,tgs}, [ID_C \| AD_C \| TS_3])$$

**(b) Ticket-Granting Service Exchange to obtain service-granting ticket**

**(5) C → V** $Ticket_v \| Authenticator_c$

**(6) V → C** $E(K_{c,v}, [TS_5 + 1])$ (for mutual authentication)

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$

$$Authenticator_c = E(K_{c,v}, [ID_C \| AD_C \| TS_5])$$

**(c) Client/Server Authentication Exchange to obtain service**

The TGS can decrypt the ticket with the key that it shares with the AS. This ticket indicates that user C has been provided with the session key $K_{c,tgs}$. The TGS session key to decrypt the authenticator. The TGS can then check the name address from the authenticator with that of the ticket and with the network address of the incoming message. If all match, then the TGS is assured that the sender of ticket is indeed the ticket's real owner.. It is the authenticator that proves the client's identity.

The message is encrypted with the session key shared by the TGS and C and includes a session key to he shared between C and the server V, the ID of V, and the timestamp of the ticket. The ticket itself includes the same session key. C now has a reusable service-granting ticket for V.

When C presents this ticket, it also sends an authenticator. The server can decrypt the ticket and recover the session key and decrypt the authenticator.

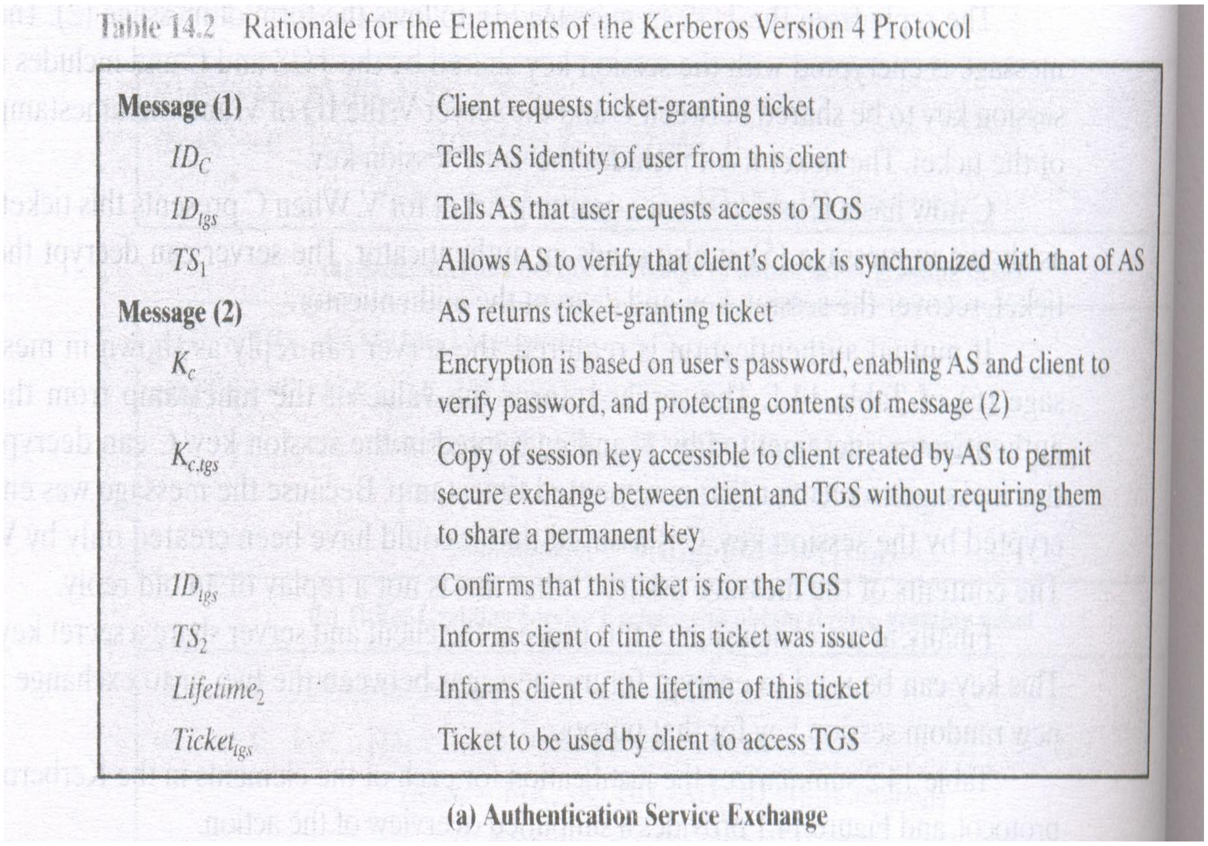
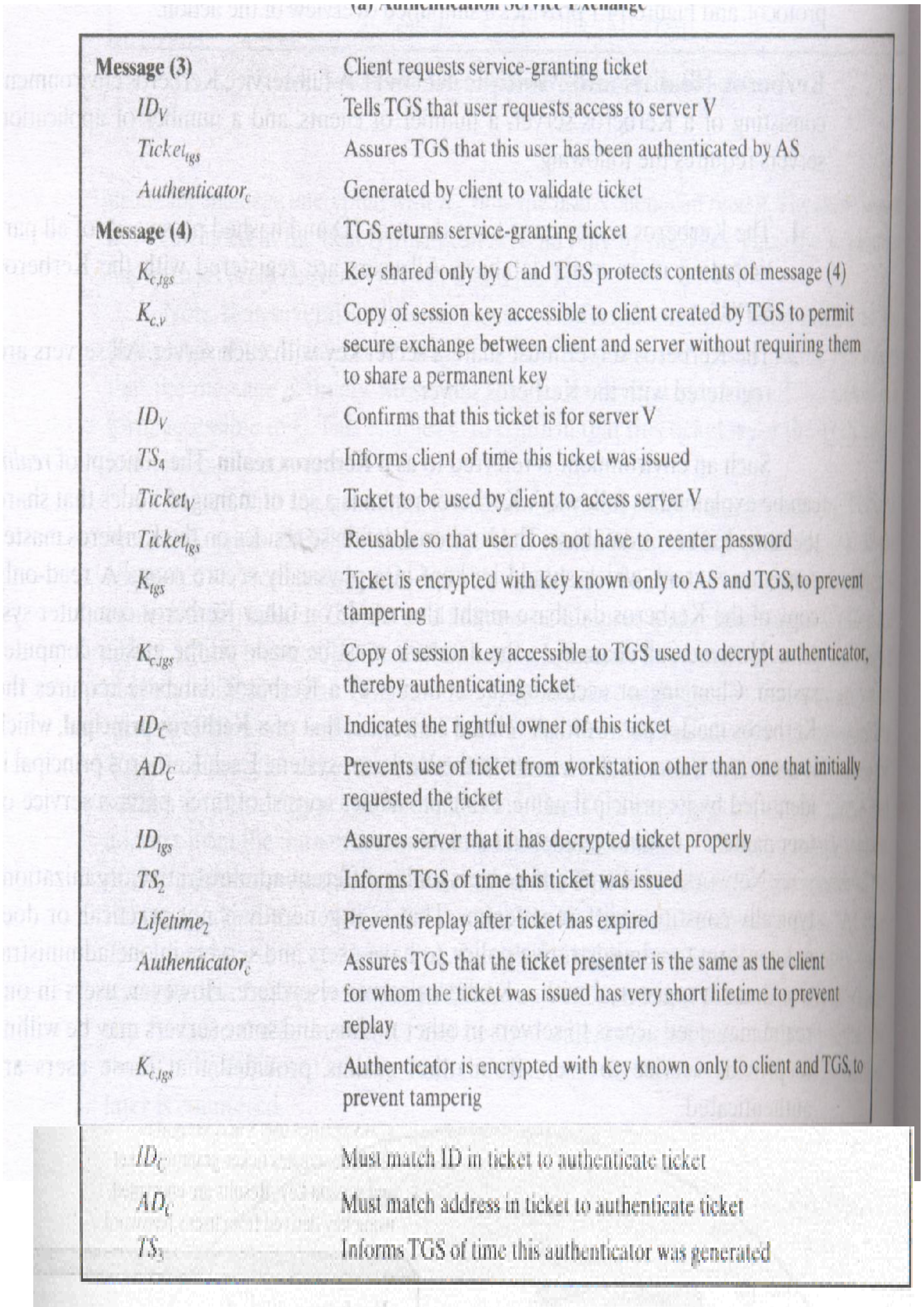


Table 14.2 Rationale for the Elements of the Kerberos Version 4 Protocol

| | |
|---|---|
| **Message (1)** | Client requests ticket-granting ticket |
| $ID_C$ | Tells AS identity of user from this client |
| $ID_{tgs}$ | Tells AS that user requests access to TGS |
| $TS_1$ | Allows AS to verify that client's clock is synchronized with that of AS |
| **Message (2)** | AS returns ticket-granting ticket |
| $K_c$ | Encryption is based on user's password, enabling AS and client to verify password, and protecting contents of message (2) |
| $K_{c,tgs}$ | Copy of session key accessible to client created by AS to permit secure exchange between client and TGS without requiring them to share a permanent key |
| $ID_{tgs}$ | Confirms that this ticket is for the TGS |
| $TS_2$ | Informs client of time this ticket was issued |
| $Lifetime_2$ | Informs client of the lifetime of this ticket |
| $Ticket_{tgs}$ | Ticket to be used by client to access TGS |

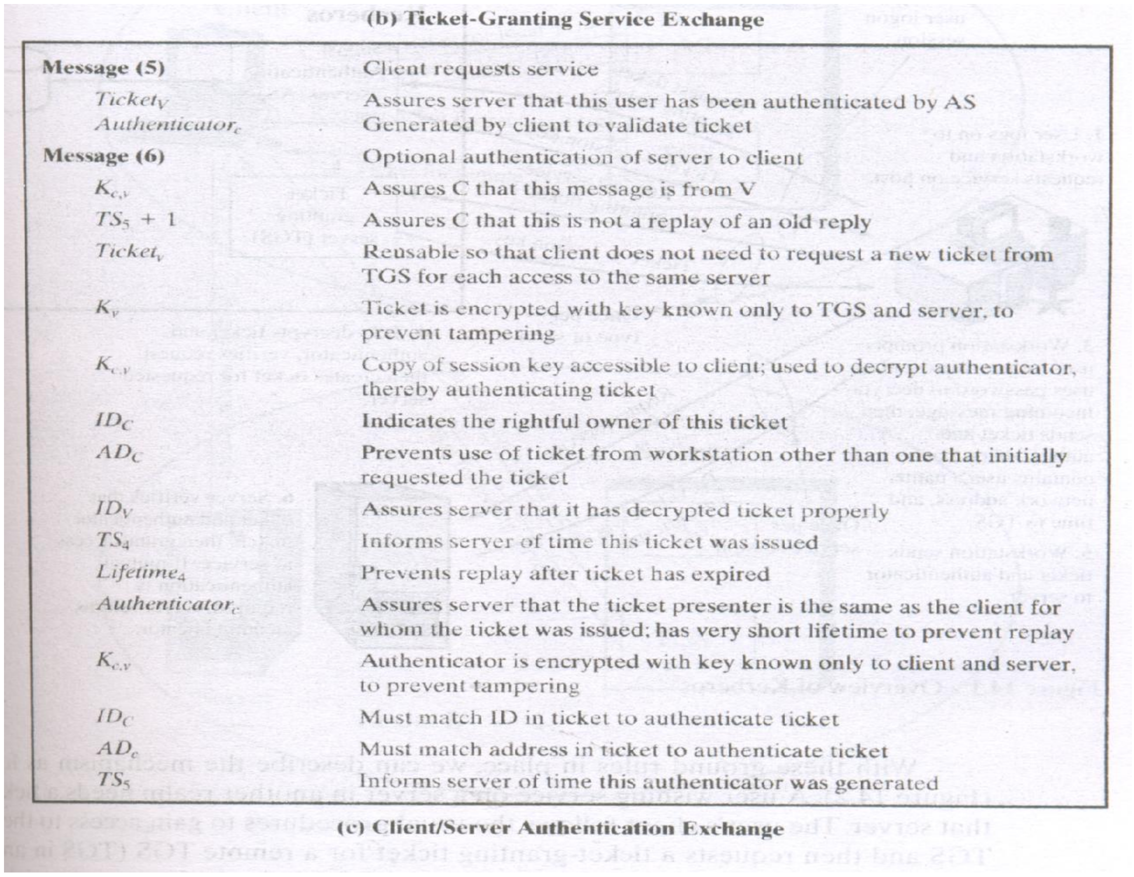
**(a) Authentication Service Exchange**

If mutual authentication is required, the server returns the value of the timestamp from the authenticator, incremented by 1, and encrypted in the session key. C can decrypt this message to recover the incremented timestamp. Because the session key encrypted the message, C is assured that only V could have created it. The contents of the message assure **C** that this is not a replay of an old reply.Finally, the client and server share a secret key. This key can he used to encrypt future messages between the two or to exchange a new random session key for that purpose.



| | |
|---|---|
| **Message (3)** | Client requests service-granting ticket |
| $ID_V$ | Tells TGS that user requests access to server V |
| $Ticket_{tgs}$ | Assures TGS that this user has been authenticated by AS |
| $Authenticator_c$ | Generated by client to validate ticket |
| **Message (4)** | TGS returns service-granting ticket |
| $K_{c,tgs}$ | Key shared only by C and TGS protects contents of message (4) |
| $K_{c,v}$ | Copy of session key accessible to client created by TGS to permit secure exchange between client and server without requiring them to share a permanent key |
| $ID_V$ | Confirms that this ticket is for server V |
| $TS_4$ | Informs client of time this ticket was issued |
| $Ticket_V$ | Ticket to be used by client to access server V |
| $Ticket_{tgs}$ | Reusable so that user does not have to reenter password |
| $K_{tgs}$ | Ticket is encrypted with key known only to AS and TGS, to prevent tampering |
| $K_{c,tgs}$ | Copy of session key accessible to TGS used to decrypt authenticator, thereby authenticating ticket |
| $ID_C$ | Indicates the rightful owner of this ticket |
| $AD_C$ | Prevents use of ticket from workstation other than one that initially requested the ticket |
| $ID_{tgs}$ | Assures server that it has decrypted ticket properly |
| $TS_2$ | Informs TGS of time this ticket was issued |
| $Lifetime_2$ | Prevents replay after ticket has expired |
| $Authenticator_c$ | Assures TGS that the ticket presenter is the same as the client for whom the ticket was issued has very short lifetime to prevent replay |
| $K_{c,tgs}$ | Authenticator is encrypted with key known only to client and TGS, to prevent tamperig |
| $ID_C$ | Must match ID in ticket to authenticate ticket |
| $AD_C$ | Must match address in ticket to authenticate ticket |
| $TS_3$ | Informs TGS of time this authenticator was generated |

**(b) Ticket-Granting Service Exchange**


(b) Ticket-Granting Service Exchange

| | |
|---|---|
| **Message (5)** | Client requests service |
| $Ticket_V$ | Assures server that this user has been authenticated by AS |
| $Authenticator_c$ | Generated by client to validate ticket |
| **Message (6)** | Optional authentication of server to client |
| $K_{c,v}$ | Assures C that this message is from V |
| $TS_5 + 1$ | Assures C that this is not a replay of an old reply |
| $Ticket_v$ | Reusable so that client does not need to request a new ticket from TGS for each access to the same server |
| $K_v$ | Ticket is encrypted with key known only to TGS and server, to prevent tampering |
| $K_{c,v}$ | Copy of session key accessible to client; used to decrypt authenticator, thereby authenticating ticket |
| $ID_C$ | Indicates the rightful owner of this ticket |
| $AD_C$ | Prevents use of ticket from workstation other than one that initially requested the ticket |
| $ID_V$ | Assures server that it has decrypted ticket properly |
| $TS_4$ | Informs server of time this ticket was issued |
| $Lifetime_4$ | Prevents replay after ticket has expired |
| $Authenticator_c$ | Assures server that the ticket presenter is the same as the client for whom the ticket was issued; has very short lifetime to prevent replay |
| $K_{c,v}$ | Authenticator is encrypted with key known only to client and server, to prevent tampering |
| $ID_C$ | Must match ID in ticket to authenticate ticket |
| $AD_c$ | Must match address in ticket to authenticate ticket |
| $TS_5$ | Informs server of time this authenticator was generated |

(c) Client/Server Authentication Exchange

**Kerberos Realms and Mu1tple Kerberi**

A full-service Kerberos environment consisting of a Kerberos server, a number of clients, and a number of application servers require the following:

1. The Kerberos server must have the user ID and hashed passwords of all participating users in its database. All users are registered with the Kerberos server.

2. The Kerberos server must share a secret key with each server. All servers are registered with the Kerberos server.

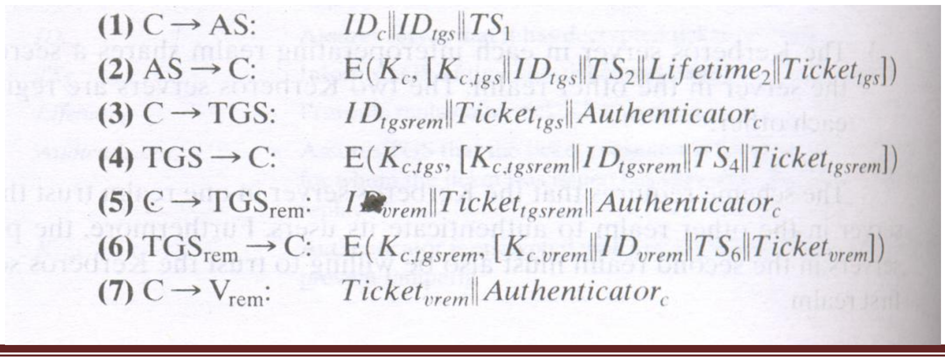
Such an environment is referred to as a **Kerberos realm.** The concept of realm can he explained as follows. A Kerberos realm is a set of managed nodes that share the same Kerberos database. The Kerberos database resides on the Kerberos master computer system, which should be kept in a physically secure room. A read-only copy of the Kerberos database might also reside on other Kerberos computer systems. However, all changes to the database must he made on the master computer system. Changing or accessing the contents of a Kerberos database requires the Kerberos master password. A related concept is that of a **Kerberos principal,** which is a service or user that is known to the Kerberos system. Each Kerberos principal is identified by its principal name. Principal names consist of three parts: a service or user name, an instance name, and a realm name.

Networks of clients and servers under different administrative organizations typically constitute different realms. The users in one realm may need access to servers in other realms, and some servers may be willing to provide service to users from other realms, provided that those users are authenticated.

Kerberos provides a mechanism for supporting such interrealm authentication. For two realms to support interrealm authentication, a third requirement is added:

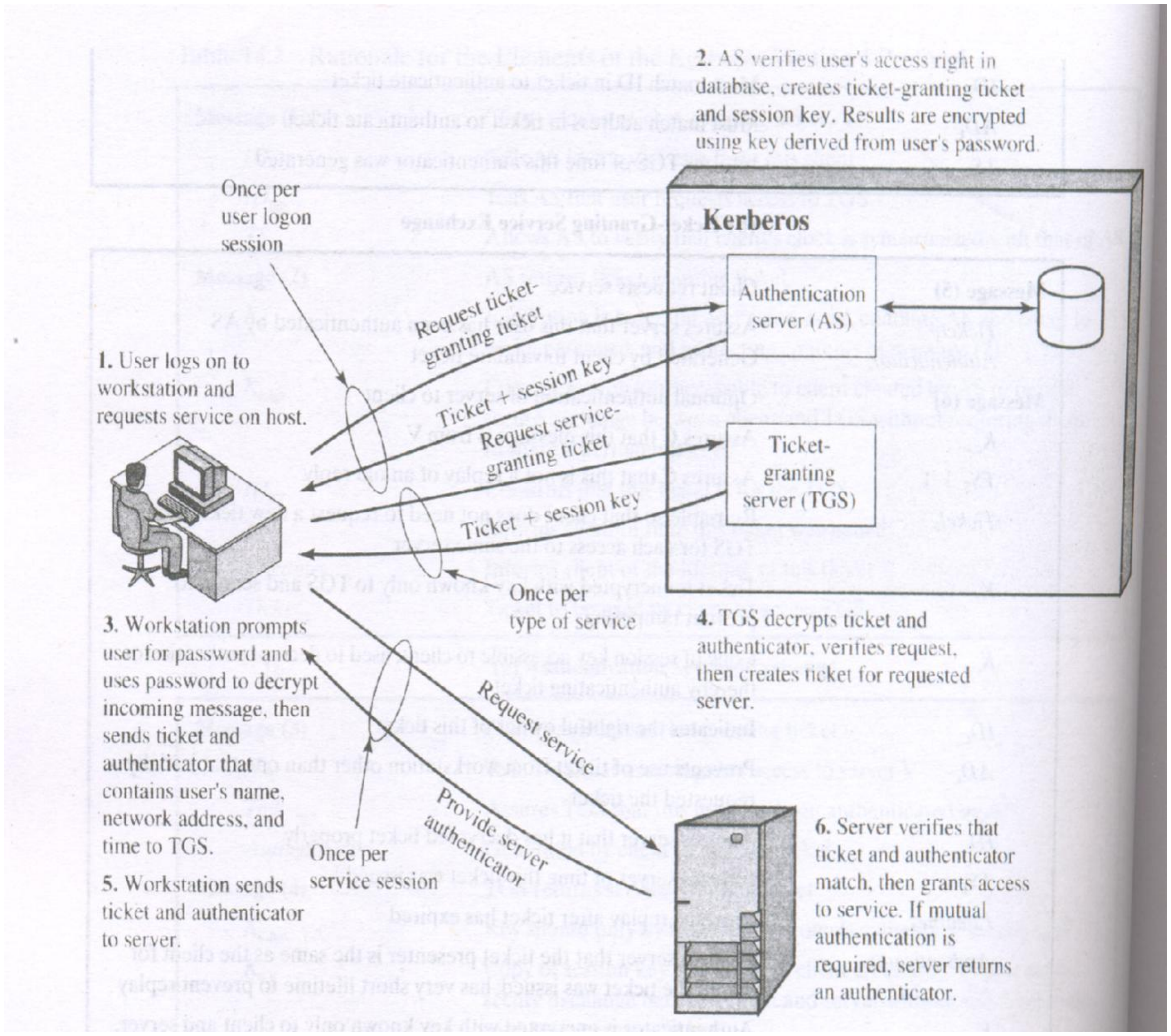
The Kerberos server in each interoperating realm shares a secret key with the server in the other realm. The two Kerberos servers are registered with each other. The scheme requires that the Kerberos server in one realm trust the Kerberos server in the other realm to authenticate its users. Furthermore, the participating servers in the second realm must also be willing to trust the Kerberos server in the first realm.

With these ground rules in place, we can describe the mechanism as shown in figure. A user wishing service on a server in another realm needs a ticket for that server. The user's client follows the usual procedures to gain access to the local TGS and then requests a ticket-granting ticket for a remote TGS (TGS in another realm). The client can then apply to the remote TGS for a service-granting ticket for the desired server in the realm of the remote TGS. The details of the exchange are given below.



| | |
|---|---|
| **(1)** $C \rightarrow AS$: | $ID_c \| ID_{tgs} \| TS_1$ |
| **(2)** $AS \rightarrow C$: | $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$ |
| **(3)** $C \rightarrow TGS$: | $ID_{tgsrem} \| Ticket_{tgs} \| Authenticator_c$ |
| **(4)** $TGS \rightarrow C$: | $E(K_{c,tgs}, [K_{c,tgsrem} \| ID_{tgsrem} \| TS_4 \| Ticket_{tgsrem}])$ |
| **(5)** $C \rightarrow TGS_{rem}$: | $ID_{vrem} \| Ticket_{tgsrem} \| Authenticator_c$ |
| **(6)** $TGS_{rem} \rightarrow C$: | $E(K_{c,tgsrem}, [K_{c,vrem} \| ID_{vrem} \| TS_6 \| Ticket_{vrem}])$ |
| **(7)** $C \rightarrow V_{rem}$: | $Ticket_{vrem} \| Authenticator_c$ |

The ticket presented to the remote server (Vrem) indicates the realm in which the user was originally authenticated. The server chooses whether to honor the remote request.


2. AS verifies user's access right in database, creates ticket-granting ticket and session key. Results are encrypted using key derived from user's password.
Once per user logon session
Kerberos
Authentication server (AS)
Request ticket-granting ticket
Ticket + session key
Request service-granting ticket
Ticket + session key
Ticket-granting server (TGS)
1. User logs on to workstation and requests service on host.
Once per type of service
3. Workstation prompts user for password and uses password to decrypt incoming message, then sends ticket and authenticator that contains user's name, network address, and time to TGS.
4. TGS decrypts ticket and authenticator, verifies request, then creates ticket for requested server.
Request service
Provide server authenticator
Once per service session
5. Workstation sends ticket and authenticator to server.
6. Server verifies that ticket and authenticator match, then grants access to service. If mutual authentication is required, server returns an authenticator.

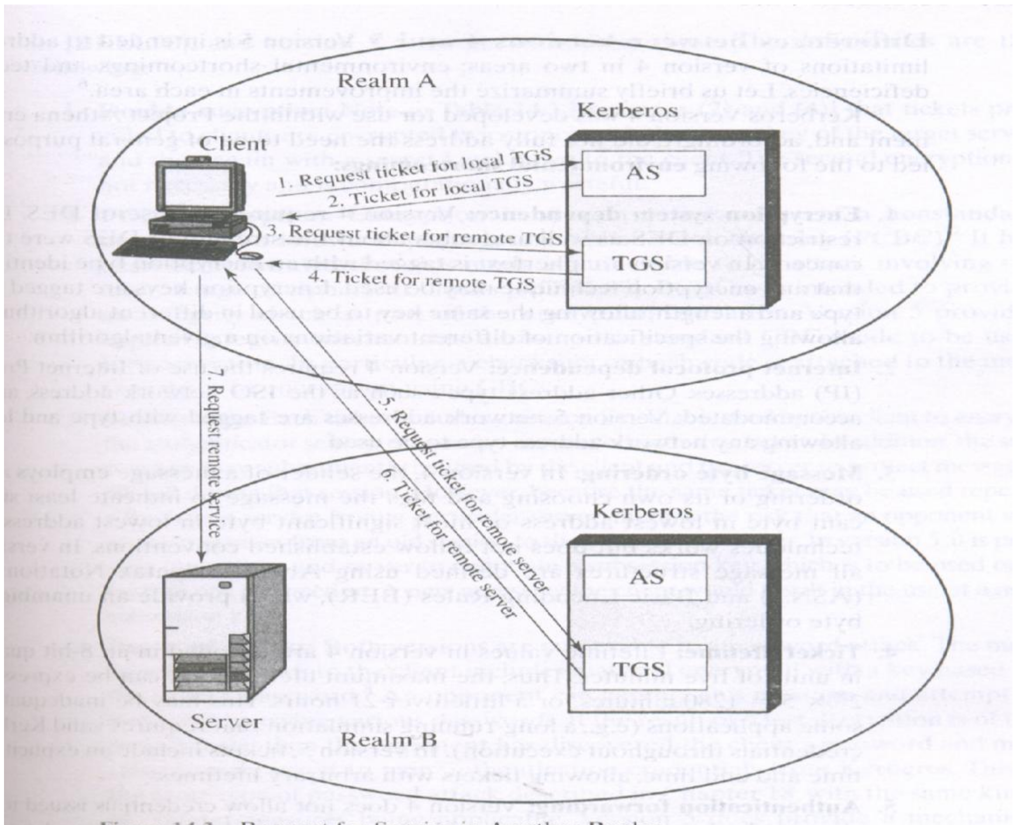

Realm A
Kerberos
Client
AS
TGS
1. Request ticket for local TGS
2. Ticket for local TGS
3. Request ticket for remote TGS
4. Ticket for remote TGS
5. Request ticket for remote server
6. Ticket for remote server
7. Request remote service
Kerberos
AS
TGS
Server
Realm B

**Figure 14.2** Request for Service in Another Realm

**Differences between Versions 4 and 5**

The environmental shortcomings of version 4 when compared with version 5 are:

1) Encryption system dependence: Version 4 requires the use of DES. Export restriction on DES as well as doubts about the strength of DES were thus of concern. In version 5, ciphertext is tagged with an encryption type identifier so that any encryption technique may be used. Encryption keys are tagged with a type and a length, allowing the same key to be used in different algorithms and allowing the specification of different variations on a given algorithm.
2) Internet protocol dependence: Version 4 requires the use of Internet Protocol (IP) addresses. Other address types, such as the ISO network address, are not accommodated.

Version 5 network addresses are tagged with type and length, allowing any network address type to be used.

3) Message byte ordering: In version 4, the sender of a message employs a byte ordering of its own choosing and tags the message to indicate least significant byte in lowest address or most significant byte in lowest address. This techniques works but does not follow established conventions. In version 5, all message structures are defined using Abstract Syntax Notation One (ASN.1) and Basic Encoding Rules (BER), which provide an unambiguous byte ordering.
4) Ticket lifetime: Lifetime values in version 4 are encoded in an 8-bit quantity in units of five minutes. Thus, the maximum lifetime that can be expressed is $2^8$ x 5 = 1280 minutes, or a little over 21 hours. This may be inadequate for some applications (e.g., a long-running simulation that requires valid Kerberos credentials throughout execution). In version 5, tickets include an explicit start time and end time, allowing tickets with arbitrary lifetimes.
5) Authentication forwarding: Version 4 does not allow credentials issued to one client to be forwarded to some other host and used by some other client. This capability would enable a client to access a server and have that server access another server on behalf of the client. For example, a client issues a request to a print server that then accesses the client's file from a file server, using the client's credentials for access. Version 5 provides this capability.
6) Interrealm authentication: In version 4, interoperability among N realms requires on the order of $N^2$ Kerberos-to-Kerberos relationships, as described earlier. Version 5 supports a method that requires fewer relationships, as described shortly.

The technical deficiencies in the version 4 protocol itself when compared to version 5 are:

1) Double encryption: The tickets provided to clients are encrypted twice, once with the secret key of the target server and then again with a secret key known to the client. The second encryption is not necessary and is computationally wasteful.
2) PCBC encryption: Encryption in version 4 makes use of a nonstandard mode of DES known as propagating cipher block chaining (PCBC). It has been demonstrated that this mode is vulnerable to an attack involving the interchange of ciphertext blocks. PCBC was

intended to provide an integrity check as part of the encryption operation. Version 5 provides explicit integrity mechanisms, allowing the standard CBC mode to be used for encryption. In particular, a checksum or hash code is attached to the message prior to encryption using CBC.

3) Session keys: Each ticket includes a session key that is used by the client to encrypt the authenticator sent to the service associated with that ticket. In addition, the session key may subsequently be used by the client and the server to protect messages passed during that session. However, because the same ticket may be used repeatedly to gain service from a particular server, there is the risk that an opponent will replay messages from an old session to the client or the server. In version 5, it is possible for a client and server to negotiate a subsession key, which is to be used only for that one connection. A new access by the client would result in the use of a new subsession key.
4) Password attacks: Both versions are vulnerable to a password attack. The message from the AS to the client includes material encrypted with a key based on the client's password. An opponent can capture this message and attempt to decrypt it by trying various passwords. If the result of a test decryption is of the proper form, then the opponent has discovered the client's password and may subsequently use it to gain authentication credentials from Kerberos. Version 5 does provide a mechanism known as preauthentication, which should make password attacks more difficult, but it does not prevent them.

**The Version 5 Authentication Dialogue**

Consider the authentication service exchange. Message (1) is a client request for a ticket-granting ticket. As before, it includes the ID of the user and the TGS. The following new elements are added:

Realm: Indicates realm of user

Options: Used to request that certain flags be set in the returned ticket

Times: Used by the client to request the following time settings in the ticket:

from: the desired start time for the requested ticket

till: the requested expiration time for the requested ticket

rtime: requested renew-till time

Nonce : A random value to be repeated in message (2) to assure that the response is fresh and has not been replayed by an opponent.

Message (2) returns a ticket-granting ticket, identifying information for the client, and a block encrypted using the encryption key based on the user's password. This block includes the session key to be used between the client and the TGS, times specified in message (1), the nonce from message (1), and TGS identifying information. The ticket itself includes the session key, identifying information for the client, the requested time values, and flags that reflect the status of this ticket and the requested options. These flags introduce significant new functionality to version 5.

The message (3) for both versions includes an authenticator, a ticket, and the name of the requested service. In addition, version 5 includes requested times and options for the ticket and a nonce, all with functions similar to those of message (1). The authenticator itself is essentially the same as the one used in version 4. Message (4) has the same structure as message (2), returning a ticket plus information needed by the client, the latter encrypted with the session key now shared by the client and the TGS.

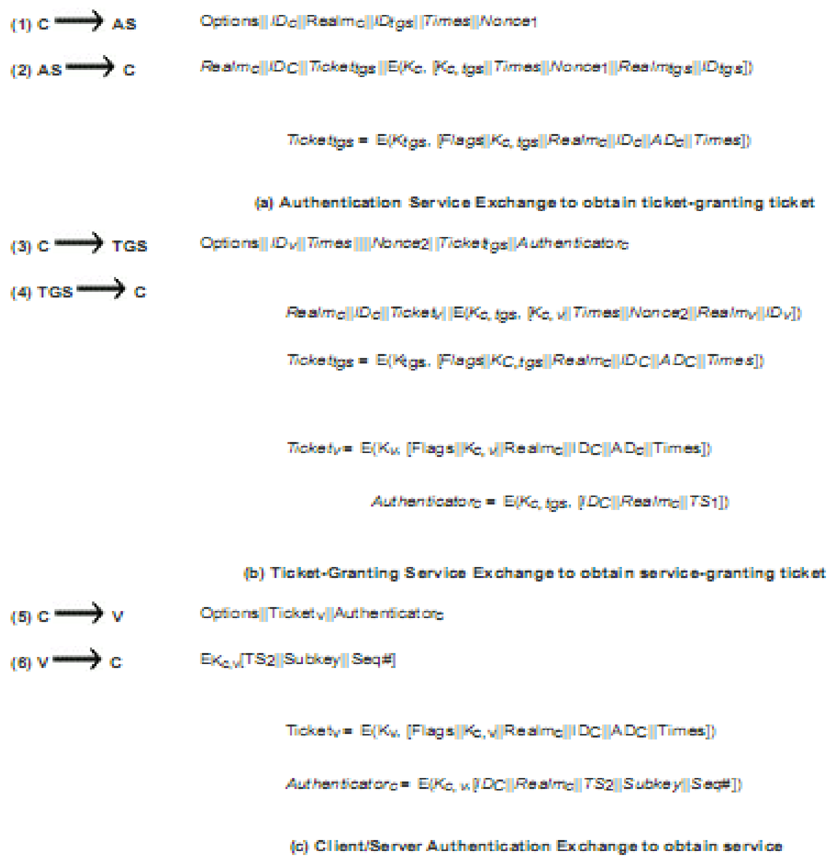
Finally, for the client/server authentication exchange, several new features appear in version 5. In message (5), the client may request as an option that mutual authentication is required. The authenticator includes several new fields as follows:

Subkey: The client's choice for an encryption key to be used to protect this specific application session. If this field is omitted, the session key from the ticket ( $K_{c,v}$ ) is used.

Sequence number: An optional field that specifies the starting sequence number to be used by the server for messages sent to the client during this session. Messages may be sequence numbered to detect replays.

If mutual authentication is required, the server responds with message (6). This message includes the timestamp from the authenticator. Note that in version 4, the timestamp was incremented by one. This is not necessary in version 5 because the nature of the format of messages is such that it is not possible for an opponent to create message (6) without knowledge of the appropriate encryption keys. The optional sequence number field specifies the starting sequence number to be used by the client.

**Table .3. Summary of Kerberos Version 5 Message Exchanges**



(1) $C \longrightarrow AS$ $\quad Options \| ID_c \| Realm_c \| ID_{tgs} \| Times \| Nonce_1$

(2) $AS \longrightarrow C$ $\quad Realm_c \| ID_C \| Ticket_{tgs} \| E(K_c, [K_{c,tgs} \| Times \| Nonce_1 \| Realm_{tgs} \| ID_{tgs}])$

$Ticket_{tgs} = E(K_{tgs}, [Flags \| K_{c,tgs} \| Realm_c \| ID_c \| AD_c \| Times])$

**(a) Authentication Service Exchange to obtain ticket-granting ticket**

(3) $C \longrightarrow TGS$ $\quad Options \| ID_v \| Times \|\| Nonce_2 \| Ticket_{tgs} \| Authenticator_c$

(4) $TGS \longrightarrow C$

$Realm_c \| ID_c \| Ticket_v \| E(K_{c,tgs}, [K_{c,v} \| Times \| Nonce_2 \| Realm_v \| ID_v])$

$Ticket_{tgs} = E(K_{tgs}, [Flags \| K_{C,tgs} \| Realm_c \| ID_C \| AD_C \| Times])$

$Ticket_v = E(K_v, [Flags \| K_{c,v} \| Realm_c \| ID_C \| AD_c \| Times])$

$Authenticator_c = E(K_{c,tgs}, [ID_C \| Realm_c \| TS_1])$

**(b) Ticket-Granting Service Exchange to obtain service-granting ticket**

(5) $C \longrightarrow V$ $\quad Options \| Ticket_v \| Authenticator_c$

(6) $V \longrightarrow C$ $\quad E_{K_{c,v}}[TS_2 \| Subkey \| Seq\#]$

$Ticket_v = E(K_v, [Flags \| K_{c,v} \| Realm_c \| ID_C \| AD_C \| Times])$

$Authenticator_c = E(K_{c,v}, [ID_C \| Realm_c \| TS_2 \| Subkey \| Seq\#])$

**(c) Client/Server Authentication Exchange to obtain service**

## X.509 Authentication Services

X.509 is a directory service. The directory is a server or distributed set of servers that maintains a database of information about all users. The information includes a mapping from user name to network address, as well as other attributes and information about the users. X.509 defines authentication protocol for public key certificates. It is based on the use of public key cryptography and digital signatures. Figure illustrates the generation of public key certificates.

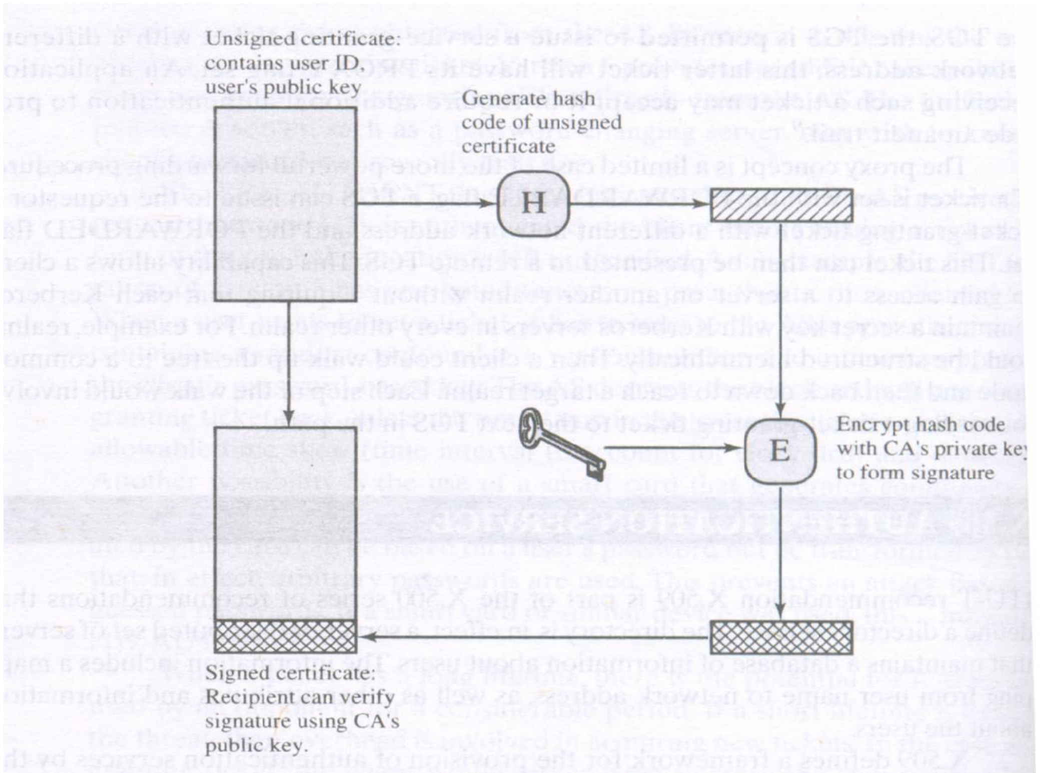

Unsigned certificate: contains user ID, user's public key
Generate hash code of unsigned certificate
H
Encrypt hash code with CA's private key to form signature
E
Signed certificate: Recipient can verify signature using CA's public key.

**Figure 14.3** Public-Key Certificate Use

**X.509 Certificates**

The heart of the X.509 scheme is the public-key certificate associated with each user. These user certificates are assumed to be created by some trusted certification authority (CA) and placed in the directory by the CA or by the user. The directory server itself is not responsible for the creation of public keys or for the certification function; it merely provides an easily accessible location for obtaining certificates. The format of certificate includes the following

**Version**

Differentiates among successive versions of the certificate format, the default is version 1. If the Issuer Unique identifier or Subject Unique Identifier are present, the value must he version 2. If one or more extensions are present the version must be version 3.

**Serial number**

An integer value, unique within the issuing CA, that is associated with this certificate.

Signature algorithm **identifier**

The algorithm used to sign the certificate, together with any associated parameters. This information is repeated in the Signature field at the end of the certificate.

**Issuer name**

It is the X.500 name of the CA that created and signed this certificate.

**Period of validity**

Consists of two dates: the first and last on which the certificate is valid.

**Subject name**

The name of the user to whom this certificate refers. That is, this certificate certifies the public key of the subject who holds the corresponding private key.

**Subject’s public-key information**

It is the public key of the subject, plus an identifier of the algorithm for which this key is to be used, together with any associated parameters.

**Issuer unique identifier**

An optional bit string field used to identify uniquely the issuing CA in the event the X.500 name has been reused for different entities.

**Subject unique identifier**

An optional bit string field used to identify uniquely the subject in the event the X.500 name has been reused for different entities.

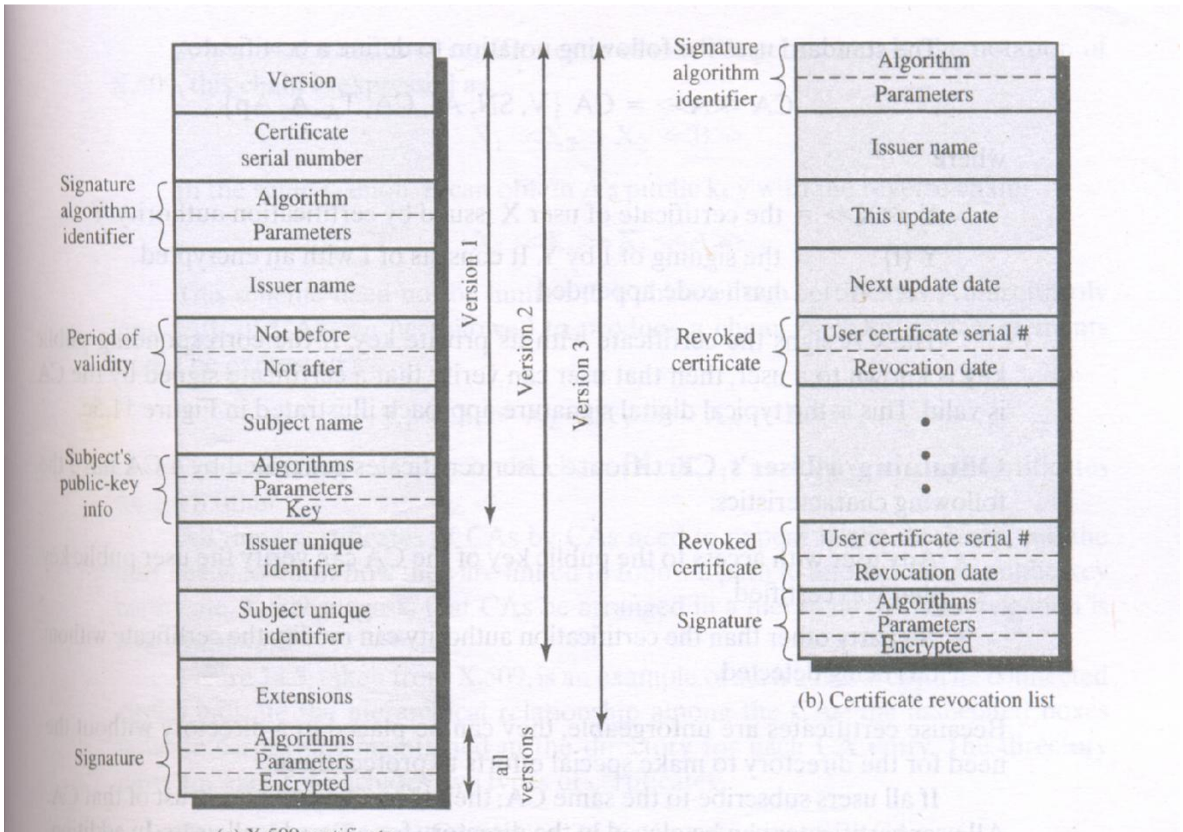


**Figure 14.4** X.509 Formats

**Extensions**

They are a set of one or more extension fields.

**Signature**

Covers all of the other fields of the certificate; it contains the hash code of the other fields, encrypted with the CA's private key. This field includes the signature algorithm identifier.

The standard uses the following notation to define a certificate:

CA <<A>> CA {V, SN, Al, CA, TA, A, Ap}

where Y <<X>> = the cettificate of user X issued by certification authority Y

Y {I} = the signing of I by Y. It consists of I with an encrypted hash code appended

The CA signs the certificate with its private key. If the corresponding key is known to a user, then that user can verify that a certificate signed by the CA is valid.

Obtaining a User’s Certificate User certificates generated by a CA have following characteristics:

Any user with access to the public key of the CA can verify the user public that was certified.

No party other than the certification authority can modify the certificate’ this being detected.

Because certificates are unforgeable, they can be placed in a directory without the need for the directory to make special efforts to protect them.

If all users subscribe to the same CA, then there is a common trust of that CA. All users can place all user certificates in the directory for access. In addition, a user can transmit his or her certificate directly to other users. In either case, once B is in possession of A’s certificate, B has confidence that messages it encrypts with A’s public key will be secure from eavesdropping and that messages signed with A private key are unforgeable.

If there is a large community of users, it may not be practical for all users to subscribe to the same CA. Because it is the CA that signs certificates, each participating user must have a copy of the CA’s own public key to verify signatures. This public key must be provided to each user in an absolutely secure (with respect to integrity and authenticity) way so that the user has confidence in the associated certificates. Thus with many users, it may be more practical for there to be a number of CAs, each of which securely provides its public key to some fraction of the users.

Now suppose that A has obtained a certificate from certification authorityX1 and B has obtained a certificate from CA X2. If A does not securely know the public key of X2, then B’s certificate, issued by X2, is useless to A. A can read B's certificate, but A cannot verify the signature. However, if the two CAs have securely exchanged their own public keys, the following procedure will enable A to obtain B's public key:

- A obtains, from the directory, the certificate of X2 signed by X1. Because A securely knows X1's public key, A can obtain X2's public key from its certificate and verify it by means of X1's signature on the certificate.

- A then goes back to the directory and obtains the certificate of B signed by X Because A now has a trusted copy of X2's public key, A can verify the signature and securely obtain B's public key.

A has used a chain of certificates to obtain B's public key. In the notation of X.509, this chain is expressed as

**X1 <<X>> X2 <<B>>**

In the same fashion, B can obtain A's public key with the reverse chain:

X2<<X1>>X1<<A>>

This scheme need not be limited to a chain of two certificates. An arbitrarily long path of CAs can be followed to produce a chain. A chain with N elements would be expressed as

X1 <<X2>> X2 <<X3>> . X <<B>>

In this case, each pair of CAs in the chain (Xi, Xi+1) must have created certificates for each other.

X.509 suggests that CAs be arranged in a hierarchy so that navigation is straightforward. Figure is an example of such a hierarchy. The connected circles indicate the hierarchical relationship among the CAs; the associated boxes indicate certificates maintained in the directory for each CA entry. The directory entry for each CA includes two types of certificates:

- **Forward certificates:** Certificates of X generated by other CAs

- **Reverse certificates:** Certificates generated by X that are the certificates of other CAs .

In this example, user A can acquire the following certificates from the directory to establish a certification path to B:

**X <<W>> W <<V>> V** <<Y>>Y **<<Z>> Z <<B>>**

When A has obtained these certificates, it can unwrap the certification path in sequence to recover a trusted copy of B's public key. Using this public key, A can send encrypted messages to **B.** if A wishes to receive encrypted messages back from B, or to sign messages sent to B, then B will require A's public key, which can be obtained from the following certification path:

**Z <<Y>> Y <<V>> V <<W>> W <<X>> X <<A>>**

B can obtain this set of certificates from the directory, or A can provide them as part of its initial message to B.

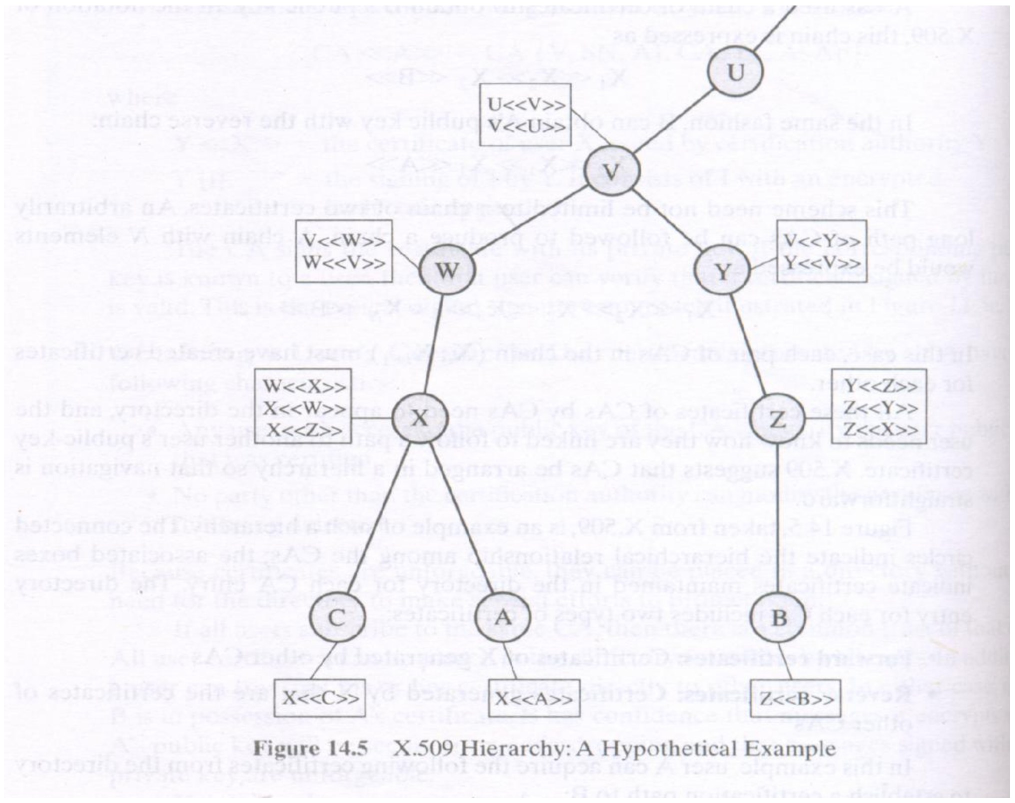


**Figure 14.5** X.509 Hierarchy: A Hypothetical Example

**Revocation of Certificates**

Each certificate includes a period of validity. Typically, a new certificate is issued just before the expiration of the old one. In addition, it may be desirable on occasion to revoke a certificate before it expires, for one of the following reasons:

- The user's private key is assumed to be compromised.
- This CA no longer certifies the user.
- The CA's certificate is assumed to be compromised.

Each CA must maintain a list consisting of all revoked but not expired certificates issued by that CA, including both those issued to users and to other CAs. These, lists should also be posted on the directory.

Each certificate revocation list (CRL) posted to the directory is signed by the issuer and includes the issuer's name, the date the list was created, the date the next CRL is scheduled to be issued, and an entry for each revoked certificate. Each entry consists of the serial number of a certificate and revocation date for that certificate. Because serial numbers are unique within a CA, the serial number sufficient to identify the certificate.

When a user receives a certificate in a message, the user must determine whether the certificate has been revoked. The user could check the directory each time a certificate is received. To avoid the delays associated with directory searches, it is likely that the user would maintain a local cache of certificates and lists of revoked certificates.

**Authentication Procedures**

X.509 also includes three alternative authentication procedures that are intended for use across a variety of applications. All these procedures make use of public-key signatures.

It is assumed that the two parties know each other's public key, either by obtaining each other's certificates from the directory or because the certificate is included in the initial message from each side.

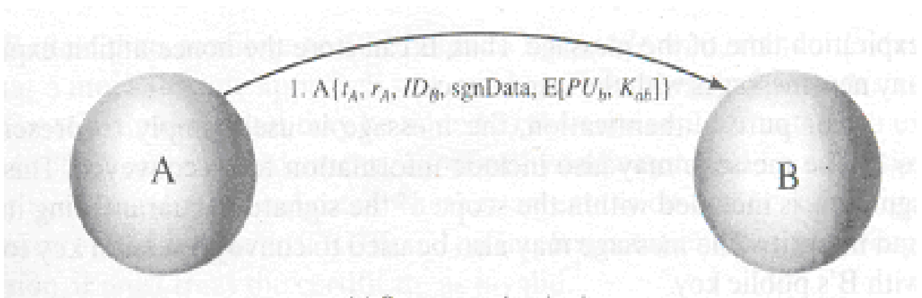
**One-Way Authentication**

One way authentication involves a single transfer of information from one user (A) to another (B), and establishes the following:

- The identity of A and that the message was generated by A
- That the message was intended for B
- The integrity and originality of the message

Note that only the identity of the initiating entity is verified in this process, not that of the responding entity.

At a minimum, the message includes a timestamp **$t_A$,** a nonce **$r_A$,** and the identity of B and is signed with A's private key. The timestamp consists of an optional generation time and an expiration time. This prevents delayed delivery of messages. lhe nonce can be used to detect replay attacks. The nonce value must be unique within the expiration time of the message. Thus B can store the nounce until it expires and reject the new message with the same nounce.



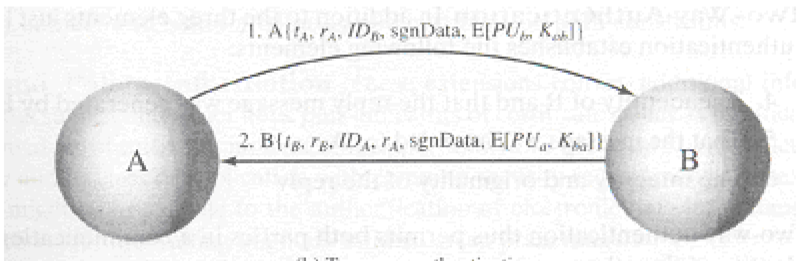
(a) One-way authentication

**Two-Way Authentication**

In addition to the three elements just listed, it establishes the following elements.

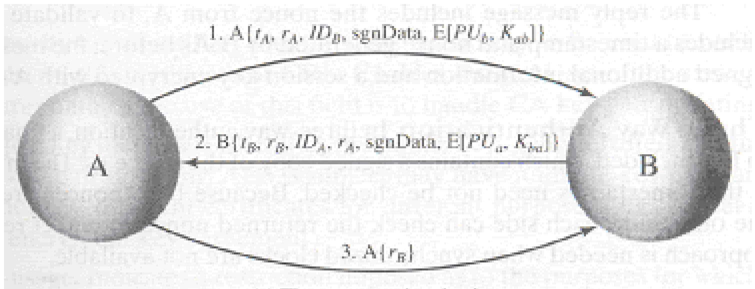
- The identity of B and that the reply message was generated by B
- That the message was intended for A
- The integrity and originality of reply

Thus it permits both parties of the communication to authenticate each other.


1. A{$t_A$, $r_A$, $ID_B$, sgnData, E[$PU_b$, $K_{ab}$]}
2. B{$t_B$, $r_B$, $ID_A$, $r_A$, sgnData, E[$PU_a$, $K_{ba}$]}
A
B
(b) Two-way authentication

**Three-way authentication**

In this a final message from A to B is included, which contains a signed copy of a nounce $r_{B.}$ the indent of this design is that timestamp need not be checked, because both the nounce are echoed back by the other side, each side can check the returned nounce to detect repay attacks.


1. A{$t_A$, $r_A$, $ID_B$, sgnData, E[$PU_b$, $K_{ab}$]}
2. B{$t_B$, $r_B$, $ID_A$, $r_A$, sgnData, E[$PU_a$, $K_{ba}$]}
3. A{$r_B$}
A
B
(c) Three-way authentication