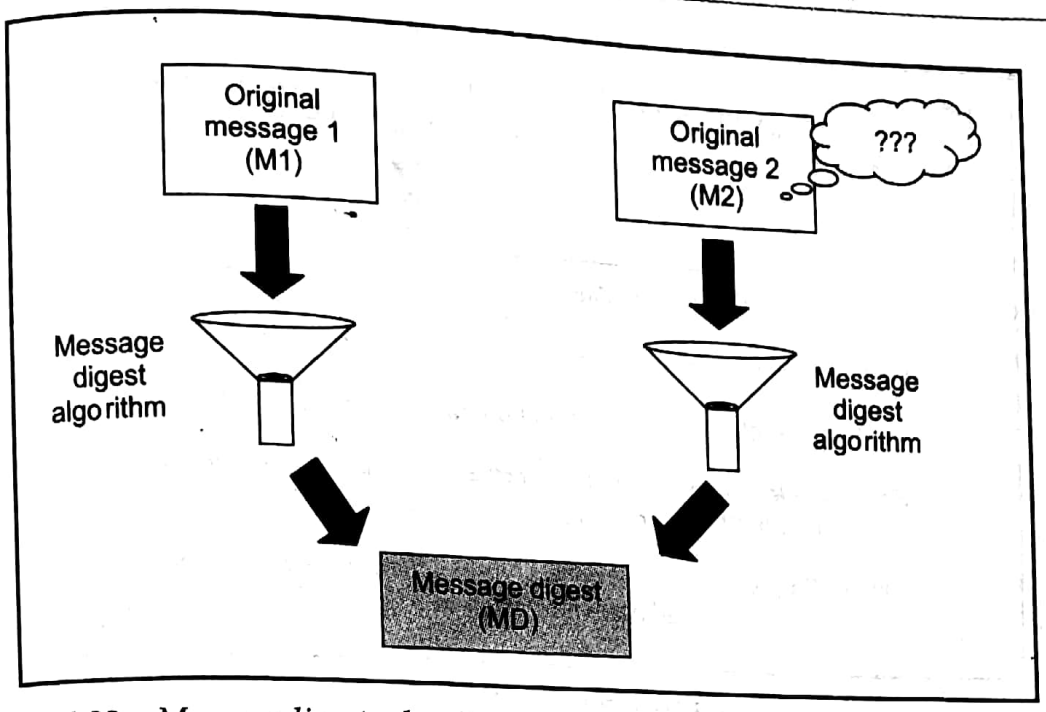

Fig. 4.22 *Message digests should not reveal anything about the original message*

Rivest. The original message digest algorithm was called as **MD**. He soon came up with its next version, **MD2**. Rivest first developed it, but it was found to be quite weak. Therefore, Rivest began working on **MD3**, which was a failure (and therefore was never released). Then, Rivest developed **MD4**. However, soon, MD4 was also found to be wanting. Consequently, Rivest released MD5.

MD5 is quite fast and produces 128-bit message digests. Over the years, researchers have developed potential weaknesses in MD5. However, so far, MD5 has been able to successfully defend itself against collisions. This may not be guaranteed for too long, though.

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.

How MD5 Works?

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 bits less than an exact multiple of 512. For example, if the length of the original message is 1000 bits, we add a padding of 472 bits to make the length of the message 1472 bits. This is because, if we add 64 to 1472, we get 1536, which is a multiple of 512 (because $1536 = 512 \times 3$).

Thus, after padding, the original message will have a length of 448 bits (64 bits less than 512), 960 bits (64 bits less than 1024), 1472 bits (64 bits less than 1536), etc.

The padding consists of a single 1-bit, followed by as many 0-bits, as required. Note that padding is always added, even if the message length is already 64 bits less than a multiple of 512. Thus, if the message were already of length say 448 bits, we will add a padding of 512 bits to make its length 960 bits. Thus, the padding length is any value between 1 and 512.

The padding process is shown in Fig. 4.23.

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. How is this done?

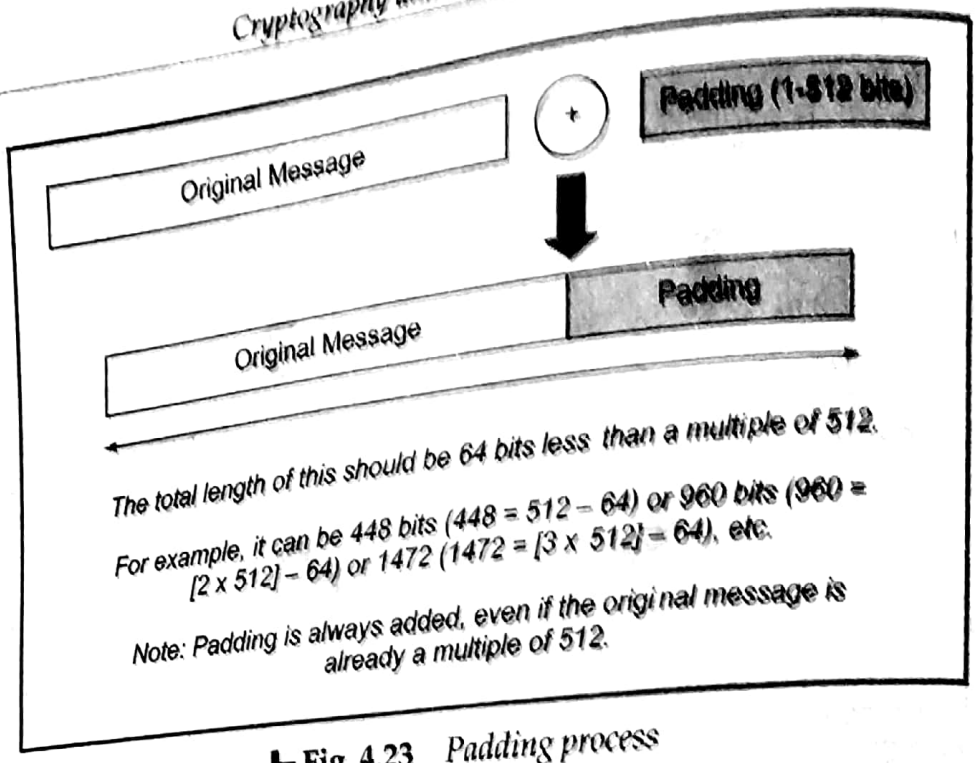

Fig. 4.23 *Padding process*

The length of the message is calculated, excluding the padding bits (i.e. it is the length before the padding bits were added). For instance, if the original message consisted of 1000 bits and we added padding of 472 bits to make the length of the message 64 bits less than 1536 (a multiple of 512), the length is considered as 1000 and not 1472 for the purpose of this step.

This length of the original message is now expressed as a 64-bit value and these 64 bits are appended to the end of the original message + padding. This is shown in Fig. 4.24. Note that if the length of the

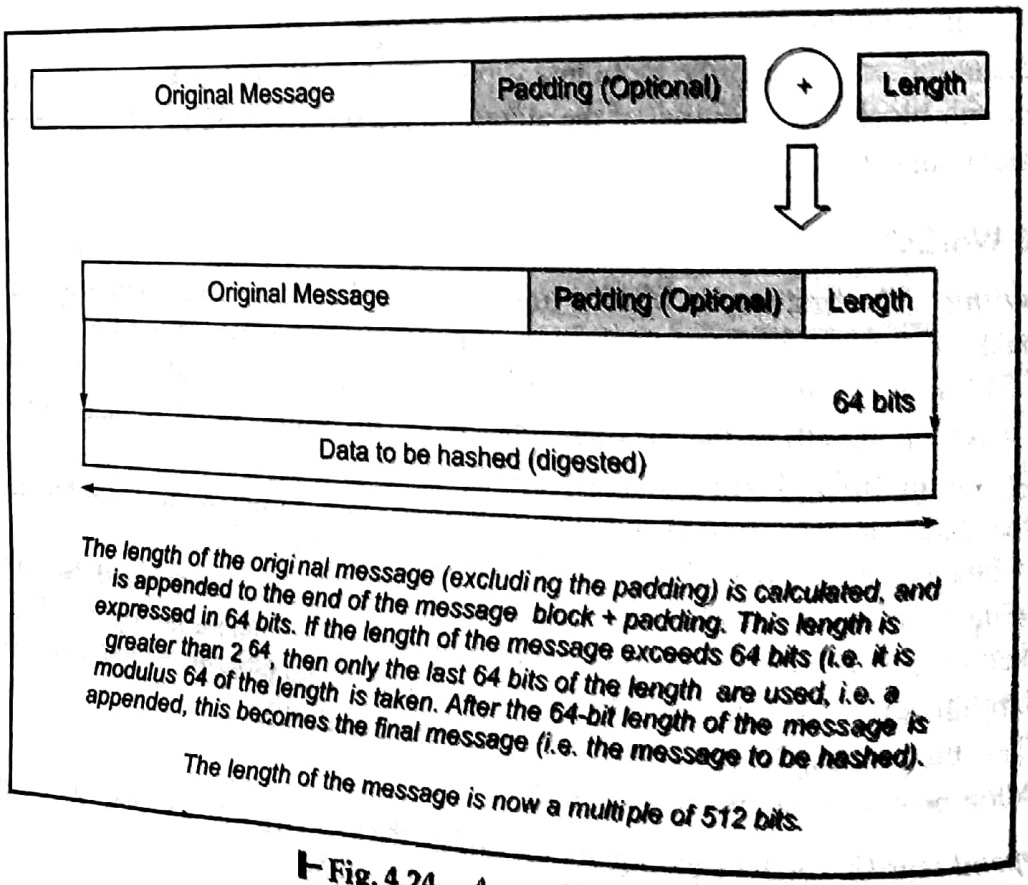

Fig. 4.24 *Append length*

message exceeds 2^{64} bits (i.e. 64 bits are not enough to represent the length, which is possible in the case of a really long message), we use only the low-order 64 bits of the length. That is, in effect, we calculate the length mod 2^{64} in that case.

We will realize that the length of the message is now an exact multiple of 512. This now becomes the message whose digest will be calculated.

Step 3: Divide the input into 512-bit blocks Now, we divide the input message into blocks, each of length 512 bits. This is shown in Fig. 4.25.

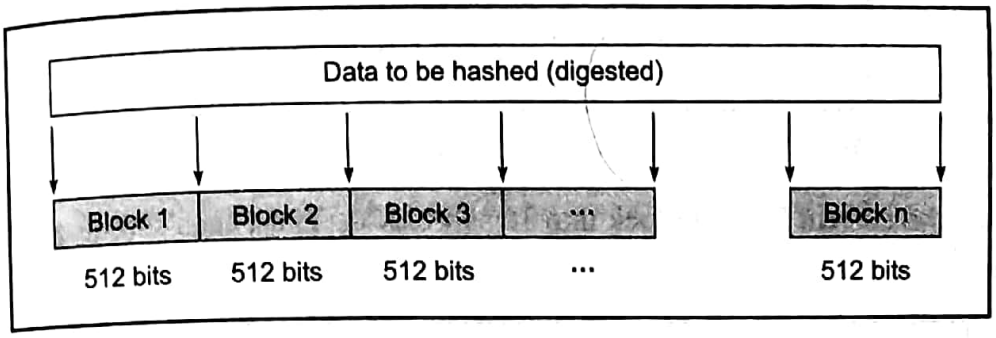

Fig. 4.25 *Data is divided into 512-bit blocks*

Step 4: Initialize chaining variables In this step, four variables (called as **chaining variables**) are initialized. They are called as A, B, C and D. Each of these is a 32-bit number. The initial hexadecimal values of these chaining variables are shown in Fig. 4.26.

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

Fig. 4.26 *Chaining variables*

Step 5: Process blocks After all the initializations, the real algorithm begins. It is quite complicated and we shall discuss it step-by-step to simplify it to the maximum extent possible.

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 (note the smaller case). Thus, we now have a = A, b = B, c = C and d = D. This is shown in Fig. 4.27.

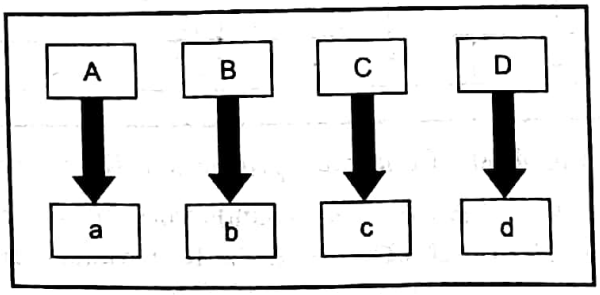

Fig. 4.27 *Copying chaining variables into temporary variables*

Actually, the algorithm considers the combination of a, b, c and d as a 128-bit single register (which we shall call as abcd). This register (abcd) is useful in the actual algorithm operation for holding intermediate as well as final results. This is shown in Fig. 4.28.

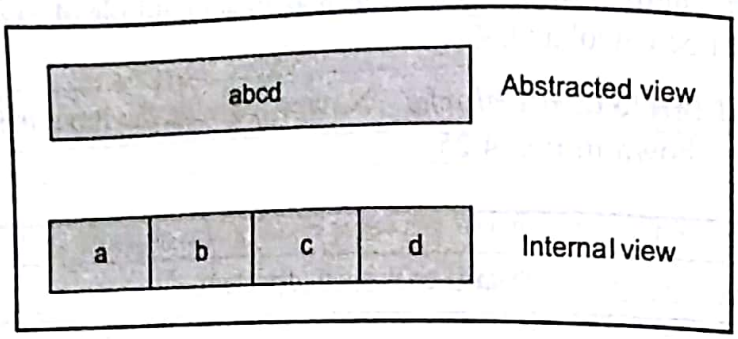

Fig. 4.28 *Abstracted view of the chaining variables*

Step 5.2: Divide the current 512-bit block into 16 sub-blocks. Thus, each sub-block contains 32 bits, as shown in Fig. 4.29.

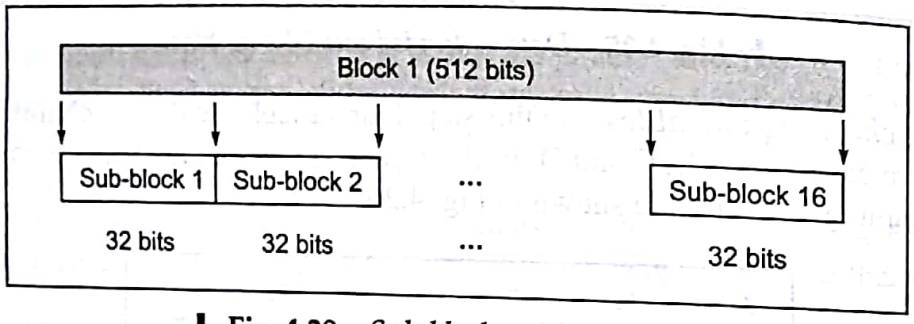

Fig. 4.29 *Sub-blocks within a block*

Step 5.3: Now, we have four *rounds*. In each round, we process all the 16 sub-blocks belonging to a block. The inputs to each round are: (a) all the 16 sub-blocks, (b) the variables a, b, c, d and (c) some constants, designated as t. This is shown in Fig. 4.30.

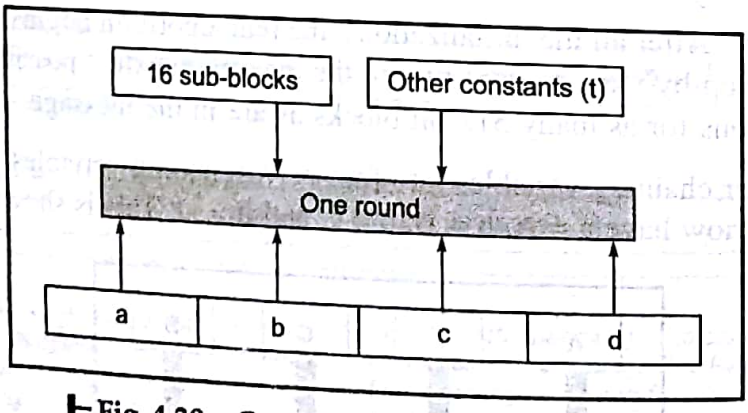

Fig. 4.30 *Conceptual process within a round*

What is done in these four rounds? All the four rounds vary in one major way: Step 1 of the four rounds has different processing. The other steps in all the four rounds are the same.

- In each round, we have 16 input sub-blocks, named M[0], M[1], ..., M[15] or in general, M[i], where i varies from 0 to 15. As we know, each sub-block consists of 32 bits.

• Also, t is an array of constants. It contains 64 elements, with each element consisting of 32 bits. We denote the elements of this array t as t[1], t[2], ... t[64] or in general as t[k], where k varies from 1 to 64. Since there are four rounds, we use 16 out of the 64 values of t in each round.

Let us summarize these iterations of all the four rounds. In each case, the output of the intermediate as well as the final iteration is copied into the register abcd. Note that we have 16 such iterations in each round.

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

This is shown in Fig. 4.31.

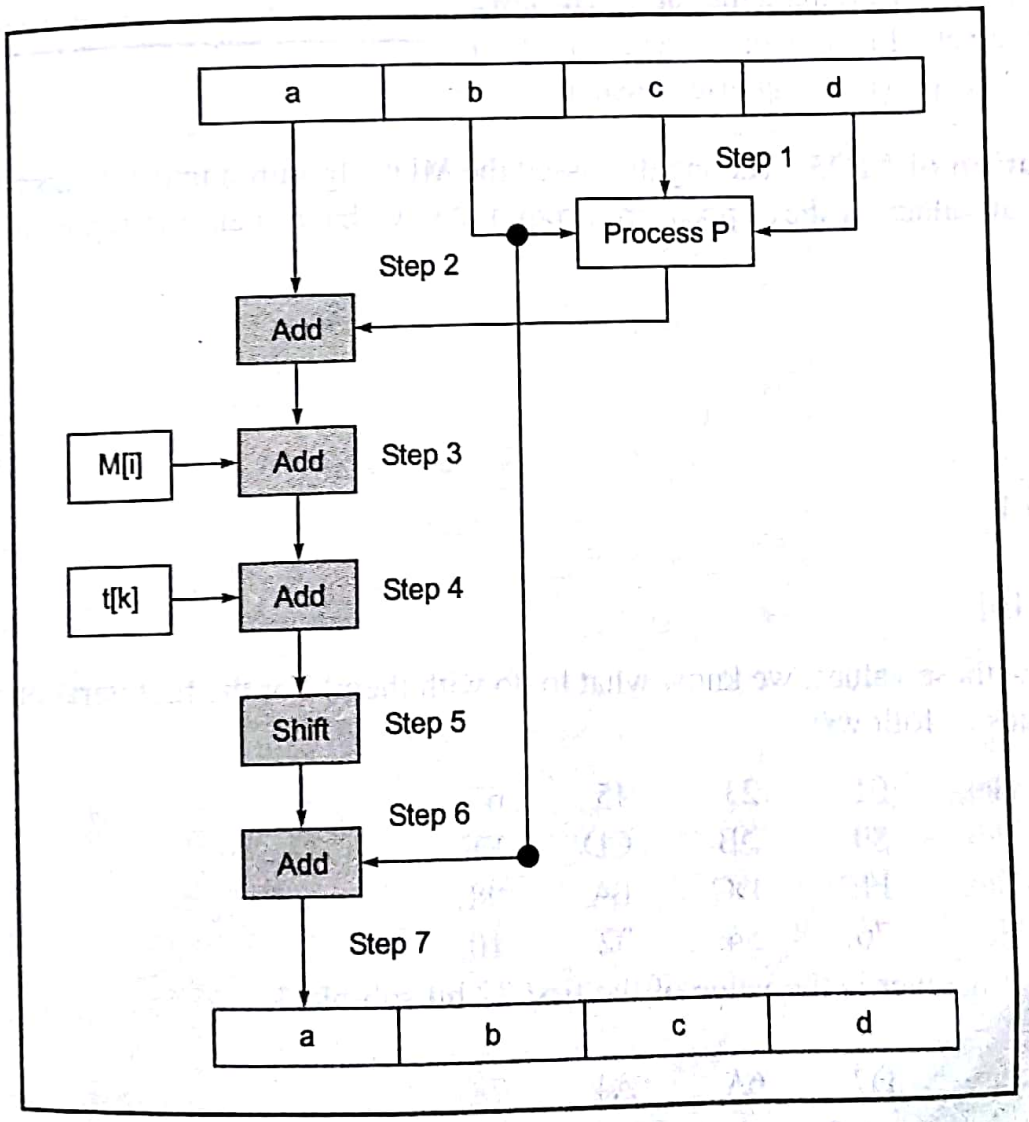

Fig. 4.31 *One MD5 operation*

We can mathematically express a single MD5 operation as follows:

$$a = b + ((a + \text{Process } P\ (b, c, d) + M[i] + T[k]) <<< s)$$

Where,

a, b, c, d	= Chaining variables, as described earlier
Process P	= A non-linear operation, as described subsequently
M[i]	= M[q × 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

Understanding the process P

As we can see, the most crucial aspect here is to understand the process P, as it is different in the four rounds. In simple terms, the process P is nothing but some basic Boolean operations on b, c and d. This is shown in Table 4.3.

Note that in the four rounds, only the process P differs. All the other steps remain the same. Thus, we can substitute the actual details of process P in each of the round and keep everything else constant.

Table 4.3 *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))

Sample Execution of MD5 Having discussed the MD5 algorithm in detail, now it is time to take a look at the actual values as they appear in a round. As we have seen, the inputs to any round are as follows:

- a
- b
- c
- d
- M[i = 0 to 15]
- s,
- t[k = 1 to 16]

Once we know these values, we know what to do with them! For the first iteration of the first round, we have the values as follows:

a	=	Hex	01	23	45	67
b	=	Hex	89	AB	CD	EF
c	=	Hex	FE	DC	BA	98
d	=	Hex	76	54	32	10
M[0]	=	Whatever is the value of the first 32 bit sub-block				
s	=	7				
t[1]	=	Hex	D7	6A	A4	78

For the second iteration, we move the positions a, b, c and d one position right. So, for the second iteration, we now have:

d = Output value of d of iteration 1
a = Output value of a of iteration 1
b = Output value of b of iteration 1
c = Output value of c of iteration 1
d = Whatever is the value of the second 32 bit sub-block
M[1] = 12
s = Hex E8 C7 B7 56
t[2] =

This process will continue for the remaining 14 iterations of round 1, as well as for all the 16 iterations of rounds 2, 3 and 4. In each case, before the iteration begins, we move a, b, c and d to one position right and use a different s and t[i] defined by MD5 for that step/iteration combination. The actual four rounds are tabulated in Table 4.4. The 64 possible values of t are shown after that.

Table 4.4 *(a) Round 1*

Iteration	*a*	*b*	*c*	*d*	*M*	*s*	*t*
1	a	b	c	d	M[0]	7	t[1]
2	d	a	b	c	M[1]	12	t[2]
3	c	d	a	b	M[2]	17	t[3]
4	b	c	d	a	M[3]	22	t[4]
5	a	b	c	d	M[4]	7	t[5]
6	d	a	b	c	M[5]	12	t[6]
7	c	d	a	b	M[6]	17	t[7]
8	b	c	d	a	M[7]	22	t[8]
9	a	b	c	d	M[8]	7	t[9]
10	d	a	b	c	M[9]	12	t[10]
11	c	d	a	b	M[10]	17	t[11]
12	b	c	d	a	M[11]	22	t[12]
13	a	b	c	d	M[12]	7	t[13]
14	d	a	b	c	M[13]	12	t[14]
15	c	d	a	b	M[14]	17	t[15]
16	b	c	d	a	M[15]	22	t[16]

Table 4.4 *(b) Round 2*

Iteration	*a*	*B*	*c*	*d*	*M*	*s*	*t*
1	a	b	c	d	M[1]	5	t[17]
2	d	a	b	c	M[6]	9	t[18]
3	c	d	a	b	M[11]	14	t[19]
4	b	c	d	a	M[0]	20	t[20]
5	a	b	c	d	M[5]	5	t[21]
6	d	a	b	c	M[10]	9	t[22]
7	c	d	a	b	M[15]	14	t[23]
8	b	c	d	a	M[4]	20	t[24]
9	a	b	c	d	M[9]	5	t[25]

Contd.

Table 4.4 (b) Contd.

10	d	a	b	c	M[14]	9	
11	c	d	a	b	M[3]	14	t[25]
12	b	c	d	a	M[8]	20	t[26]
13	a	b	c	d	M[13]	5	t[27]
14	d	a	b	c	M[2]	9	t[28]
15	c	d	a	b	M[7]	14	t[29]
16	b	c	d	a	M[12]	20	t[30]
							t[31]
							t[32]

Table 4.4 (c) Round 3

Iteration	a	b	c	d	M	s	t
1	a	b	c	d	M[5]	4	t[33]
2	d	a	b	c	M[8]	11	t[34]
3	c	d	a	b	M[11]	16	t[35]
4	b	c	d	a	M[14]	23	t[36]
5	a	b	c	d	M[1]	4	t[37]
6	d	a	b	c	M[4]	11	t[38]
7	c	d	a	b	M[7]	16	t[39]
8	b	c	d	a	M[10]	23	t[40]
9	a	b	c	d	M[13]	4	t[41]
10	d	a	b	c	M[0]	11	t[42]
11	c	d	a	b	M[3]	16	t[43]
12	b	c	d	a	M[6]	23	t[44]
13	a	b	c	d	M[9]	4	t[45]
14	d	a	b	c	M[12]	11	t[46]
15	c	d	a	b	M[15]	16	t[47]
16	b	C	d	a	M[2]	23	t[48]

Table 4.4 (d) Round 4

Iteration	a	B	c	d	M	s	t
1	a	b	c	d	M[0]	6	t[49]
2	d	a	b	c	M[7]	10	t[50]
3	c	d	a	b	M[14]	15	t[51]
4	b	c	d	a	M[5]	21	t[52]
5	a	b	c	d	M[12]	6	t[53]
6	d	a	b	c	M[3]	10	t[54]
7	c	d	a	b	M[10]	15	t[55]
8	b	c	d	a	M[1]	21	t[56]
9	a	b	c	d	M[8]	6	t[57]
10	d	a	b	c	M[15]	10	t[58]
11	c	d	a	b	M[6]	15	t[59]
12	b	c	d	a	M[13]	21	t[60]
13	a	b	c	d	M[4]	6	t[61]
14	d	a	b	c	M[11]	10	t[62]
15	c	d	a	b	M[2]	15	t[63]
16	b	c	d	a	M[9]	21	t[64]

The table t contains values (in hex) as shown in Table 4.5.

Table 4.5 *Values of the table t*

t[i]	Value	t[i]	Value	t[i]	Value	t[i]	Value
t[1]	D76AA478	t[17]	F61E2562	t[33]	FFFA3942	t[49]	F4292244
t[2]	E8C7B756	t[18]	C040B340	t[34]	8771F681	t[50]	432AFF97
t[3]	242070DB	t[19]	265E5A51	t[35]	699D6122	t[51]	AB9423A7
t[4]	C1BDCEEE	t[20]	E9B6C7AA	t[36]	FDE5380C	t[52]	FC93A039
t[5]	F57C0FAF	t[21]	D62F105D	t[37]	A4BEEA44	t[53]	655B59C3
t[6]	4787C62A	t[22]	02441453	t[38]	4BDECFA9	t[54]	8F0CCC92
t[7]	A8304613	t[23]	D8A1E681	t[39]	F6BB4B60	t[55]	FFEFF47D
t[8]	FD469501	t[24]	E7D3FBC8	t[40]	BEBFBC70	t[56]	85845DD1
t[9]	698098D8	t[25]	21E1CDE6	t[41]	289B7EC6	t[57]	6FA87E4F
t[10]	8B44F7AF	t[26]	C33707D6	t[42]	EAA127FA	t[58]	FE2CE6E0
t[11]	FFFF5BB1	t[27]	F4D50D87	t[43]	D4EF3085	t[59]	A3014314
t[12]	895CD7BE	t[28]	455A14ED	t[44]	04881D05	t[60]	4E0811A1
t[13]	6B901122	t[29]	A9E3E905	t[45]	D9D4D039	t[61]	F7537E82
t[14]	FD987193	t[30]	FCEFA3F8	t[46]	E6DB99E5	t[62]	BD3AF235
t[15]	A679438E	t[31]	676F02D9	t[47]	1FA27CF8	t[63]	2AD7D2BB
t[16]	49B40821	t[32]	8D2A4C8A	t[48]	C4AC5665	t[64]	EB86D391

MD5 versus MD4 Let us list down the key differences between MD5 and its predecessor, MD4, as shown in Table 4.6.

Table 4.6 *Differences between MD5 and MD4*

Point of discussion	*MD4*	*MD5*
Number of rounds	3	4
Use of additive constant t	Not different in all the iterations	Different in all the iterations
Process P in round 2	((b AND c) OR (b AND d) OR (c AND d))	(b AND d) OR (c AND (NOT d)) *– This is more random*

Additionally, the order of accessing the sub-blocks in rounds 2 and 3 is changed, to introduce more randomness.

The Strength of MD5 We can see how complex MD5 can get! The attempt of Rivest was to add as much of complexity and randomness as possible to the MD5 algorithm, so that no two message digests produced by MD5 on any two different messages are equal. MD5 has a property that every bit of the message digest is some function of every bit in the input. The possibility that two messages produce the same message digest using MD5 is in the order of 2^{64} operations. Given a message digest, working backwards to find the original message can lead up to 2^{128} operations.

The following attacks have been launched against MD5.

1. Tom Berson could find two messages that produce the same message digest for each of the four individual rounds. However, he could not come up with two messages that produce the same message digest for all the four rounds taken together.

2. den Boer and Bosselaers showed that the 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 as **pseudocollision**. However, they could not extend this to a full MD5 consisting of four rounds, each containing 16 steps.
3. Dobbertin provided the most serious attack on MD5. Using his attack, the operation of MD5 on two different 512-bit blocks produces the same 128-bit output. However, this has not been generalized to a full message block.

The general recommendation now is not to trust MD5 (although it is not practically broken into, as yet). Consequently, the quest for better message algorithms has led to the possible alternative, called as **SHA-1**. We will study it now.

4.6.4 Secure Hash Algorithm (SHA)

Introduction The National Institute of Standards and Technology (NIST) along with NSA developed the **Secure Hash Algorithm (SHA)**. In 1993, SHA was published as a Federal Information Processing Standard (FIPS PUB 180). It was revised to FIPS PUB 180-1 in 1995 and the name was changed to SHA-1. As we shall study, SHA is a modified version of MD5 and its design closely resembles MD5.

SHA works with any input message that is less than 2^{64} bits in length. The output of SHA is a message digest, which is 160 bits in length (32 bits more than the message digest produced by MD5). The word *Secure* in SHA was decided based on two features. SHA is designed to be computationally infeasible to:

(a) Obtain the original message, given its message digest and
(b) Find two messages producing the same message digest

How SHA Works? As we have mentioned before, SHA is closely modeled after MD5. Therefore, we shall not discuss in detail those features of SHA, which are similar to MD5. Instead, we shall simply mention them and point out the differences. The reader can go back to the appropriate descriptions of MD5 to find out more details.

Step 1: Padding Like MD5, the first step in SHA is to add padding to the end of the original message in such a way that the length of the message is 64 bits short of a multiple of 512. Like MD5, the padding is always added, even if the message is already 64 bits short of a multiple of 512.

Step 2: Append length The length of the message excluding the length of the padding is now calculated and appended to the end of the padding as a 64-bit block.

Step 3: Divide the input into 512-bit blocks The input message is now divided into blocks, each of length 512 bits. These blocks become the input to the message digest processing logic.

Step 4: Initialize chaining variables Now, five *chaining variables* A through E are initialized. Remember that we had four chaining variables, each of 32 bits in MD5 (which made the total length of the variables $4 \times 32 = 128$ bits). Recall that we stored the intermediate as well as the final results into the combined register made up of these four chaining variables, i.e. abcd. Since in the case of SHA, we want to produce a message digest of length 160 bits, we need to have five chaining variables here ($5 \times 32 = 160$ bits). In SHA, the variables A through D have the same values as they had in MD5. Additionally, E is initialized to Hex C3 D2 E1 F0.

Step 5: Process blocks Now the actual algorithm begins. Here also, the steps are quite similar to those in MD5.

Step 5.1: Copy the chaining variables A-E into variables a-e. The combination of a-e, called as abcde will be considered as a single register for storing the temporary intermediate as well as the final results.

Step 5.2: Now, divide the current 512-bit block into 16 sub-blocks, each consisting of 32 bits.

Step 5.3: SHA has four rounds, each round consisting of 20 steps. Each round takes the current 512-bit block, the register abcde and a constant K[t] (where t = 0 to 79) as the three inputs. It then updates the contents of the register abcde using the SHA algorithm steps. Also notable is the fact that we had 64 constants defined as t in MD5. Here, we have only four constants defined for K[t], one used in each of the four rounds. The values of K[t] are as shown in Table 4.7.

Table 4.7 *Values of K[t]*

Round	*Value of t between*	*K[t] in hexadecimal*	*K[t] in decimal (Only integer portion of the value shown)*
1	1 and 19	5A 92 79 99	$2^{30} \times \sqrt{2}$
2	20 and 39	6E D9 EB A1	$2^{30} \times \sqrt{3}$
3	40 and 59	9F 1B BC DC	$2^{30} \times \sqrt{5}$
4	60 and 79	CA 62 C1 D6	$2^{30} \times \sqrt{10}$

Step 5.4: SHA consists of four rounds, each round containing 20 iterations. This makes it a total of 80 iterations. The logical operation of a single SHA iteration looks as shown in Fig. 4.32.

Mathematically, an iteration consists of the following operations:

$$abcde = (e + \text{Process P} + s^5 (a) + W[t] + K[t]), a, s^{30} (b), c, d$$

Where,

abcde = The register made up of the five variables a, b, c, d and e
Process P = The logical operation, which we shall study later
s^t = Circular-left shift of the 32-bit sub-block by t bits
W[t] = A 32-bit derived from the current 32-bit sub block, as we shall study later
K[t] = One of the five additive constants, as defined earlier

We will notice that this is very similar to MD5, with a few variations (which are induced to try and make SHA more complicated in comparison with MD5). We have to now see what are the meanings of the process P and W[t] in the above equation.

Table 4.8 illustrates the process P.

Table 4.8 *Process P in each SHA-1 round*

Round	*Process P*
1	(b AND c) OR ((NOT b) AND (d))
2	B XOR c XOR d
3	(b AND c) OR (b and D) OR (c AND d)
4	B XOR c XOR d

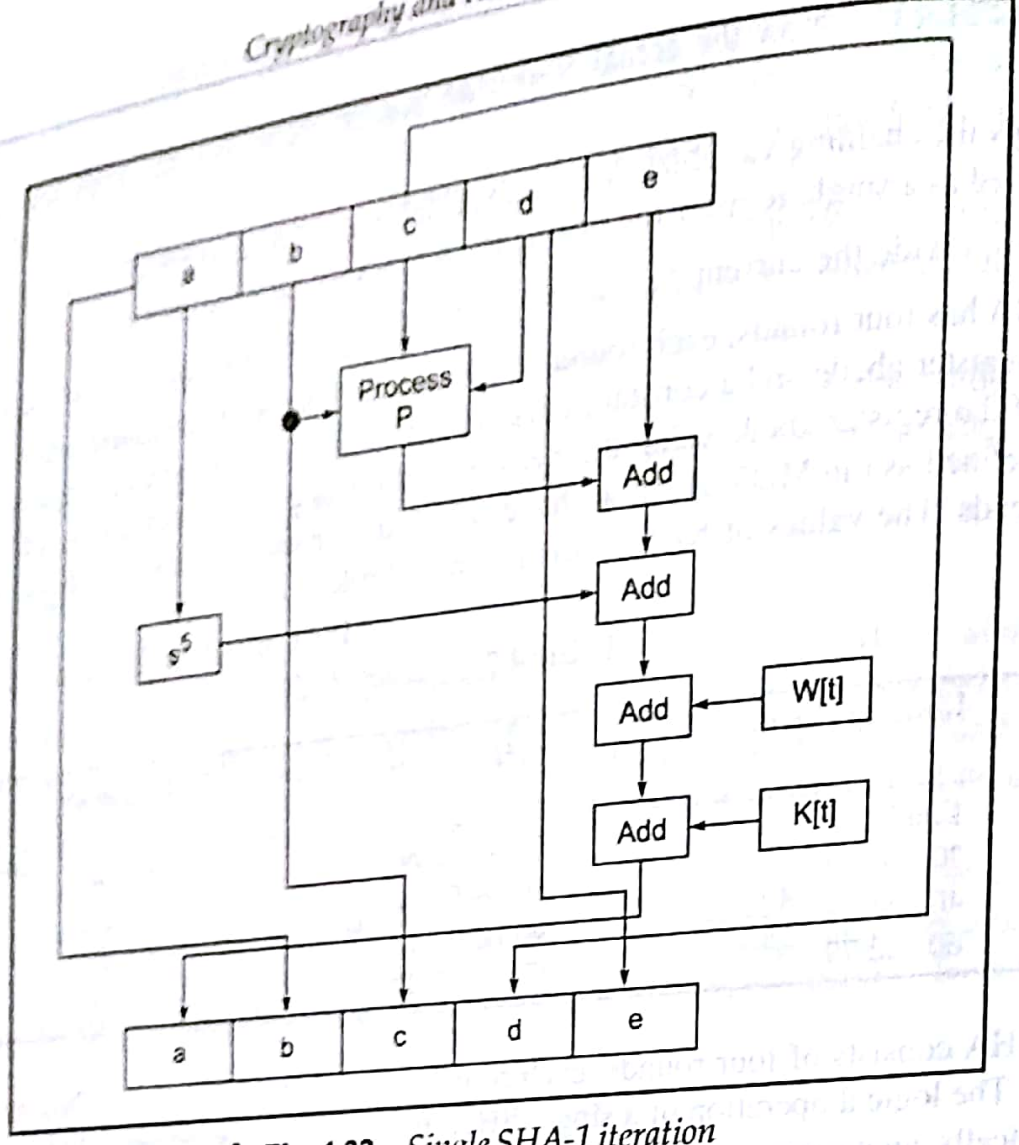

Fig. 4.32 *Single SHA-1 iteration*

The values of W[t] are calculated as follows:

For the first 16 words of W (i.e. t = 0 to 15), the contents of the input message sub-block M[t] become the contents of W[t] straightaway. That is, the first 16 blocks of the input message M are copied to W.

The remaining 64 values of W are derived using the equation:

$$W[t] = s^1 (W[t-16] \text{ XOR } W[t-14] \text{ XOR } W[t-8] \text{ XOR } W[t-3])$$

As before, s^1 indicates a circular-left shift (i.e. rotation) by 1 bit position.

Thus, we can summarize the values of W as shown in Table 4.9.

Table 4.9 *Values of W in SHA-1*

For t = 0 to 15	*Value of W[t]*
W[t] =	Same as M[t]
W[t] =	s^1 (W [t – 16] XOR W[t – 14] XOR W[t – 8] XOR W[t – 3])

Comparison of MD5 and SHA-1 As we know, the basis for both MD5 and SHA is the MD4 algorithm. Therefore, it is quite reasonable to compare MD5 and SHA and see what is the difference between the two, as depicted in Table 4.10.

Table 4.10 *Comparison of MD5 and SHA-1*

Point of discussion	*MD5*	*SHA-1*
Message digest length in bits	128	160
Attack to try and find the original message given a message digest	Requires 2^{128} operations to break in	Requires 2^{160} operations to break in, therefore more secure
Attack to try and find two messages producing the same message digest	Requires 2^{64} operations to break in	Requires 2^{80} operations to break in
Successful attacks so far	There have been reported attempts to some extent (as we discussed earlier)	No such claims so far
Speed	Faster (64 iterations and 128-bit buffer)	Slower (80 iterations and 160-bit buffer)
Software implementation	Simple, does not need any large programs or complex tables	Simple, does not need any large programs or complex tables

Security of SHA-1 In 2005, a possible vulnerability was discovered in SHA-1. Just before that, NIST had announced its intentions to seek more secure versions of SHA by 2010. Hence, various options are being discussed as follows: In 2002, NIST had come up with a new version of SHA in standard document FIPS 1802, called as SHA-256, SHA-384 and SHA-512; with the number after the word SHA indicating the length of the message digest in bits. Table 4.11 summarizes the various SHA versions.

Table 4.11 *Parameters for the Various Versions of SHA*

Parameter	*SHA-1*	*SHA-256*	*SHA-384*	*SHA-512*
Message digest size (in bits)	160	256	384	512
Message size (in bits)	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size (in bits)	512	512	1024	1024
Word size (in bits)	32	32	64	64
Steps in algorithm	80	64	80	80

As a consequence, we need to understand how a *more secure SHA* algorithm works. Therefore, we now discuss SHA-512. Naturally, SHA-512 works similar to SHA-1 with some additions/changes.

4.6.5 SHA-512

The SHA-512 algorithm takes a message of length 2^{128} bits and produces a message digest of size 512 bits. The input is divided into blocks of size 1024 bits each.

SHA-512 is closely modeled after SHA-1, which itself is modeled on MD5. Therefore, we shall not discuss in detail those features of SHA-512, which are similar to these two algorithms. Instead, we shall simply mention them and point out the differences. The reader can go back to the appropriate descriptions of MD5 to find out more details.

Step 1: Padding Like MD5 and SHA-1, the first step in SHA is to add padding to the end of the original message in such a way that the length of the message is 128 bits short of a multiple of 1024. Like MD5 and SHA-1, the padding is always added, even if the message is already 128 bits short of a multiple of 1024.

Step 2: Append length The length of the message excluding the length of the padding is now calculated and appended to the end of the padding as a 128-bit block. Hence, the length of the message is exactly a multiple of 1024 bits.

Step 3: Divide the input into 1024-bit blocks The input message is now divided into blocks, each of length 1024 bits. These blocks become the input to the message digest processing logic.

Step 4: Initialize chaining variables Now, eight *chaining variables* a through h are initialized. Remember that we had (a) Four chaining variables, each of 32 bits in MD5 (which made the total length of the variables 4 x 32 = 128 bits) and (b) Five chaining variables each of 32 bits (which made the total length of the variables 5 x 32 = 160 bits) in SHA-1. Recall that we stored the intermediate as well as the final results into the combined register made up of these chaining variables, i.e. abcd in MD5 and abcde in SHA-1. Since in the case of SHA-256, we want to produce a message digest of length 512 bits, we need to have eight chaining variables, each containing 64 bits here (8x 64 = 512 bits). In SHA-512, these eight variables have values as shown in Table 4.12.

Table 4.12 *SHA-512 Chaining Variables*

A = 6A09E667F3BCC908	B = BB67AE8584CAA73B
C = 3C6EF372FE94F82B	D = A54FF53A5F1D36F1
E = 510E527FADE682D1	F = 9B05688C2B3E6C1F
G = 1F83D9ABFB41BD6B	H = 5BE0CD19137E2179

Step 5: Process blocks Now the actual algorithm begins. Here also, the steps are quite similar to those in MD5 and SHA-1.

Step 5.1: Copy the chaining variables A-H into variables a-h. The combination of a-h, called as abcdefgh will be considered as a single register for storing the temporary intermediate as well as the final results.

Step 5.2: Now, divide the current 1024-bit block into 16 sub-blocks, each consisting of 64 bits.

Step 5.3: SHA-512 has 80 rounds, Each round takes the current 1024-bit block, the register abcdefgh and a constant K[t] (where t = 0 to 79) as the three inputs. It then updates the contents of the register abcdefgh using the SHA-512 algorithm steps. The operation of a single round is shown in Fig. 4.33.

Each round consists of the following operations:

$$Temp1 = h + Ch(e, f, g) + Sum(e_i \text{ for } i = 1 \text{ to } 512) + W_t + K_t$$

$$Temp2 = Sum(a_i \text{ for } i = 0 \text{ to } 512) + Maj(a, b, c)$$

$$a = Temp1 + Temp2$$

$$b = a$$

$$c = b$$

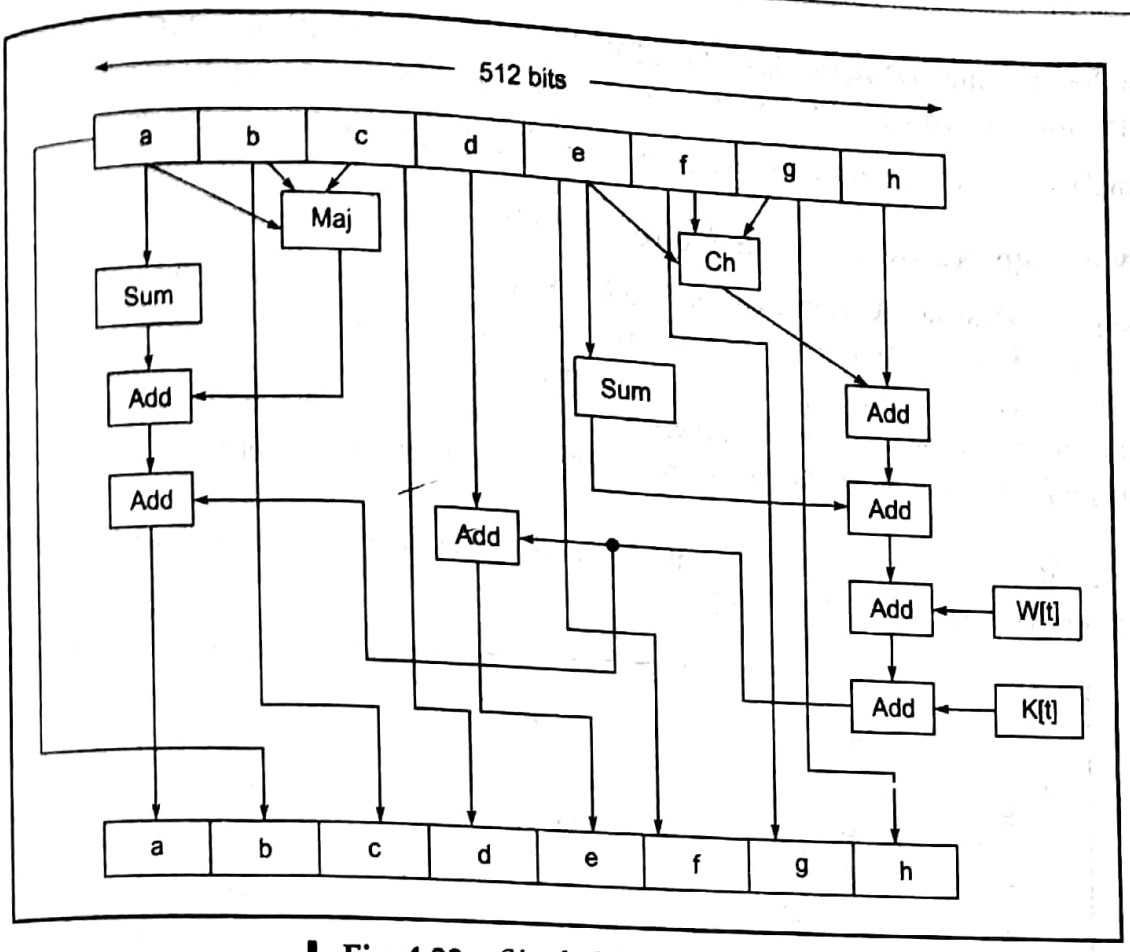

Fig. 4.33 *Single SHA-512 iteration*

$d = c$
$e = d + \text{Temp1}$
$f = e$
$g = f$
$h = g$

where:

t	= Round number
Ch (e, f, g)	= (e AND f) XOR (NOT e AND g)
Maj (a, b, c)	= (a AND b) XOR (a AND c) XOR (b AND c)
Sum (a_i)	= ROTR (a_i by 28 bits) XOR ROTR (a_i by 34 bits) XOR (ROTR a_i by 39 bits)
Sum (e_i)	= ROTR (e_i by 14 bits) XOR ROTR (e_i by 18 bits) XOR (ROTR e_i by 41 bits)
ROTR (x)	= Circular right shift, i.e. rotation, of the 64-bit array x by the specified number of bits
W_t	= 64-bit word derived from the current 512-bit input block
K_t	= 64-bit additive constant
+ (or Add)	= Addition mod 2^{64}

The 64-bit word values for W_t are derived from the 1024-bit message using certain mappings, which we shall not describe here. Instead, we will simply point out this:

1. For the first 16 rounds (0 to 15), the value of W_t is equal to the corresponding word in the message block.

2. The receiver (in this case, B) is assured that the message indeed came from the correct sender (in this case, A). Since only the sender and the receiver (A and B, respectively, in this case) know the secret key (in this case, K), no one else could have calculated the MAC (in this case, H1) sent by the sender (in this case, A).

Interestingly, although the calculation of the MAC seems to be quite similar to an encryption process, it is actually different in one important respect. As we know, in symmetric key cryptography, the cryptographic process must be reversible. That is, the encryption and decryption are the mirror images of each other. However, note that in the case of MAC, both the sender and the receiver are performing encryption process only. Thus, a MAC algorithm need not be reversible – it is sufficient to be a one-way function (encryption) only.

We have already discussed two main message digest algorithms, namely MD5 and SHA-1. Can we reuse these algorithms for calculating a MAC, in their original form? Unfortunately, we cannot reuse them, because they do not involve the usage of a secret key, which is the basis of MAC. Consequently, we must have a separate practical algorithm implementation for MAC. The solution is **HMAC**, a practical algorithm to implement MAC.

4.6.7 HMAC

Introduction HMAC stands for **Hash-based Message Authentication Code**. HMAC has been chosen as a mandatory security implementation for the Internet Protocol (IP) security and is also used in the Secure Socket Layer (SSL) protocol, widely used on the Internet.

The fundamental idea behind HMAC is to reuse the existing message digest algorithms, such as MD5 or SHA-1. Obviously, there is no point in reinventing the wheel. Therefore, what HMAC does it to work with any message digest algorithm. That is, it treats the message digest as a black box. Additionally, it uses the shared symmetric key to encrypt the message digest, which produces the output MAC. This is shown in Fig. 4.35.

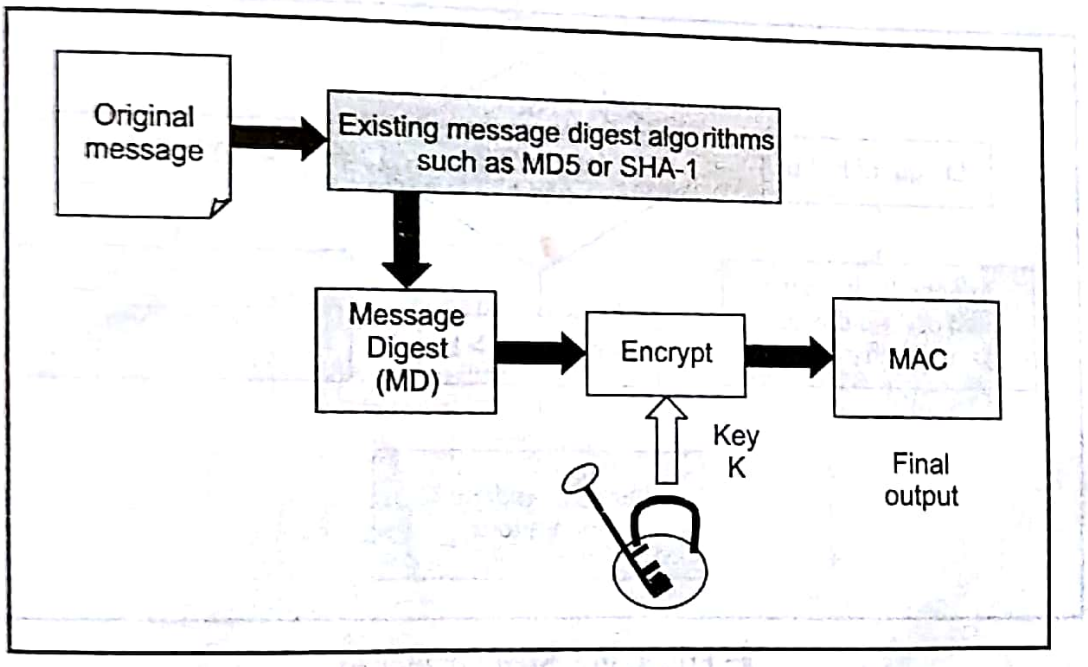

Fig. 4.35 *HMAC concept*

How HMAC Works? Let us now take a look at the internal working of HMAC. For this, let us start with the various variables that will be used in our HMAC discussion.

MD	=	The message digest/hash function used (e.g. MD5, SHA-1, etc.)
M	=	The input message whose MAC is to be calculated
L	=	The number of blocks in the message M
b	=	The number of bits in each block
K	=	The shared symmetric key to be used in HMAC
ipad	=	A string 00110110 repeated b/8 times
opad	=	A string 01011010 repeated b/8 times

Armed with these inputs, we shall use a step-by-step approach to understand the HMAC operation.

Step 1: Make the length of K equal to b The algorithm starts with three possibilities, depending on the length of the key K:

- ***Length of K < b***

In this case, we need to expand the key (K) to make the length of K equal to the number of bits in the original message block (i.e. b). For this, we add as many 0 bits as required to the left of K. For example, if the initial length of K = 170 bits and b = 512, then we add 342 bits, all with a value 0, to the left of K. We shall continue to call this modified key as K.

- ***Length of K = b***

In this case, we do not take any action and proceed to Step 2.

- ***Length of K > b***

In this case, we need to trim K to make the length of K equal to the number of bits in the original message block (i.e. b). For this, we pass K through the message digest algorithm (H) selected for this particular instance of HMAC, which will give us a key K, trimmed so that its length is equal to b. This is shown in Fig. 4.36.

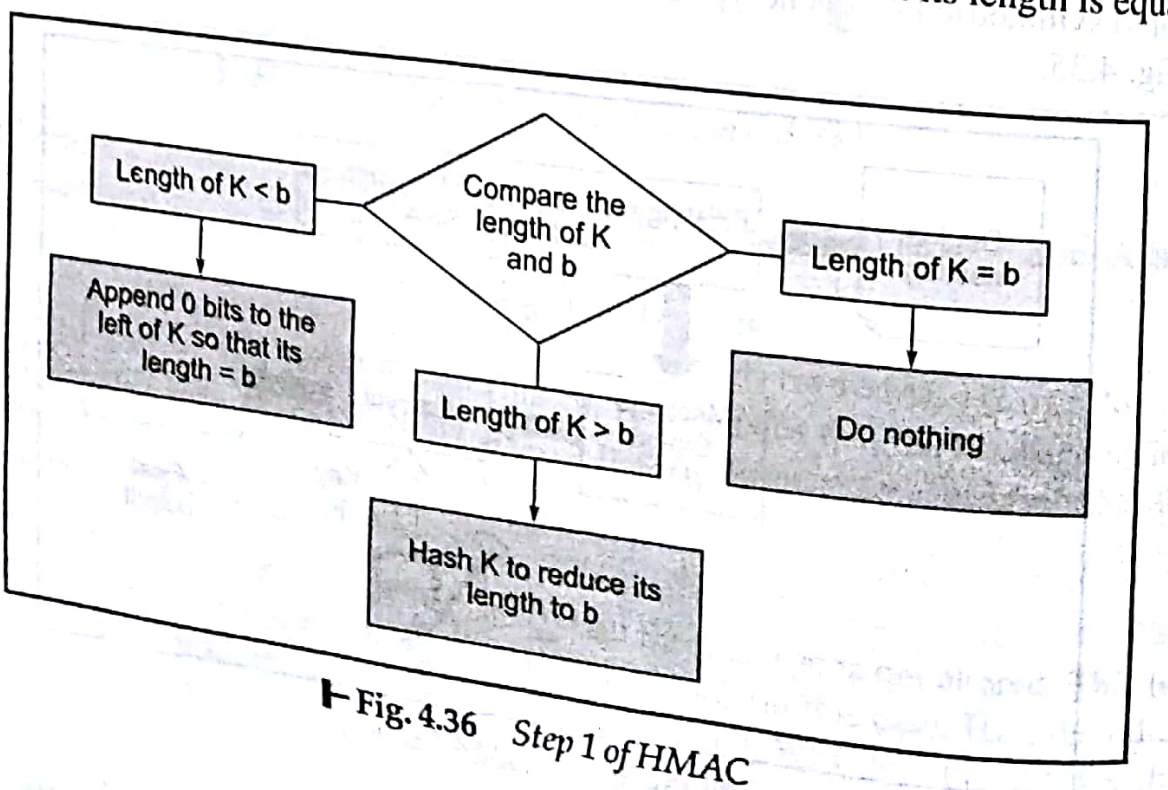

Fig. 4.36 *Step 1 of HMAC*

Step 2: XOR K with ipad to produce S1 We XOR K (the output of Step 1) and ipad to produce a variable called as S1. This is shown in Fig. 4.37.

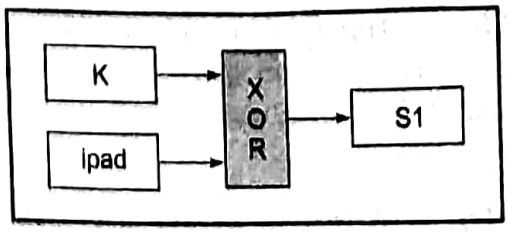

Fig. 4.37 *Step 2 of HMAC*

Step 3: Append M to S1 We now take the original message (M) and simply append it to the end of S1 (which was calculated in Step 2). This is shown in Fig. 4.38.

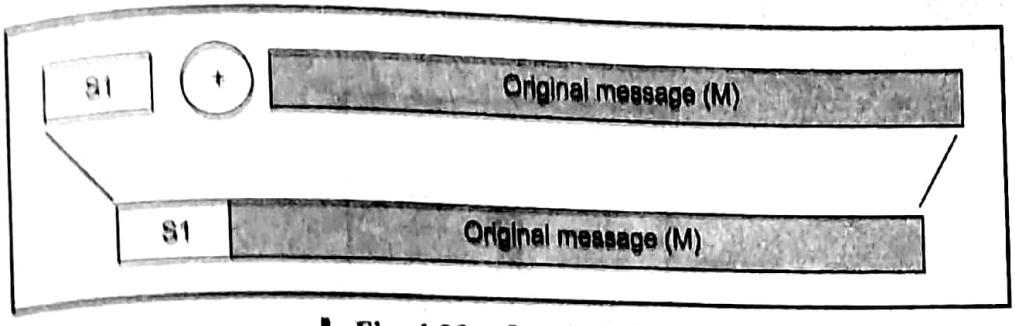

Fig. 4.38 *Step 3 of HMAC*

Step 4: Message digest algorithm Now, the selected message digest algorithm (e.g. MD5, SHA-1, etc) is applied to the output of Step 3 (i.e. to the combination of S1 and M). Let us call the output of this operation as H. This is shown in Fig. 4.39.

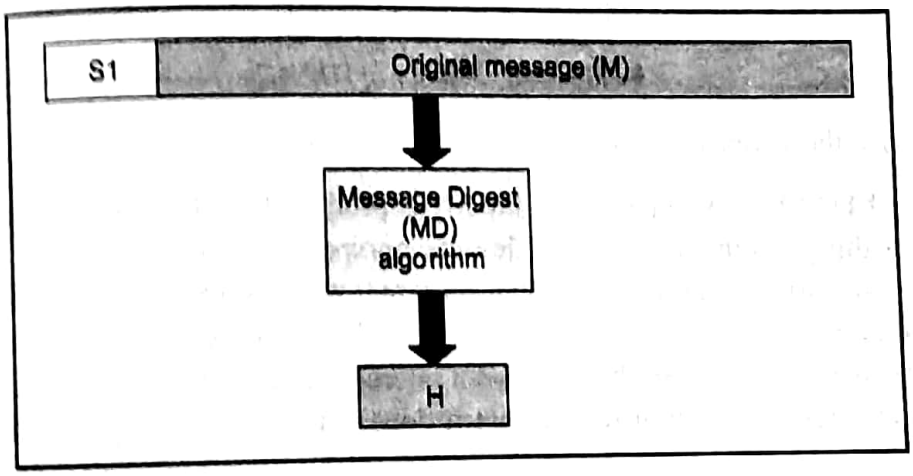

Fig. 4.39 *Step 4 of HMAC*

Step 5: XOR K with opad to produce S2 Now, we XOR K (the output of Step 1) with opad to produce a variable called as S2. This is shown in Fig. 4.40.

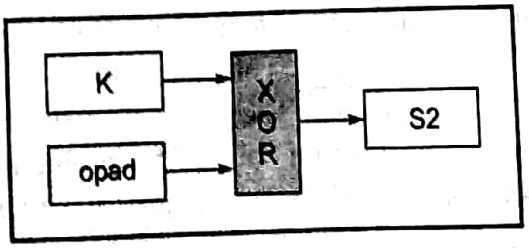

Fig. 4.40 *Step 5 of HMAC*

Step 6: Append H to S2 In this step, we take the message digest calculated in step 4 (i.e. H) and simply append it to the end of S2 (which was calculated in Step 5). This is shown in Fig. 4.41.

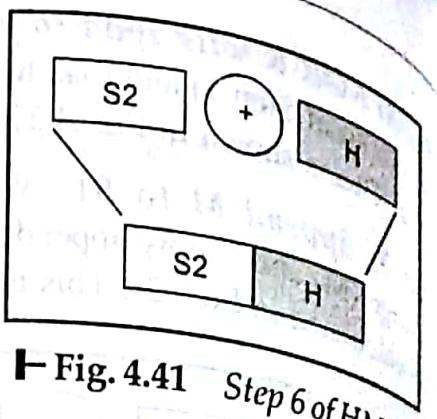

Fig. 4.41 *Step 6 of HMAC*

Step 7: Message digest algorithm Now, the selected message digest algorithm (e.g. MD5, SHA-1, etc) is applied to the output of Step 6 (i.e. to the concatenation of S2 and H). This is the final MAC that we want. This is shown in Fig. 4.42.

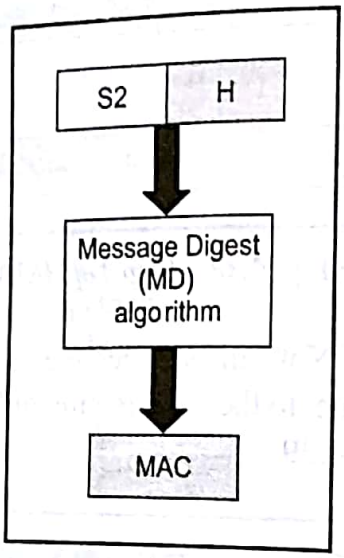

Fig. 4.42 *Step 7 of HMAC*

Let us summarize the seven steps of HMAC, as shown in Fig. 4.43.

Disadvantages of HMAC It appears that the MAC produced by the HMAC algorithm fulfills our requirements of a digital signature. From a logical perspective, firstly we calculate a fingerprint (message digest) of the original message and then encrypt it with a symmetric key, which is known only to the sender and the receiver. This gives sufficient confidence to the receiver that the message came from the correct sender only and also that it was not altered during the transit. However, if we observe the HMAC scheme carefully, we will realize that it does not solve all our problems. What are these problems?

1. We assume in HMAC that the sender and the receiver only know about the symmetric key. However, we have studied in great detail that the problem of symmetric key exchange is quite serious and cannot be solved easily. The same problem of key exchange is present in the case of HMAC.
2. Even if we assume that somehow the key exchange problem is resolved, HMAC cannot be used if the number of receivers is greater than one. This is because, to produce a MAC by using HMAC, we need to make use of a symmetric key. The symmetric key is supposed to be shared only by two parties: one sender and one receiver. Of course, this problem can be solved if multiple parties (one sender and all the receivers) share the same symmetric key. However, this resolution leads to a third problem.
3. The new problem is that how does a receiver know that the message was prepared and sent by the sender and not by one of the *other* receivers? After all, all the co-receivers also know the

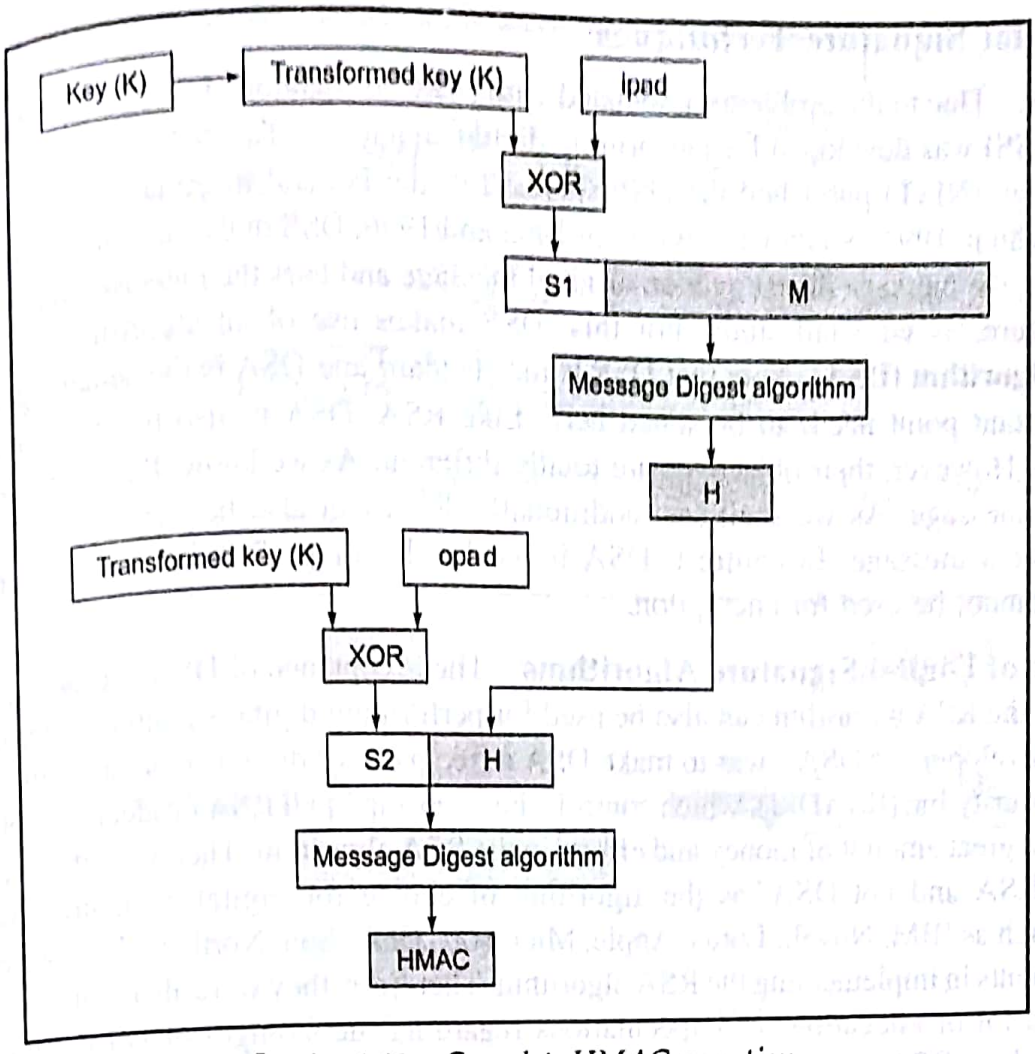

Fig. 4.43 *Complete HMAC operation*

symmetric key. Therefore, it is quite possible that one of the co-receivers might have created a false message on behalf of the alleged sender and using HMAC, the co-receiver might have prepared a MAC for the message and sent the message and the MAC as if it originated at the alleged sender! There is no way to prevent or detect this!

4. Even if we somehow solve the above problem, one major concern remains. Let us go back to our simple case of one sender and one receiver. Now, only two parties – the sender (say A, a bank customer) and the receiver (say B, a bank) share the symmetric key secret. Suppose that one fine day, B transfers all the balance standing in the account of A to a third person's account and closes A's bank account. A is shocked and files a suit against B. In the court of law, B argues that A had sent an electronic message in order to perform this transaction and produces that message as evidence. A claims that she never sent that message and that it is a forged message. Fortunately, the message produced by B as evidence also contained a MAC, which was produced on the original message. As we know, only encrypting the message digest of the original message with the symmetric key shared by A and B could have produced it – and here is where the trouble is! Even though we have a MAC, how in the world are we now going to prove that the MAC was produced by A or by B? After all, both know about the shared secret key! It is equally possible for either of them to have created the original message and the MAC!

As it turns out, even if we are able to somehow resolve the first three problems, we have no solution for the fourth problem. Therefore, we cannot trust HMAC to be used in digital signatures. Better schemes are required.