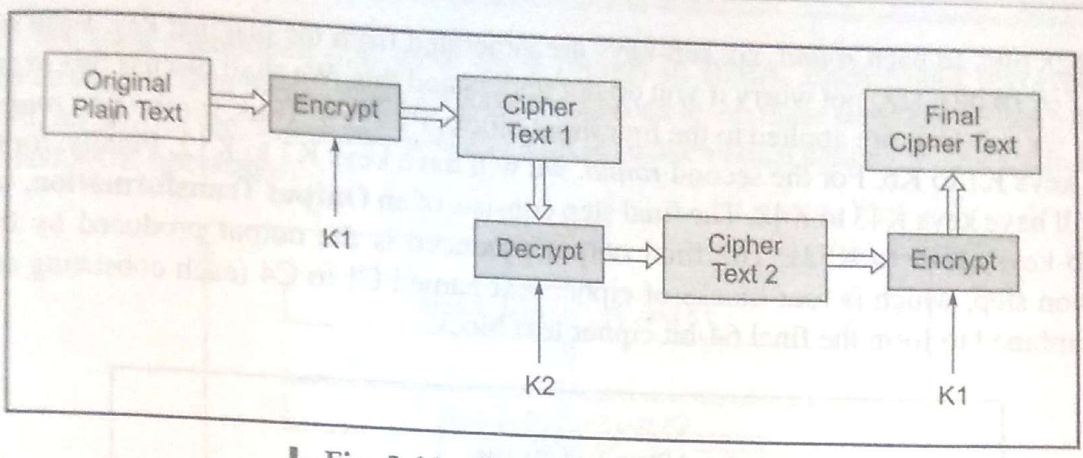

Fig. 3.44 *Triple DES with two keys*

3.5 International Data Encryption Algorithm (IDEA)

3.5.1 Background and History

The **International Data Encryption Algorithm (IDEA)** is perceived as one of the strongest cryptographic algorithms. It was launched in 1990 and underwent certain changes in names and capabilities as shown in Table 3.3.

Table 3.3 *Progress of IDEA*

Year	*Name*	*Description*
1990	Proposed Encryption Standard (PES)	Developed by Xuejia Lai and James Massey at the Swiss Federal Institute of Technology
1991	Improved Proposed Encryption Standard (IPES)	Improvements in the algorithm as a result of cryptanalysts finding some areas of weakness
1992	International Data Encryption Algorithm (IDEA)	No major changes, simply renamed

Although it is quite *strong*, IDEA is not as popular as DES for two primary reasons. Firstly, it is patented unlike DES and therefore, must be licensed before it can be used in commercial applications Secondly DES has a long history and track record as compared to IDEA. However one popular email privacy technology known as **Pretty Good Privacy (PGP)** is based on IDEA.

3.5.2 How IDEA Works

Basic Principles Technically, IDEA is a block cipher. Like DES, it also works on 64-bit plain text blocks. The key is longer and consists of 128 bits. IDEA is reversible like DES, that is, the same algorithm is used for encryption and decryption. Also, IDEA uses both *diffusion* and *confusion* for encryption.

The working of IDEA can be visualized at a broad level as shown in Fig. 3.45. The 64-bit input plain text block is divided into four portions of plain text (each of size 16 bits), say P1 to P4. Thus, P1 to P4 are the inputs to the first *round* of the algorithm. There are eight such *rounds*. As we mentioned, the key

consists of 128 bits. In each *round*, six sub-keys are generated from the original key. Each of the sub-keys consists of 16 bits. (Do not worry if you do not understand this. We shall discuss this in great detail soon). These six sub-keys are applied to the four input blocks P1 to P4. Thus, for the first *round*, we will have the six keys K1 to K6. For the second *round*, we will have keys K7 to K12. Finally, for the eighth round, we will have keys K43 to K48. The final step consists of an **Output Transformation**, which uses just four sub-keys (K49 to K52). The final output produced is the output produced by the Output Transformation step, which is four blocks of cipher text named C1 to C4 (each consisting of 16 bits). These are combined to form the final 64-bit cipher text block.

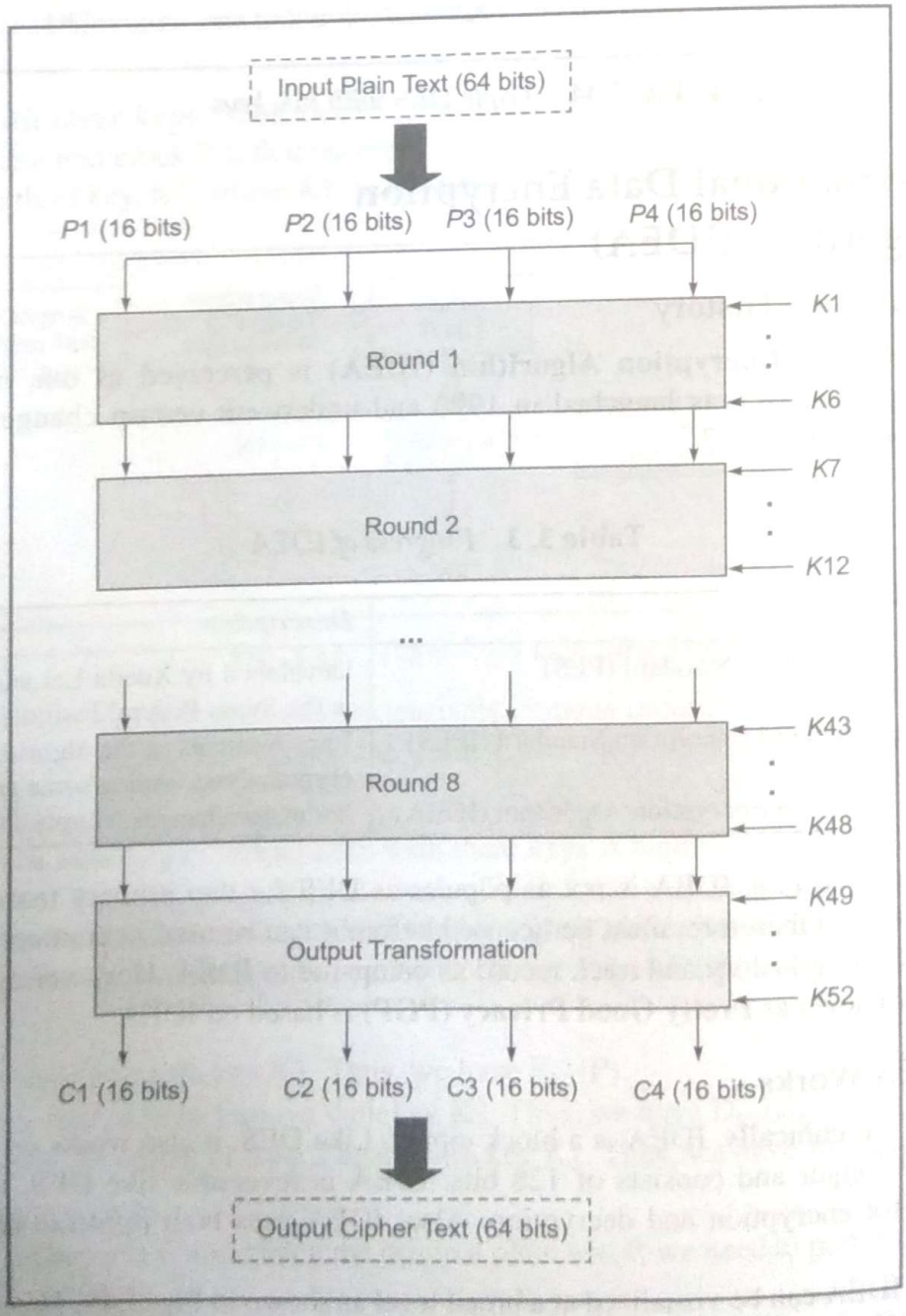

Fig. 3.45 *Broad level steps in IDEA*

Rounds We have mentioned that there are 8 rounds in IDEA. Each round involves a series of operations on the four data blocks using six keys. At a broad level, these steps can be described as shown in Fig. 3.46. As we can see, these steps perform a lot of mathematical actions. There are multiplications, additions and XOR operations.

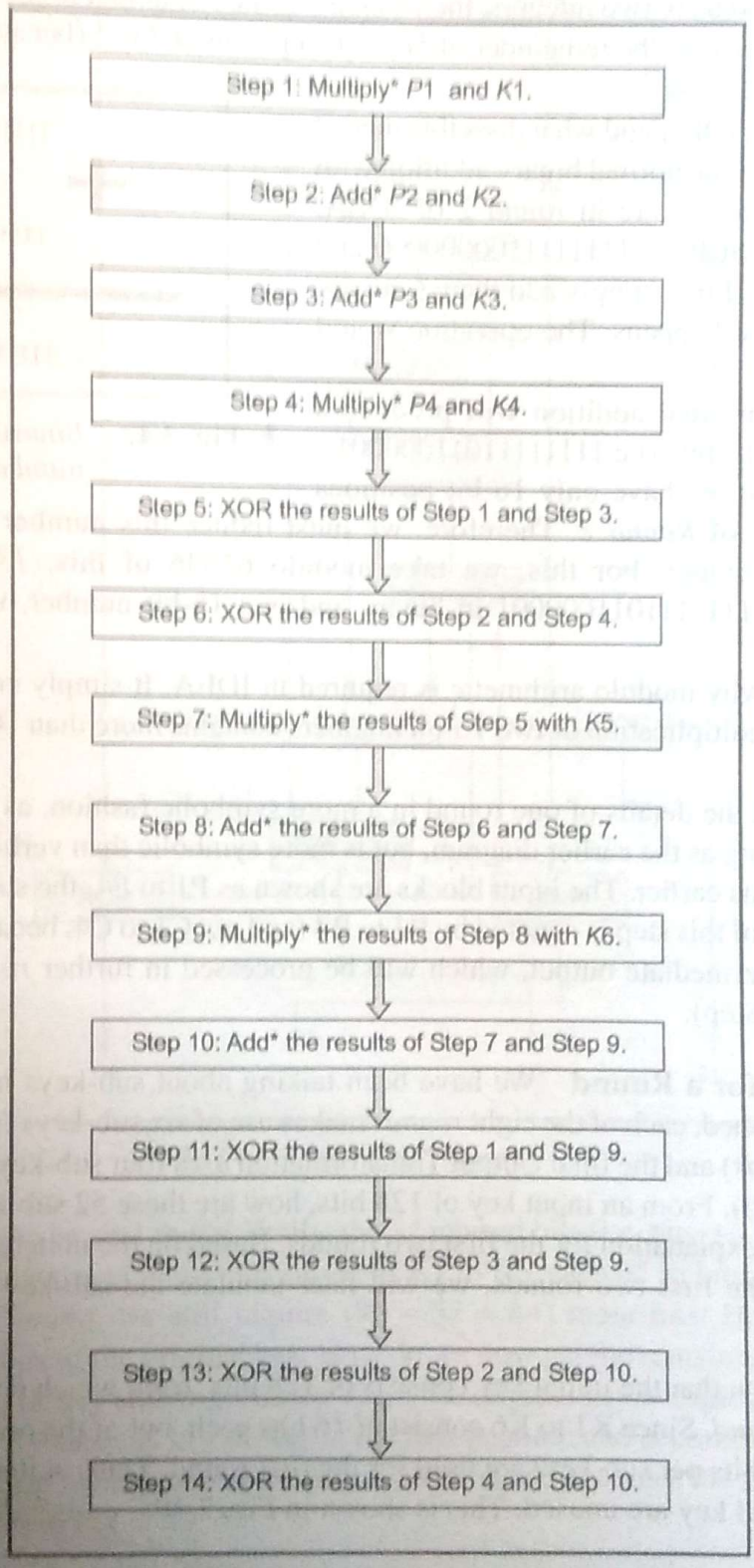

Fig. 3.46 *Details of one round in IDEA*

Note that we have put an asterisk in front of the words *Add* and *Multiply*, causing them to be shown as Add* and Multiply*, respectively. The reason behind this is that these are not mere additions and multiplications. Instead, these are addition modulo 2^{16} (i.e. addition modulo 65536) and multiplication modulo $2^{16} + 1$ (i.e. multiplication modulo 65537), respectively. [For those who are not conversant with modulo arithmetic, if a and b are two integers, then *a mod b* is the remainder of the division a/b. Thus, for example, *5 mod 2* is 1 (because the remainder of 5/2 is 1) and *5 mod 3* is 2 (because the remainder of 5/3 is 2)].

Why is this required in IDEA and what does this mean? Let us examine the case of the normal binary addition with an example. Suppose that we are in *round* 2 of IDEA. Further, let us assume that P2 = 1111111100000000 and K2 = 1111111111000001. First, simply add them (*and not add* them!*) and see what happens. The operation would look like as shown in Fig. 3.47.

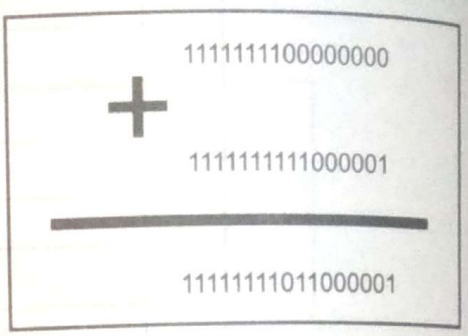

Fig. 3.47 *Binary addition of two 16-bit numbers*

As we can see, the normal addition will produce a number that consists of 17 bits (i.e.11111111011000001). However, remember that we have only 16 bit positions available for the output of *Round* 2. Therefore, we must reduce this number (which is 130753 in decimal) to a 16-bit number. For this, we take modulo 65536 of this. *130753 modulo 65536* yields 65217, which is 1111111011000001 in binary and is a 16-bit number, which fits well in our scheme.

This should explain why modulo arithmetic is required in IDEA. It simply ensures that even if the result of an addition or multiplication of two 16-bit numbers contains more than 16 bits, we bring it back to 16 bits.

Let us now re-look at the details of one round in a more symbolic fashion, as shown in Fig. 3.48. It conveys the same meaning as the earlier diagram, but is more symbolic than verbose in nature. We have depicted the same steps as earlier. The input blocks are shown as P1 to P4, the sub-keys are denoted by K1 to K6 and the output of this step is denoted by R1 to R4 (and *not* C1 to C4, because this is *not* the final cipher text – it is an intermediate output, which will be processed in further *rounds* as well as in the *Output Transformation* Step).

Sub-key Generation for a Round We have been talking about sub-keys in our discussion quite frequently. As we mentioned, each of the eight *rounds* makes use of six sub-keys (so, $8 \times 6 = 48$ sub-keys are required for the *rounds*) and the final Output Transformation uses four sub-keys (making a total of 48 + 4 = 52 sub-keys overall). From an input key of 128 bits, how are these 52 sub-keys generated? Let us understand this with the explanation for the first two rounds. Based on the understanding of the sub-key generation process for the first two rounds, we will later tabulate the sub-key generation for all the rounds.

- ***First round*** We know that the initial key consists of 128 bits, from which 6 sub-keys K1 to K6 are generated for the first *round*. Since K1 to K6 consist of 16 bits each, out of the original 128 bits, the first 96 bits (6 sub-keys × 16 bits per sub-key) are used for the first round. Thus, at the end of the first round, bits 97-128 of the original key are unused. This is shown in Fig. 3.49.

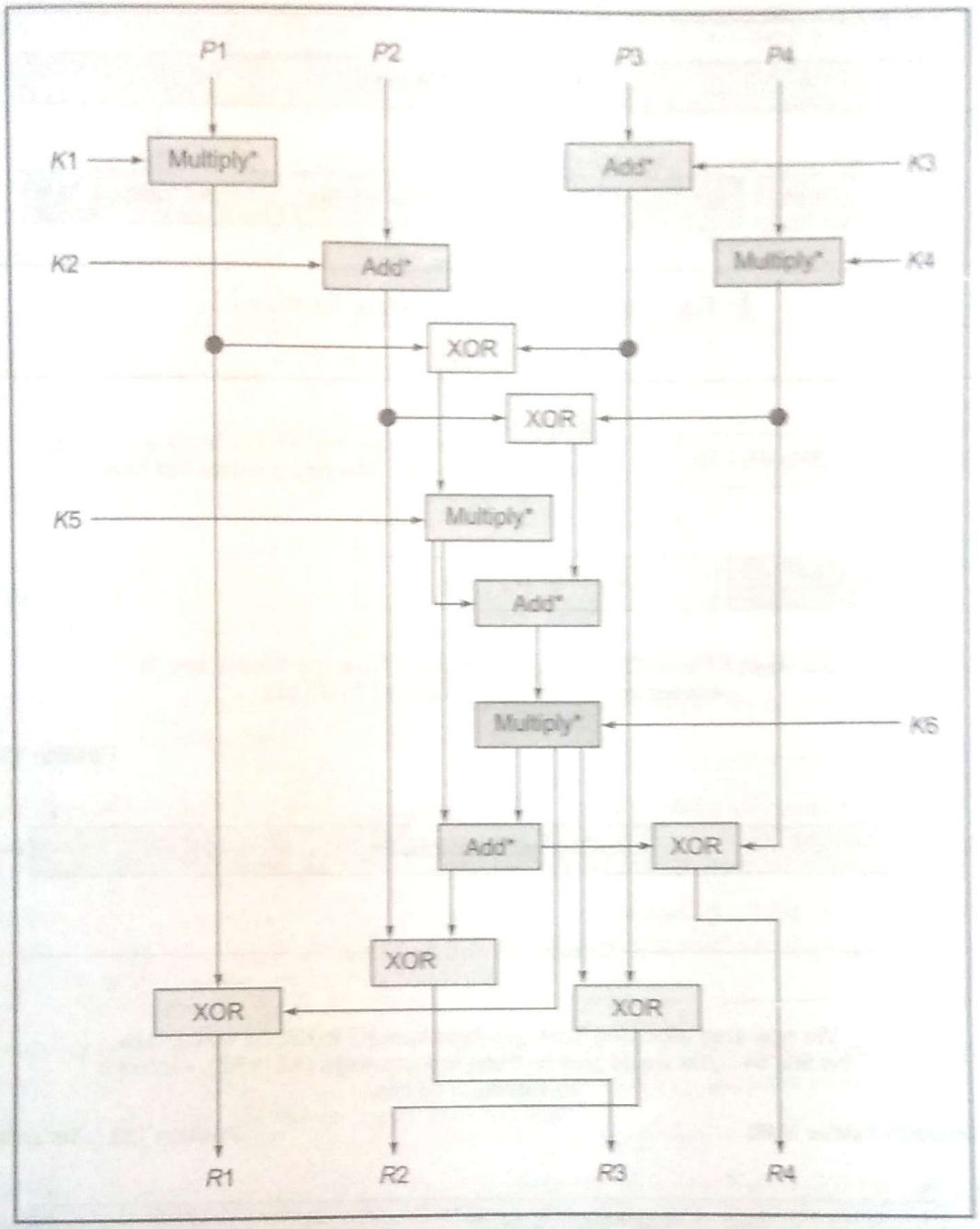

Fig. 3.48 *One round of IDEA*

• ***Second round*** In the second *round*, firstly, the 32 unused bits (i.e. bits 97-128) of the first *round* are used. We know that each round requires 6 sub-keys K1 to K6, each of 16 bits, making a total of 96 bits. Thus, for the second *round*, we still require (96 – 32 = 64) more bits. However, we have already exhausted all the 128 bits of the original key. How do we now get the remaining 64 bits? For this, IDEA employs the technique of **key shifting**. At this stage, the original key is *shifted left circularly* by 25 bits. That is, the 26th bit of the original key moves to the first position and becomes the first bit after the shift and the 25th bit of the original key moves to the last position and becomes the 128th bit after the shift. The whole process is shown in Fig. 3.50.

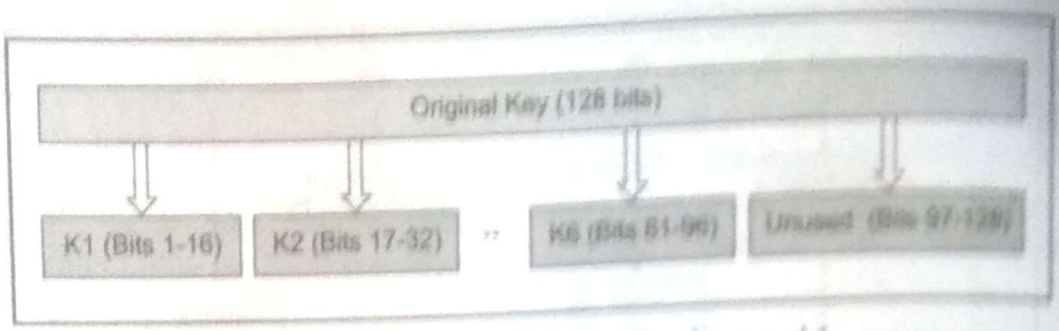

Fig. 3.49 *Sub-key generation for round 1*

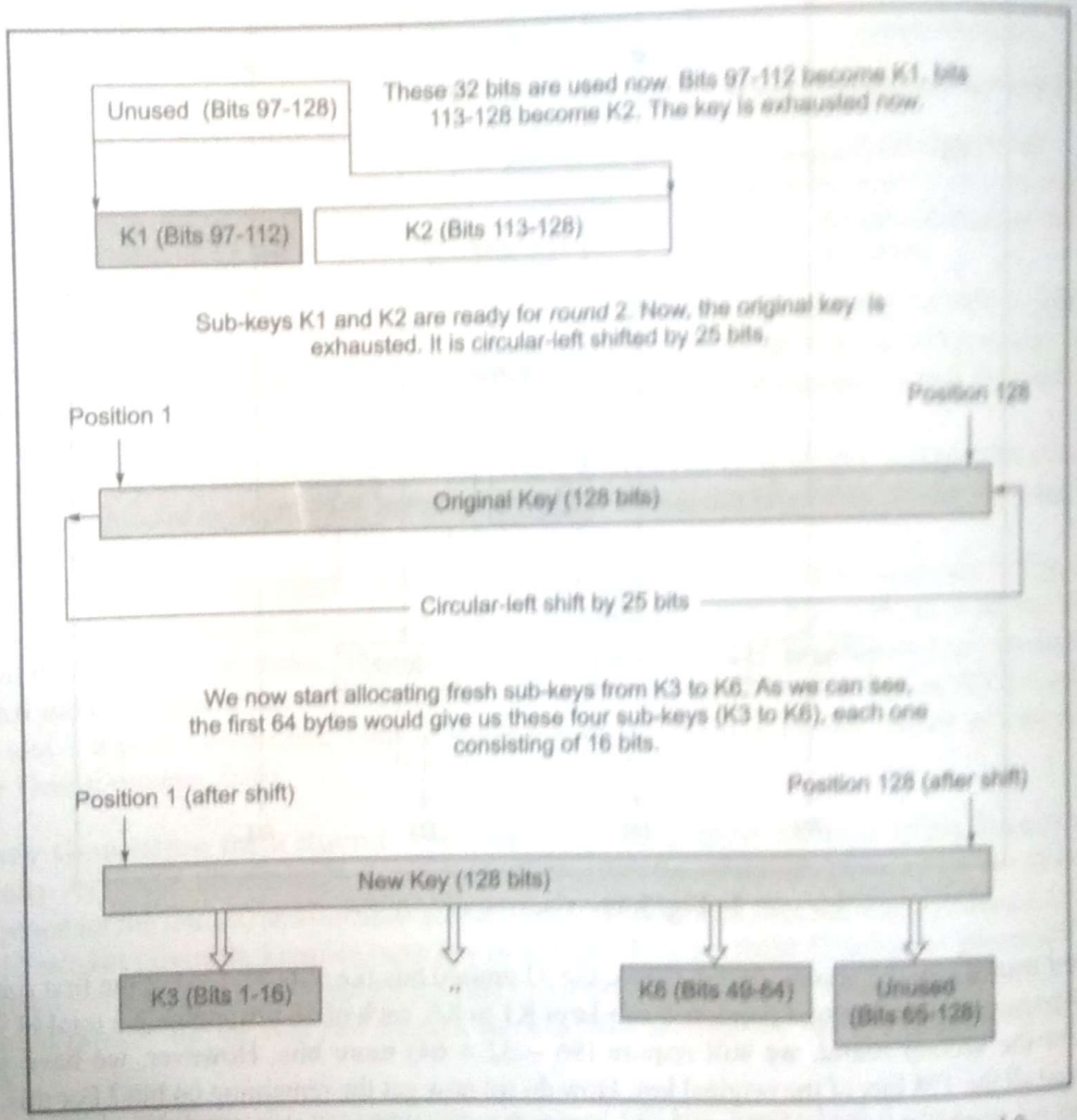

Fig. 3.50 *Circular-left key shift and its use in sub-key generation for round 2*

We can now see that the second round uses the bits 97-128 of the first round, and after a 25-bit shift bits 1-64. For the third round, we then use the remaining bits, i.e. bits 65-128 (i.e. a total of 64 bits). Again a shift on 25 bits occurs, and post this shifting, bits 1-32 are used in the third round, and so on.

Thus, is every round, 96 bits are obtained in the same manner, and this is how a 128-bit Key is divided into 96-bit subkeys.

Table 3.4 *Sub-key Generation Process for Each Round*

Round	*Details of the sub-key generation and use*
1	Bit positions 1-96 of the initial 128-bit key would be used. This would give us 6 sub-keys K1 to K6 for Round 1. Key bits 97 to 128 are available for the next round.
2	Key bits 97 to 128 make up sub-keys K1 and K2 for this round. A 25-bit shift on the original key happens, as explained. Post this shifting; the first 64 bits are used as sub-keys K3 to K6 for this round. This leaves bits 65 to 128 unused for the next round.
3	Unused key bits 65 to 128 are used as sub-keys K1 to K4 of this round. Upon key exhaustion, another 25-bit shift happens, and bits 1 to 32 of the shifted key are used as sub-keys K5 and K6. This leaves bits 33 to 128 unused for the next round.
4	Bits 33 to 128 are used for this round, which is perfectly adequate. No bits are unused at this stage. After this, the current key is again shifted.
5	This is similar to Round 1. Bit positions 1-96 of the current 128-bit key would be used. This would give us 6 sub-keys K1 to K6 for Round 1. Key bits 97 to 128 are available for the next round.
6	Key bits 97 to 128 make up sub-keys K1 and K2 for this round. A 25-bit shift on the original key happens, as explained. Post this shifting; the first 64 bits are used as sub-keys K3 to K6 for this round. This leaves bits 65 to 128 unused for the next round.
7	Unused key bits 65 to 128 are used as sub-keys K1 to K4 of this round. Upon key exhaustion, another 25-bit shift happens, and bits 1 to 32 of the shifted key are used as sub-keys K5 and K6. This leaves bits 33 to 128 unused for the next round.
8	Bits 33 to 128 are used for this round, which is perfectly adequate. No bits are unused at this stage. After this, the current key is again shifted for the *Output Transformation* round.

At the end of the last round, we have no unused bits. They are used in the **Output Transformation**.

Output Transformation The **Output Transformation** is a one-time operation. It takes place at the end of the 8^{th} *round*. The input to the Output Transformation is, of course, the output of the 8^{th} *round*. This is, as usual, a 64-bit value divided into four sub-blocks (say R1 to R4, each consisting of 16 bits). Also, four sub-keys are applied here and not six. We will describe the key generation process for the Output Transformation later. For now, we shall assume that four 16-bit sub-keys K1 to K4 are available to the Output Transformation. The process of the Output Transformation is described in Fig. 3.51.

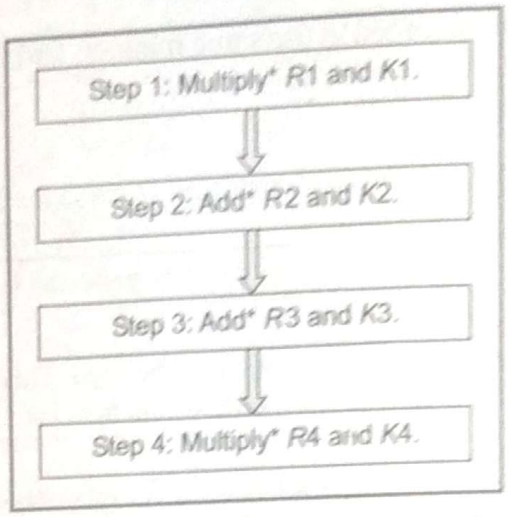

Fig. 3.51 *Details of the output transformation*

This process can be shown diagrammatically as illustrated by Fig. 3.52. The output of this process is the final 64-bit cipher text, which is the combination of the four cipher text sub-blocks C1 to C4.

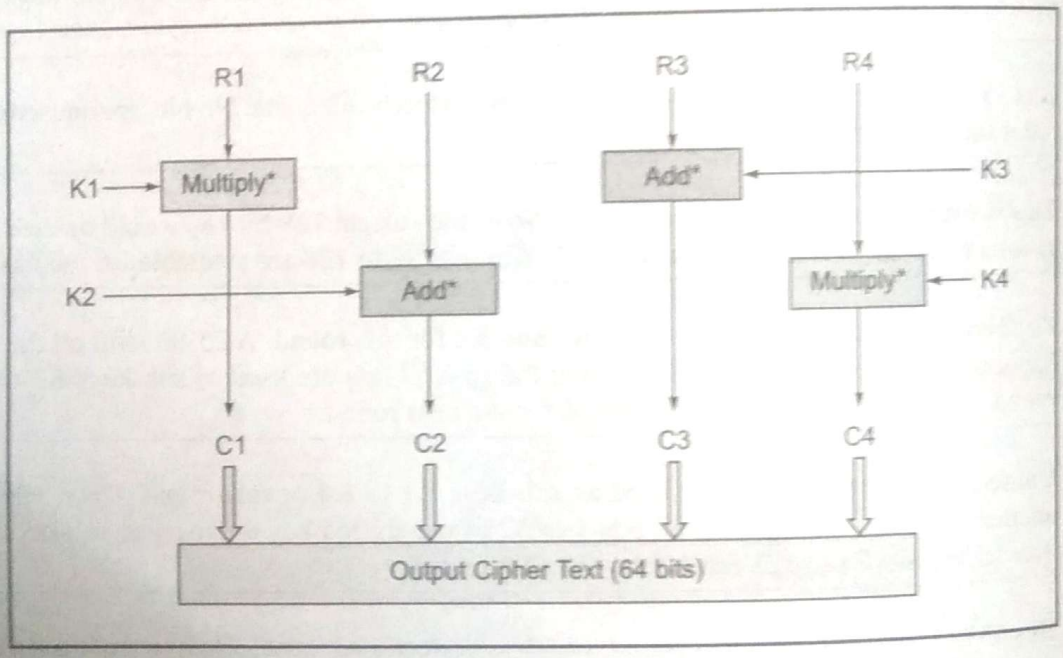

Fig. 3.52 *Output transformation process*

Sub-key Generation for the Output Transformation The process for the sub-key generation for the Output Transformation is exactly similar to the sub-key generation process for the eight *rounds*. Recall that at the end of the eighth and the final *round*, the key was exhausted. Hence, the key is again shifted by 25 bits. Post this shift operation, the first 64 bits of the key are taken, and are called as sub-keys K1 to K4 for the final output Transformation round.

IDEA Decryption The decryption process is exactly the same as the encryption process. There are some alterations in the generation and pattern of sub-keys. The decryption sub-keys are actually inverses of the encryption sub-keys. We shall not discuss this any further, as the basic concepts remain the same.

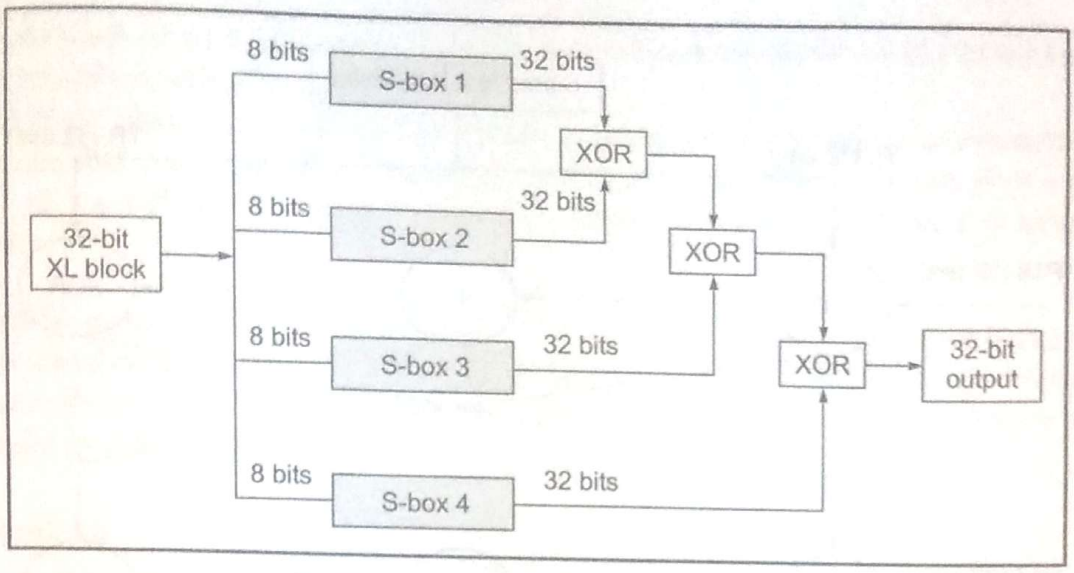

Fig. 3.72 *Function F in Blowfish*

The decryption process is shown in Fig. 3.73. As we can see, it is quite similar to the encryption process, with reversal of P-array values.

3.9 Advanced Encryption Standard (AES)

3.9.1 Introduction

In the 1990s, the US Government wanted to standardize a cryptographic algorithm, which was to be used universally by them. It was to be called as the **Advanced Encryption Standard (AES)**. Many proposals were submitted and after a lot of debate, an algorithm called as **Rijndael** was accepted. Rijndael was developed by Joan Daemen and Vincent Rijmen (both from Belgium). The name Rijndael was also based on their surnames (Rijmen and Daemen).

The need for coming up with a new algorithm was actually because of the perceived weakness in DES. The 56-bit keys of DES were no longer considered safe against attacks based on exhaustive key searches and the 64-bit blocks were also considered as weak. AES was to be based on 128-bit blocks, with 128-bit keys.

In June 1998, the Rijndael proposal was submitted to NIST as one of the candidates for AES. Out of the initial 15 candidates, only 5 were short listed in August 1999, which were as follows:

(i) Rijndael (From Joan Daemen and Vincent Rijmen; 86 votes)
(ii) Serpent (From Ross Anderson, Eli Biham and Lars Knudsen; 59 votes)
(iii) Twofish (From Bruce Schneier and others, 31 votes)
(iv) RC6 (From RSA Laboratories, 23 votes)
(v) MARS (From IBM, 13 votes)

In October 2000, Rijndael was announced as the final selection for AES. In November 2001, Rijndael became a US Government standard published as Federal Information Processing Standard 197 (FIPS 197).

According to its designers, the main features of AES are as follows.

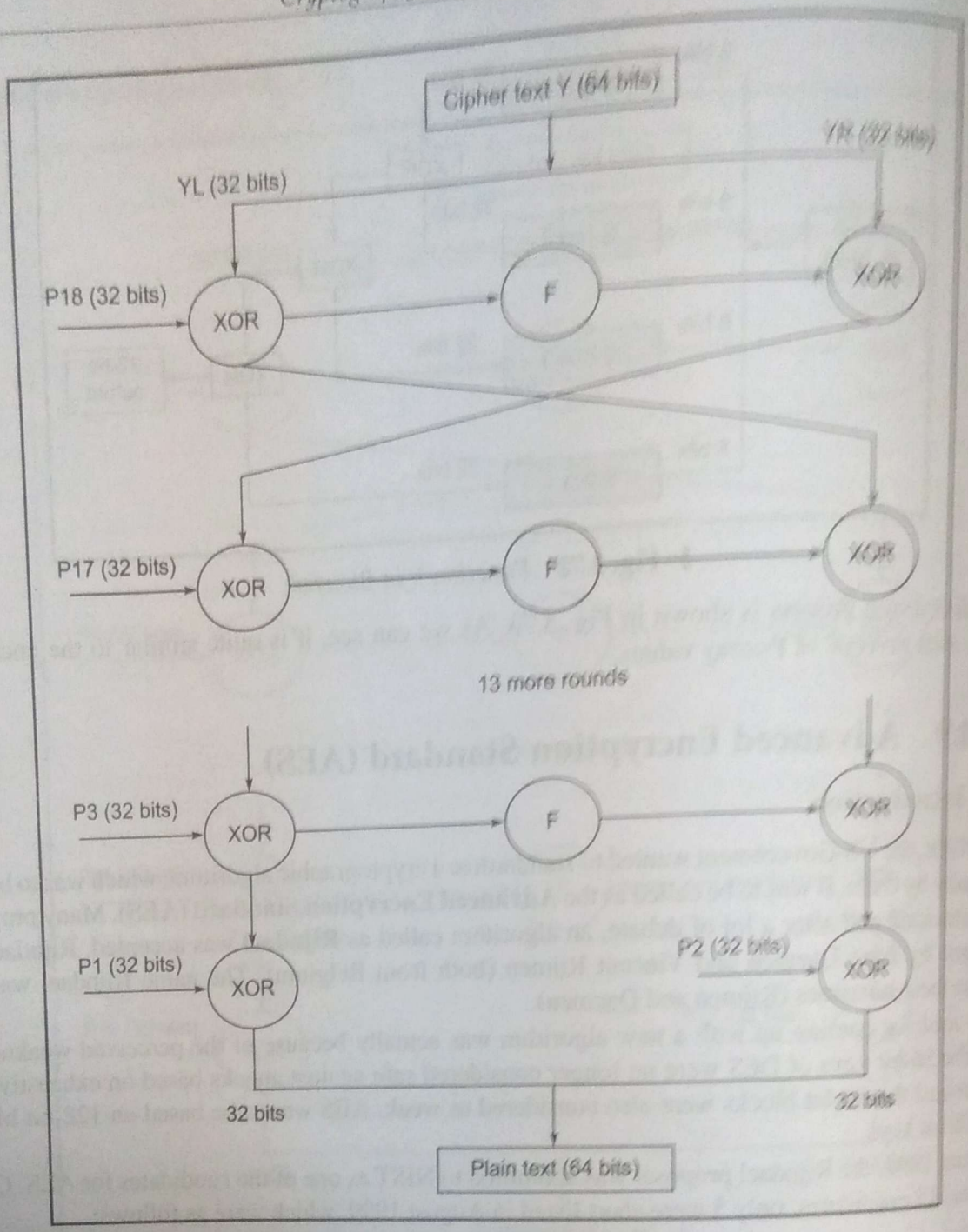

Fig. 3.73 *Blowfish decryption*

(i) Symmetric and parallel structure – This gives the implementers of the algorithm a lot of flexibility. It also stands up well against cryptanalysis attacks.

(ii) Adapted to modern processors – The algorithm works well with modern processors (Pentium, RISC, parallel).

(iii) Suited to smart cards – The algorithm can work well with smart cards.

Rijndael supports key lengths and plain text block sizes from 128 bits to 256 bits, in the steps of 32 bits. The key length and the length of the plain text blocks need to be selected independently. AES mandates that the plain text block size must be 182 bits and key size should be 128, 192 or 256 bits. In

general, two versions of AES are used: 128-bit plain text block combined with 128-bit key block and 128-bit plain text block with 256-bit key block.

Since the 128-bit plain text block and 128-bit key length are likely to pair as a commercial standard, we will examine that case only. Other principles remain the same. Since 128 bits give a possible key range of 2^{128} or 3×10^{38} keys, Andrew Tanenbaum outlines the strength of this key range in his unimitable style:

> *Even if NSA manages to build a machine with 1 billion parallel procesors, each being able to evaluate one key per picosecond, it would take such a machine about 10^{10} years to search the key space. By then the sun would have burned out, so the folks then present will have to read the results by candlelight.*

3.9.2 Operation

The basics of Rijndael are in a mathematical concept called as Galois field theory. For the current discussion, we will keep those concepts out and try and figure out how the algorithm itself works at a conceptual level.

Similar to the way DES functions, Rijndael also uses the basic techniques of substitution and transposition (i.e. permutation). The key size and the plain text block size decide how many rounds need to be executed. The minimum number of rounds is 10 (when key-size and the plain text block size are each 128 bits) and the maximum number of rounds is 14. One key differentiator between DES and Rijndael is that all the Rijndael operations involve entire byte and not individual bits of a byte. This provides for more optimized hardware and software implementation of the algorithm.

Figure 3.74 describes the steps in Rijndael at a high level.

(i) Do the following one-time initialization processes:
  (a) Expand the 16-byte key to get the actual *Key block* to be used.
  (b) Do one time initialization of the 16-byte plain text block (called as *State*).
  (c) XOR the *state* with the *key block*.
(ii) For each round, do the following:
  (a) Apply S-box to each of the plain text bytes.
  (b) Rotate row *k* of the plain text block (i.e. *state*) by *k* bytes.
  (c) Perform a *mix columns* operation.
  (d) XOR the *state* with the *key block*

Fig. 3.74 *The description of Rijndael*

Let us now understand how this algorithm works, step-by-step.

3.9.3 One-time Initialization Process

We will describe each of the algorithm steps in detail now.

(a) Expand the 16-byte Key to get the Actual Key Block to be Used The inputs to the algorithm are the key and the plain text, as usual. The key size is 16 bytes, in this case. This Step expands this 16-byte key into 11 arrays, each array contains 4 rows and 4 columns. Conceptually, the key expansion process can be depicted as shown in Fig. 3.75.

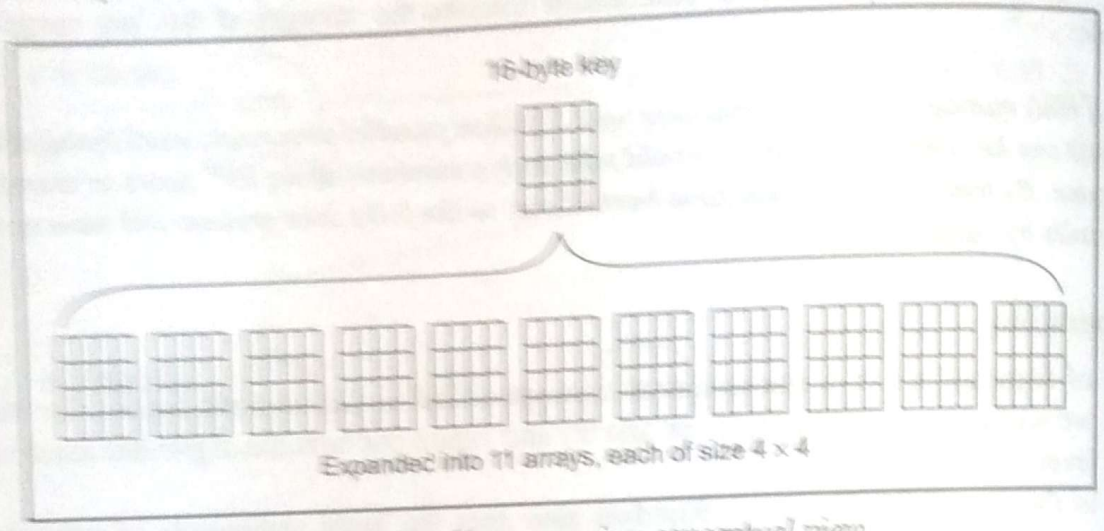

Fig. 3.75 *Key expansion: conceptual view*

In other words, the original 16-byte key array is expanded into a key containing $11 \times 4 \times 4 = 176$ bytes. One of these 11 arrays is used in the initialization process and the other 10 arrays are used in the 10 rounds, one array per round.

The key expansion process is quite complex and can be safely ignored. We nevertheless describe it below for the sake of completeness. Now, let us start using the terminology of *word*, in the context of AES. A word means 4 bytes. Therefore, in the current context, our 16-byte initial key (i.e. 16 / 4 = 4-word key) will be expanded into 176-byte key (i.e.176 / 4 words, i.e. 44 words).

1. Firstly, the original 16-byte key is copied into the first 4 words of the expanded key (i.e. the first 4 × 4 array of our diagram), as is. This is shown in Fig. 3.76.

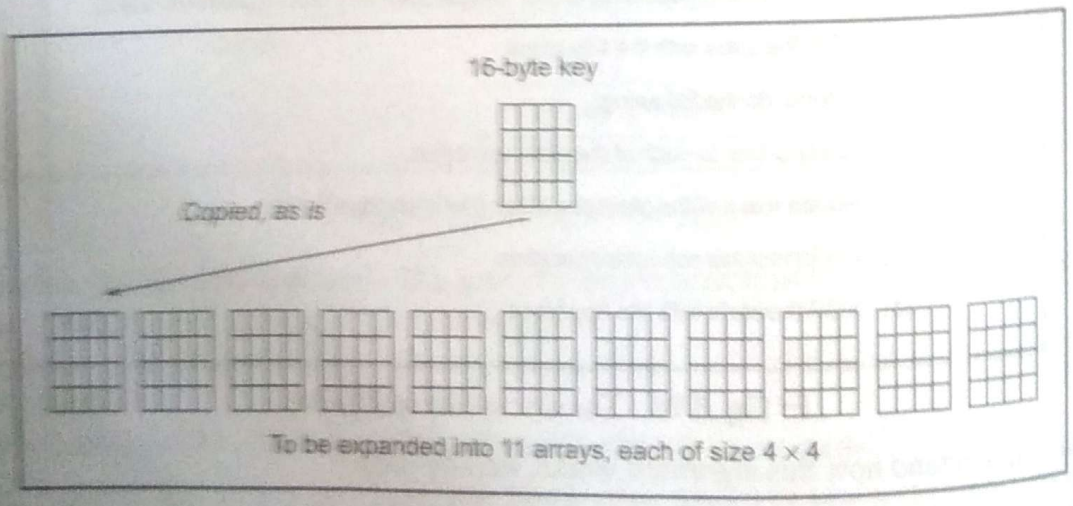

Fig. 3.76 *Key expansion: Step 1*

Another representation of the same concept is shown in Fig. 3.77.

2. After filling the first array (for words numbered W0 to W3) of the expanded key block as explained above, the remaining 10 arrays (for words numbered W4 to W43) are filled one-by-one. Every time, one such 4 × 4 array (i.e. four words) gets filled. Every added key array block depends on the immediately preceding block and the block 4 positions earlier to it. That is, every added word w [i] depends on w [i – 1] and w [i – 4]. We have said that this fills four words at a time. For filling these four words at a time, the following logic is used:

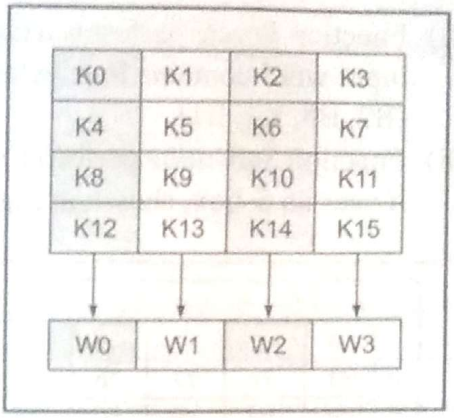

Fig. 3.77 *Key expansion: First Step*

   (a) If the word in the W array is a multiple of four, some complex logic is used, explained later. That is, for words W [4], W [8], W [12]... W [40], this complex logic would come into picture.
   (b) For others, a simple XOR is used.

This logic is described in an algorithm shown in Fig. 3.78.

```
ExpandKey (byte K [16], word W [44]) {
        word tmp;

        // First copy all the 16 input key blocks into first four words of output key
        for (i = 0; i < 4; i++) {
                W [i] = K [4*i], K [4*i + 1], K [4*i + 2], K [4*i + 3];
        }

        // Now populate the remaining output key words (i.e. W5 to W43)
        for (i = 4; i < 44; i++) {
                tmp = W [i-1];

                if (i mod 4 == 0)
                        tmp = Substitute (Rotate (temp)) XOR Constant [i/4];

                w [i] = w [i-4] XOR tmp;
        }
}
```

Fig. 3.78 *Key expansion described as an algorithm*

We have already discussed the portion of copying all the 16 input key blocks (i.e. 16 bytes) into the first four words of the output key block. Therefore, we will not discuss this again. This is described by the first *for* loop.

In the second *for* loop, we check if the current word being populated in the output key block is a multiple of 4. If it is, we perform three functions, titled *Substitute*, *Rotate* and *Constant*. We will describe them shortly. However, if the current word in the output key block is not a multiple of four, we simply perform an XOR operation of the previous word and the word four places earlier and store it as the output word. That is, for word W [5], we would XOR W [4] and W [1] and store their output as W [5]. This should be clear from the algorithm. Note that a temporary variable called as *tmp* is created, which stores W [i – 1], which is then XORed with W [i – 4].

Let us now understand the three functions, titled *Substitute*, *Rotate* and *Constant*.

(ii) Function *Rotate* performs a circular left shift on the contents of the word by one byte. Thus, if an input word contains four bytes numbered [B1, B2, B3, B4]; then the output word would contain [B2, B3, B4, B1].

(iii) Function *Substitute* performs a byte substitution on each byte of the input word. For this purpose, it uses an S-box, shown in Fig. 3.79.

	Y															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

X

Fig. 3.79 *AES S-box*

(iii) In the function *Constant*, the output of the above steps is XORed with a constant. This constant is a word, consisting of 4 bytes. The value of the constant depends on the round number. The last three bytes of a constant word always contain 0. Thus, XORing any input word with such a constant is as good as XORing only with the first byte of the input word. These constant values per round are listed in Fig. 3.80.

Round number	1	2	3	4	5	6	7	8	9	10
Value of constant to be used in Hex	01	02	04	08	10	20	40	80	1B	36

Fig. 3.80 *Values of constants per round, to be used in the Constant function*

Let us understand how the whole thing works, with an example.

1. Suppose that our original unexpanded 4-word (i.e. 16-byte) key is as shown in Fig. 3.81.

Byte position (Decimal)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value (Hex)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

Fig. 3.81 *Key expansion example – Step 1 – Original 4-word key*

2. In the first four rounds, the original 4-word input key would be copied into the first 4-word output key, as per our algorithm step as follows:

```
// First copy all the 16 input key blocks into first four words of output key
for (i = 0; i < 4; i++) {
        W [i] = K [4*i], K [4*i + 1], K [4*i + 2], K [4*i + 3];
}
```

Thus, the first 4 words of the output key (i.e. from W [0] to W [3]) would contain values as shown in Fig. 3.82. This is constructed from the input 4-word (i.e. 16-byte) key as follows:

(a) Firstly, the first four bytes, i.e. one word, of the input key, namely 00 01 02 03 is copied into the first word of the output key W [0].

(b) Now, the next four bytes of the input key, i.e. 04 05 06 07 would be copied into the second word of the output key, i.e. W [1].

Needless to say, W [2] and W [3] would get populated with the remaining contents of the input key bytes, as shown in the figure.

W [0]	W [1]	W [2]	W [3]	W [4]	W [5]	...	W [44]
00 01 02 03	04 05 06 07	08 00 0A 0B	0C 0D 0E 0F	?	?	?	?

Fig. 3.82 *Key expansion example – Step 2 – Filling of first 4 output key words*

3. Now let us understand how the next word of the output key block, i.e. W [4] is populated. For this purpose, the following algorithm would be executed:

```
// Now populate the remaining output key words (i.e. W5 to W43)
for (i = 4; i < 44; i++) {
    tmp = W [i - 1];

    if (i mod 4 == 0)
        tmp = Substitute (Rotate (temp)) XOR Constant [i/4];

    w [i] = w [i - 4] XOR tmp;
}
```

Based on this, we will have the following:

tmp = W [i – 1] = W [4 – 1] = W [3] = 0C 0D 0E 0F

Since $i = 4$, i mod 4 is 0. Therefore, we will now have the following Step:

tmp = Substitute (Rotate (temp)) XOR Constant [i/4];

- We know that *Rotate (temp)* will produce Rotate (0C 0D 0E 0F), which equals 0D 0E 0F 0C.
- Now, we need to do *Substitute (Rotate (temp))*. For this purpose, we need to take one byte at a time and look up in the S-box for substitution. For example, our first byte is 0D. Looking it up in the S-box with $x = 0$ and $y = D$ produces D7. Similarly, 0E produces AB, 0F produces 76 and 0C produces FE. Thus, at the end of the *Substitute (Rotate (temp))* step, our input of *0D 0E 0F 0C* is transformed into the output of *D7 AB 76 FE*.
- We now need to XOR this value with *Constant [i/4]*. Since $i = 4$, we need to obtain the value of *Constant [4/4]*, i.e. *Constant [1]*, which is 01, as per our earlier table of constants. As we know, we

need to pad this with three more bytes, all set to 00. Therefore, our constant value is 01 00 00 00. So, we have:

```
        D7 AB 76 FE
XOR
        10 00 00 00
=       D6 AB 76 FE
```

Thus, our new *tmp* value is *D6 AB 76 FE*.

- Finally, we need to XOR this *tmp* value with W [i – 4], i.e. with W [4 – 4], i.e. with W [0]. Thus, we have:

```
        D6 AB 76 FE
XOR
        00 01 02 03
=       D6 AA 74 FD
```

Thus, W [4] = D6 AA 74 FD.

We can use the same logic to derive the remaining expanded key blocks (W [5] to W [44]).

(b) Do One Time Initialization of the 16-byte Plain Text Block (Called as State) This step is relatively simple. Here, the 16-byte plain text block is copied into a two-dimensional 4 × 4 array called as *state*. The order of copying is in the column order. That is, the first four bytes of the plain text block get copied into the first column of the *state* array, the next four bytes of the plain text block get copied into the second column of the *state* array and so on. This is shown in Fig. 3.83 for every byte (numbered from B1 to B16).

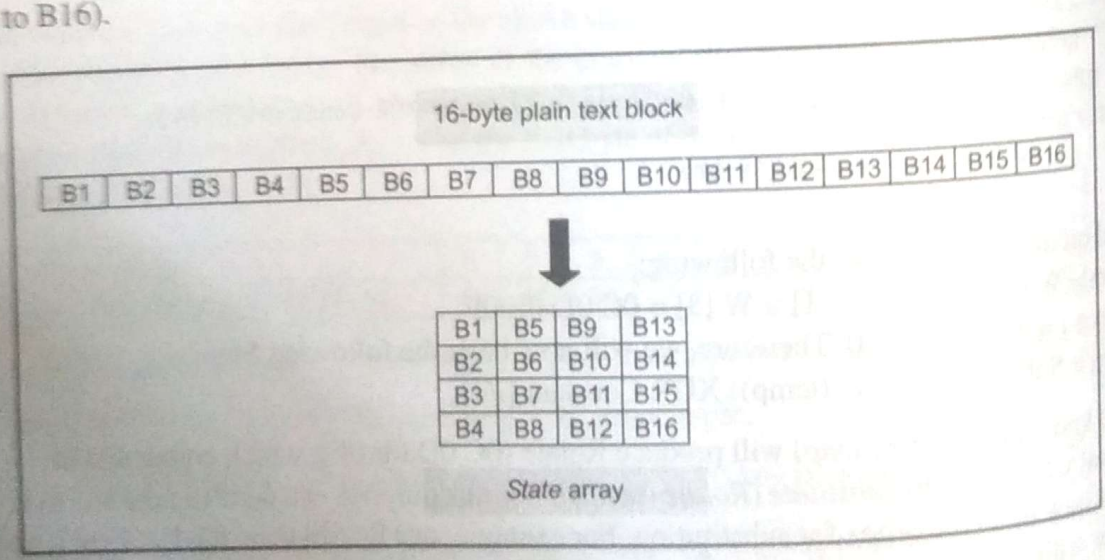

Fig. 3.83 *Copying of the input text block into state array*

(c) XOR the State with the Key block Now, the first 16 bytes (i.e. four words W [0], W [1], W [2] and W [3]) of the expanded key are XORed into the 16-byte *state* array (B1 to B16 shown above). Thus, every byte in the *state* array is replaced by the XOR of itself and the corresponding byte in the expanded key.

At this stage, the initialization is complete and we are ready for rounds.

3.9.4 Processes in Each Round

The following steps are executed 10 times, one per round.

(a) Apply S-box to Each of the Plain Text Bytes. This step is very straightforward. The contents of the *state* array are looked up into the S-box. Byte by byte substitution is done to replace the contents of the *state* array with the respective entries in the S-box. Note that only one S-box is used, unlike DES, which has multiple S-boxes.

(b) Rotate Row k of the Plain Text Block (i.e. state) by k Bytes. Here, each of the four rows of the *state* array are rotated to the left. Row 0 is rotated 0 bytes (i.e. not rotated at all), row 1 is rotated by 1 byte, row 2 is rotated 2 bytes and row 2 is rotated 3 bytes. This helps in diffusion of data. Thus, if the original 16 bytes of the *state* array contain values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 then the rotate operation would change the values as follows:

Original array	Modified array
1 5 9 13	1 5 9 13
2 6 10 14	6 10 14 2
3 7 11 15	11 15 3 7
4 8 12 16	16 4 8 12

(c) Perform a Mix Columns Operation Now, each column is mixed independent of the other. Matrix multiplication is used. The output of this step is the matrix multiplication of the old values and a constant matrix.

This is perhaps the most complex step to understand and also to explain. Nevertheless, we will make an attempt! This is based on a nice article by Adam Berent.

There are two aspects of this step. The first explains which parts of the state are multiplied against which parts of the matrix. The second explains how this multiplication is implemented over what's called as a **Galois Field**.

(c.1) Matrix Multiplication We know that the state is arranged into a 4×4 matrix.

The multiplication is performed one column at a time (i.e. on 4 bytes at a time). Each value in the column is eventually multiplied against every value of the matrix (i.e. 16 total multiplications are performed). The results of these multiplications are XORed together to produce only 4 resulting bytes for the next state. We together have 4 bytes input, 16 multiplications, 12 XORs and 4 bytes output. The multiplication is performed one matrix row at a time against each value of a *state* column.

For example, consider that our matrix is as shown in Fig. 3.84.

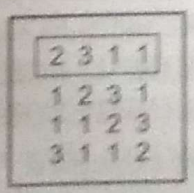

Fig. 3.84 *Multiplication matrix*

Next, let us imagine that the 16-byte state array is as shown in Fig. 3.85.

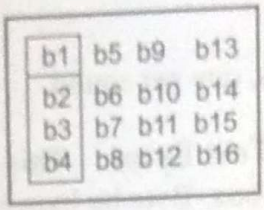

Fig. 3.85 *16-byte state array*

The first result byte is calculated by multiplying four values of the *state* column with the four values of the first row of the matrix. The result of each multiplication is then XORed to produce one byte. For this, the following calculation is used:

$$b1 = (b1 * 2) \text{ XOR } (b2 * 3) \text{ XOR } (b3 * 1) \text{ XOR } (b4 * 1)$$

Next, the second result byte is calculated by multiplying the same four values of the *state* column against four values of the second row of the matrix. The result of each multiplication is then XORed to produce one byte.

$$b2 = (b1 * 1) \text{ XOR } (b2 * 2) \text{ XOR } (b3 * 3) \text{ XOR } (b4 * 1)$$

The third result byte is calculated by multiplying the same four values of the *state* column against four values of the third row of the matrix. The result of each multiplication is then XORed to produce one byte.

$$b3 = (b1 * 1) \text{ XOR } (b2 * 1) \text{ XOR } (b3 * 2) \text{ XOR } (b4 * 3)$$

The fourth result byte is calculated by multiplying the same four values of the *state* column against four values of the fourth row of the matrix. The result of each multiplication is then XORed to produce one byte.

$$b4 = (b1 * 3) \text{ XOR } (b2 * 1) \text{ XOR } (b3 * 1) \text{ XOR } (b4 * 2)$$

This procedure is repeated again with the next column of the *state*, until there are no more state columns.

To summarize:

The first column will include *state* bytes 1-4 and will be multiplied against the matrix in the following manner:

$$b1 = (b1 * 2) \text{ XOR } (b2*3) \text{ XOR } (b3*1) \text{ XOR } (b4*1)$$
$$b2 = (b1 * 1) \text{ XOR } (b2*2) \text{ XOR } (b3*3) \text{ XOR } (b4*1)$$
$$b3 = (b1 * 1) \text{ XOR } (b2*1) \text{ XOR } (b3*2) \text{ XOR } (b4*3)$$
$$b4 = (b1 * 3) \text{ XOR } (b2*1) \text{ XOR } (b3*1) \text{ XOR } (b4*2)$$

($b1$= specifies the first byte of the state)

The second column will be multiplied against the second row of the matrix in the following manner:

$$b5 = (b5 * 2) \text{ XOR } (b6*3) \text{ XOR } (b7*1) \text{ XOR } (b8*1)$$
$$b6 = (b5 * 1) \text{ XOR } (b6*2) \text{ XOR } (b7*3) \text{ XOR } (b8*1)$$
$$b7 = (b5 * 1) \text{ XOR } (b6*1) \text{ XOR } (b7*2) \text{ XOR } (b8*3)$$
$$b8 = (b5 * 3) \text{ XOR } (b6*1) \text{ XOR } (b7*1) \text{ XOR } (b8*2)$$

And so on until all columns of the *state* are exhausted.

(c.2) Galois Field Multiplication The multiplication mentioned above is performed over a Galois Field. The mathematics behind this is quite complex and beyond the scope of the current text. However, we will discuss the implementation of the multiplication, which can be done quite easily with the use of the following two tables, shown in hexadecimal formats. Refer to Fig. 3.86 and Fig. 3.87.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

Fig. 3.86 *E table*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

Fig. 3.87 *L Table*

The result of the multiplication is actually the output of a lookup of the **L** table, followed by the addition of the results, followed by a lookup of the **E** table. Note that the term *addition* means a traditional mathematical addition and not a bit-wise AND operation.

All numbers being multiplied using the *Mix Column* function converted to Hex will form a maximum of 2 digit Hex number. We use the first digit in the number on the vertical index and the second number on the horizontal index. If the value being multiplied is composed of only one digit we use 0 on the

vertical index. For example if the two Hex values being multiplied are AF * 8 we first lookup L (AF) index which returns B7 and then lookup L (08) which returns 4B. Once the L table lookup is complete we can then simply add the numbers together. The only trick being that if the addition result is greater than FF, we subtract FF from the addition result. For example B7+4B= 102. Because 102 > FF, we perform: 102-FF which gives us 03.

The last step is to look up the addition result on the E table. Again we take the first digit to look up the vertical index and the second digit to look up the horizontal index.

For example E(03)=0F. Therefore, the result of multiplying AF * 8 over a Galois Field is 0F.

(d) XOR the State with the Key Block. This Step XORs the key for this round into the *state* array.

For decryption, the process can be executed in the reverse order. There is another option, though. The same encryption process, run with some different table values, can also perform decryption.

SUMMARY

- In symmetric key cryptography, the sender and the receiver share a single key and the same key is used for both encryption and decryption.
- Some of the important symmetric key cryptographic algorithms are DES (and its variations), IDEA, RC4, RC5 and Blowfish.
- An algorithm type defines what size of plain text should be encrypted in each step of the algorithm.
- There are two main types of algorithms: stream cipher and block cipher.
- In stream cipher, each bit/byte of text is encrypted/decrypted individually.
- In block cipher, a block of text is encrypted/decrypted at a time.
- An algorithm mode defines the details of the cryptographic algorithm, once the type is decided.
- Confusion is a technique of ensuring that a cipher text gives no clue about the original plain text.
- Diffusion increases the redundancy of the plain text by spreading it across rows and columns.
- There are four important algorithm modes: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB) and Output Feedback (OFB).
- There is a variation of the OFB mode, called as Counter (CTR).
- Electronic Code Book (ECB) is the simplest mode of operation. Here, the incoming plain text message is divided into blocks of 64 bits each. Each such block is then encrypted independently of the other blocks. For all blocks in a message, the same key is used for encryption.
- The Cipher Block Chaining (CBC) mode ensures that even if a block of plain text repeats in the input, these two (or more) identical plain text blocks yield totally different cipher text blocks in the output. For this, a feedback mechanism is used.

2. Now, calculate 41^{77} mod 119. This gives 6.
3. Decode 6 back to F from our alphabet numbering scheme. This gives back the decrypted (original) plain text.

4.4.3 Understanding the Crux of RSA

Based on the calculations done in our examples, it should be clear that the RSA algorithm itself is quite simple: choosing the right keys is the real challenge. Suppose B wants to receive a confidential message from A. B must generate a private key (D), a public key (E) by using the mechanisms discussed earlier. B must then give the public key (E) and the number N to A. Using E and N, A encrypts the message and then sends the encrypted message to B. B uses her private key (D) to decrypt the message.

The question is: if B can calculate and generate D, anyone else should be able to do that as well. However, this is not straightforward, herein lies the real crux of RSA.

It might appear that anyone (say an attacker) knowing about the public key E (5, in our second example) and the number N (119, in our second example) could find the private key D (77, in our second example) by trial and error. What does the attacker need to do? The attacker needs to first find out the values of P and Q using N (because we know that $N = P \times Q$). In our example, P and Q are quite small numbers (7 and 17, respectively). Therefore, it is quite easy to find out P and Q, given N. However, in actual practice, P and Q are chosen as very large numbers. Therefore, factoring N to find P and Q is not at all easy. It is quite complex and time-consuming. Since the attacker cannot find out P and Q, she cannot proceed further to find out D, because D depends on P, Q and E. Consequently, even if the attacker knows N and E, she cannot find D, therefore, cannot decrypt the cipher text.

Mathematical research suggests that it would take more than 70 years to find P and Q if N is a 100-digit number!

It is estimated that if we implement a symmetric algorithm such as DES and an asymmetric algorithm such as RSA in hardware, DES is faster by about 1000 times as compared to RSA. If we implement these algorithms in software, DES is faster by about 100 times.

4.5 Symmetric and Asymmetric Key Cryptography Together

4.5.1 Comparison Between Symmetric and Asymmetric Key Cryptography

Asymmetric key cryptography (or the use of the receiver's public key for encryption) solves the problem of key agreement and key exchange, as we have seen. However, this does not solve all the problems in a practical security infrastructure. More specifically, symmetric key cryptography and asymmetric key cryptography differ in certain other respects, with both depicting certain advantages as compared to the other. Let us summarize them to appreciate how they are practically used in real life, as shown in Table 4.2. We have not discussed the last point (*Usage*) in the table. However, it is mentioned for the sake of completeness. We shall explore those aspects shortly.

The above table shows that both symmetric key cryptography and asymmetric key cryptography have nice features. Also, both have some areas where better alternatives are generally desired. Asymmetric key cryptography solves the major problem of key agreement/key exchange as well as

Table 4.2 *Symmetric versus Asymmetric Key Cryptography*

Characteristic	*Symmetric key cryptography*	*Asymmetric key cryptography*
Key used for encryption/ decryption	Same key is used for encryption and decryption	One key used for encryption and another, different key is used for decryption
Speed of encryption/decryption	Very fast	Slower
Size of resulting encrypted text	Usually same as or less than the original clear text size	More than the original clear text size
Key agreement/exchange	A big problem	No problem at all
Number of keys required as compared to the number of participants in the message exchange	Equals about the square of the number of participants, so scalability is an issue	Same as the number of participants, so scales up quite well
Usage	Mainly for encryption and decryption (confidentiality), cannot be used for digital signatures (integrity and non-repudiation checks)	Can be used for encryption and decryption (confidentiality) as well as for digital signatures (integrity and non-repudiation checks)

scalability. However, it is far slower and produces huge chunks of cipher text as compared to symmetric key cryptography (essentially because it uses large keys and complex algorithms as compared to symmetric key cryptography).

4.5.2 The Best of both Worlds

It would be very effective, if we could combine the two cryptography mechanisms, so as to achieve the better of the two and yet do not compromise on any of the features? More specifically, we need to ensure that the following objectives are met:

1. The solution should be completely secure.
2. The encryption and decryption processes must not take a long time.
3. The generated cipher text should be compact in size.
4. The solution should scale to a large number of users easily, without introducing any additional complications.
5. The key distribution problem must be solved by the solution.

Indeed, in practice, symmetric key cryptography and asymmetric key cryptography are combined to have a very efficient security solution. The way it works is as follows, assuming that A is the sender of a message and B is its receiver.

1. A's computer encrypts the original plain text message (P T) with the help of a standard symmetric key cryptography algorithm, such as DES, IDEA or RC5, etc. This produces cipher text message (CT) as shown in Fig. 4.6. The key used in this operation (K1) is called as one-time symmetric key, as it is used once and then discarded.

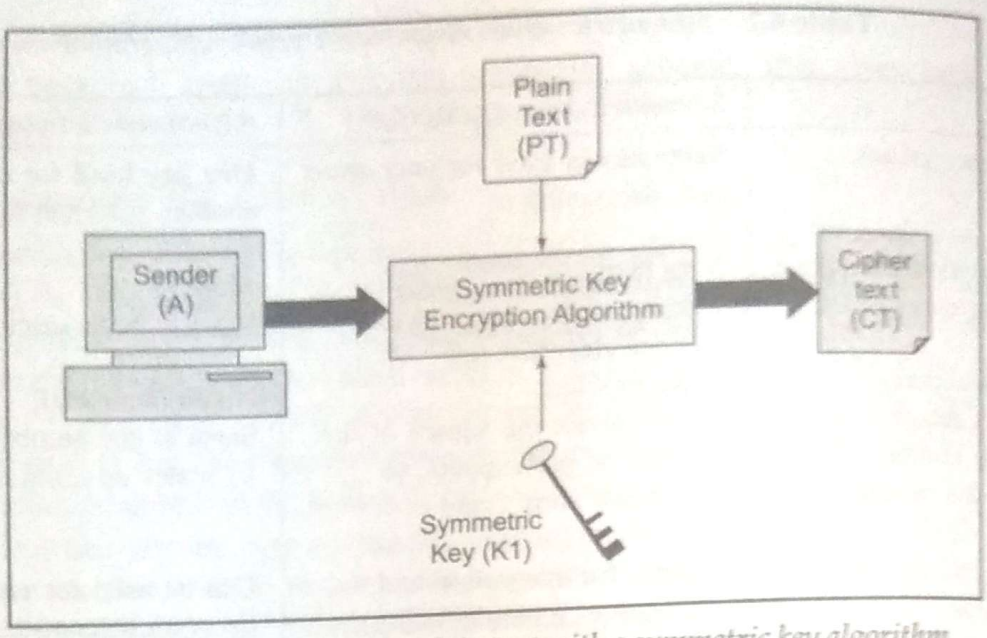

Fig. 4.6 *Encrypting the plain text with a symmetric key algorithm*

2. We would now think, we are back to square one! We have encrypted the plain text (PT) with a symmetric key operation. We must now transport this one-time symmetric key (K1) to the server so that the server can decrypt the cipher text (CT) to get back the original plain text message (PT). Does this not again lead us to the key exchange problem? Well, a novel concept is used now. A now takes the one-time symmetric key of Step 1 (i.e. K1) and encrypts K1 with B's public key (K2). This process is called as **key wrapping** of the symmetric key and is shown in Fig. 4.7. We have shown that the symmetric key K1 key goes inside a logical box, which is sealed by B's public key (i.e. K2).

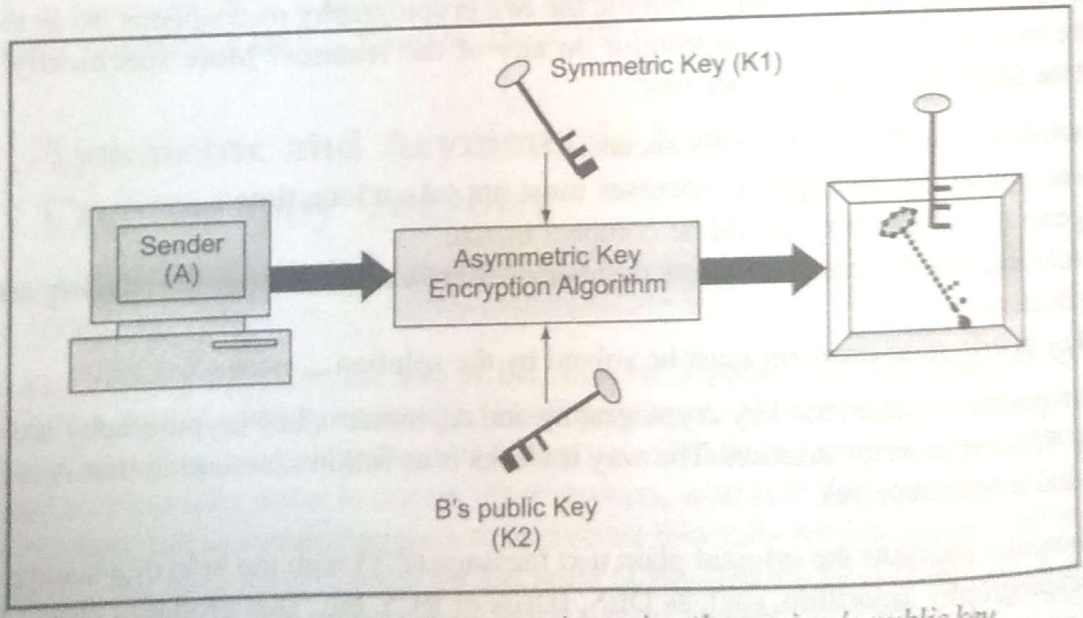

Fig. 4.7 *Symmetric key wrapping using the receiver's public key*

3. Now, A now puts the cipher text CT1 and the encrypted symmetric key together inside a digital **envelope**. This is shown in Fig. 4.8.

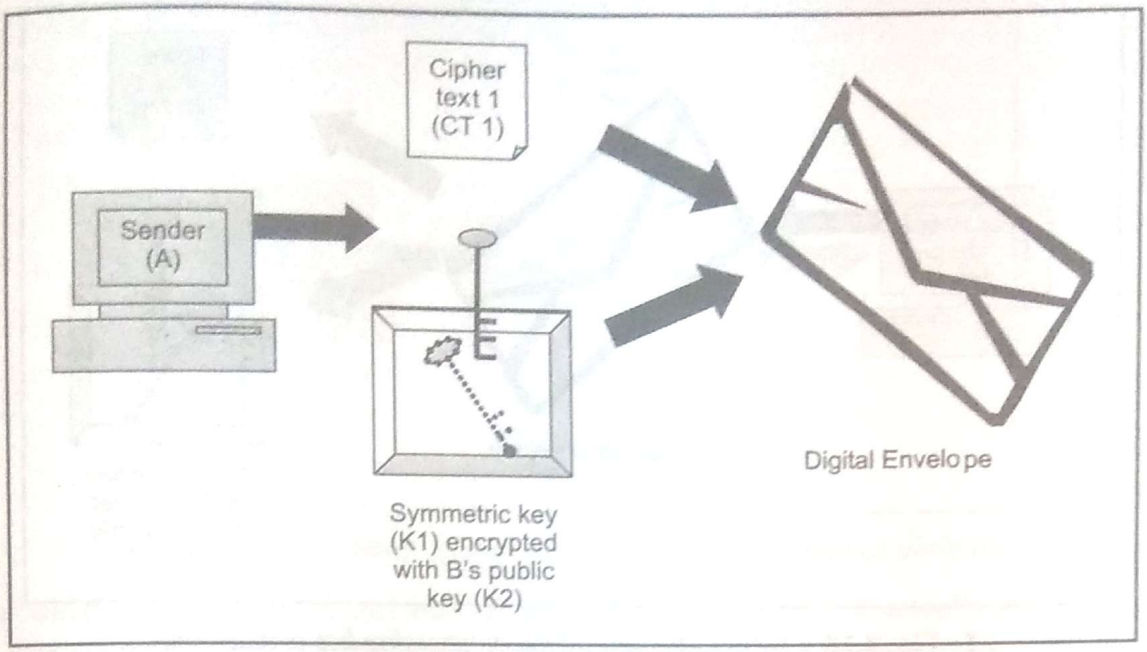

Fig. 4.8 *Digital envelope*

4. The sender (A) now sends the digital envelope [which contains the cipher text (CT) and the one-time symmetric key (K1) encrypted with B's public key, (K2)] to B using the underlying transport mechanism (network). This is shown in Fig. 4.9. We do not show the contents of the envelope and assume that the envelope contains the two entities, as discussed.

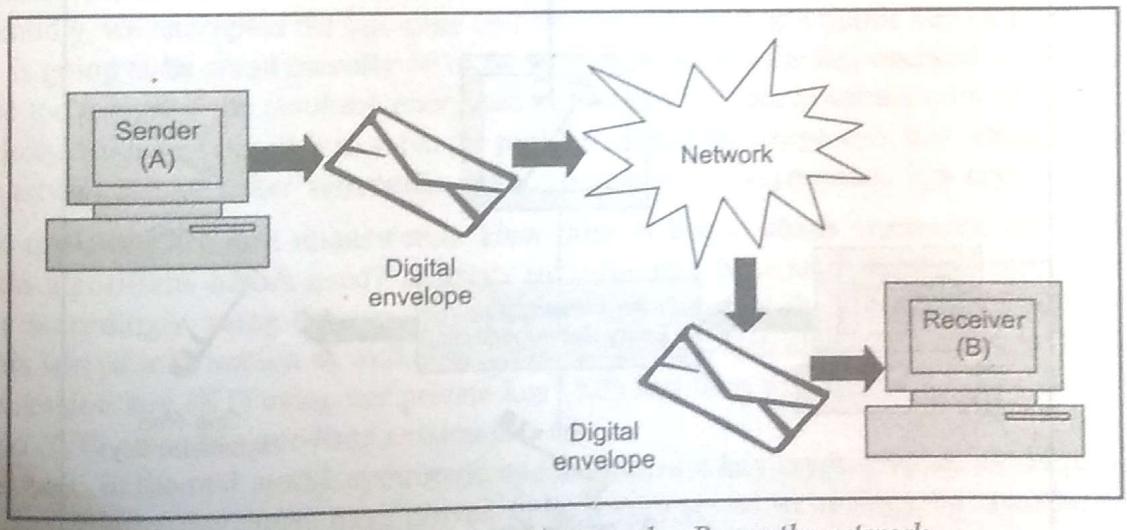

Fig. 4.9 *Digital envelope reaches B over the network*

5. B receives and opens the digital envelope. After B opens the envelope, it receives two things: cipher text (CT) and the one-time session key (K1) encrypted using B's private key (K2). This is shown in Fig. 4.10.

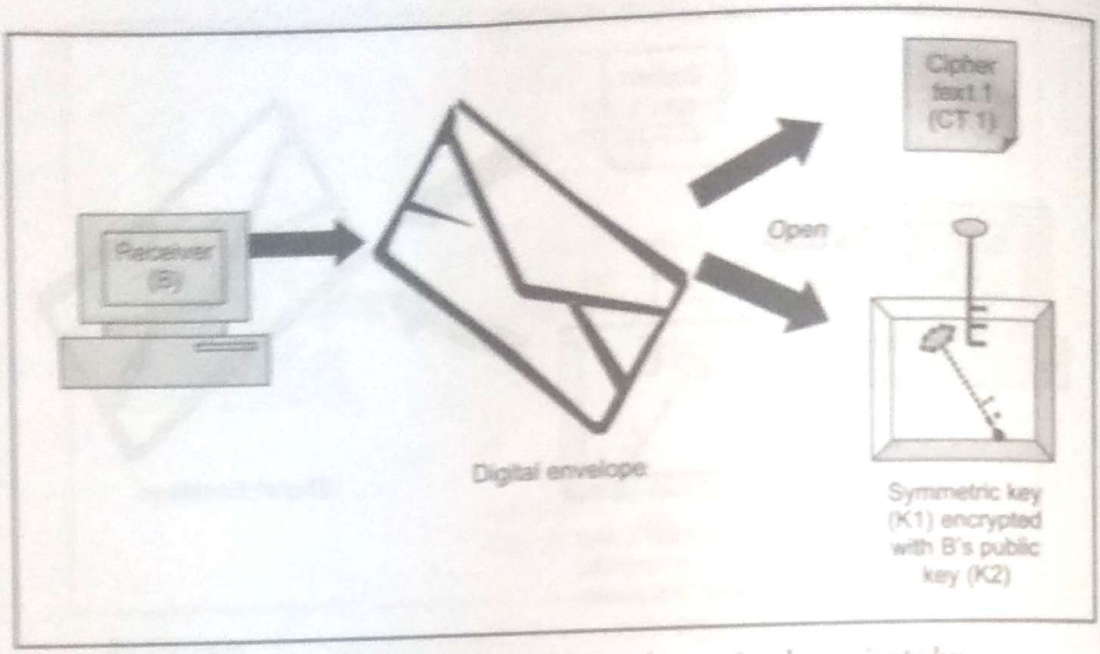

Fig. 4.10 *B opens the digital envelope using her private key*

6. B now uses the same asymmetric key algorithm as was used by A and her private key (K3) to decrypt (i.e. open up) the logical box that contains the symmetric key (K1), which was encrypted with B's public key (K2). This is shown in Fig. 4.11. Thus, the output of the process is the one-time symmetric key K1.

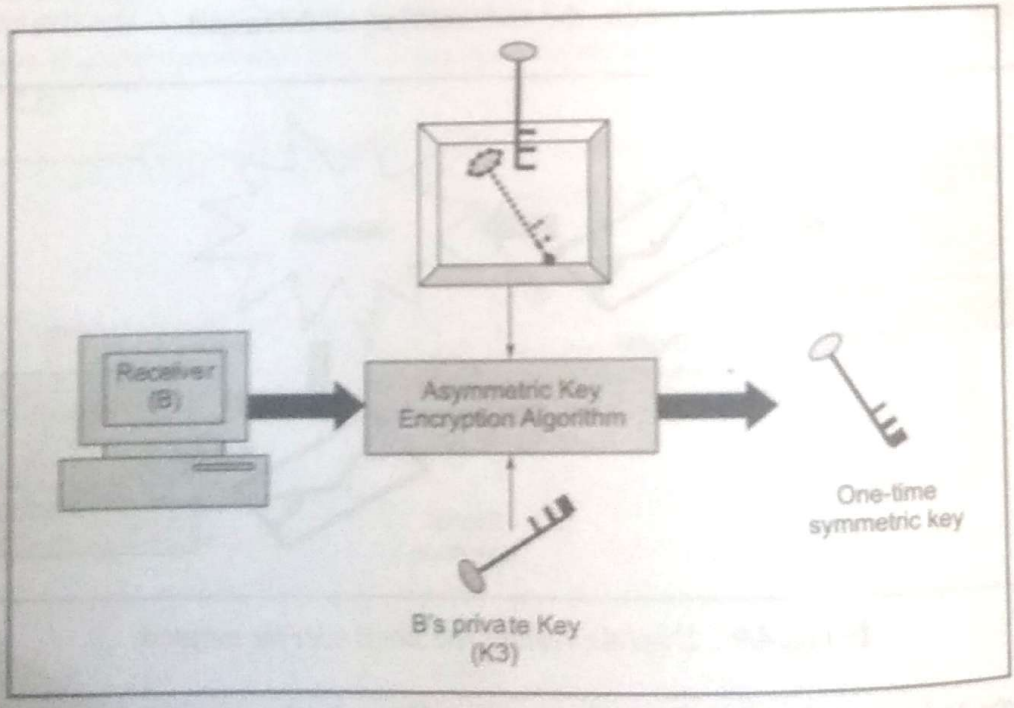

Fig. 4.11 *Retrieval of one-time symmetric key*

7. Finally, B applies the same symmetric key algorithm as was used by A and the symmetric key K1 to decrypt the cipher text (CT). This process yields the original plain text (PT), as shown in Fig. 4.12.

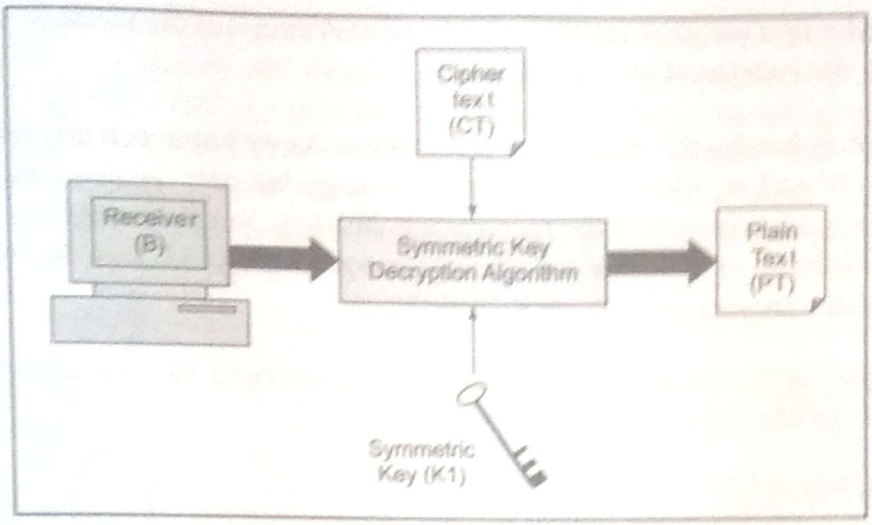

Fig. 4.12 *Using symmetric key to retrieve the original plain text*

One would think that this process looks complicated. How come this is actually efficient? The reason that this process based on digital envelopes is efficient is because of the following reasons:

(a) Firstly, we encrypt the plain text (PT) with the one-time session key (K1) using a symmetric key cryptographic algorithm. As we know, symmetric key encryption is fast and the generated cipher text (CT) is of the same size of the original plain text (PT). Instead, if we had used an asymmetric key encryption here, the operation would have been quite slow, especially if the plain text was of a large size, which it can be. Also, the output cipher text (CT) produced would have been of a size greater than the size of the original plain text (PT).

(b) Secondly, we encrypted the one-time session key (K1) with B's public key (K2). Since the size of K1 is going to be small (usually 56 or 64 bits), this asymmetric key encryption process would not take too long and the resulting encrypted key would not also consume a lot of space.

(c) Thirdly, we have been able to solve the problem of key exchange with this scheme, without losing the advantages of either symmetric key cryptography or asymmetric key cryptography!

A few questions are still unanswered. How does B know which symmetric or asymmetric key encryption algorithms has A used? B needs to know this because it must perform the decryption processes accordingly, using the same algorithms. Well, actually the digital envelope sent by A to B carries this sort of information as well. Therefore, B knows which algorithms to use to first decrypt the one-time session key (K1) using her private key (K3) and then which algorithm to use to decrypt the cipher text (CT) using the one-time session key (K1).

This is how, in the real world, symmetric and asymmetric key cryptographic techniques are used in combination. Digital envelopes have proven to be a very sound technology for transferring messages from the sender to the receiver, achieving confidentiality.

4.6 Digital Signatures

4.6.1 Introduction

All along, we have been talking of the following general scheme in the context of asymmetric key cryptography:

If A is the sender of a message and B is the receiver, A encrypts the message with B's public key and sends the encrypted message to B.

We have deliberately hidden the internals of this scheme. As we know, actually this is based on digital envelopes as discussed earlier, wherein not the entire message but only the one-time session key used to encrypt the message is encrypted with the receiver's public key. But for simplicity, we shall ignore this technical detail and instead, assume that the whole message is encrypted with the receiver's public key.

Let us now consider another scheme, as follows:

If A is the sender of a message and B is the receiver, A encrypts the message with A's private key and sends the encrypted message to B.

This is shown in Fig. 4.13.

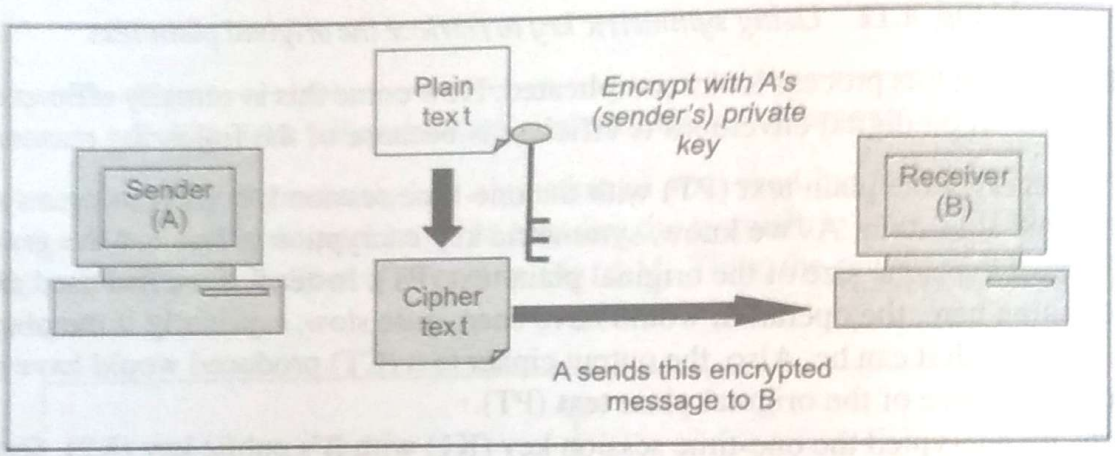

Fig. 4.13 *Encrypting a message with the sender's private key*

Our first reaction to this would be, what purpose would this serve? After all, A's public key would be, well, *public*, i.e. accessible to anybody. This means that anybody who is interested in knowing the contents of the message sent by A to B can simply use A's public key to decrypt the message, thus causing the failure of this encryption scheme!

Well, this is quite true. But here, when A encrypts the message with her private key, her intention is not to hide the contents of the message (i.e. not to achieve *confidentiality*), but it is something else. What can that intention be? If the receiver (B) receives such a message encrypted with A's private key, B can use A's public key to decrypt it and therefore, access the plain text. Does this ring a bell? If the decryption is successful, it assures B that this message was indeed sent by A. This is because if B can decrypt a message with A's public key, it means that the message must have been initially encrypted with A's private key (remember that a message encrypted with a public key can be decrypted only with the corresponding private key and vice versa). This is also because only A knows her private key. Therefore, someone posing as A (say C) could not have sent a message encrypted with A's private key to B. A must have sent it. Therefore, although this scheme does not achieve confidentiality, it achieves *authentication* (identifying and proving A as the sender). Moreover, in the case of a dispute tomorrow, B can take the encrypted message and decrypt it with A's public key to prove that the message indeed came from A. This achieves the purpose of *non-repudiation* (i.e. A cannot refuse that she had sent this message, as the message was encrypted with her private key, which is supposed to be known only to her).

Even if someone (say C) manages to intercept and access the encrypted message while it is in transit, then uses A's public key to decrypt the message, changes the message, that would not achieve any purpose. Because C does not have A's private key, C cannot encrypt the changed message with A's private key again. Therefore, even if C now forwards this changed message to B, B will not be fooled into believing that it came from A, as it was not encrypted with A's private key.

Such a scheme, wherein the sender encrypts the message with her private key, forms the basis of digital signatures, as is shown in Fig. 4.14.

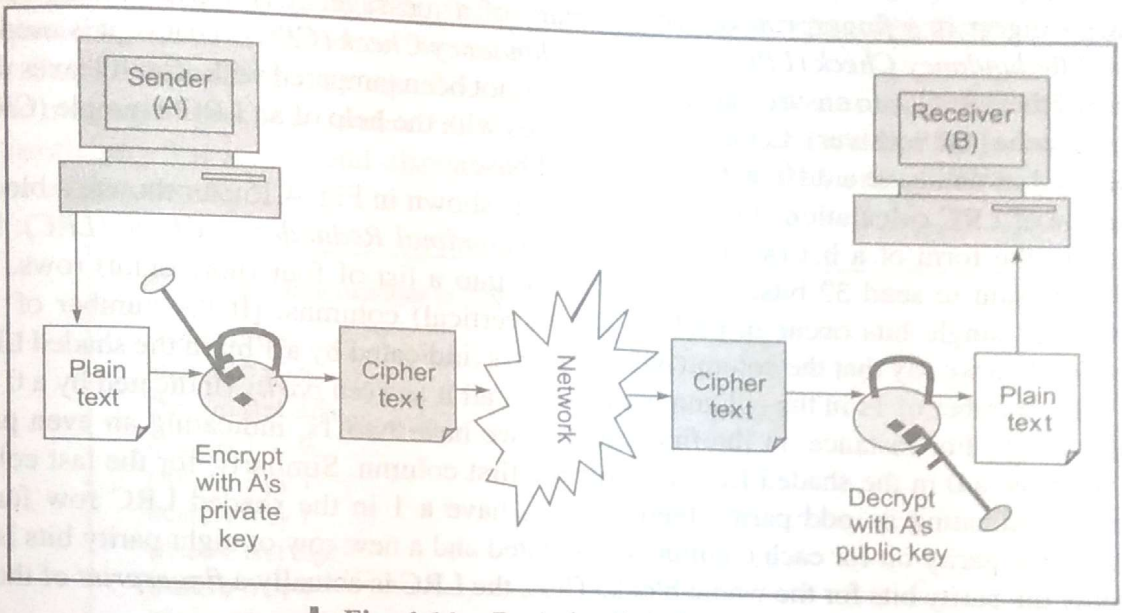

Fig. 4.14 *Basis for digital signatures*

Digital signatures have assumed great significance in the modern world of Web-commerce. Most countries have already made provisions for recognizing a digital signature as a valid authorization mechanism, just like paper-based signatures. Digital signatures have legal status now. For example, suppose you send a message to your bank over the Internet, to transfer some amount from your account to your friend's account and digitally sign the message, this transaction has the same status as the one wherein you fill in and sign the bank's paper-based money transfer slip.

We have seen the theory behind digital signatures. However, there are some undesirable elements in this scheme, which we shall study in the following sections.

4.6.2 Message Digests

Introduction If we examine the conceptual process of digital signatures, we will realize that it does not deal with the problems associated with asymmetric key encryption, namely slow operation and large cipher text size. This is because we are encrypting the whole of the original plain text message with the sender's private key. As the size of the original plain text can be quite large, this encryption process can be really very slow.

We can tackle this problem using the digital envelope approach, as before. That is, A encrypts the original plain text message (PT) with a one-time symmetric key (K1) to form the cipher text (CT). It then encrypts the one-time symmetric key (K1) with her private key (K2). She creates a digital envelope containing CT and K1 encrypted with K2 and sends the digital envelope to B. B opens the digital

envelope, uses A's public key (K3) to decrypt the encrypted one-time symmetric key and obtains the symmetric key K1. It then uses K1 to decrypt the cipher text (CT) and obtains the original plain text (PT). Since B uses A's public key to decrypt the encrypted one-time symmetric key (K1), B can be assured that only A's private key could have encrypted K1. Thus, B can be assured that the digital envelope came from A.

Such a scheme could work perfectly. However, in real practice, a more efficient scheme is used. It involves the usage of a **message digest** (also called as **hash**).

A message digest is a *fingerprint* or the summary of a message. It is similar to the concepts of *Longitudinal Redundancy Check (LRC)* or *Cyclic Redundancy Check (CRC)*. That is, it is used to verify the *integrity* of the data (i.e. to ensure that a message has not been tampered with after it leaves the sender but before it reaches the receiver). Let us understand this with the help of an LRC example (CRC would work similarly, but will have a different mathematical base).

An example of LRC calculation at the sender's end is shown in Fig. 4.15. As shown, a block of bits is organized in the form of a list (as rows) in the *Longitudinal Redundancy Check (LRC)*. Here, for instance, if we want to send 32 bits, we arrange them into a list of four (horizontal) rows. Then we count how many single bits occur in each of the 8 (vertical) columns. [If the number of 1s in the column is odd, then we say that the column has *odd parity* (indicated by a 1 bit in the shaded LRC row); otherwise if the number of 1s in the column is even, we call it as *even parity* (indicated by a 0 bit in the shaded LRC row).] For instance, in the first column, we have two 1s, indicating an even parity and therefore, we have a 0 in the shaded LRC row for the first column. Similarly, for the last column, we have three 1s, indicating an odd parity, therefore, we have a 1 in the shaded LRC row for the last column. Thus, the parity bit for each column is calculated and a new row of eight parity bits is created. These become the parity bits for the whole block. Thus, the LRC is actually a *fingerprint* of the original message.

The data along with the LRC is then sent to the receiver. The receiver separates the data block from the LRC block (shown shaded). It performs its own LRC on the data block alone. It then compares its LRC values with the ones received from the sender. If the two LRC values match, then the receiver has a reasonable confidence that the message sent by the sender has not been changed, while in transit.

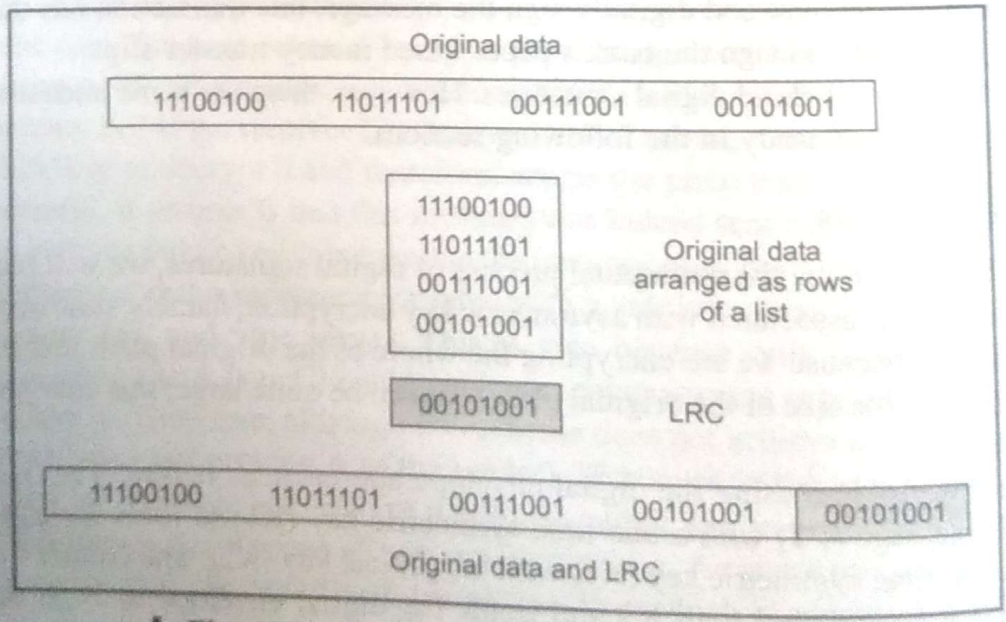

Fig. 4.15 *Longitudinal redundancy check (LRC)*

Idea of a Message Digest The concept of message digests is based on similar principles. However, it is slightly wider in scope. For instance, suppose we have a number 4000 and we divide it by 4 to get 1000, 4 becomes a fingerprint of the number 4000. Dividing 4000 by 4 will always yield 1000. If we change either 4000 or 4, the result will not be 1000.

Another important point is, if we are simply given the number 4, but are not given any further information, we would not be able to trace back the equation 4 x 1000 = 4000. Thus, we have one more important concept here. The fingerprint of a message (in this case, the number 4) does not tell anything about the original message (in this case, the number 4000). This is because there are infinite other possible equations, which can produce the result 4.

Another simple example of message digest is shown in Fig. 4.16. Let us assume that we want to calculate the message digest of a number 7391753. Then, we multiply each digit in the number with the next digit (excluding it if it is 0) and disregarding the first digit of the multiplication operation, if the result is a two-digit number.

- Original number is 7391743

Operation	Result
Multiply 7 by 3	21
Discard first digit	1
Multiply 1 by 9	9
Multiply 9 by 1	9
Multiply 9 by 7	63
Discard first digit	3
Multiply 3 by 4	12
Discard first digit	2
Multiply 2 by 3	6

- Message digest is 6

Fig. 4.16 *Simplistic example of message digest*

Thus, we perform a hashing operation (or a message digest algorithm) over a block of data to produce its hash or message digest, which is smaller in size than the original message. This concept is shown in Fig. 4.17.

So far, we are considering very simple cases of message digests. Actually, the message digests are not so small and straightforward to compute. Message digests usually consist of 128 or more bits. This means that the chance of any two message digests being the same is anything between 0 to at least 2^{128}. The message digest length is chosen to be so long with a purpose. This ensures that the scope for two message digests being the same.

Requirements of a Message Digest We can summarize the requirements of the message digest concept, as follows:

1. Given a message, it should be very easy to find its corresponding message digest. This is shown in Fig. 4.18. Also, for a given message, the message digest must always be the same.
2. Given a message digest, it should be very difficult to find the original message for which the digest was created. This is shown in Fig. 4.19.

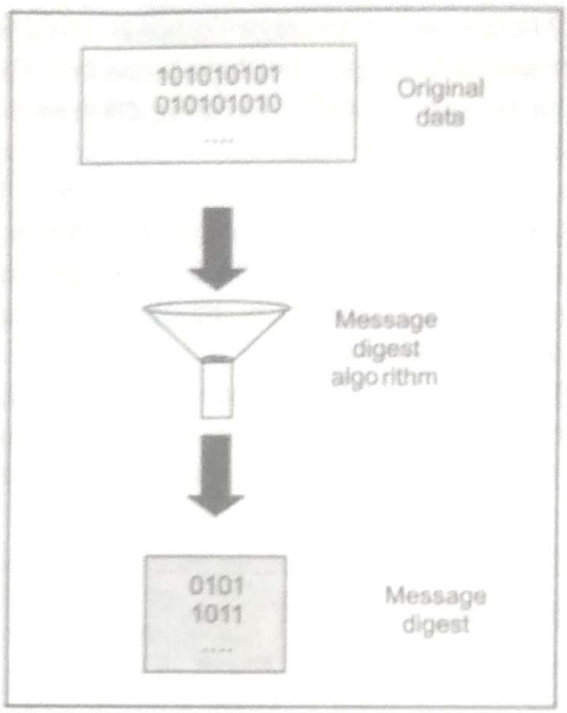

Fig. 4.17 *Message digest concept*

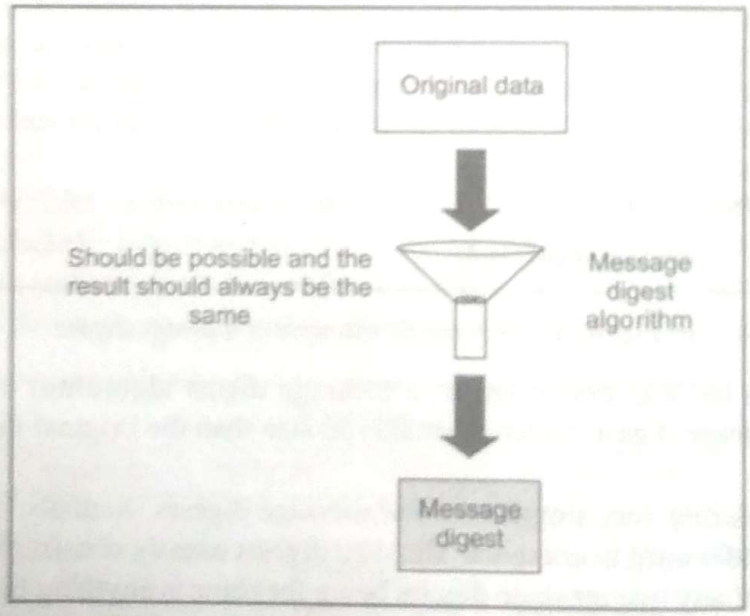

Fig. 4.18 *Message digest for the same original data should always be the same*

3. Given any two messages, if we calculate their message digests, the two message digests must be different. This is shown in Fig. 4.20.

If any two messages produce the same message digest, thus violating our principle, it is called as a **collision**. That is, if two message digests *collide*, they meet at the digest! As we shall study soon, the message digest algorithms usually produce a message digest of length 128 bits or 160 bits. This means that the chances of any two message digests being the same are one in 2^{128} or 2^{160}, respectively. Clearly, this seems possible only in theory, but extremely rare in practice.

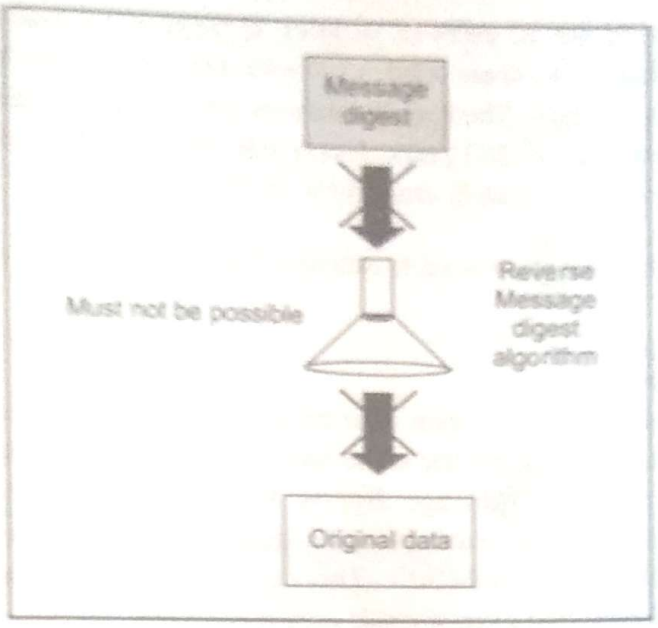

Fig. 4.19 *Message digest should not work in the opposite direction*

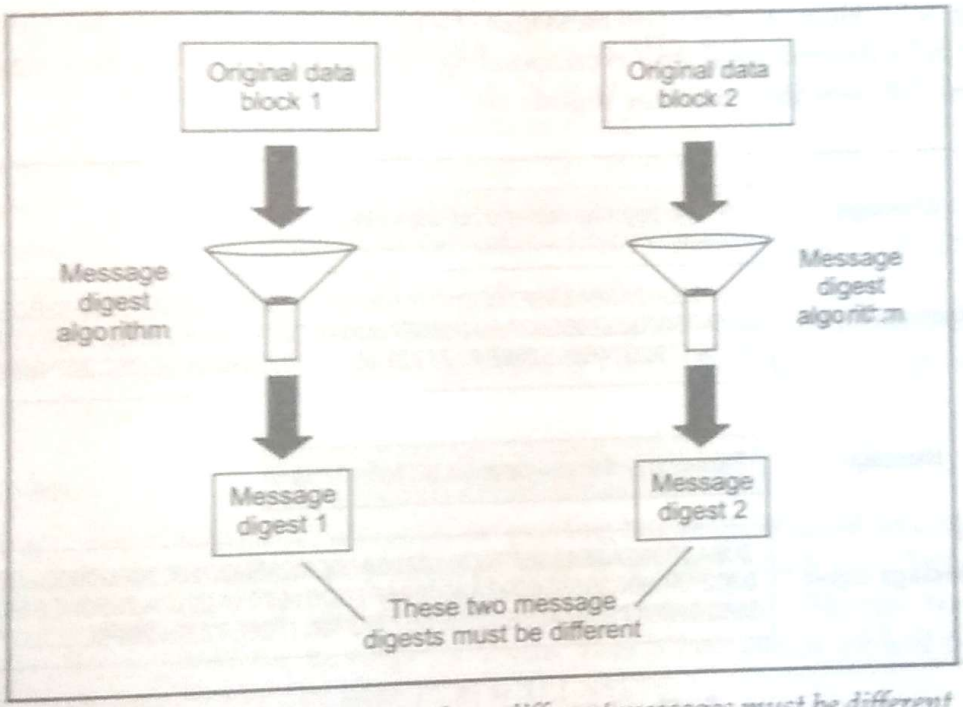

Fig. 4.20 *Message digests of two different messages must be different*

A specific type of security attack called as **birthday attack** is used to detect collisions in message digest algorithms. It is based on the principle of *Birthday Paradox*, which states that if there are 23 people in a room, chances are more than 50% that two of the people will share the same birthday. At first, this may seem to be illogical. However, we can understand this in another manner. We need to keep in mind we are just talking about *any two* people (out of the 23) sharing the same birthday. Moreover, we are not talking about this sharing with a specific person. For instance, suppose that we have Alice, Bob and Carol as three of the 23 people in the room. Therefore, Alice has 22 possibilities to share a birthday

with anyone else (since there are 22 pairs of people). If there is no matching birthday for Alice, she leaves. Bob now has 21 chances to share a birthday with anyone else in the room. If he fails to have a match too, the next person is Carol. She has 20 chances and so on. 22 pairs + 21 pairs + 20 pairs ... + 1 pair means that there is a total of 253 pairs. Every pair has a $1/365^{th}$ chance of finding a matching birthday. Clearly, the chances of a match cross 50% at 253 pairs.

The birthday attack is most often used to attempt discover collisions in hash functions, such as MD5 or SHA1.

This can be explained as follows:

If a message digest uses 64-bit keys, then after trying 2^32 transactions, an attacker can expect that for two different messages, we may get the same message digests. In general, for a given message, if we can compute up to N different message digests, then we can expect the first collision after the number of message digests computed exceeds square-root of N. In other words, a collision is expected when the probability of collision exceeds 50%. This can lead to birthday attacks.

It might surprise us to know that even a small difference between two original messages can cause the message digests to differ vastly. The message digests of two extremely similar messages are so different that they provide no clue at all that the original messages were very similar to each other. This is shown in Fig. 4.21. Here, we have two messages (*Please pay the newspaper bill today* and *Please pay the newspaper bill tomorrow*) and their corresponding message digests. Note how similar the messages are and yet how different their message digests are.

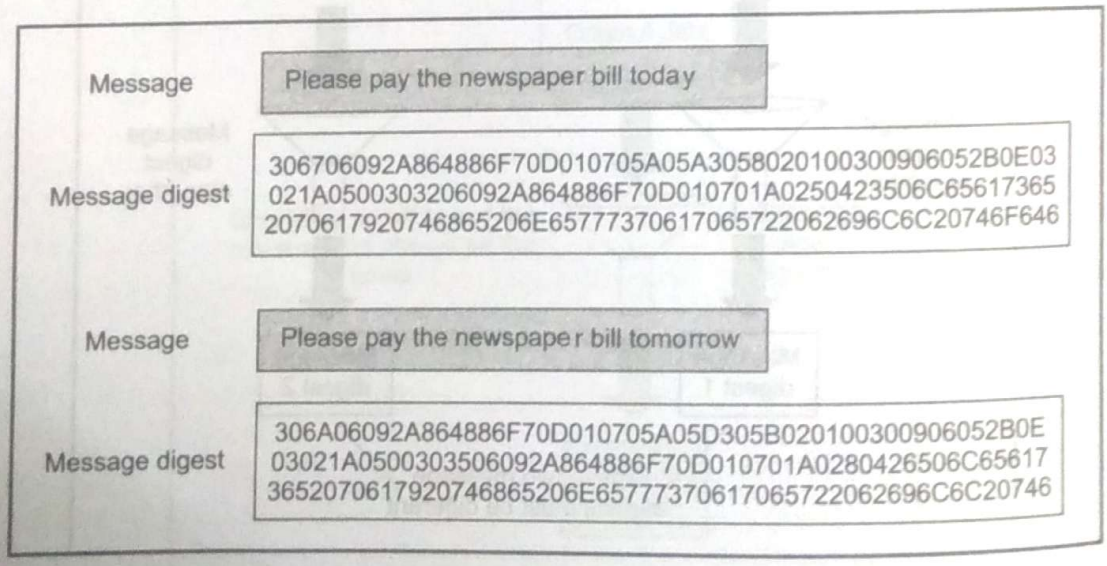

Fig. 4.21 *Message digest example*

Looked another way, we are saying that given one message (M1) and its message digest (MD), it is simply not feasible to find another message (M2), which will also produce MD exactly the same, bit-by-bit. The message digest scheme should try and prevent this to the maximum extent possible. This is shown in Fig. 4.22.

4.6.3 MD5

Introduction **MD5** is a message digest algorithm developed by Ron Rivest. MD5 actually has its roots in a series of message digest algorithms, which were the predecessors of MD5, all developed by

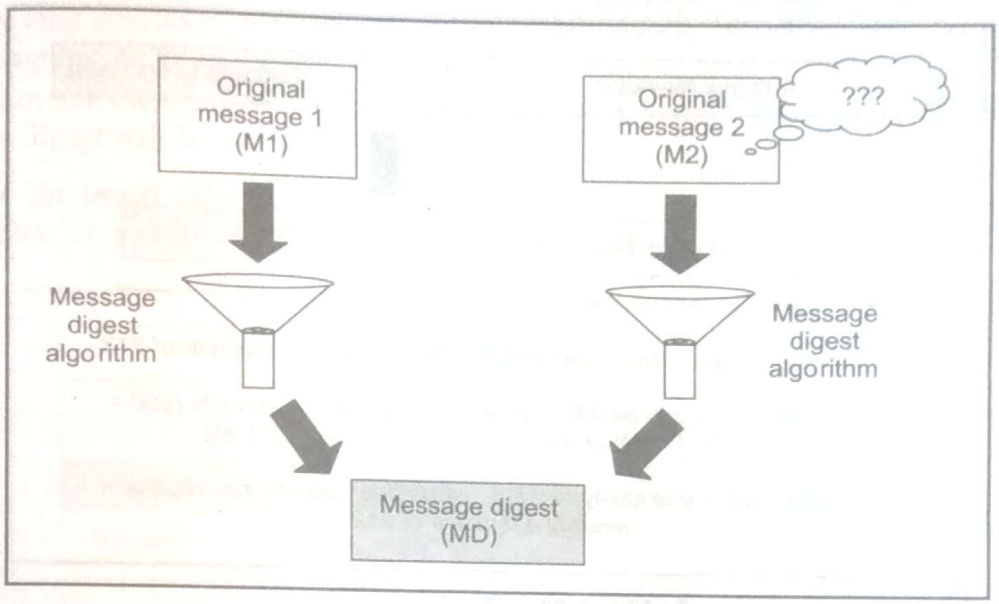

Fig. 4.22 *Message digests should not reveal anything about the original message*

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?

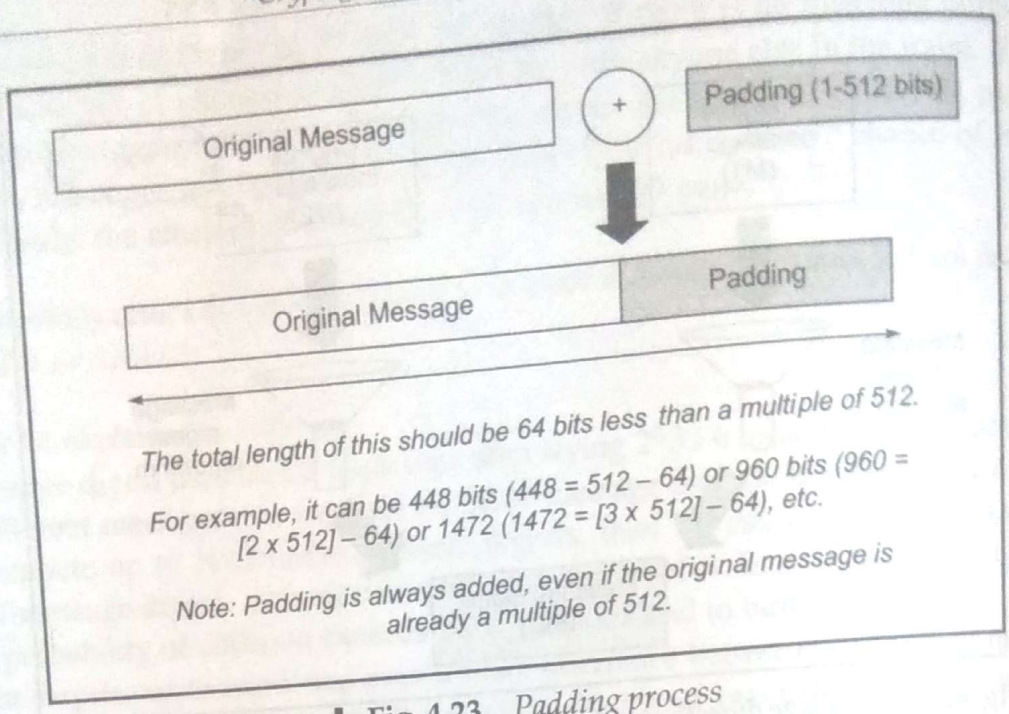

Fig. 4.23 *Padding process*

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 a 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

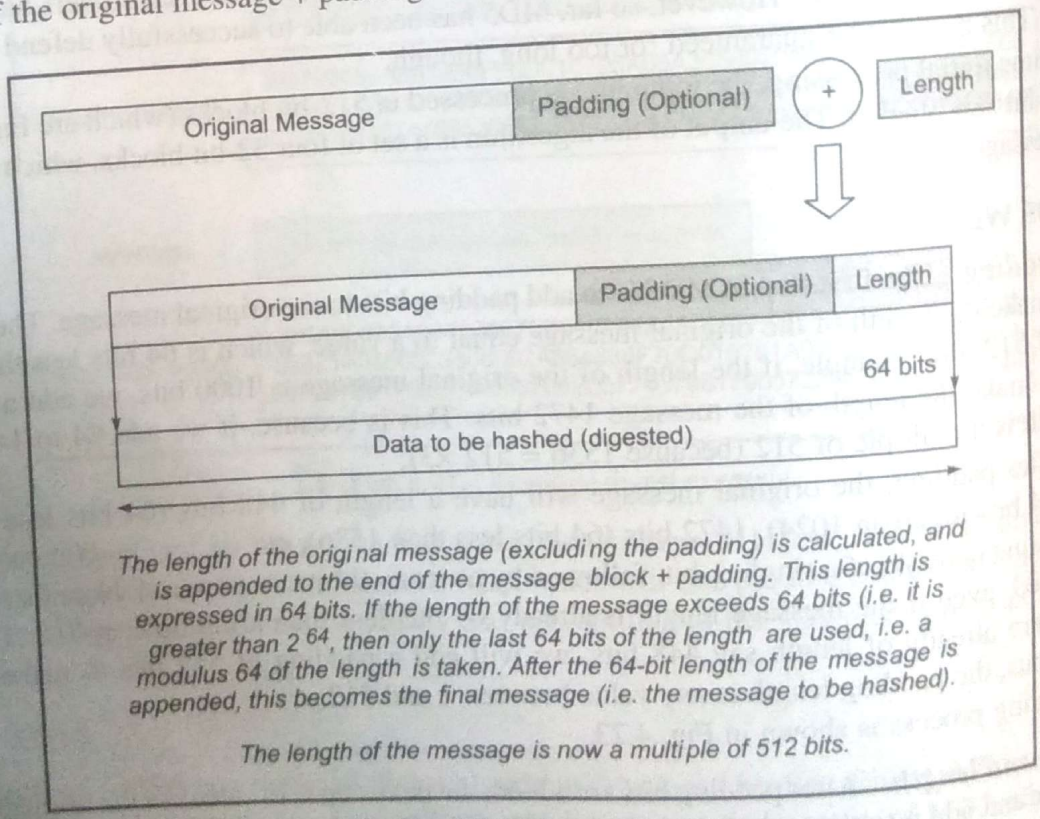

Fig. 4.24 *Append length*

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.

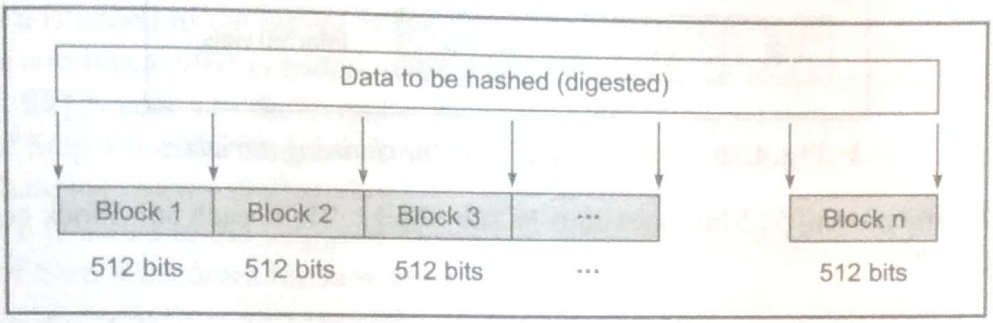

Fig. 4.25 *Data is divided into 512-bit blocks*

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.

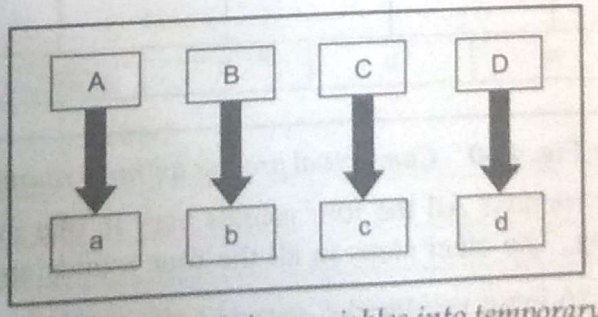

Fig. 4.27 *Copying chaining variables into temporary variables*

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.

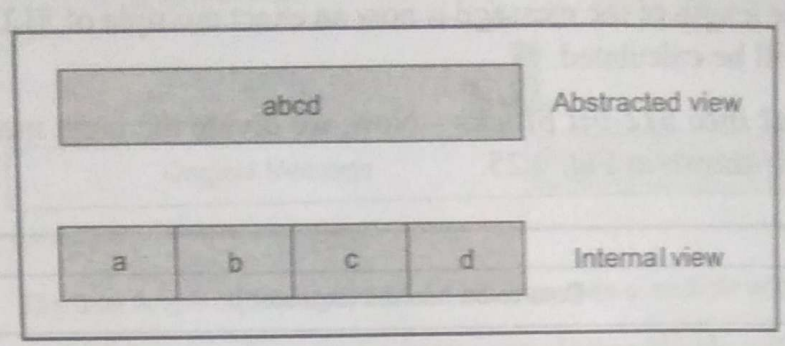

Fig. 4.28 *Abstracted view of the chaining variables*

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.

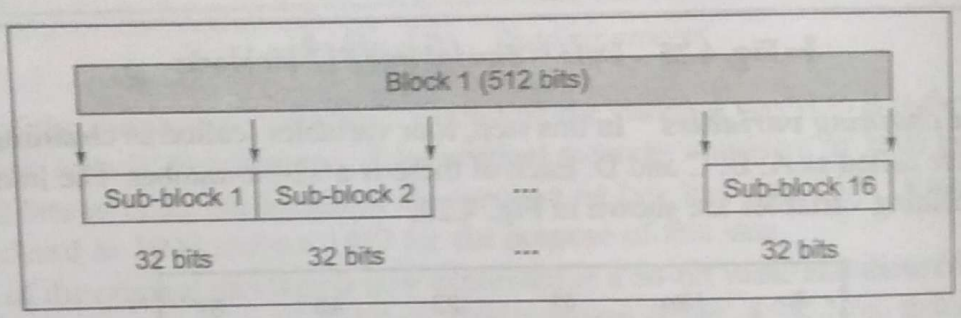

Fig. 4.29 *Sub-blocks within a block*

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.

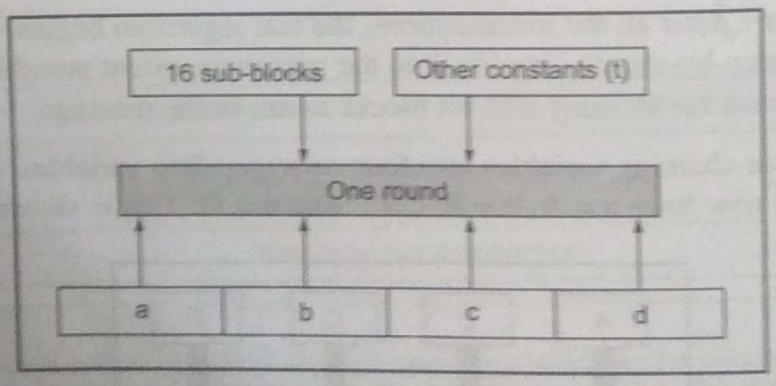

Fig. 4.30 *Conceptual process within a round*

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.

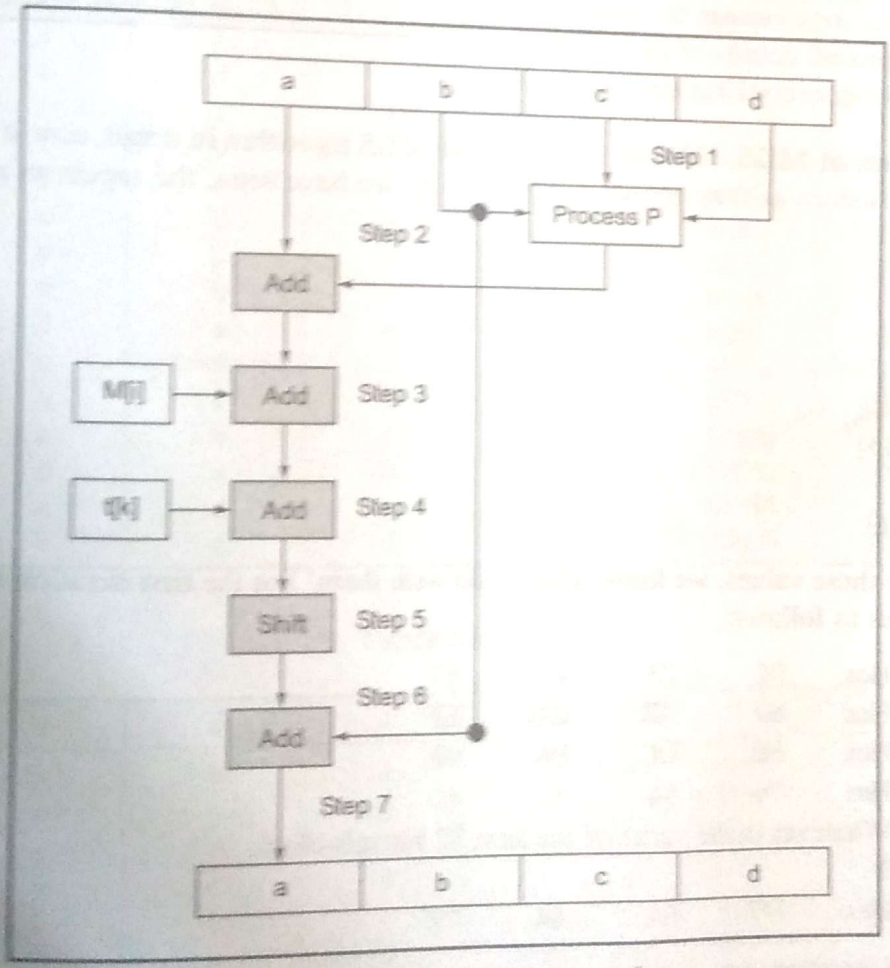

Fig. 4.31 *One MD5 operation*

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 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

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.

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))

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.

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:

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

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	t[26]
11	c	d	a	b	M[3]	14	t[27]
12	b	c	d	a	M[8]	20	t[28]
13	a	b	c	d	M[13]	5	t[29]
14	d	a	b	c	M[2]	9	t[30]
15	c	d	a	b	M[7]	14	t[31]
16	b	c	d	a	M[12]	20	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. For the remaining 64 steps, the value of W_t is equal to the circular left shift by one bit of the XOR of the four preceding values of W_t, with two of them subjected to shift and rotate operations.

This makes the message digest more complex and difficult to break.

4.6.6 Message Authentication Code (MAC)

The concept of **Message Authentication Code (MAC)** is quite similar to that of a message digest. However, there is one difference. As we have seen, a message digest is simply a fingerprint of a message. There is no cryptographic process involved in the case of message digests. In contrast, a MAC requires that the sender and the receiver should know a shared symmetric (secret) key, which is used in the preparation of the MAC. Thus, MAC involves cryptographic processing. Let us see how it works.

Let us assume that the sender A wants to send a message M to a receiver B. How the MAC processing works is shown in Fig. 4.34.

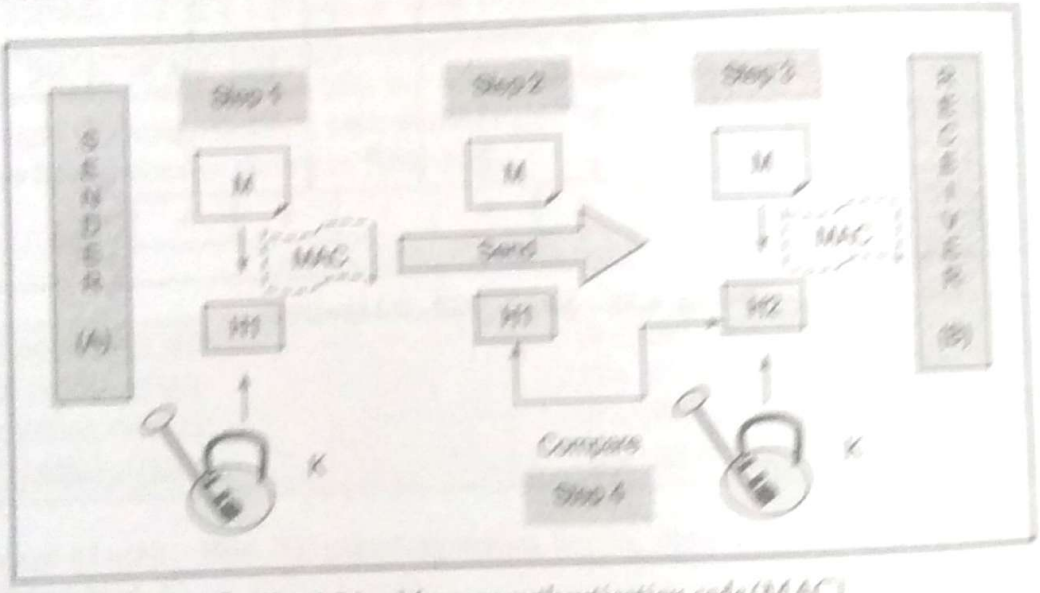

Fig. 4.34 Message authentication code (MAC)

1. A and B share a symmetric (secret) key K, which is not known to anyone else. A calculates the MAC by applying key K to the message M.
2. A then sends the original message M and the MAC H1 to B.
3. When B receives the message, B also uses K to calculate its own MAC H2 over M.
4. B now compares H1 with H2. If the two match, B concludes that the message M has not been changed during transit. However, if H1 ≠ H2, B rejects the message, realizing that the message was changed during transit.

The significances of a MAC are as follows:

1. The MAC assures the receiver (in this case, B) that the message is not altered. This is because if an attacker alters the message but does not alter the MAC (in this case, H1), then the receiver's calculation of the MAC (in this case, H2) will differ from it. Why does the attacker then not alter the MAC? Well, as we know, the key used in the calculation of the MAC (in this case, K)

assumed to be known only to the sender and the receiver (in this case, A and B). Therefore, the attacker does not know the key, K and therefore, she cannot alter the MAC.

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.

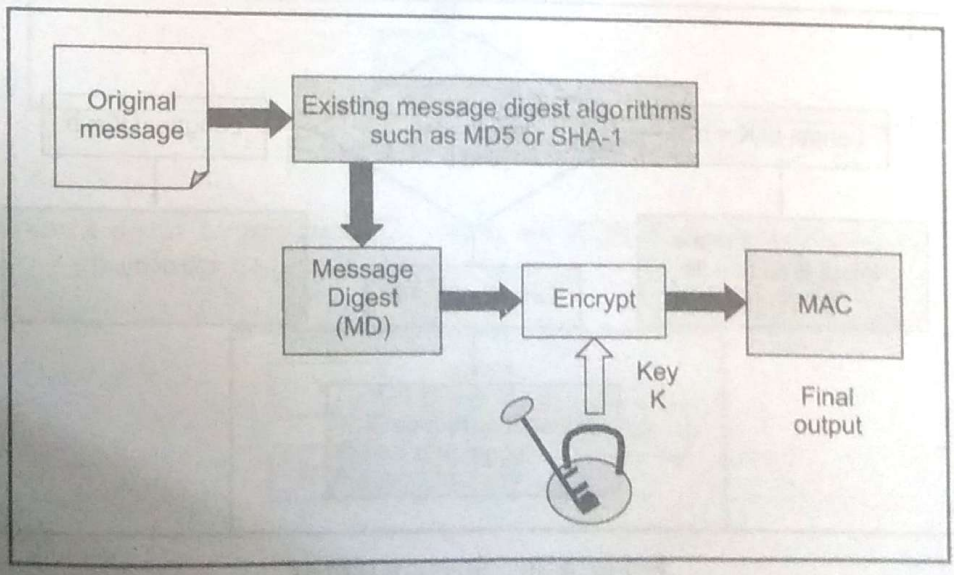

Fig. 4.35 *HMAC concept*

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.

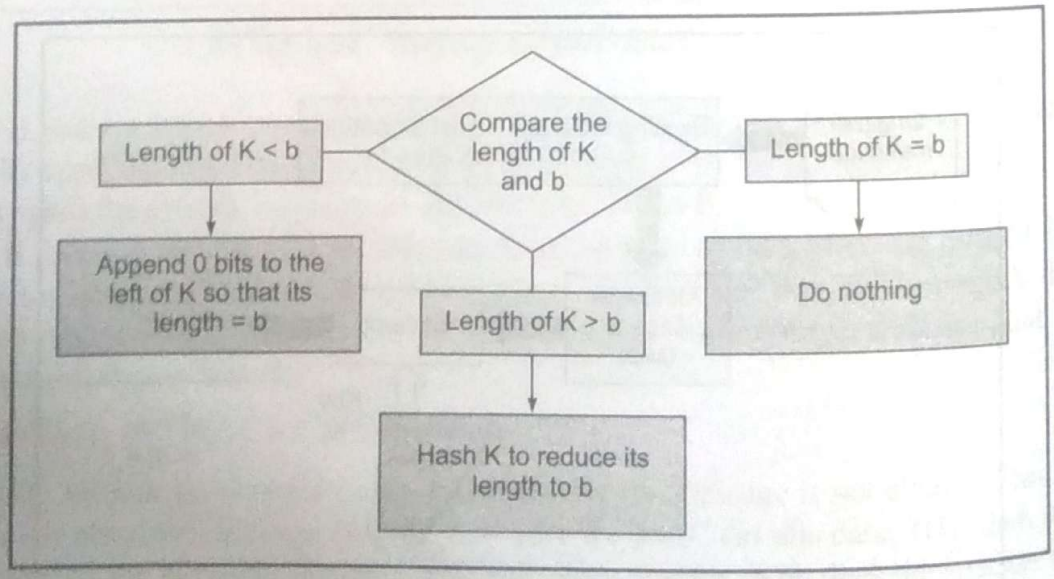

Fig. 4.36 *Step 1 of HMAC*

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.

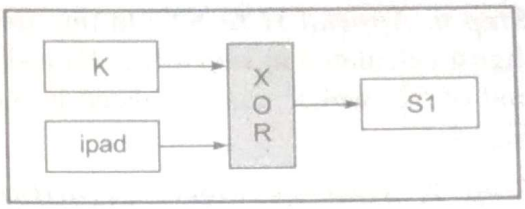

Fig. 4.37 *Step 2 of HMAC*

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.

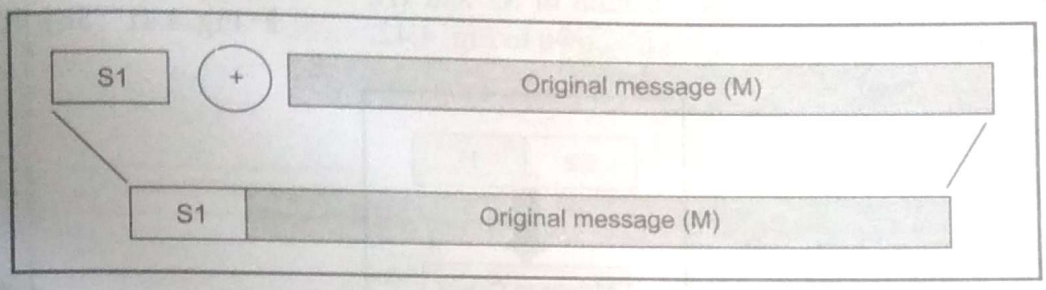

Fig. 4.38 *Step 3 of HMAC*

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.

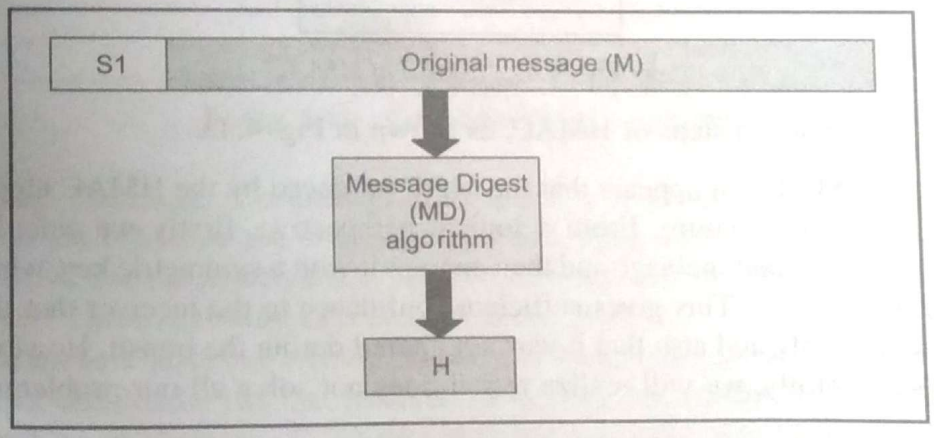

Fig. 4.39 *Step 4 of HMAC*

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.

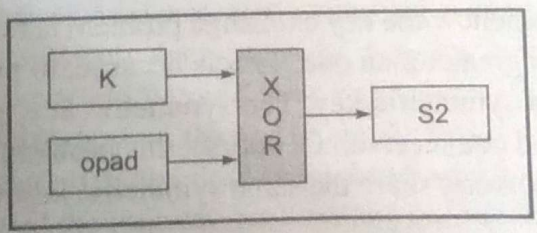

Fig. 4.40 *Step 5 of HMAC*

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.

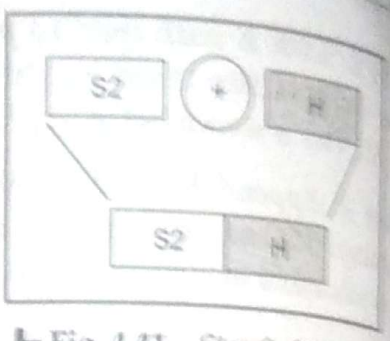

Fig. 4.41 *Step 6 of HMAC*

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.

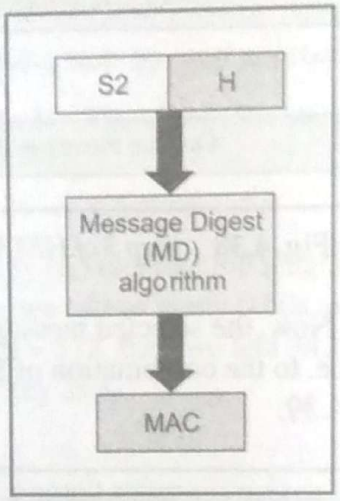

Fig. 4.42 *Step 7 of HMAC*

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

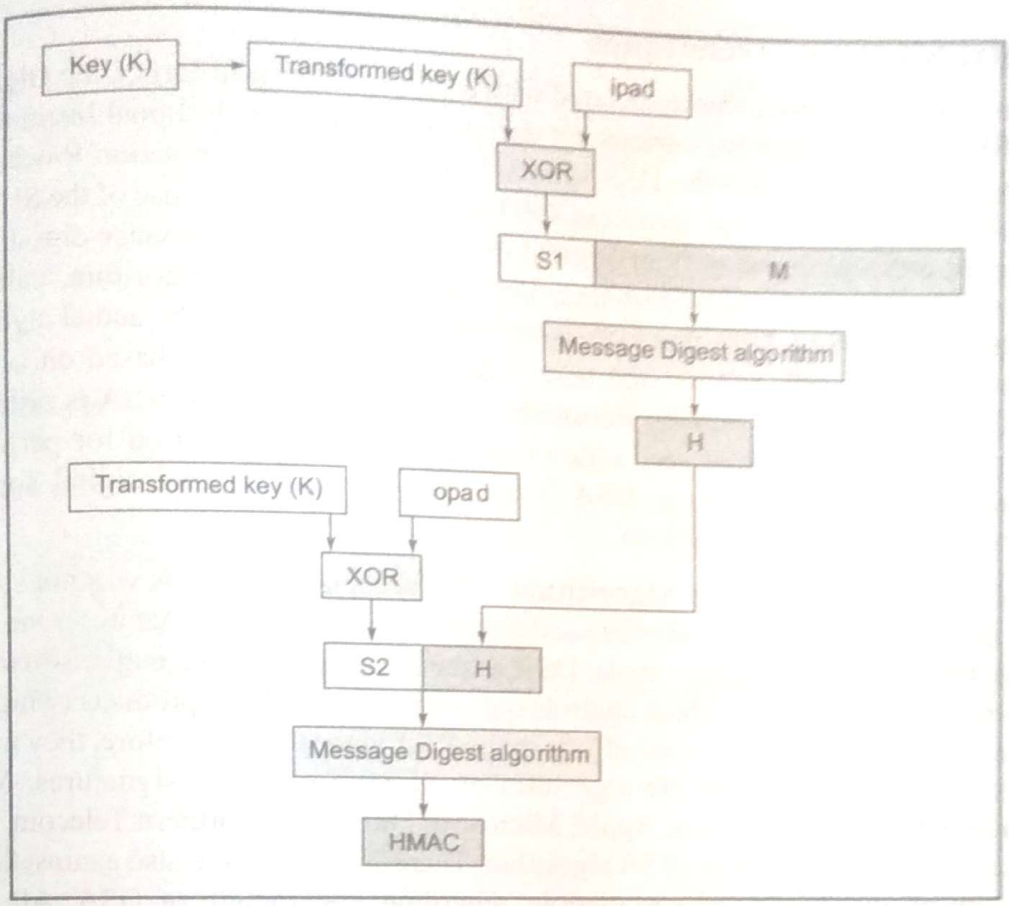

Fig. 4.43 *Complete HMAC operation*

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.