Unit 2
AES

AES- Advanced Encryption Standards

- The Advanced Encryption Standard (AES) was published by the National Institute of Standards and Technology (NIST) in 2001.
- AES is a block cipher intended to replace DES for commercial applications.
- It uses a 128-bit block size and a key size of 128, 192, or 256 bits.
- AES does not use a **Feistel structure**. Instead, each full round consists of four separate functions: byte substitution, permutation, arithmetic operations over a finite field, and XOR with a key.
- Compared to public-key ciphers such as RSA and most symmetric ciphers, the structure of AES and is quite complex

AES- Advanced Encryption Standards

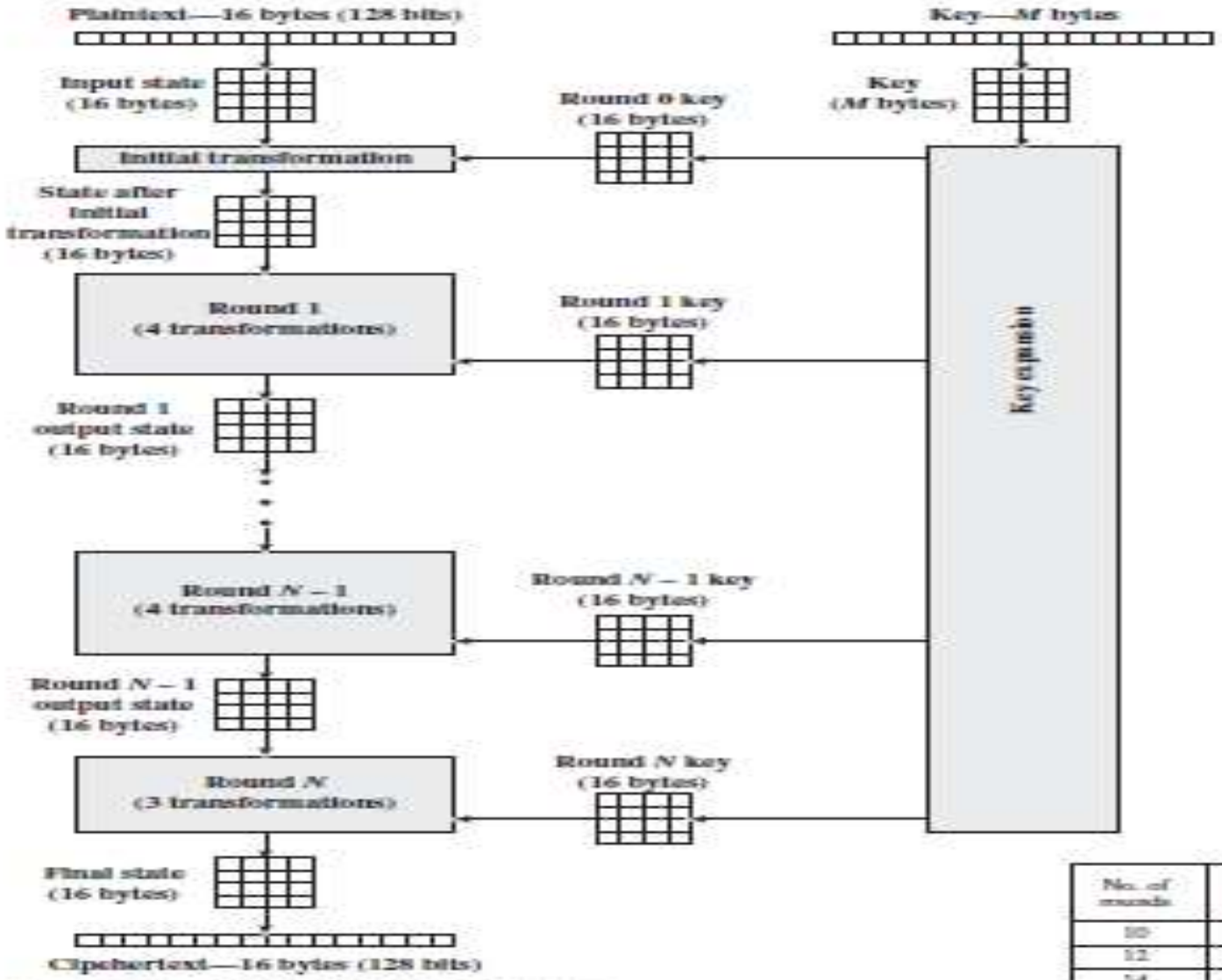
- The input to the encryption and decryption algorithms is a single 128-bit block.
- This block is depicted as a square matrix of bytes –**Input matrix.**
- This block is copied into the **State** array, which is modified at each stage of encryption or decryption.
- After the final stage, **State** is copied to an **output matrix**. These operations are depicted in Figure 5.2a.
- Similarly, the key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words. Figure 5.2b shows the expansion for the 128-bit key.
- Each word is four bytes, and the total key schedule is 44 words for the 128-bit key. Note that the **ordering of bytes within a matrix is by column.**
- So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the **in** matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the **w** matrix.

AES- Advanced Encryption Standards

- The cipher consists of rounds, where the number of rounds depends on the key length: 10 rounds for a 16-byte key, 12 rounds for a 24-byte key, and 14 rounds for a 32-byte key (Table 5.1).
- The first rounds consist of four distinct transformation functions: **SubBytes, ShiftRows, MixColumns, and AddRoundKey**. The **final round** contains only three transformations, namely - **SubBytes, ShiftRows, and AddRoundKey**
- There is a initial single transformation (AddRoundKey) before the first round,ie before Round 1.
- Each transformation takes one or more 4*4 matrices as input and produces a 4*4 matrix as output. Figure 5.1 shows that the output of each round is a matrix, with the output of the final round being the ciphertext.
- Also, the key expansion function generates round keys, each of which is a distinct matrix.
- Each round key serve as one of the inputs to the AddRoundKey transformation in each round.

AES Structure

General Structure



No. of rounds	Key Length (bytes)
10	16
12	24
14	32

Figure 5.1 AES Encryption Process

AES Structure

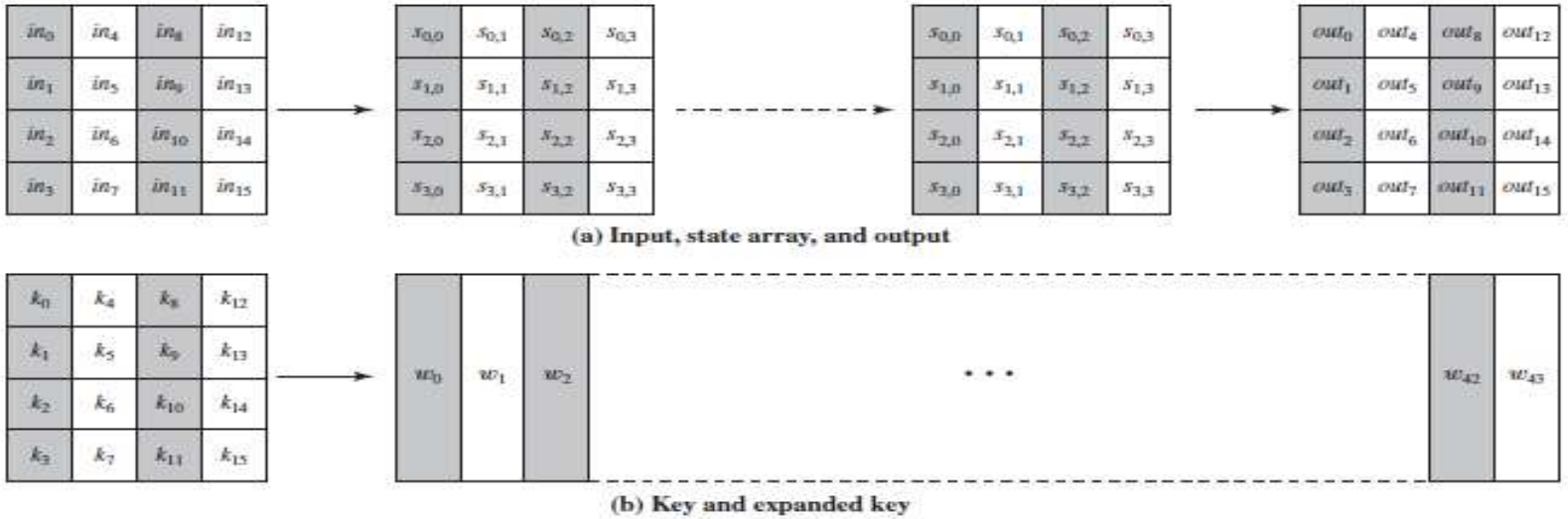


Figure 5.2 AES Data Structures

Table 5.1 AES Parameters

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

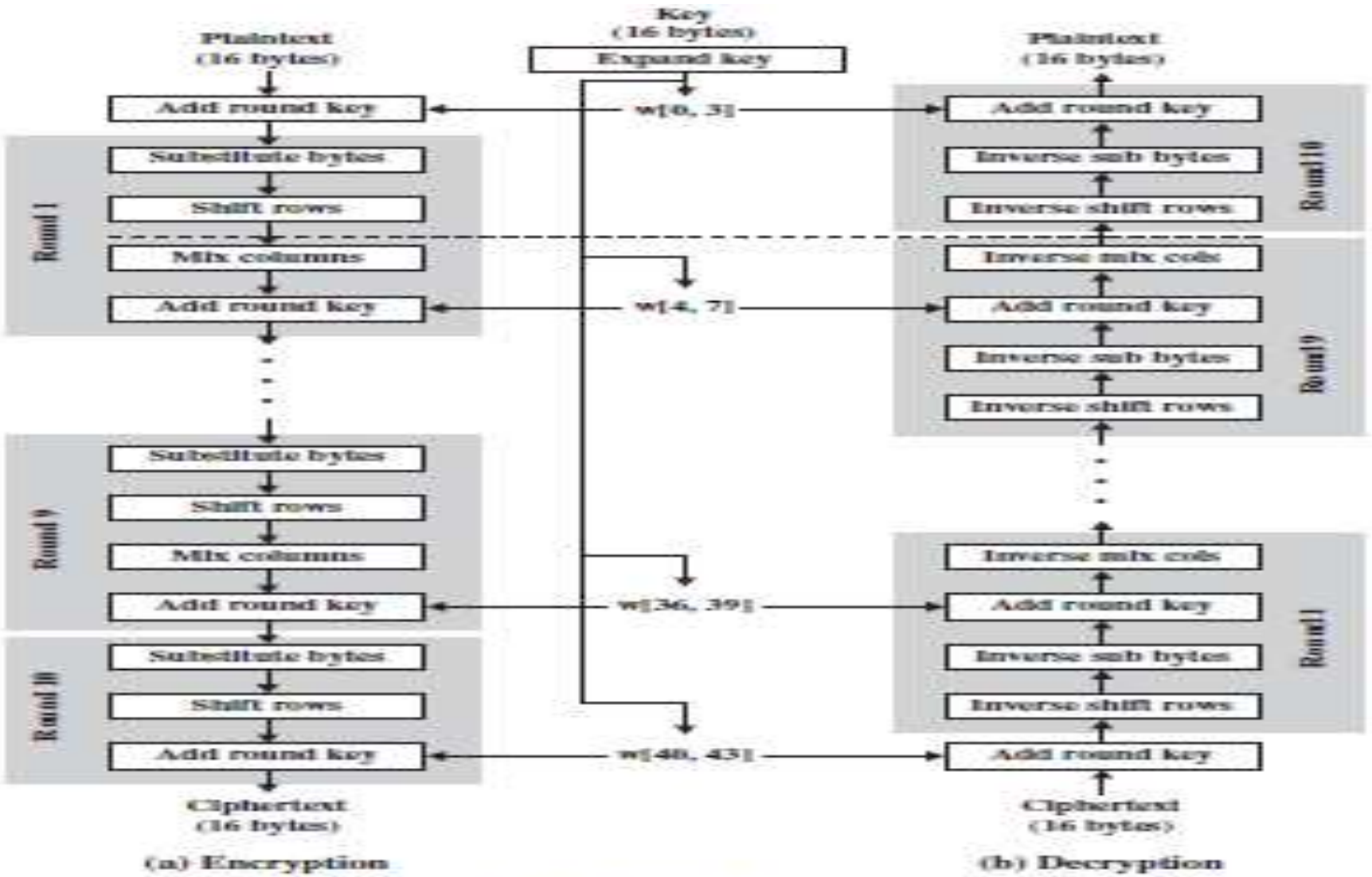
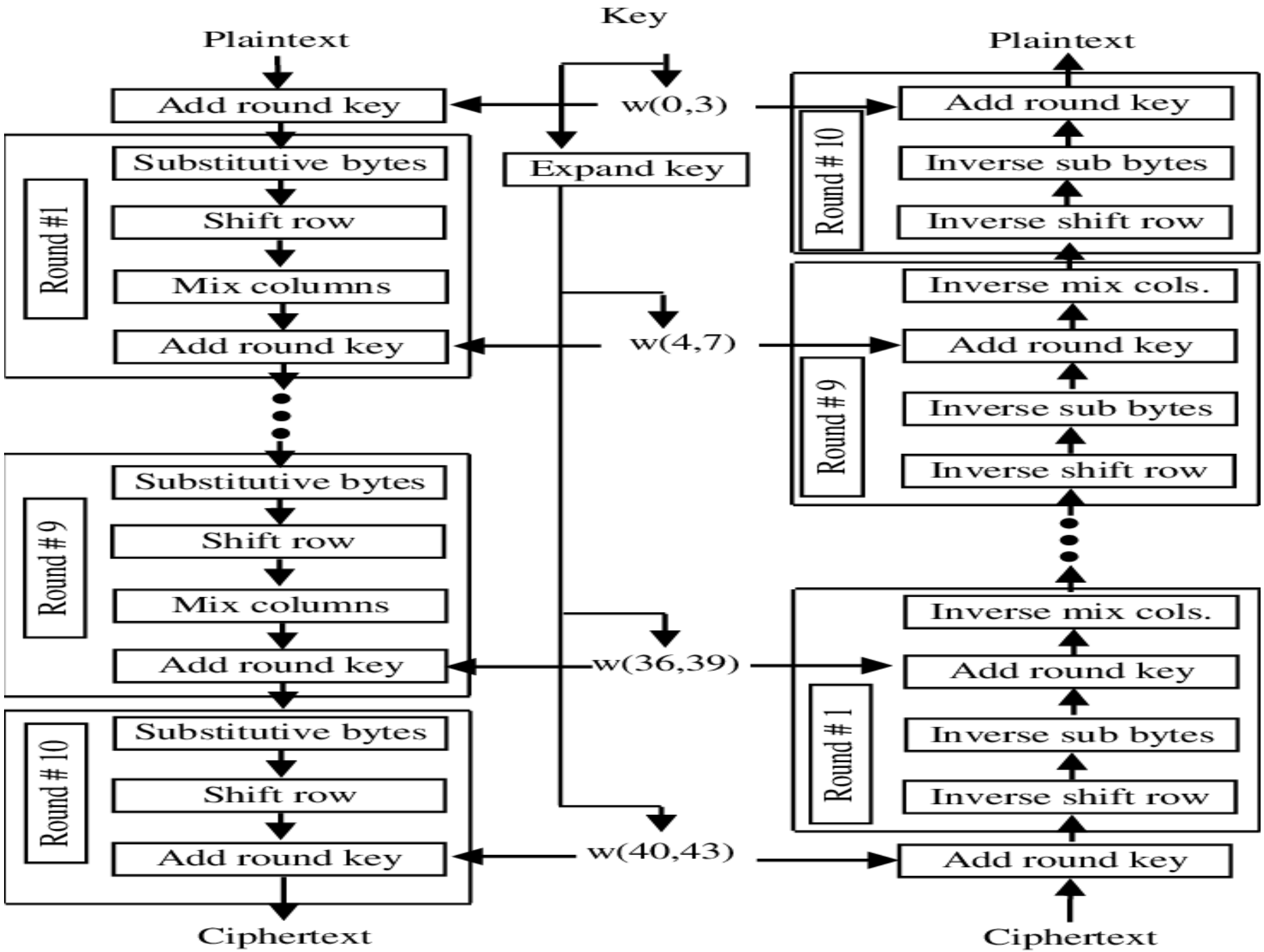


Figure 5.3 AES Encryption and Decryption


Key
Plaintext
Add round key
w(0,3)
Expand key
Round #1
Substitutive bytes
Shift row
Mix columns
Add round key
w(4,7)
Round #9
Substitutive bytes
Shift row
Mix columns
Add round key
w(36,39)
Round #10
Substitutive bytes
Shift row
Add round key
w(40,43)
Ciphertext
Plaintext
Round #10
Add round key
Inverse sub bytes
Inverse shift row
Round #9
Inverse mix cols.
Add round key
Inverse sub bytes
Inverse shift row
Round #1
Inverse mix cols.
Add round key
Inverse sub bytes
Inverse shift row
Add round key
Ciphertext

AES Encryption Round

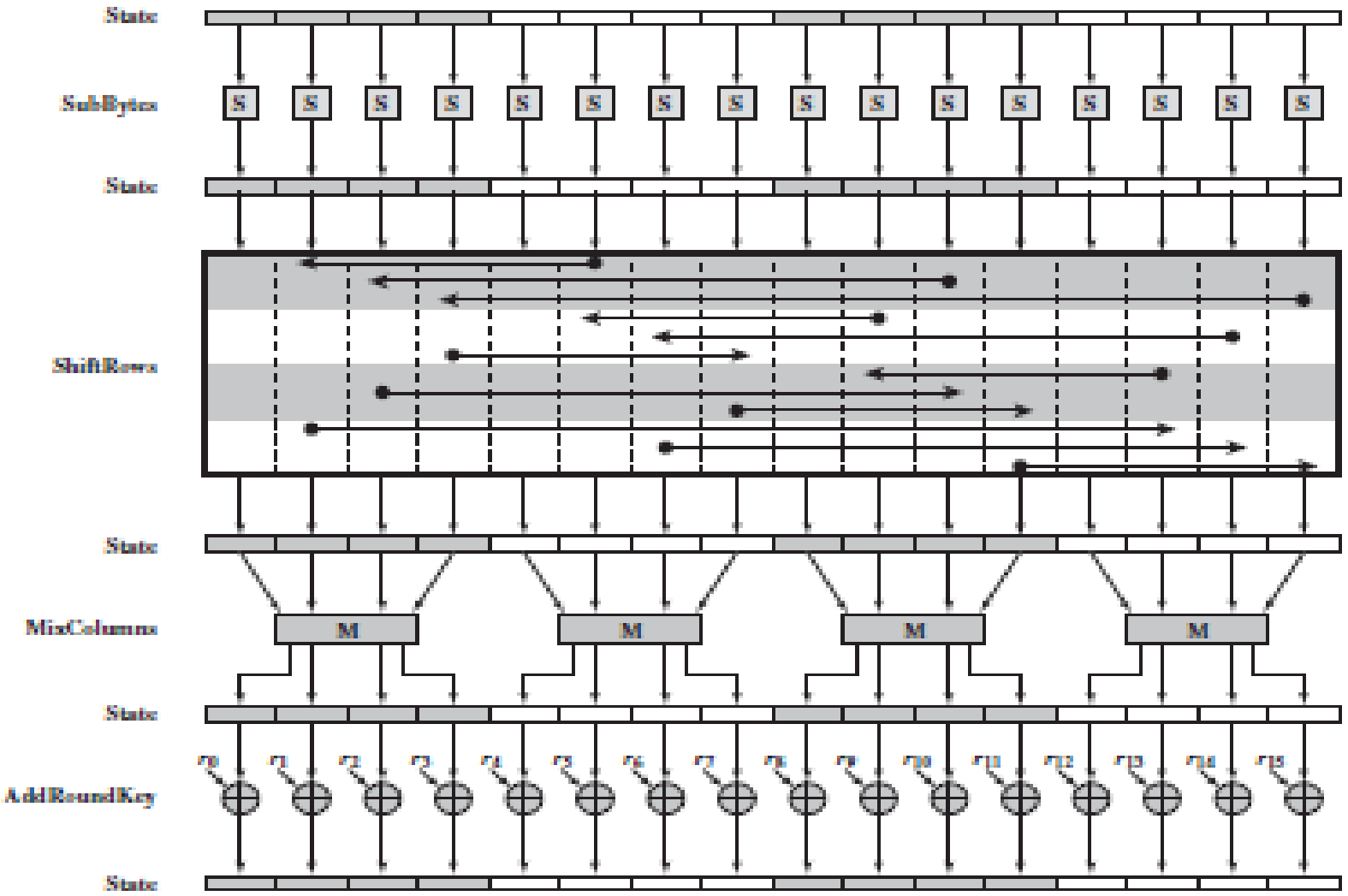


Figure 5.4 AES Encryption Round

AES Transformation Function

Forward and Inverse Transformation:

- The **forward substitute byte transformation**, called SubBytes, is a simple table lookup (Figure 5.5a).

- AES defines a matrix of byte values, called an S-box (Table 5.2a), that contains a permutation of all possible 256 8-bit values.

- Each individual byte of **State** is mapped into a new byte in the following way:

  * The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value.
  *These row and column values serve as indexes into the S-box to select a unique 8-bit output value.
  * For example, the hexadecimal value {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value is mapped into the value .

Here is an example of the SubBytes transformation:

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

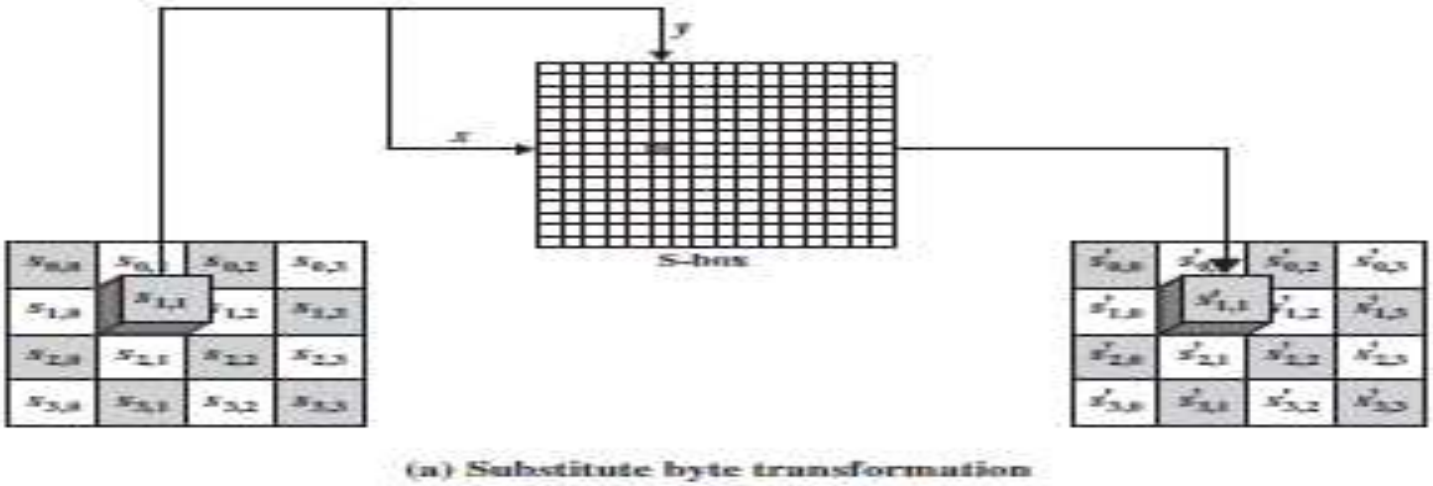
→

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

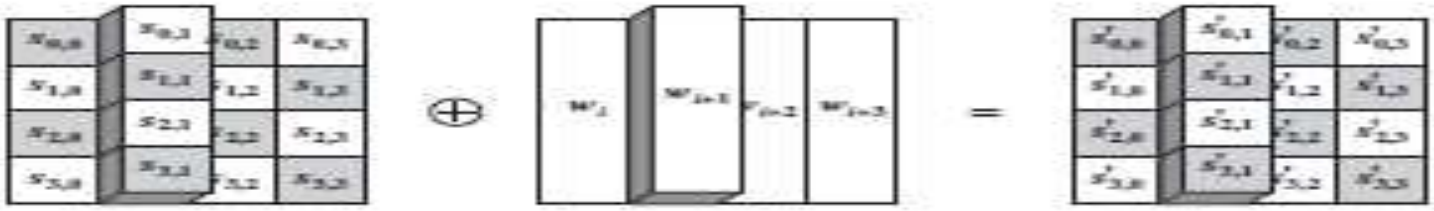
The S-box is constructed in the following fashion (Figure 5.6a).

AES Transformation Function

Forward and Inverse Transformation:



(a) Substitute byte transformation



(b) Add round key transformation

Figure 5.5 AES Byte-Level Operations

AES Transformation Function

Forward and Inverse Transformation:

Table 5.2 AES S-Boxes

x \ 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

(a) S-box

AES Transformation Function

Inverse Substitute Transformation:

- Also called InvSubBytes,
- makes use of the inverse S-box shown in Table 5.2b. Note, for example, that the input is {2A} produces the output {95}

AES Transformation Function

Forward and Inverse Transformation:

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

(b) Inverse S-box

AES Transformation Function

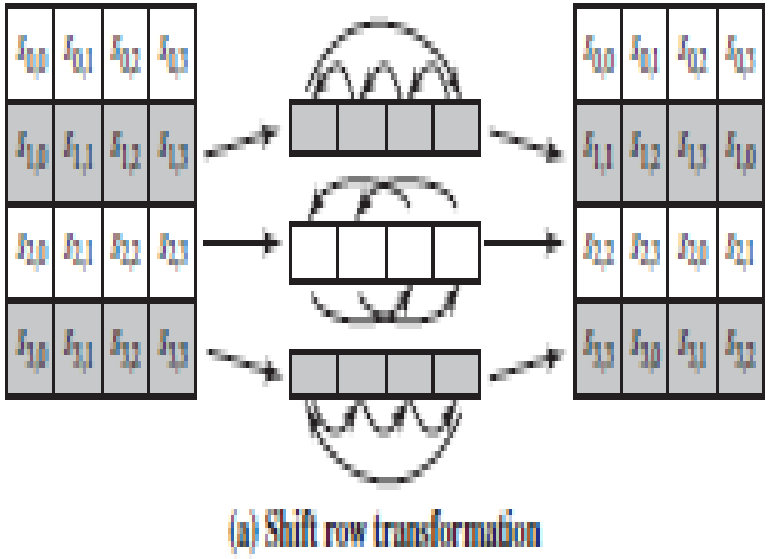
ShiftRow Transformation:

Forward Shift Row Tansformation:

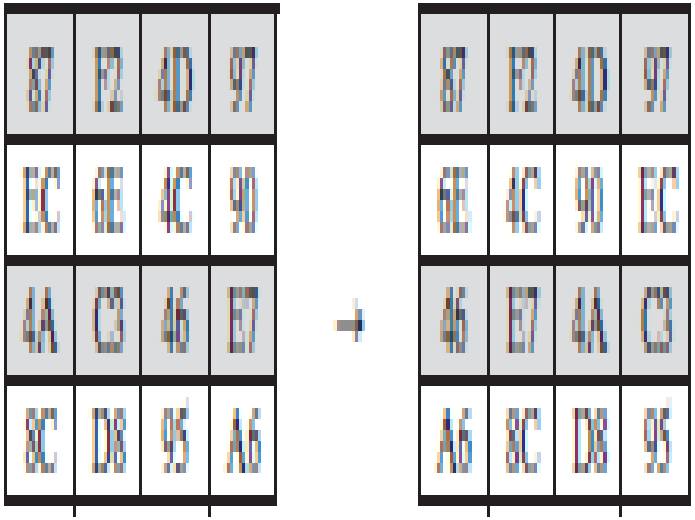
- Also called Shiftrows

- a circular byte shift in each row is as follows and is depicted in below figure:
  - * 1st row - unchanged
  - * 2nd row - 1 byte circular shift to left
  - * 3rd row - 2 byte circular shift to left
  - * 4th row - 3 byte circular shift to left

- The following is an example for shiftrows



(a) Shift row transformation



AES Transformation Function

Inverse Shift Row Transformation:

- Also called invshiftrows.
- performs the circular shifts in the opposite direction for each of the last three rows with :

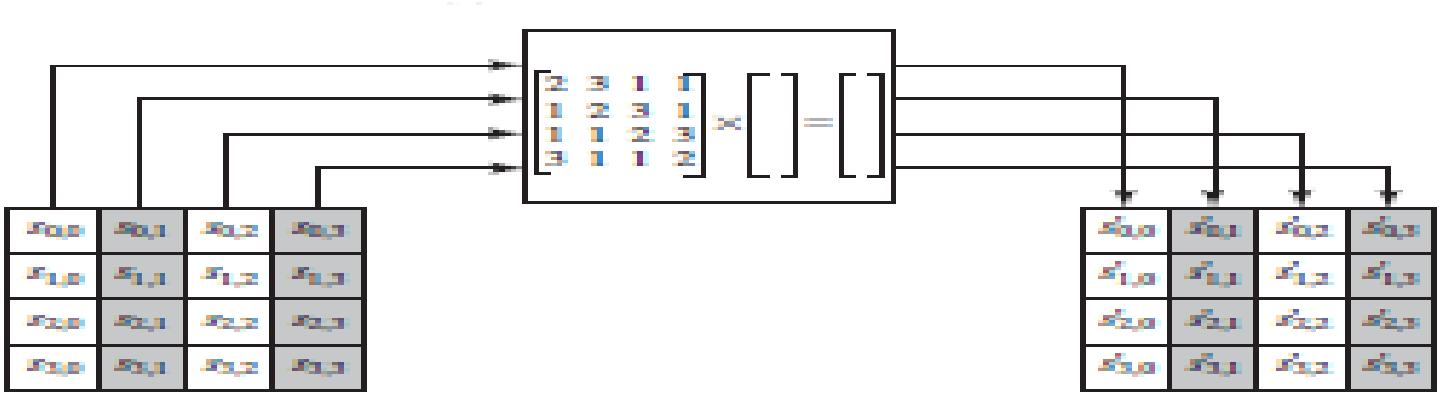
    * 1st row - unchanged
    * 2nd row - 1 byte circular shift to right
    * 3rd row - 2 byte circular shift to right
    * 4th row - 3 byte circular shift to right

AES Transformation Function

Mixcolumn Transformation:

Forward mix column transformation,
- Also called MixColumns,
- operates on each column individually.
- Each byte of a column is mapped into a new value that is a function of all four bytes in that column.
- The transformation can be defined by the following matrix multiplication on **State.**
- Each element in the product matrix is the sum of products of elements of one row and one column.

The MixColumns transformation on a single column of **State** can be expressed as



(b) Mix column transformation

Figure 5.7 AES Row and Column Operations

AES Transformation Function

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

The MixColumns transformation on a single column of **State** can be expressed as

$$s'_{0,j} = (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$
$$s'_{1,j} = s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j}$$
$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j})$$
$$s'_{3,j} = (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j})$$

- The following is an example for Mixcolumn

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

AES Transformation Function

Inverse Mixcolumn Transformation:

- Also called InvMixColumns defined by the following matrix multiplication.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

AES Transformation Function

AddRoundkey:

forward add round key transformation:

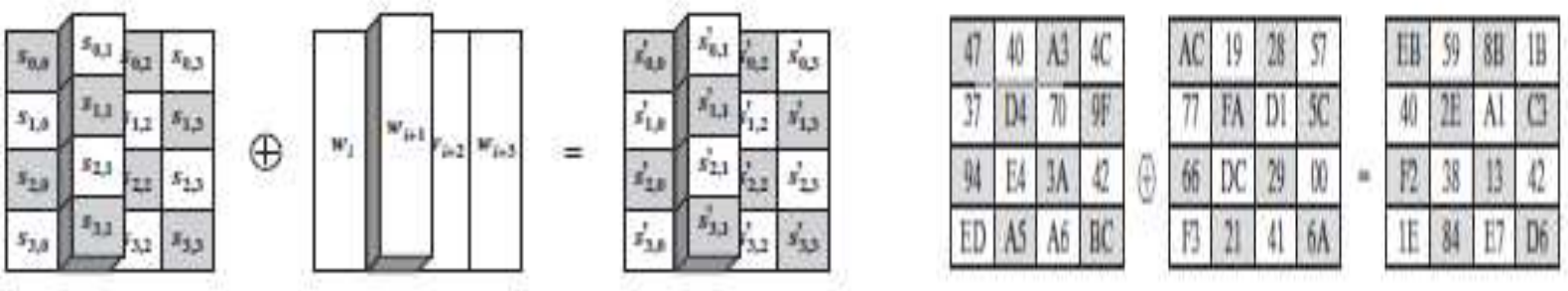
- Also called AddRoundKey,
- the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. As shown in Figure 5.5b,
- the operation is viewed as a columnwise operation between the 4 bytes of a **State**column and one word of the round key; it can also be viewed as a byte-level operation.
- The following is an example of AddRoundKey:

AES Transformation Function

AddRoundkey:

forward add round key transformation:

- Also called AddRoundKey,
- the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. As shown in Figure 5.5b,
- the operation is viewed as a columnwise operation between the 4 bytes of a **State**column and one word of the round key; it can also be viewed as a byte-level operation.
- The following is an example of AddRoundKey:



(b) Add round key transformation

AES Transformation Function

The **inverse add round key transformation** is identical to the forward add round key transformation, because the XOR operation is its own inverse.

AES Transformation Function

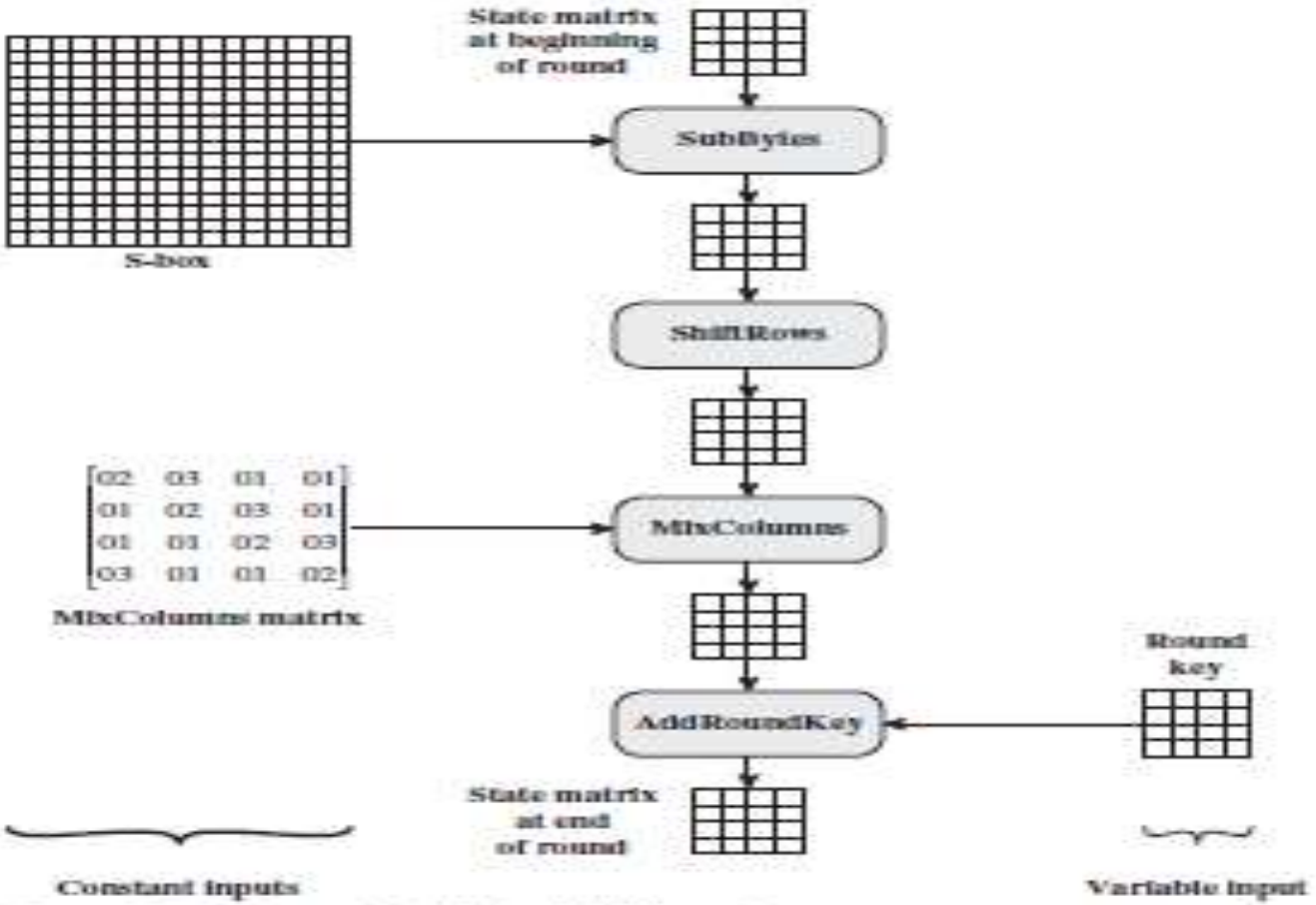
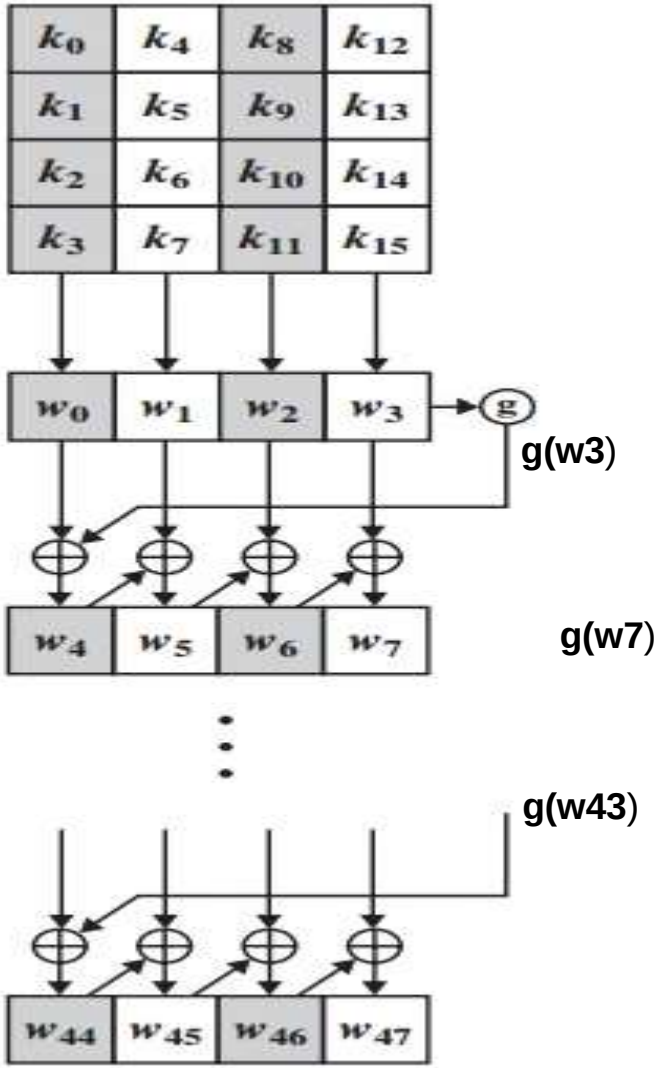
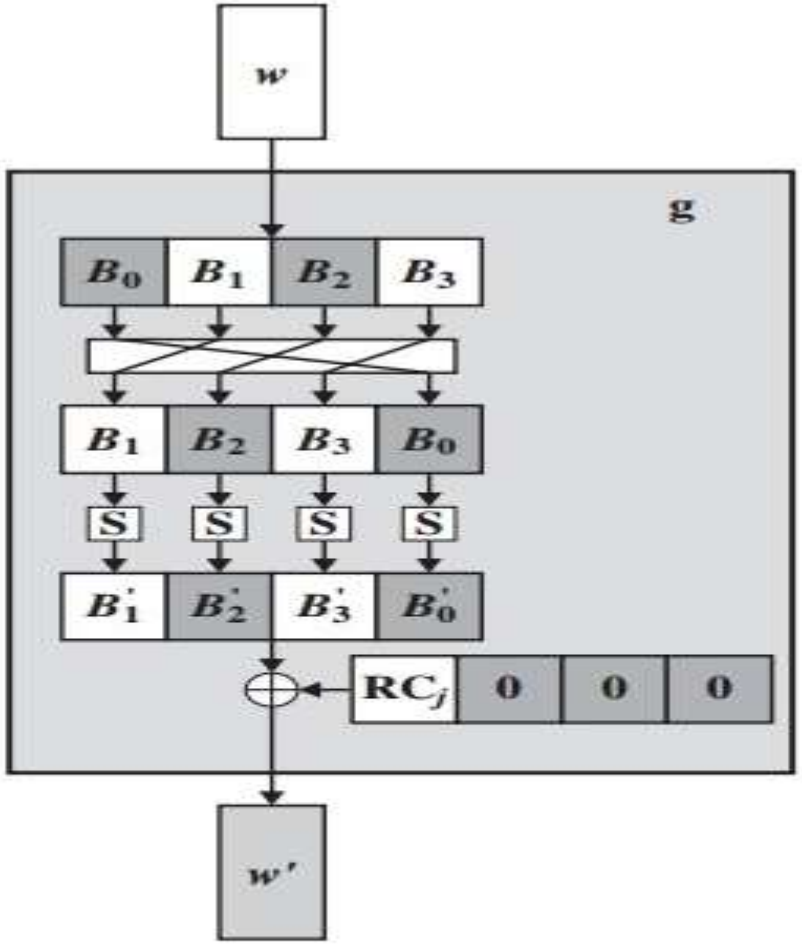


Figure 5.8 Inputs for Single AES Round

AES Key Expansion Algorithm



(a) Overall algorithm



(b) Function g

AES Key Expansion Algorithm

```
KeyExpansion (byte key[16], word w[44])
{
    word temp
    for (i = 0; i < 4; i++)    w[i] = (key[4*i], key[4*i+1],
                                       key[4*i+2],
                                       key[4*i+3]);

    for (i = 4; i < 44; i++)
    {
     temp = w[i - 1];
     if (i mod 4 = 0)    temp = SubWord (RotWord (temp))
                                ⊕ Rcon[i/4];
     w[i] = w[i-4] ⊕ temp
    }
}
```

AES Key Expansion Algorithm

- The AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes).This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher.

Steps:

- The key is copied into the first four words of the expanded key.
- The remainder of the expanded key is filled in four words at a time.
- Each added word depends on the immediately preceding word, , and

  the word four positions back .
- In three out of four cases, a simple XOR is used. For a word whose position in the **w** array is a multiple of 4, a more complex function is used.
- Figure 5.9 illustrates the generation of the expanded key, using the symbol **g** to represent that complex function.
- The function g consists of the following subfunctions are:

  1. RotWord performs a one-byte circular left shift on a word. This means that an input word [B3,B2,B1,B0] is transformed into [B2,B1,B0,B3] .

  2. SubWord performs a byte substitution on each byte of its input word, using the S-box (Table 5.2a).

  3. The result of steps 1 and 2 is XORed with a round constant(Rcon[i])

Rcon[i] : The round constant is a word in which the three rightmost bytes are always 0. Thus, the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word.

AES Key Expansion Algorithm

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

For example, suppose that the round key for round 8 is

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as follows:

i (decimal)	temp	After RotWord	After SubWord	Rcon (9)	After XOR with Rcon	w[i−4]	w[i] = temp ⊕ w[i−4]
36	7F8D292F	8D292F7F	5DA515D2	1B000000	46A515D2	EAD27321	AC7766F3

Avalanche Effect

- Small Change in plaintext or key or state matrix – significant changes in further rounds status and ciphertext.
- State matrix change affects significant change in a round, and the magnitude of change after all subsequent rounds is roughly half the bits. Thus, based on this example, AES exhibits a very strong avalanche effect. (Refer to table 5.5 and 5.6 in the text book)
- Note that this **avalanche effect is stronger than that for DES** (Table 3.5), which requires three rounds to reach a point at which approximately half the bits are changed, both for a bit change in the plaintext and a bit change in the key.

AES Example & Implementation

- **Refer to text book and notes(problems solved in class)**

Unit-3

ASYMMETRIC CIPHERS/ PUBLIC-KEY CRYPTOGRAPHY

Public Key Cryptography

Table 9.1 Terminology Related to Asymmetric Encryption

Asymmetric Keys Two related keys, a public key and a private key, that are used to perform complementary operations, such as encryption and decryption or signature generation and signature verification.
Public Key Certificate A digital document issued and digitally signed by the private key of a Certification Authority that binds the name of a subscriber to a public key. The certificate indicates that the subscriber identified in the certificate has sole control and access to the corresponding private key.
Public Key (Asymmetric) Cryptographic Algorithm A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys have the property that deriving the private key from the public key is computationally infeasible.
Public Key Infrastructure (PKI) A set of policies, processes, server platforms, software and workstations used for the purpose of administering certificates and public-private key pairs, including the ability to issue, maintain, and revoke public key certificates.

Source: *Glossary of Key Information Security Terms*, NIST IR 7298 [KISS06]

Public key cryptography

- The concept of public-key cryptography evolved from an attempt to attack two of the most difficult problems associated with symmetric encryption.

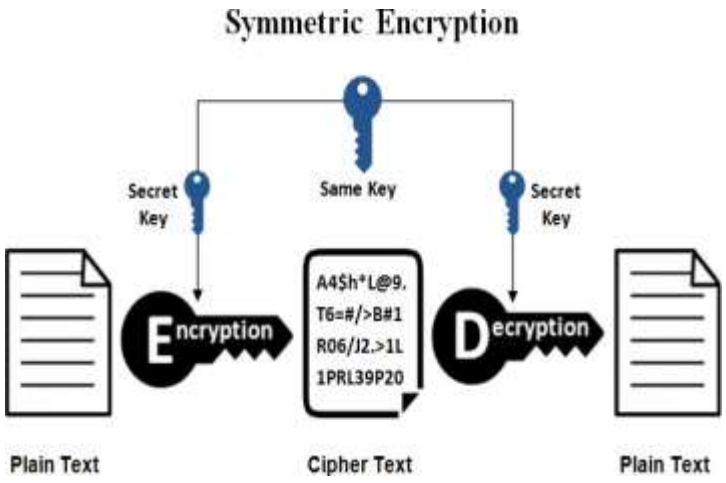
(1)The first problem is that of key distribution. Two communicants already share a key, which somehow has been distributed to them; or

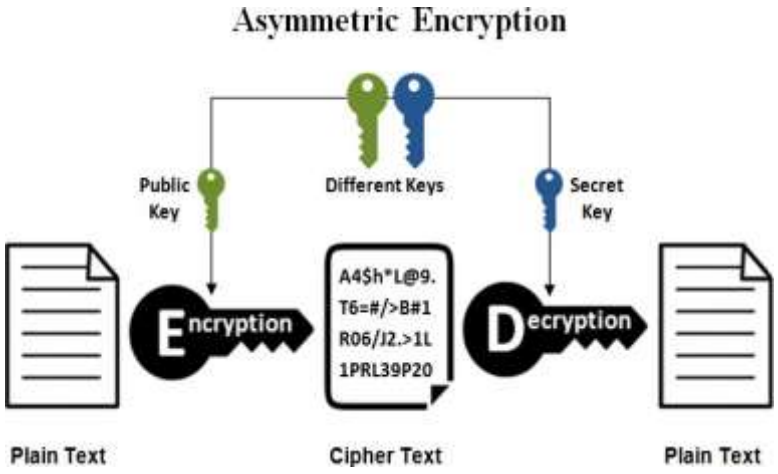
(2) The use of a key distribution center.

Public key cryptography

- Asymmetric encryption is a form of cryptosystem in which encryption and decryption are performed using the different keys—one a public key and one a private key. It is also known as public-key encryption.

- EX: RSA, Diffie Hellman





Public key cryptography

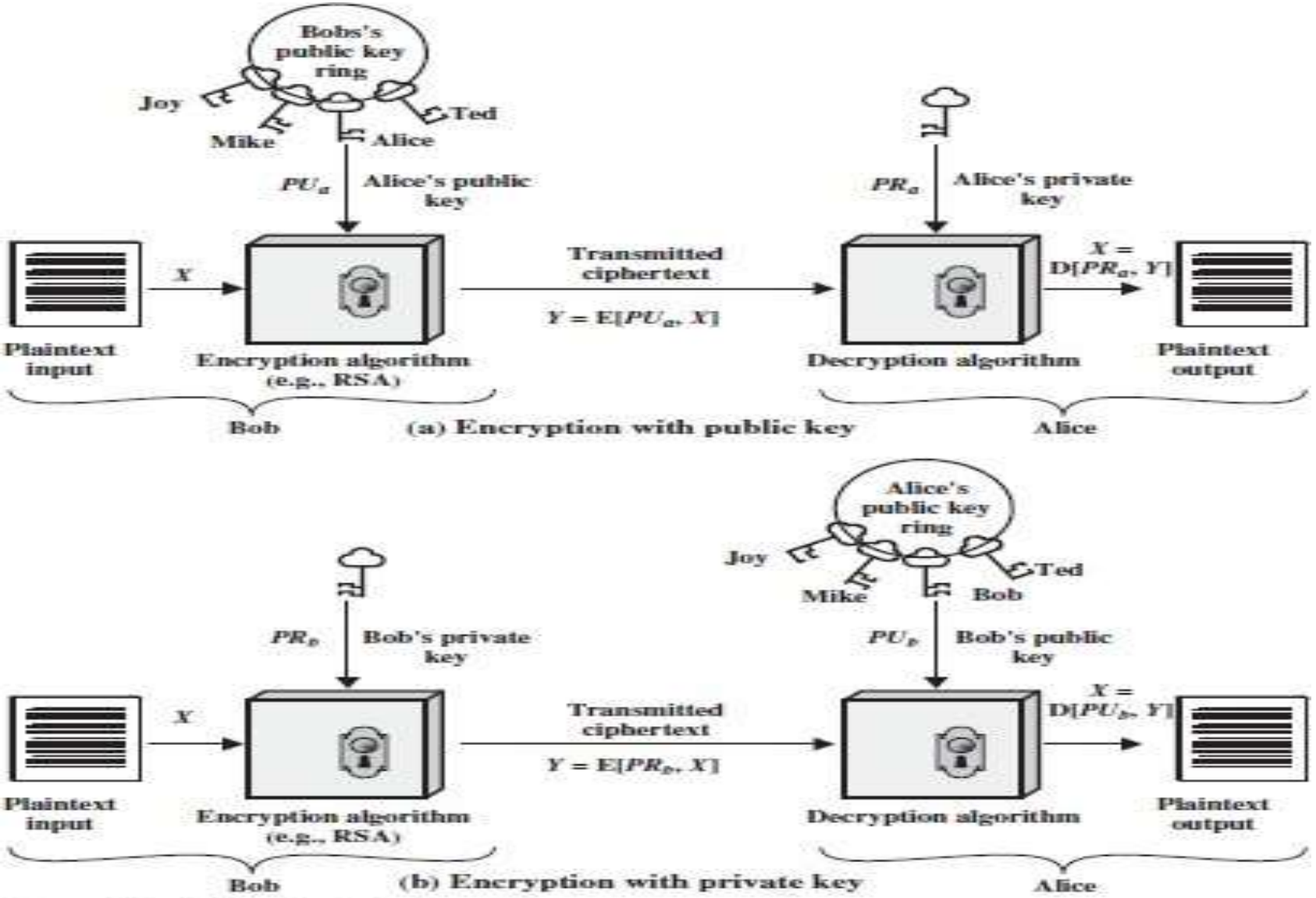


Figure 9.1 Public-Key Cryptography

A public-key encryption scheme ingredients

- **Plaintext:** This is the readable message or data that is fed into the algorithm as input.
- **Encryption algorithm:** The encryption algorithm performs various transformations on the plaintext.
- **Public and private keys:** This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.
- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
- **Decryption algorithm:** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

The essential steps are the following

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.

2.Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. Each user maintains a collection of public keys obtained from others.

3. If **Bob** wishes to send a confidential message to **Alice**, Bob encrypts the message using Alice's **public key**.

4.When Alice receives the message, she decrypts it using **her private key**. No other recipient can decrypt the message because only Alice knows Alice's private key.

Comparison

Characteristic	Symmetric key cryptography	Asymmetric key cryptography
Key used for encryption/decryption	Same key is used	One key is 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 plain text size.	More than the original plain text size
Known keys	Both parties should know the key in symmetric key encryption	Only, either one of the keys is known by the two parties in public key encryption.
Usage	Confidentiality	Confidentiality, digital signature etc.
Key agreement / exchange	A big problem	No problem at all

Examples

- RC4
- AES
- DES
- 3DES
- QUAD

- RSA
- Diffie-Hellman
- ECC
- El Gamal
- DSA

Comparison

Table 9.2 Conventional and Public-Key Encryption

Conventional Encryption	Public-Key Encryption
Needed to Work: 1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key. *Needed for Security:* 1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	*Needed to Work:* 1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one). *Needed for Security:* 1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Applications for Public-Key Cryptosystems

- **Encryption /decryption:** The sender with the encrypts a message recipient's public key.

- **Digital signature:** The sender "signs" a message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.

- **Key exchange:** Two sides cooperate to exchange a session key. Several different approaches are possible, involving the private key(s) of one or both parties.

Table 9.3 Applications for Public-Key Cryptosystems

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Requirements for Public-Key Cryptography

1. It is computationally easy for a party B to generate a pair (public key PU_b, private key PR_b).
2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext:

$$C = \mathrm{E}(PU_b, M)$$

3. It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message:

$$M = \mathrm{D}(PR_b, C) = \mathrm{D}[PR_b, \mathrm{E}(PU_b, M)]$$

4. It is computationally infeasible for an adversary, knowing the public key, PU_b, to determine the private key, PR_b.
5. It is computationally infeasible for an adversary, knowing the public key, PU_b, and a ciphertext, C, to recover the original message, M.

We can add a sixth requirement that, although useful, is not necessary for all public-key applications:

6. The two keys can be applied in either order:

$$M = \mathrm{D}[PU_b, \mathrm{E}(PR_b, M)] = \mathrm{D}[PR_b, \mathrm{E}(PU_b, M)]$$

THE RSA ALGORITHM

Step 1 : Generate two large random primes, *p* and *q.*

Step 2: Compute n = pq

Step 3 : Compute (phi) φ(n) = (p-1)(q-1).

Step 4: Choose an integer *e*, 1 < e < phi,
such that gcd(e, phi) = 1.

Step 5: Compute the secret exponent *d*, 1 < d < phi, such that

ed ≡ 1 (mod phi).

Step 6: The public key is (**e,** n) and the
private key (**d**,n) or private key (d, p, q).

Keep all the values d, p, q and phi secret.

- **Encryption:**

$$c = m^e \bmod n$$

- **Decryption:**

$$m = c^d \bmod n$$

RSA Example

p

12131072439211271897323671531612440428472427633701410925634549312301964
37304208561932419736532241686654101705736136521417171171379797429933487
1062829803541

q

12027524255478748885956220793734512128733387803682075433653899983955179
85098879789986914690080913161115334681705083209602216014636634639181247
0987105415233

With these two large numbers, we can calculate n and $\phi(n)$

n

14590676800758332323018693934907063529240187237535716439958187101987343
87990053589383695714026701498021218180862924674228281570229220767469065
43401224889672472407926969987100581290103199317858753663710862357656510
50788371429711563734278891146353510271203276516651841172685983798867211
1837205085526346618740053

$\phi(n)$

14590676800758332323018693934907063529240187237535716439958187101987343
87990053589383695714026701498021218180862924674228281570229220767469065
43401224889648313811232279966317301397777852365301547848273478871297222
05858745715289160645926971811926897116355507080264399952954964411681194
75165139381842966835212 80

e - the public key

65537 has a gcd of 1 with $\phi(n)$, so lets use it as the public key. To calculate the private key, use extended euclidean algorithm to find the multiplicative inverse with respect to $\phi(n)$.

d - the private key

89489425009274444368228545921773093919669586065884257445497854456487674
83962981839093494197326287961679797060891728367987549933157416111385408
88132754881105882471930775825272784379065040156806234235500672400424666
65654232383502922215493623289472138866445818789127946123407807725702626
6440910365023725451397 13

Encryption: $1976620216402300889624482718775150^{e} \bmod n$

```
350521113386730266902124239370533285118807608115799816206428023466858106231098502359430490809733862411137840407947041939782153784997654130836464387847409523069325349451950801838615742252262188798272324539128205968864403775360824656817500744174591514854074458625110234722355608230534977915189288202722577877 86
```

Decryption:

```
350521113386730266902124239370533285118807608115799816206428023466858106231098502359430490809733862411137840407947041939782153784997654130836464387847409523069325349451950801838615742252262188798272324539128205968864403775360824656817500744174591514854074458625110234722355608230534977915189288202722577877 86^{d} \bmod n
```

1976620216402300889624482718775150 (which is our plaintext "attack at dawn")

Security of RSA

There are Five possible approaches to attacking the RSA algorithm are :

- **Brute force:** This involves trying all possible private keys.
- **Mathematical attacks:** There are several approaches, all equivalent in effort to factoring the product of two primes.
- **Timing attacks:** These depend on the running time of the decryption algorithm.
- **Chosen ciphertext attacks:** This type of attack exploits properties of the RSA algorithm.
- **Hardware Fault Based Attack:** This involves inducing hardware faults in the processor that is generating digital signatures.

Security of RSA

THE FACTORING PROBLEM We can identify three approaches to attacking RSA mathematically.

1. Factor n into its two prime factors. This enables calculation of $\phi(n) = (p-1) \times (q-1)$, which in turn enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
2. Determine $\phi(n)$ directly, without first determining p and q. Again, this enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
3. Determine d directly, without first determining $\phi(n)$.

Table 9.5 Progress in Factorization

Number of Decimal Digits	Approximate Number of Bits	Date Achieved	MIPS-Years	Algorithm
100	332	April 1991	7	Quadratic sieve
110	365	April 1992	75	Quadratic sieve
120	398	June 1993	830	Quadratic sieve
129	428	April 1994	5000	Quadratic sieve
130	431	April 1996	1000	Generalized number field sieve
140	465	February 1999	2000	Generalized number field sieve
155	512	August 1999	8000	Generalized number field sieve
160	530	April 2003	—	Lattice sieve
174	576	December 2003	—	Lattice sieve
200	663	May 2005	—	Lattice sieve

Security of RSA

In addition to specifying the size of n, a number of other constraints have been suggested by researchers. To avoid values of n that may be factored more easily, the algorithm's inventors suggest the following constraints on p and q.

1. p and q should differ in length by only a few digits. Thus, for a 1024-bit key (309 decimal digits), both p and q should be on the order of magnitude of 10^{75} to 10^{100}.
2. Both $(p-1)$ and $(q-1)$ should contain a large prime factor.
3. $\gcd(p-1, q-1)$ should be small.

Security of RSA

Timing Attack :

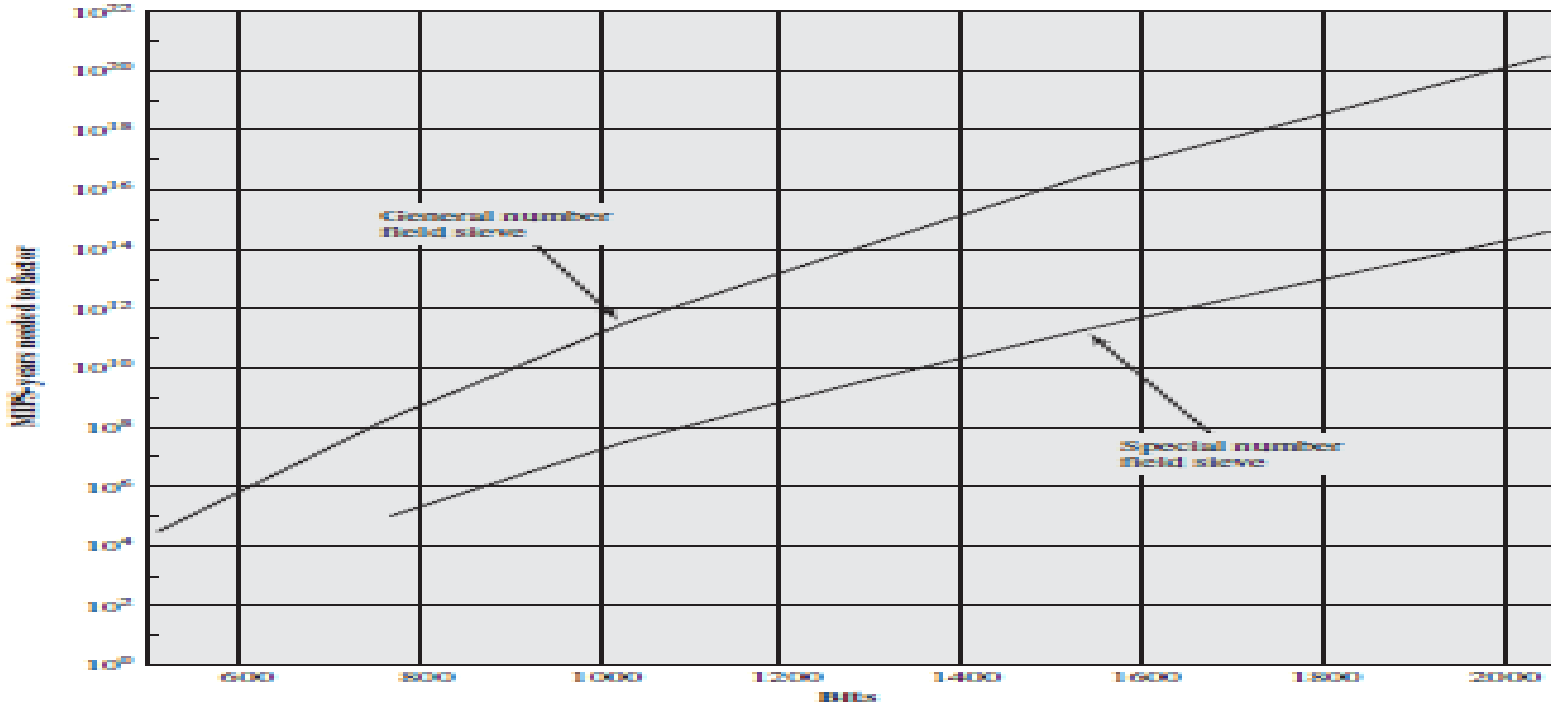


Figure 9.9 MIPS-years Needed to Factor

Security of RSA

Timing Attack :

Although the timing attack is a serious threat, there are simple countermeasures that can be used, including the following.

- **Constant exponentiation time:** Ensure that all exponentiations take the same amount of time before returning a result. This is a simple fix but does degrade performance.
- **Random delay:** Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack. Kocher points out that if defenders don't add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.
- **Blinding:** Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.

RSA Data Security incorporates a blinding feature into some of its products. The private-key operation $M = C^d \bmod n$ is implemented as follows.

1. Generate a secret random number r between 0 and $n - 1$.
2. Compute $C' = C(r^e) \bmod n$, where e is the public exponent.
3. Compute $M' = (C')^d \bmod n$ with the ordinary RSA implementation.
4. Compute $M = M'r^{-1} \bmod n$. In this equation, r^{-1} is the multiplicative inverse of $r \bmod n$; see Chapter 4 for a discussion of this concept. It can be demonstrated that this is the correct result by observing that $r^{ed} \bmod n = r \bmod n$.

RSA Data Security reports a 2 to 10% performance penalty for blinding.

Security of RSA

Chosen Ciphertext Attack(CCA) & Optimal Asymetric Encryption Padding :

A simple example of a CCA against RSA takes advantage of the following property of RSA:

$$\mathrm{E}(PU, M_1) \times \mathrm{E}(PU, M_2) = \mathrm{E}(PU, [M_1 \times M_2]) \quad (9.2)$$

We can decrypt $C = M^e \bmod n$ using a CCA as follows.

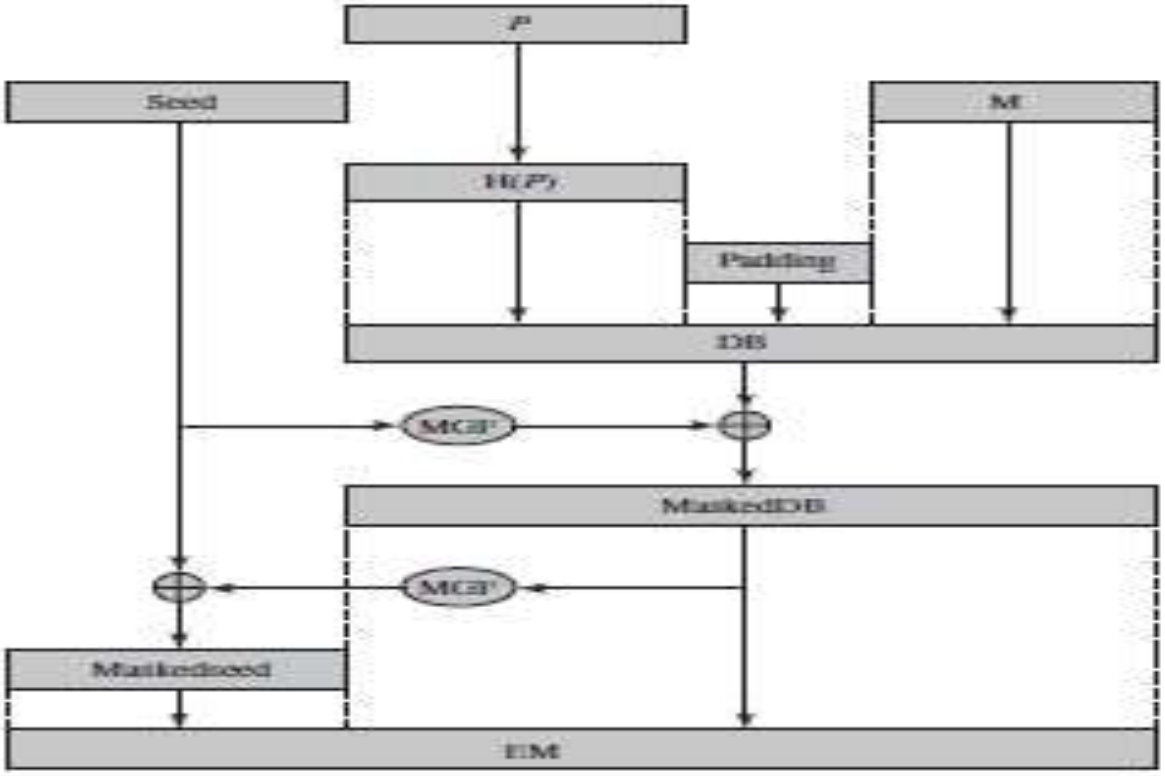
1. Compute $X = (C \times 2^e) \bmod n$.
2. Submit X as a chosen ciphertext and receive back $Y = X^d \bmod n$.

But now note that

$$\begin{aligned} X &= (C \bmod n) \times (2^e \bmod n) \\ &= (M^e \bmod n) \times (2^e \bmod n) \\ &= (2M)^e \bmod n \end{aligned}$$

Security of RSA

Chosen Ciphertext Attack & Optimal Asymmetric Encryption Padding :



P = encoding parameters
M = message to be encoded
H = hash function
DB = data block
MGF = mask generating function
EM = encoded message

Figure 9.10 Encryption Using Optimal Assymetric Encryption Padding (OAEP)

Security of RSA

Fault Based Attack :

- Approach is an attack on a processor that is generating RSA digital signatures.
- The attack includes : faults in signature computation by reducing the power to the processor.
- Fault cause the software to produce invalid signatures., which can be analyzed by the attacker to recover the private key.
- The attack algorithm includes single bit errors & observing the results.

Diffie Hellman Key Exchange

The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent encryption of messages.
The algorithm itself is limited to the exchange of secret values.

Primitive Root :

if **a** is primitive root of the prime number **p** , then the numbers.

$$\mathbf{a \bmod p,\ a^2 \bmod p,\ a^3 \bmod p\ \ \ldots\ldots\ a^{p-1} \bmod p}$$

are distinct and consist of the integers from 1 through p-1 in some permutation **(i,e must results in 1,2,3,……. p-1)**

Algorithm :

there are two publicly known numbers: a prime number and an integer α that is a primitive root of q.

Diffie Hellman Key Exchange

Global Public Elements

q	prime number
α	$\alpha < q$ and α a primitive root of q

User A Key Generation

Select private X_A	$X_A < q$
Calculate public Y_A	$Y_A = \alpha^{XA} \bmod q$

User B Key Generation

Select private X_B	$X_B < q$
Calculate public Y_B	$Y_B = \alpha^{XB} \bmod q$

Calculation of Secret Key by User A

$K = (Y_B)^{XA} \bmod q$

Calculation of Secret Key by User B

$K = (Y_A)^{XB} \bmod q$

Figure 10.1 The Diffie-Hellman Key Exchange Algorithm

Diffie Hellman Key Exchange

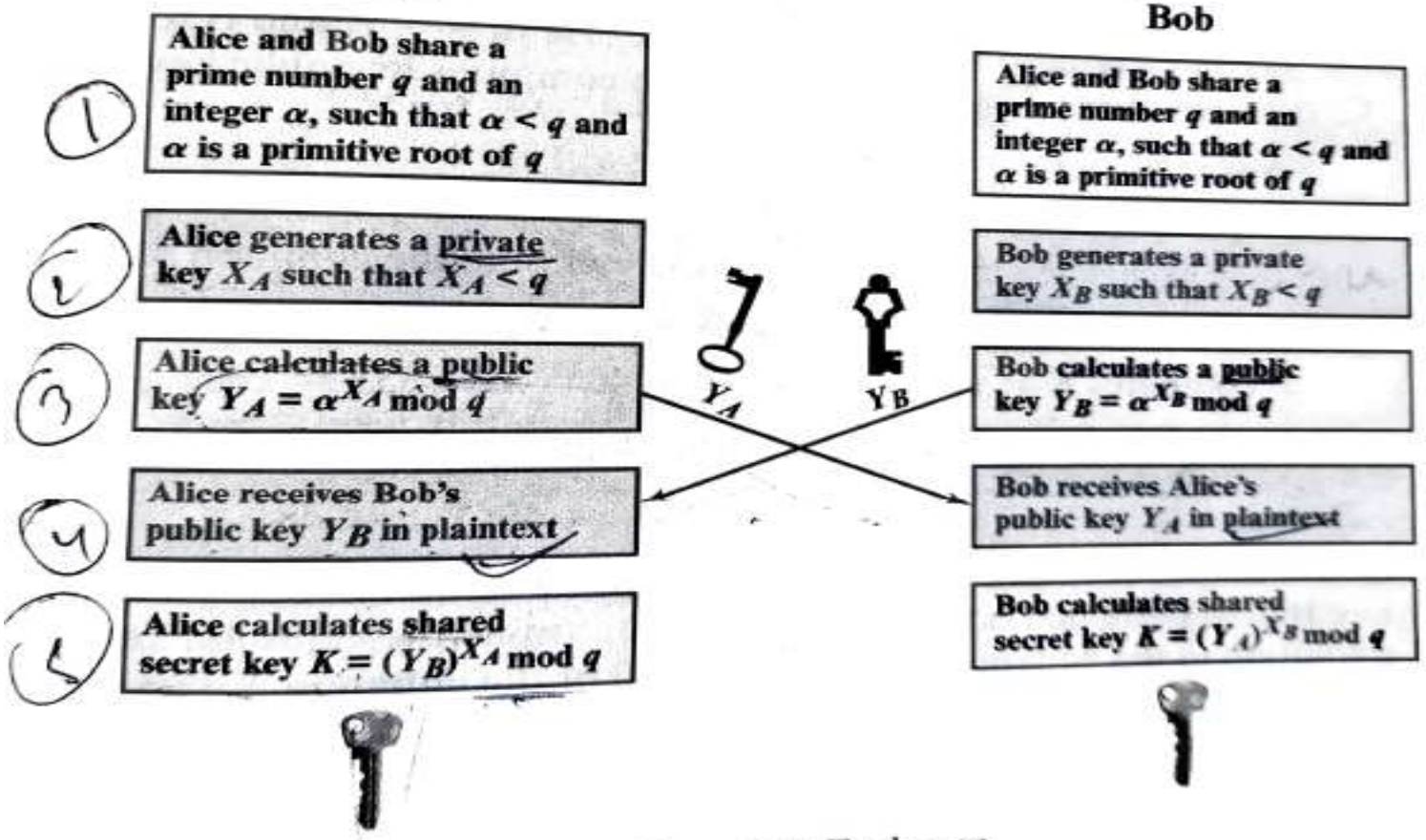


Figure 10.1 The Diffie–Hellman Key Exchange

Diffie Hellman Key Exchange

$$
\begin{aligned}
K &= (Y_B)^{X_A} \bmod q \\
&= (\alpha^{X_B} \bmod q)^{X_A} \bmod q \\
&= (\alpha^{X_B})^{X_A} \bmod q \\
&= \alpha^{X_B X_A} \bmod q \\
&= (\alpha^{X_A})^{X_B} \bmod q \\
&= (\alpha^{X_A} \bmod q)^{X_B} \bmod q \\
&= (Y_A)^{X_B} \bmod q
\end{aligned}
$$

= K

Key Exchange Protocol

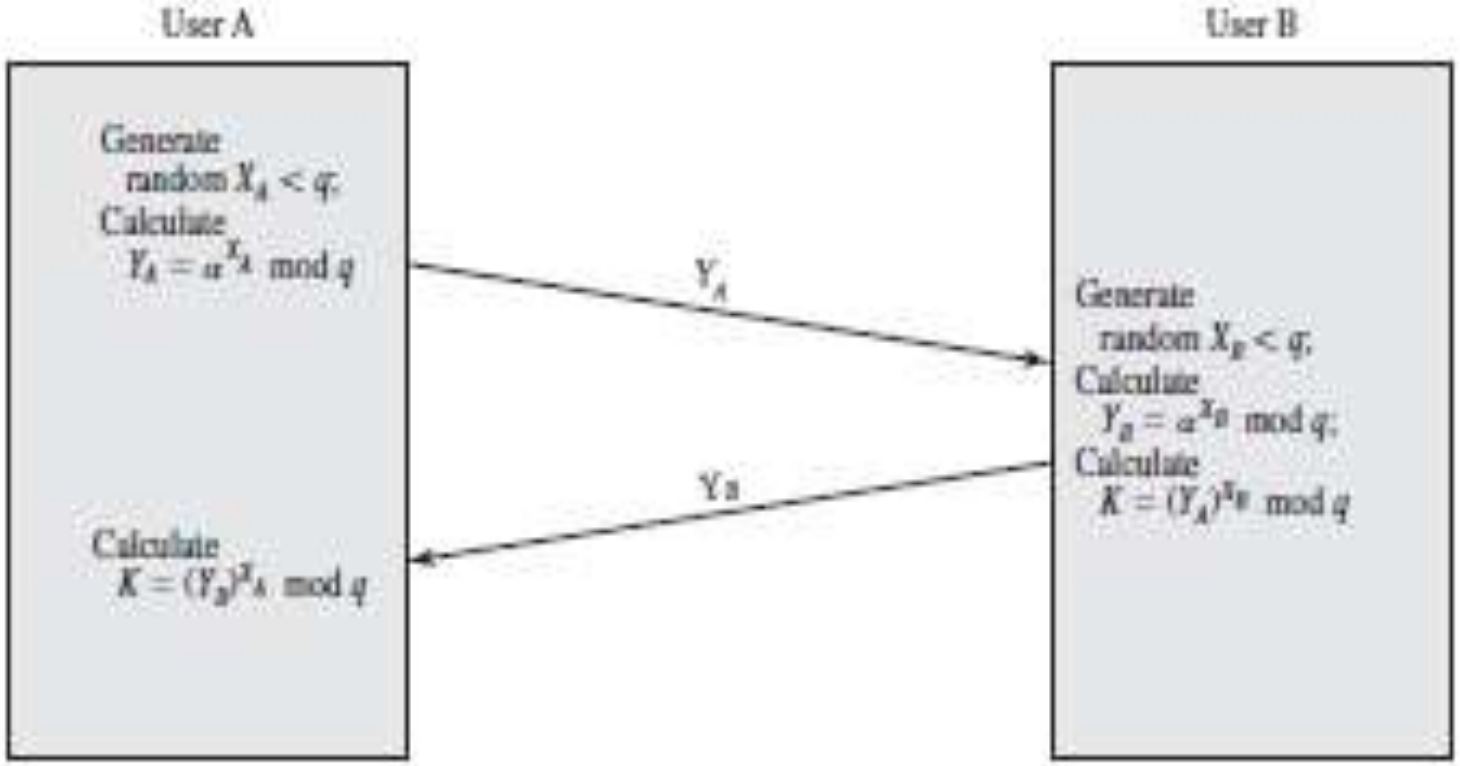


Figure 10.2 Diffie-Hellman Key Exchange

Man in the Middle Attack

The protocol depicted in Figure 10.2 is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is the adversary. The attack proceeds as follows:

1. Darth prepares for the attack by generating two random private keys X_{D1} and X_{D2} and then computing the corresponding public keys Y_{D1} and Y_{D2}.
2. Alice transmits Y_A to Bob.
3. Darth intercepts Y_A and transmits Y_{D1} to Bob. Darth also calculates $K2 = (Y_A)^{X_{D1}} \bmod q$.
4. Bob receives Y_{D1} and calculates $K1 = (Y_{D1})^{X_B} \bmod q$.
5. Bob transmits Y_B to Alice.
6. Darth intercepts Y_B and transmits Y_{D2} to Alice. Darth calculates $K1 = (Y_B)^{X_{D1}} \bmod q$.
7. Alice receives Y_{D2} and calculates $K2 = (Y_{D2})^{X_A} \bmod q$.

Man in the Middle Attack

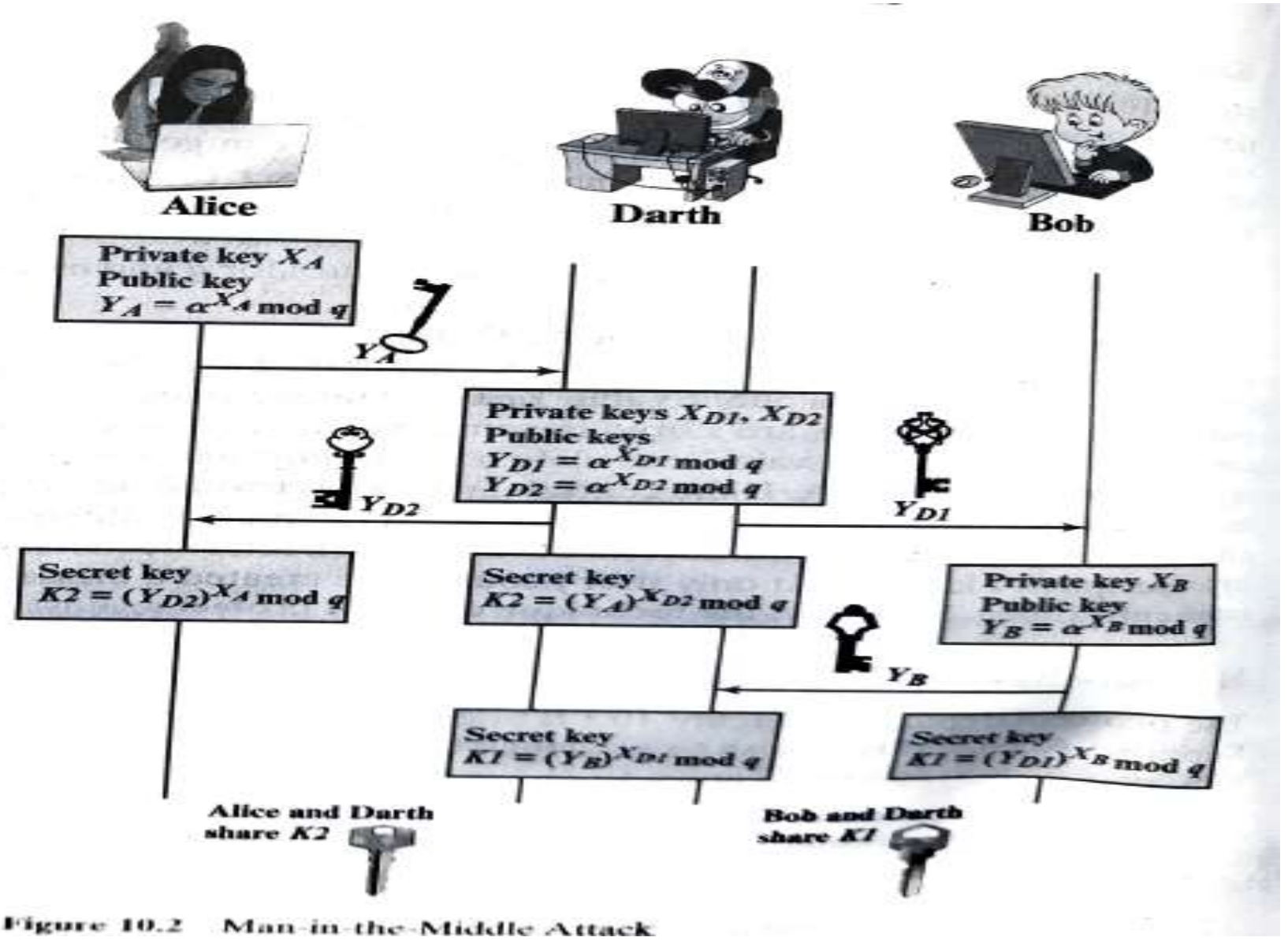


Figure 10.2 Man-in-the-Middle Attack

Man in the Middle Attack

At this point, Bob and Alice think that they share a secret key, but instead Bob and Darth share secret key and Alice and Darth share secret key . All future communication between Bob and Alice is compromised in the following way.

1. Alice sends an encrypted message M : E(K2, M) .

2. Darth intercepts the encrypted message and decrypts it to recover M .

3. Darth sends Bob E(K2, M) or E(K2, M') , where M' is any message.

In the first case, Darth simply wants to eavesdrop on the communication without altering it. In the second case, Darth wants to modify the message going to Bob.

The key exchange protocol is vulnerable to such an attack because it does not authenticate the participants. This vulnerability can be overcome with the use of digital signatures and public-key certificates;

Cryptographic Hash Function

A **hash function** H accepts a variable-length block of data as input and produces a fixed-size hash value .

A “good” hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random.

In general terms, **the principal object of a hash function** is **data integrity**.

A change to any bit or bits in results, with high probability, in a change to the hash code.

The kind of hash function needed for security applications is referred to as a **cryptographic hash function**.

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

Cryptographic Hash Function

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

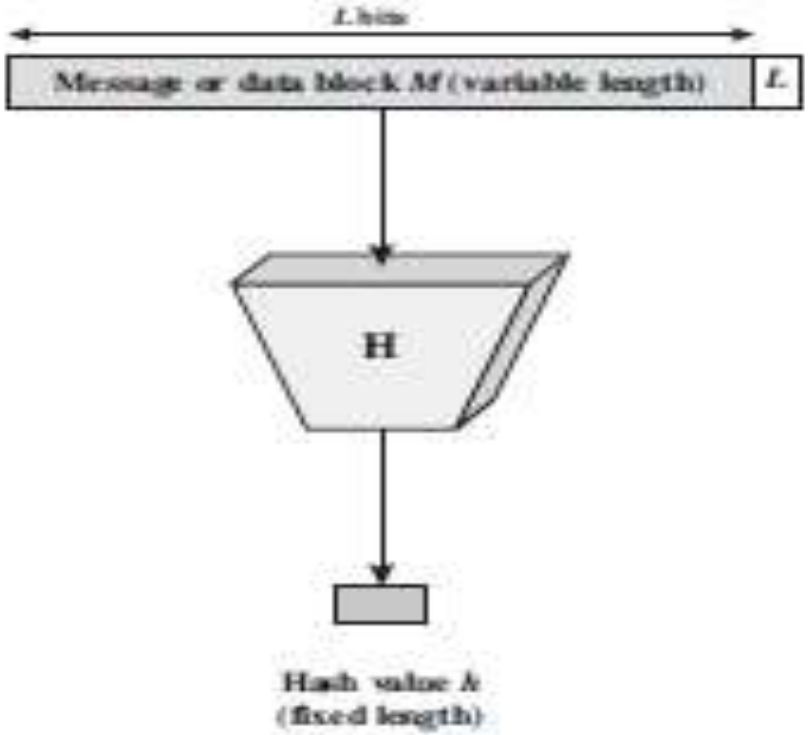


Figure 11.1 Black Diagram of Cryptographic Hash Function; $h = H(M)$

Applications of Cryptographic Hash Function

The most versatile cryptographic algorithm is the cryptographic hash function.
It is used in a wide variety of security applications and Internet protocols.

Message Authentication

Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., contain no modification, insertion, deletion, or replay).

In many cases, there is a requirement that the authentication mechanism assures that purported identity of the sender is valid. When a hash function is used to provide message authentication, the hash function value is often referred to as a **message digest**.

Figure 11.2 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows.

Applications of Cryptographic Hash Function

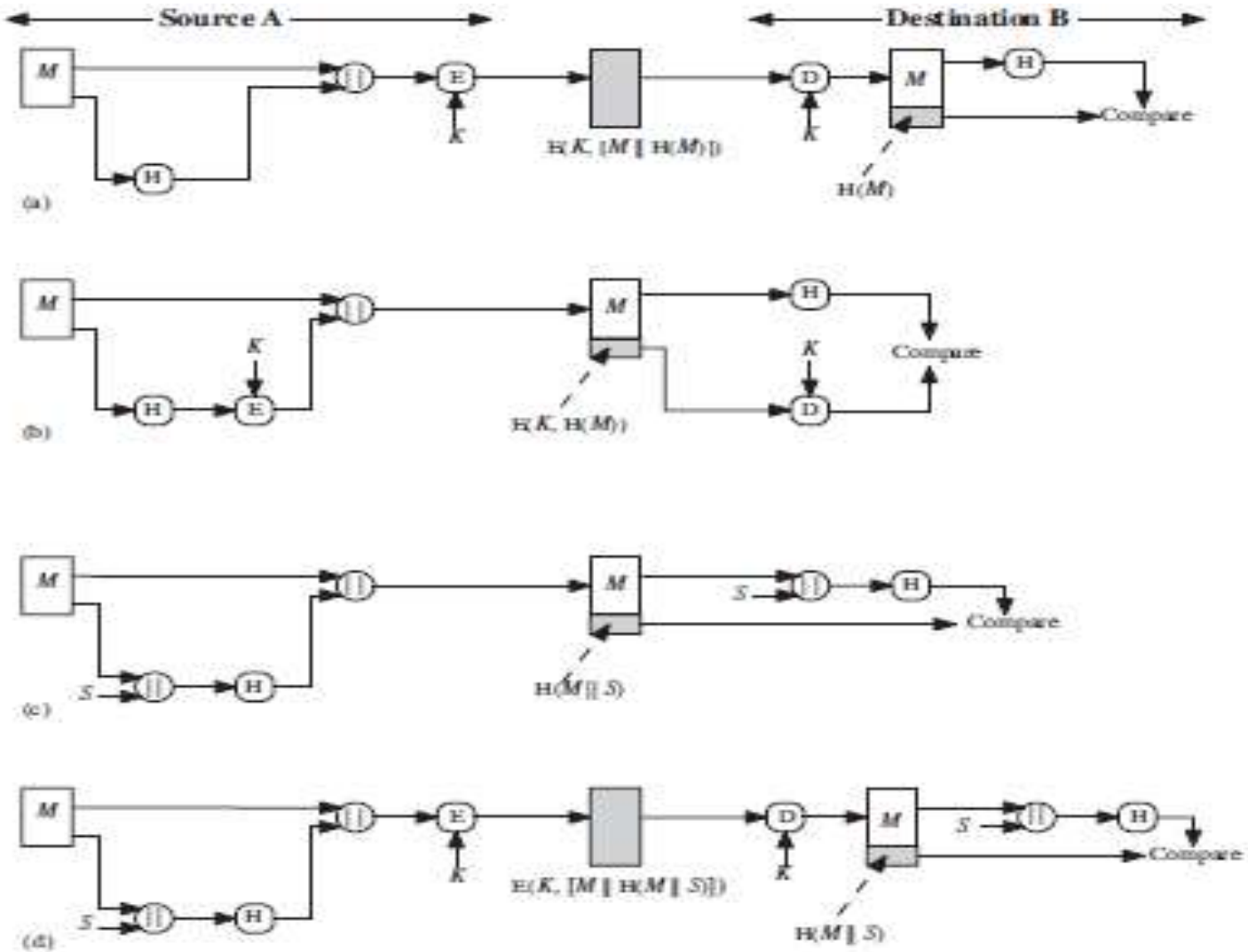


Figure 11.2 Simplified Examples of the Use of a Hash Function for Message Authentication

Applications of Cryptographic Hash Function

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

b. Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality.

c. It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value *S*. A computes the hash value over the concatenation of *M* and *S* and appends the resulting hash value to . Because B possesses , it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.

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

Applications of Cryptographic Hash Function

Reasons for avoiding encryption :

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

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

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

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

Applications of Cryptographic Hash Function

M essage authentication is achieved using a **message authentication code (MAC)**, also known as a **keyed hash function**.

Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties. A MAC function takes as input **a secret key** and **a data block** and produces **a hash value**, referred **to as the MAC**. This can then be transmitted with or stored with the protected message.

If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the stored MAC value. An attacker who alters the message will be unable to alter the MAC value without knowledge of the secret key.

Note that the verifying party also knows who the sending party is because no one else knows the secret key.

T he combination of hashing and encryption results in an overall function that is, in fact, a MAC (Figure 11.2b). That is, E(, H( )) is a function of a variable-length message and a secret key , and it produces a fixed-size output that is secure against an opponent who does not know the secret key.

Cryptographic Hash Function

Digital Signatures

The hash value of a message is encrypted with a user's private key. Anyone who knows the user's public key can verify the integrity of the message that is associated with the digital signature. In this case, an attacker who wishes to alter the message would need to know the user's private key.

Figure 11.3 illustrates, in a simplified fashion, how a hash code is used to provide a digital signature.

a. The hash code is encrypted, using public-key encryption with the sender's private key. As with Figure 11.2b, this provides authentication. It also provides a digital signature, because only the sender could have produced the encrypted hash code. In fact, this is the essence of the digital signature technique.

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

Cryptographic Hash Function

Digital Signatures :

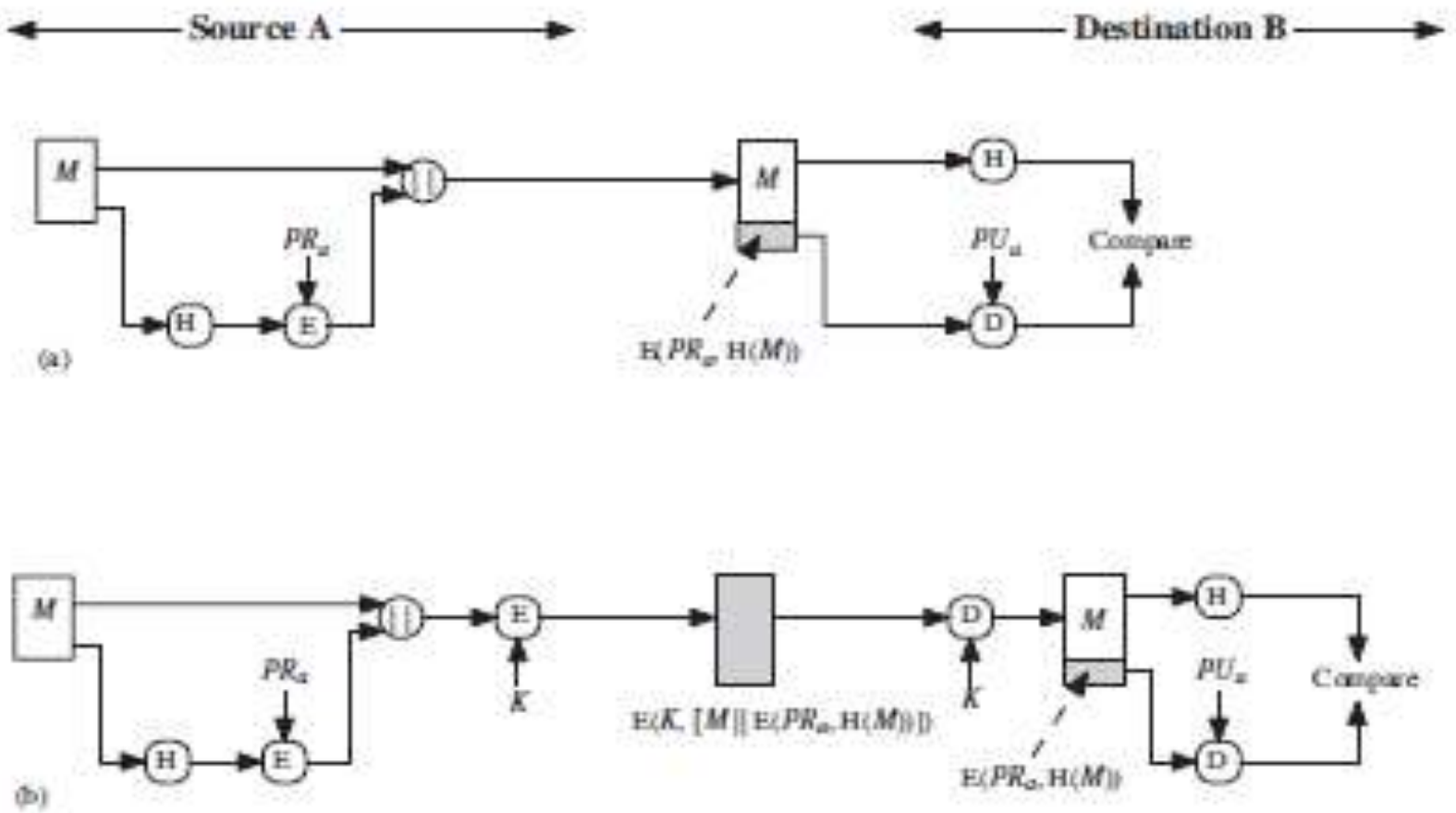


Figure 11.3 Simplified Examples of Digital Signatures

Cryptographic Hash Function

Other Applications :

- **one-way password file :** Hash functions are commonly used to create a **one-way password file**. : it is a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.

- Intrusion detection and **virus detection** : Hash functions can be used for **intrusion detection** and **virus detection**. Store H(F) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing H(F). An intruder would need to change F without changing H(F).

- PRF & PRNG :A cryptographic hash function can be used to construct a **pseudorandom function**
(PRF) or a **pseudorandom number generator (PRNG)** : A common application
for a hash-based PRF is for the generation of symmetric keys.

Two Simple Hash Functions

Self Study

Secure Hash Algorithm

- The most widely used hash function is the Secure Hash Algorithm (SHA).
- SHA was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993.
- When weaknesses were discovered in SHA, now known as **SHA-0**, a revised version was issued as FIPS 180-1 in 1995 and is referred to as **SHA-1**. The actual standards document is entitled "Secure Hash Standard." SHA is based on the hash function MD4, and its design closely models MD4. SHA-1 is also specified in RFC 3174, which essentially duplicates the material in FIPS 180-1 but adds a C code implementation.
- **SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively.**
  Collectively, these hash algorithms are known as SHA-2.

  These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version (Table 11.3).

  SHA-2 is also specified in RFC 4634, which essentially duplicates the material in FIPS 180-3 but adds a C code implementation.

Secure Hash Algorithm

These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version (Table 11.3).

SHA-2 is also specified in RFC 4634, which essentially duplicates the material in FIPS 180-3 but adds a C code implementation.

Table 11.3 Comparison of SHA Parameters

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Message Digest Size	160	224	256	384	512
Message Size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block Size	512	512	512	1024	1024
Word Size	32	32	32	64	64
Number of Steps	80	64	64	80	80

Note: All sizes are measured in bits.

SHA 512 Logic

- The algorithm takes input a message with maximum length less than 2^{128} bits and produces a output of 512 bit message digest.
- The input is processed in 1024bit blocks.
- Fig 11.8 depicts the overall processing of message to produce a digest.
- Fig 11.7 depicts a general structure.

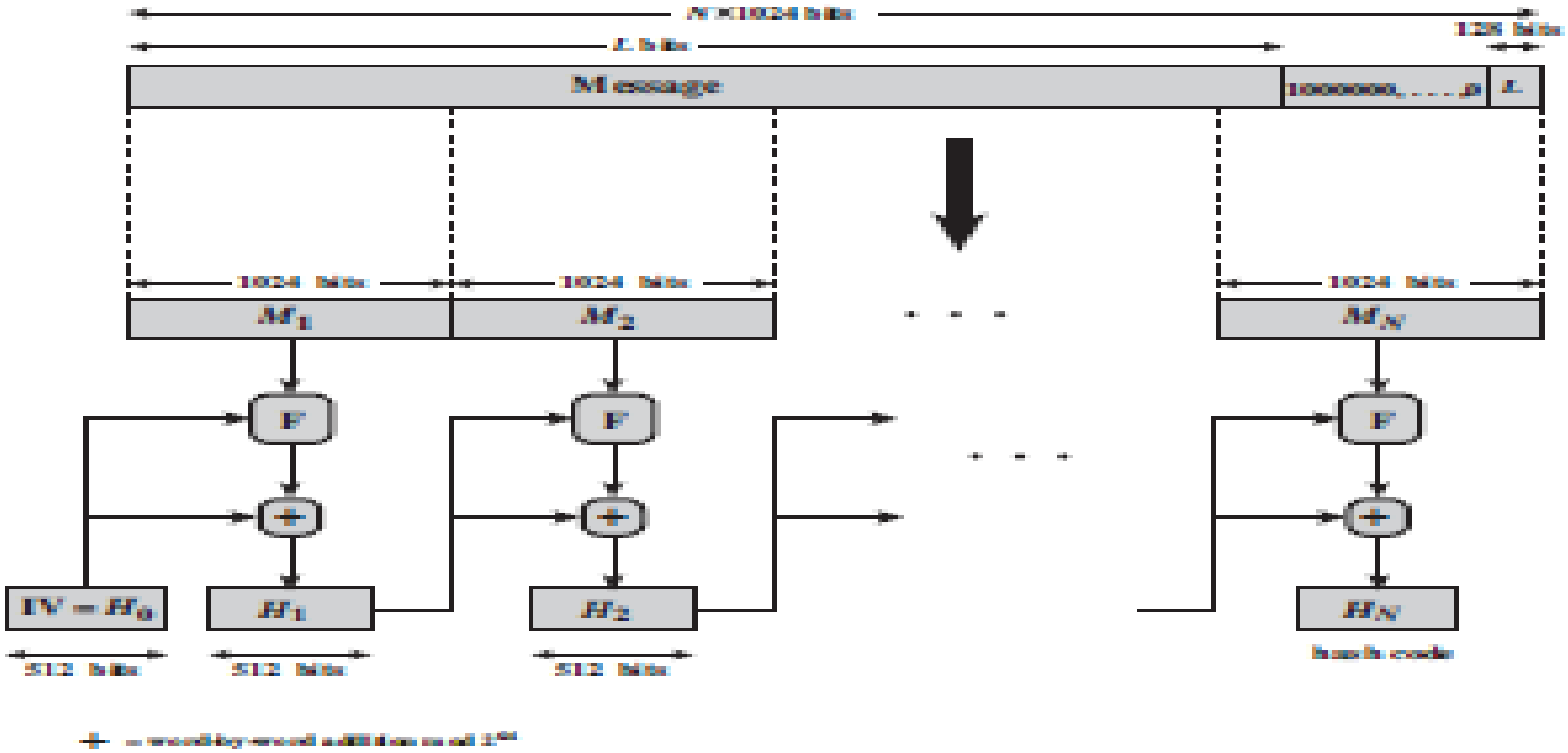


Figure 11.8 Message Digest Generation Using SHA-512

SHA 512 Logic

The processing consists of the following steps :

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

Step 2 **Append length.** A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).

The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. In Figure 11.8, the expanded message is represented as the sequence of 1024-bit blocks $M_1, M_2, \ldots, M_N$, so that the total length of the expanded message is $N \times 1024$ bits.

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

```
a = 6A09E667F3BCC908   e = 510E527FADE682D1
b = BB67AE8584CAA73B   f = 9B05688C2B3E6C1F
c = 3C6EF372FE94F82B   g = 1F83D9ABFB41BD6B
d = A54FF53A5F1D36F1   h = 5BE0CD19137E2179
```

SHA 512 Logic

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

Step 4 Process message in 1024-bit (128-word) blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 11.8. The logic is illustrated in Figure 11.9.

Each round takes as input the 512-bit buffer value, abcdefgh, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, H_{i-1}. Each round t makes use of a 64-bit value W_t derived from the current 1024-bit block being processed (M_i). These values are derived using a message schedule described subsequently. Each round also makes use of an additive constant K_t, where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data. Table 11.4 shows these constants in hexadecimal format (from left to right).

The output of the eightieth round is added to the input to the first round (H_{i-1}) to produce H_i. The addition is done independently for each of the eight

SHA 512 Logic: SHA-512 processing of single 1024-bit block

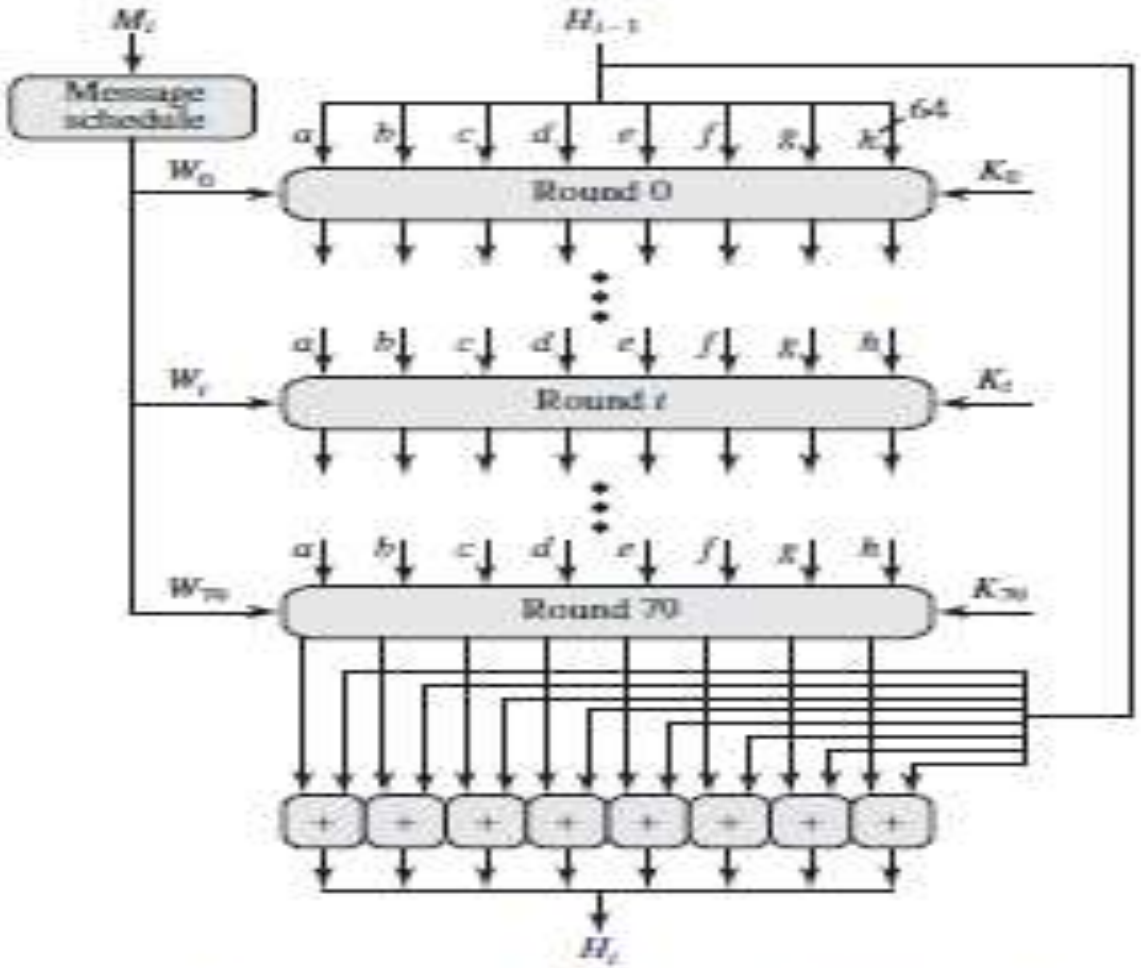


Figure 11.9 SHA-512 Processing of a Single 1024-Bit Block

words in the buffer with each of the corresponding words in H_{i-1}, using addition modulo 2^{64}.

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

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

$$H_0 = \text{IV}$$
$$H_i = \text{SUM}_{64}(H_{i-1}, \text{abcdefgh}_i)$$
$$MD = H_N$$

where

IV = initial value of the abcdefgh buffer, defined in step 3

abcdefgh_i = the output of the last round of processing of the ith message block

N = the number of blocks in the message (including padding and length fields)

SUM_{64} = addition modulo 2^{64} performed separately on each word of the pair of inputs

MD = final message digest value

SHA 512 Round Function

The detail at the logic of each 80 steps of processing of one 512 bit block (Fig 11.10). Each round is defined by the following set of equations :

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

where

t = step number; $0 \leq t \leq 79$

$\mathrm{Ch}(e, f, g) = (e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$
the conditional function: If e then f else g

SHA 512 Round Function

$\text{Maj}(a, b, c)$ – $(a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$
the function is true only of the majority (two or three) of the arguments are true

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

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

$\text{ROTR}^n(x)$ – circular right shift (rotation) of the 64-bit argument x by n bits

W_t – a 64-bit word derived from the current 512-bit input block

K_t – a 64-bit additive constant

$+$ – addition modulo 2^{64}

SHA 512 Round Function

Observations made in the Round Function :

1. Six of the eight rounds of the output of the function involve simply permutation (b,c,d,f,g,h) by means of rotation. This is indicating by shading in Fig 11.10

2. Only two of the output words (a,e) are generated by substitution. Word is a function of input variables (d ,e ,f ,g ,h), as well as the round word and the constant .Word is a function of all of the input variables except *d*, as well as the round word *Wt* and the constant *Kt*.

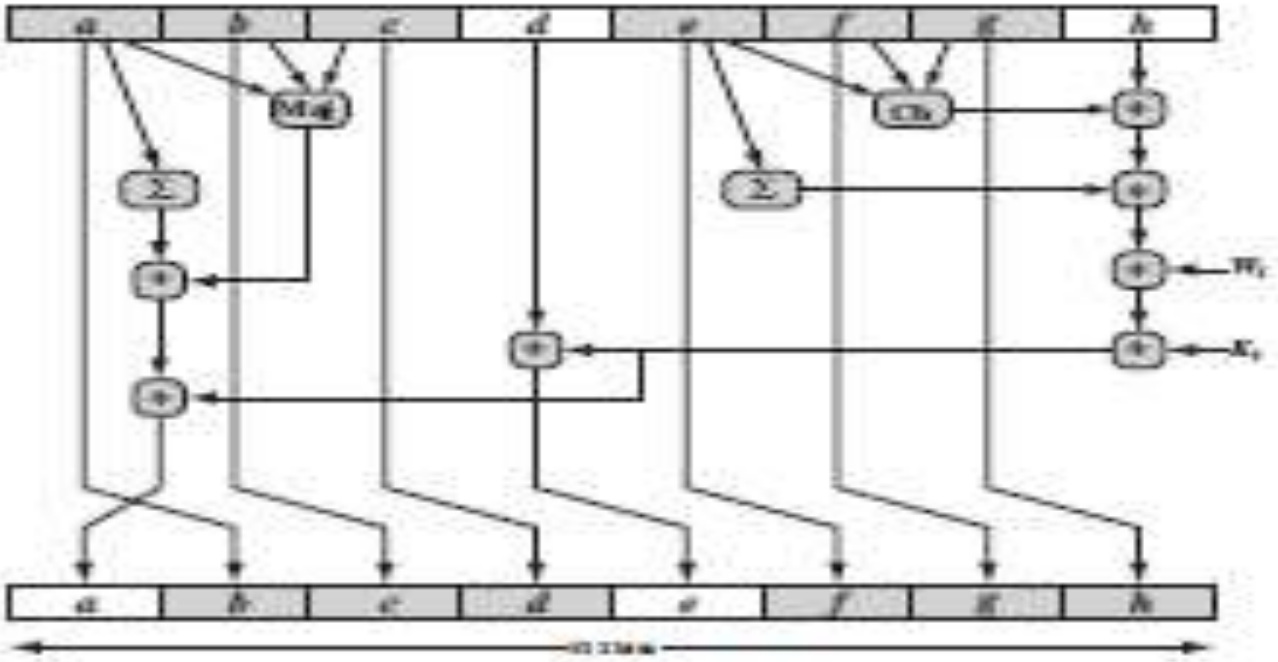


Figure 11.10 Elementary SHA-512 Operation (single round)

SHA 512 Round Function

- It remains to indicate how the 64-bit word values are derived from the 1024-bit message.
- Figure 11.11 illustrates the mapping.
- The first 16 values of are taken directly from the 16 words of the current block. The remaining values are defined as

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

where

$$\sigma_0^{512}(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$$

$$\sigma_1^{512}(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

$\text{SHR}^n(x)$ = left shift of the 64-bit argument x by n bits with padding by zeros on the right

$+$ = addition modulo 2^{64}

SHA 512 Round Function

It remains to indicate…

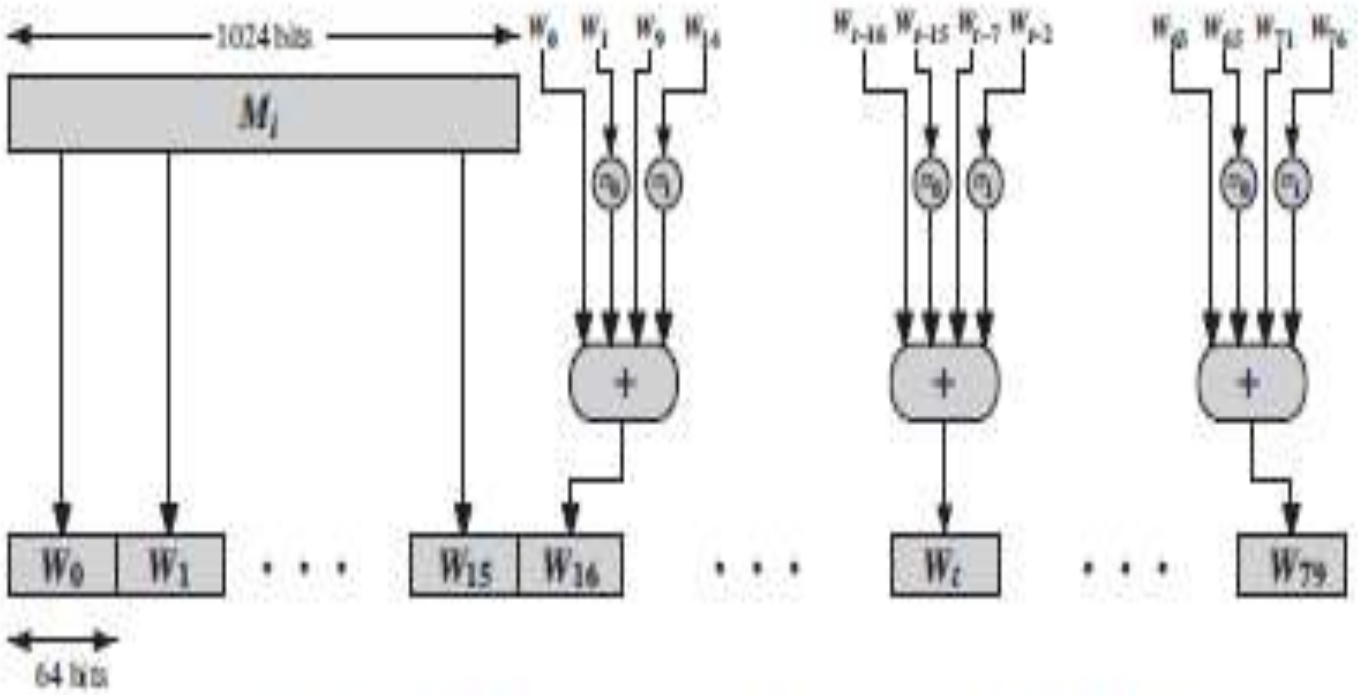


Figure 11.11 Creation of 80-word Input Sequence for SHA-512 Processing of Single Block

SHA 512 Round Function

Thus, in the first 16 steps of processing, the value of Wt is equal to the corresponding word in the message block. For the remaining 64 steps, the value of Wt consists of the circular left shift by one bit of the XOR of four of the preceding values of , with two of those values subjected to shift and rotate operations.

This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.

Figure 11.12 summarizes the SHA-512 logic.

The padded message consists blocks $M_1, M_2, \ldots, M_N$. Each message block M_i consists of 16 64-bit words $M_{i,0}, M_{i,1}, \ldots, M_{i,15}$. All addition is performed modulo 2^{64}.

$H_{0,0} = 6A09E667F3BCC908$ $H_{0,4} = 510E527FADE682D1$
$H_{0,1} = BB67AE8584CAA73B$ $H_{0,5} = 9B05688C2B3E6C1F$
$H_{0,2} = 3C6EF372FE94F82B$ $H_{0,6} = 1F83D9ABFB41BD6B$
$H_{0,3} = A54FF53A5F1D36F1$ $H_{0,7} = 5BE0CDI9137E2179$

for $i = 1$ **to** N

1. Prepare the message schedule W

for $i = 0$ **to** 15
$W_t = M_{it}$
for t = 16 **to** 79
$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$

2. Initialize the working variables

$a = H_{i-1,0}$ $e = H_{i-1,4}$
$b = H_{i-1,1}$ $f = H_{i-1,5}$
$c = H_{i-1,2}$ $g = H_{i-1,6}$
$d = H_{i-1,3}$ $h = H_{i-1,7}$

3. Perform the main hash computation

for $i = 0$ **to** 79
$T_1 = h + \text{Ch}(e,f,g) + \left(\sum_1^{512} e\right) + W_t + K_t$
$T_2 = \left(\sum_0^{512} a\right) + \text{Maj}(a,b,c)$
$h = g$
$g = f$
$f = e$
$e = d + T_1$
$d = c$
$c = b$
$b = a$
$a = T_1 + T_2$

4. Compute the inermediate hash value

$H_{i,0} = a + H_{i-1,0}$ $H_{i,4} = a + H_{i-1,4}$
$H_{i,1} = a + H_{i-1,1}$ $H_{i,5} = a + H_{i-1,5}$
$H_{i,2} = a + H_{i-1,2}$ $H_{i,6} = a + H_{i-1,6}$
$H_{i,3} = a + H_{i-1,3}$ $H_{i,7} = a + H_{i-1,7}$

return $[H_{N,0} \| H_{N,1} \| H_{N,2} \| H_{N,3} \| H_{N,4} \| H_{N,5} \| H_{N,6} \| H_{N,7}]$

Figure 11.12 SHA-512 Logic

SHA 512 Round Function

Thus, in the first 16 steps of processing, the value of Wt is equal to the corresponding word in the message block. For the remaining 64 steps, the value of Wt consists of the circular left shift by one bit of the XOR of four of the preceding values of , with two of those values subjected to shift and rotate operations.

This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.

SHA 512 : Example

Hash a one-block message consisting of three ASCII characters: “abc”, which is equivalent to the following 24-bit binary string:

01100001 01100010 01100011

- The message is padded to a length congruent to 896 modulo 1024.
- In this case of a single block, the padding consists of 896-24=872 bits, consisting of a “1” bit followed by 871 “0” bits.
- Then a 128-bit length value is appended to the message, which contains the length of the original message (before the padding). The original length is 24 bits, or a hexadecimal value of 18. Putting this all together, the 1024-bit message block, in hexadecimal, is

6162638000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018

SHA 512 : Example

This block is assigned to the words W0, . . . ,W15 of the message schedule, which appears as follows.

$W_0 = 6162638000000000$
$W_1 = 0000000000000000$
$W_2 = 0000000000000000$
$W_3 = 0000000000000000$
$W_4 = 0000000000000000$
$W_5 = 0000000000000000$
$W_6 = 0000000000000000$
$W_7 = 0000000000000000$
$W_8 = 0000000000000000$
$W_9 = 0000000000000000$
$W_{10} = 0000000000000000$
$W_{11} = 0000000000000000$
$W_{12} = 0000000000000000$
$W_{13} = 0000000000000000$
$W_{14} = 0000000000000000$
$W_{15} = 0000000000000018$

As indicated in Figure 11.12, the eight 64-bit variables, a through h, are initialized to values $H_{0,0}$ through $H_{0,7}$. The following table shows the initial values of these variables and their values after each of the first two rounds.

a	6a09e667f3bcc908	f6afceab8bcfcddf5	1320f8c9fb872cc0
b	bb67ae8584caa73b	6a09e667f3bcc908	f6afceab8bcfcddf5
c	3c6ef372fe94f82b	bb67ae8584caa73b	6a09e667f3bcc908
d	a54ff53a5f1d36f1	3c6ef372fe94f82b	bb67ae8584caa73b
e	510e527fade682d1	58cb02347ab51f91	c3d4ebfd48650ffa
f	9b05688c2b3e6c1f	510e527fade682d1	58cb02347ab51f91
g	1f83d9abfb41bd6b	9b05688c2b3e6c1f	510e527fade682d1
h	5be0cd19137e2179	1f83d9abfb41bd6b	9b05688c2b3e6c1f

SHA 512 : Example

Note that in each of the rounds, six of the variables are copied directly from variables from the preceding round.

The process continues through 80 rounds. The output of the final round is

```
73a54f399fa4b1b2 10d9c4c4295599f6 d67806db8b148677 654ef9abec389ca9
d08446aa79693ed7 9bb4d39778c07f9e 25c96a7768fb2aa3 ceb9fc3691ce8326
```

The hash value is then calculated as

$H_{1,0}$ = 6a09e667f3bcc908 + 73a54f399fa4b1b2 = ddaf35a193617aba

$H_{1,1}$ = bb67ae8584caa73b + 10d9c4c4295599f6 = cc417349ae204131

$H_{1,2}$ = 3c6ef372fe94f82b + d67806db8b148677 = 12e6fa4e89a97ea2

$H_{1,3}$ = a54ff53a5f1d36f1 + 654ef9abec389ca9 = 0a9eeee64b55d39a

$H_{1,4}$ = 510e527fade682d1 + d08446aa79693ed7 = 2192992a274fc1a8

$H_{1,5}$ = 9b05688c2b3e6c1f + 9bb4d39778c07f9e = 36ba3c23a3feebbd

$H_{1,6}$ = 1f83d9abfb41bd6b + 25c96a7768fb2aa3 = 454d4423643ce80e

$H_{1,7}$ = 5be0cd19137e2179 + ceb9fc3691ce8326 = 2a9ac94fa54ca49f

The resulting 512-bit message digest is

```
ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f
```

SHA 512 : Example

Suppose now that we change the input message by one bit, from "abc" to "cbc". Then, the 1024-bit message block is

```
6362638000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000018
```

And the resulting 512-bit message digest is

```
531668966ee79b70 0b8e593261101354 4273f7ef7b31f279 2a7ef68d53f93264
319c165ad96d9187 55e6a204c2607e27 6e05cdf993a64c85 ef9e1e125c0f925f
```

The number of bit positions that differ between the two hash values is 253, almost exactly half the bit positions, indicating that SHA-512 has a good avalanche effect.

UNIT 3
Pseudorandom Number Generation and stream ciphers

Principles of Pseudorandom Number Generation

The Use of Random Numbers:

- Key Distribution and reciprocal(mutual) authentication schemes: nonce
  nonces in authentication protocols to prevent replay attack
- Session key Generation
- Generation of keys for RSA algorithm
- Generation of bit stream for symmetric stream encryption

If numbers are not random 20, 1, 20, 3, 20, 20…. Attacker can predict what number are used as keys. Hence generating good random numbers is essential.

Ex. of a good random sequence: 27,13,1,42,16, 50000……

Principles of Pseudorandom Number Generation

- Important Cryptographic function – Random Bit Streams
- Used in wide variety of Contexts: Key Generation , Encryption
- Fundamental Different strategies for generating random bits/ random number

Deterministic Random Bit Generator:

* computes bits deterministically using an algorithm. This class o f random bit generators – **PRNGs/DRBGs**

Non Deterministic Random Bit Generator:

* produces bits non deterministically using some physical source that produces some sort of random output. This class o f random bit generators- **TRNGs/NRBGs**

Principles of Pseudorandom Number Generation

Requirements for a sequence of random numbers:

- ***RANDOMNESS***
- ***UNPREDICTABILITY***

RANDOMNESS: Sequence of numbers be in random in some well defined statistical sense.

The two criteria used to validate that the sequence of numbers is random:

- **Uniform distribution:** The distribution of bits in the sequence should be uniform, that is, the frequency of occurrence of ones and zeros should be approximately equal.

  00101101

- **Independence:** No subsequence in the sequence can be inferred from others.

  10, 20, 30,40, 50….. Dependence (bad random sequence)

UNPREDICTABILITY:

- In applications such as reciprocal authentication, session key generation, and stream ciphers, the requirement is not just that the sequence of numbers be statistically random but that the successive members of the sequence are unpredictable.
- With "true" random sequences, each number is statistically independent of other numbers in the sequence and therefore unpredictable.
- **Hard to predict next value in sequence**

  010101010101 (6 zeros and 6 ones, but not good sequence)

TRNGs , PRNGs and PRF

Pseudorandom Numbers: The algorithmic techniques used in cryptographic applications for random number generation are deterministic and produce sequences of numbers that are statistically random. If the algorithm is good, resulting sequence will pass many tests of randomness. Such numbers are referred to as Pseudorandom Numbers.

Random Number Generator : TRNGs (True Random number sequence)

- Takes the input from effective random source referred to as entrophy.
- **Entrophy source :** can be considered from physical environment (no algorithm is used) of computer such as keystroke timing pattern, Mouse/keyboard activity, I/O operations, coin flipping, mouse movement instantaneous values of system clock, Variations of radiation, electric devices generate noise, thermal noise from resistors.
- The source, or combination of sources, serve as input to an algorithm that produces random binary output. As shown in fig 7.1(a)
- The TRNG may simply involve conversion of an analog source to a binary output. The TRNG may involve additional processing to overcome any bias in the source.
- Inconvenient (physical activity is required), small number of values Non deterministic source,

TRNGs , PRNGs and PRF

a PRNG takes as input a fixed value, called the **seed**, and produces a sequence of output bits using a deterministic algorithm.

Also, the result of some algorithm is fed as input to produce additional output bits

Two forms of pseudorandom number generator:

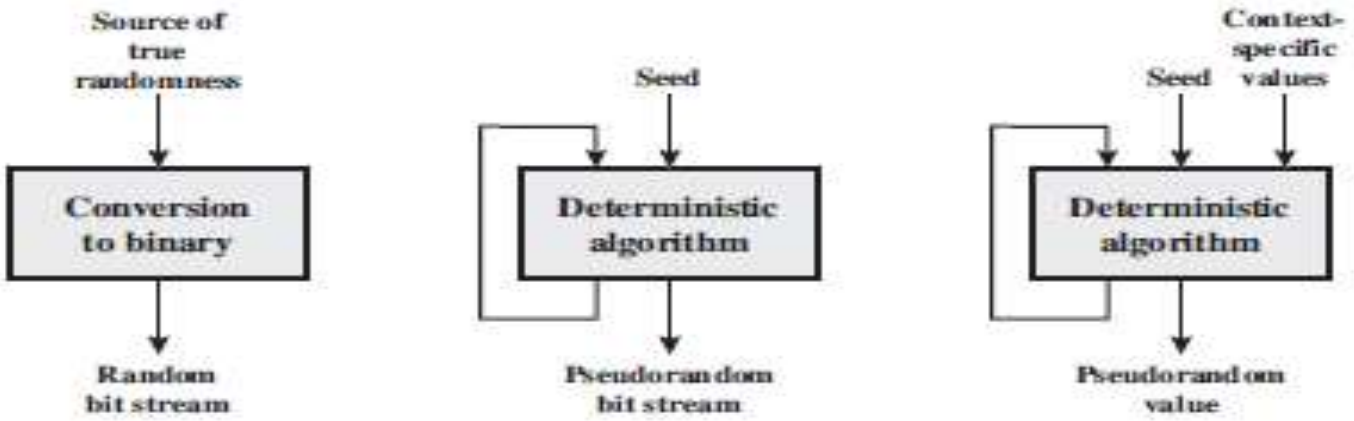
PRNGs (Pseudo Random number sequence)

- An algorithm that is used to produce an open-ended sequence of bits is referred to as a PRNG.
- a PRNG takes as input a fixed value, called the **seed**, and produces a sequence of output bits using a deterministic algorithm. Shown in fig 7.1(b)
- Not true random numbers, but appear to be random
- Deterministic algorithms to generate " relative random" sequence
- Seed is algorithm input..
- Produces continuous stream of random bits

PRF (Pseudo Random Function)

- A PRF is used to produced a pseudorandom string of bits of some fixed length.
- Examples are symmetric encryption keys and nonces. Typically, the PRF takes as input a seed plus some context specific values, such as a user ID or an application ID.
- Same as PRNG but produces string of bits of fixed length.

TRNGs and PRNGs



TRNG = true random number generator
PRNG = pseudorandom number generator
PRF = pseudorandom function

Figure 7.1 Random and Pseudorandom Number Generators

- **Pseudorandom number generator v/s Pseudorandom function (PRF):**

  - the number of bits produced

Similarities : - same algorithm used

- requires seed
- exhibits randomness and unpredictability.
- Further, PRNG application may employ context specific values

Pseudorandom Number Generators

PRNG Requirements :

RANDOMNESS:

- ***PRNG bit stream appears to be random.***
- ***No Single test to determine : PRNG generate numbers have characteristics of randomness.***
- ***Apply a sequence of tests to PRNG. If it exhibits randomness on the basis of multiple test, can be assumed to satisfy the randomness requirements.***

NIST SP 800-22 specifies that the tests should seek to establish the following **three characteristics:**

- **Uniformity:** At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely, i.e., the probability of each is exactly 1/2. The expected number of zeros (or ones) is n/2, where n = the sequence length.
- **Scalability:** If a sequence is random, then any such extracted subsequence should also be random
- **Consistency:** The behavior of a generator must be consistent across starting values (seeds). Eg: Zip compressibility

Pseudorandom Number Generators

SP 800-22 lists 15 separate tests of randomness. An understanding of these tests requires a basic knowledge of statistical analysis, so we don't attempt a technical description here. Instead, to give some flavor for the tests, we list three of the tests and the purpose of each test, as follows.

- **Frequency test:** This is the most basic test and must be included in any test suite. The purpose of this test is to **determine whether the number of ones and zeros in a sequence is approximately the same** as would be expected for a truly random sequence.
- **Runs test:** The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits bounded before and after with a bit of the opposite value. The purpose of **the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected** for a random sequence.
- **Maurer's universal statistical test:** The focus of this test is the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the **test is to detect whether or not the sequence can be significantly compressed without loss of information.** A significantly compressible sequence is considered to be non-random.

UNPREDICTABILITY

- **Forward unpredictability**: If the seed is unknown, the next output bit in the sequence should be unpredictable in spite of any knowledge of previous bits in the sequence.
- **Backward unpredictability**: It should also not be feasible to determine the seed from knowledge of any generated values.

SEED REQUIREMENTS

- The seed that serves as input to the PRNG must be secure.

- Because the PRNG is a deterministic algorithm, if the adversary can deduce the seed, then the output can also be determined. Therefore, the seed must be unpredictable.

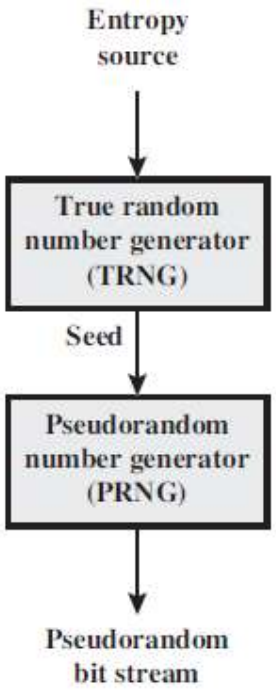


Figure 7.2 Generation of Seed Input to PRNG

Algorithm Design

- Cryptographic PRNGs have been the subject of much research over the years, and a wide variety of algorithms have been developed. These fall roughly into two categories :

• **Purpose-built algorithms:** These are algorithms designed specifically and solely for the purpose of generating pseudorandom bit streams. Some of these algorithms are used for a variety of PRNG applications; several of these are described in the next section. Others are designed specifically for use in a stream cipher.

The most important example of the latter is RC4.

• **Algorithms based on existing cryptographic algorithms:** Cryptographic algorithms have the effect of randomizing input. Indeed, this is a requirement of such algorithms. For example, if a symmetric block cipher produced ciphertext that had certain regular patterns in it, it would aid in the process of cryptanalysis. Thus, cryptographic algorithms can serve as the core of PRNGs.

Three broad categories of cryptographic algorithms are commonly used to create PRNGs:

- **Symmetric block ciphers:**
- **Asymmetric ciphers:** The number theoretic concepts used for an asymmetric cipher can also be adapted for a PRNG.
- **Hash functions and message authentication**

PSEUDORANDOM NUMBER GENERATORS

- Linear Congruential Generators (LCG)
- Blum Blum Shub Generator

Linear Congruential Generators (LCG)

The sequence of random numbers $\{X_n\}$ is obtained via the following iterative equation:

$$X_{n+1} = (aX_n + c) \bmod m$$

m	the modulus	$m > 0$
a	the multiplier	$0 < a < m$
c	the increment	$0 \leq c < m$
X_0	the starting value, or seed	$0 \leq X_0 < m$

a=7 c=0 m=32 X0=1 Sequence={ 7, 17, 23, 1, 7, 17 … etc**} Period=4**

a=5 c=0 m=32 X0=1 Sequence={5,25,29,17,21,9,13,1,5,25} period value - 8

PSEUDORANDOM NUMBER GENERATORS

Linear Congruential Generators (LCG)

Note: to generate a good random sequence in LCG

- m should be large
  - To produce a long series of distinct random numbers
- m should be prime
  - To generate full period generating function

Solve

- Find the random sequence using LCG technique when a = 7, c = 0, m = 32, X0 = 1.

PSEUDORANDOM NUMBER GENERATORS

Blum Blum Shub Generator

- Choose two large prime numbers, P and Q that both have a remainder of 3 when divided by 4.

$$p \equiv q \equiv 3 \pmod 4$$ i.e. (*7* mod 4) = (*11*mod 4) = 3

Ex: the prime numbers 7 and 11 satisfies

- Let *n*=P * Q Ex: n=7x11=77
- Choose a random number ***S***, such that ***S*** is relatively prime to *n*, this is equivalent to saying that neither *P* nor Q is a factor of ***S.***

Ex: S=3 (7,3)=1 (11,3)=1

$X_0 = s^2 \bmod n$

for $i = 1$ **to** ∞

$X_i = (X_{i-1})^2 \bmod n$

$B_i = X_i \bmod 2$

B0= 9 mod 77=9 = 0
B1=81 mod 77=4 =0
B2= 16 mod 77=16=0
B3= 256mod 77=25= 1
B4= 625mod 77=9=1
B5=81 4=0
B6= 16 mod77=16=0
Sequence: 0001100=12

X0=9
X1=4
X2=16
X3=25
.
.
.

Solve

- Find the random sequence using Blum Blum Shub generator when p = 7, q = 11, S_0 =4.

$$
\begin{aligned}
X_0 &= s^2 \bmod n \\
\textbf{for}\, i &= 1 \ \textbf{to}\ \infty \\
X_i &= (X_{i-1})^2 \bmod n \\
B_i &= X_i \bmod 2
\end{aligned}
$$

X0=16

X1= mod 2 =

X2= mod 2 =

X3= mod 2 =

X4= mod 2 =

PSEUDORANDOM NUMBER GENERATORS

p=383, q=503, n= 192642, s=101355

Table 7.1 Example Operation of BBS Generator

i	X_i	B_i
0	20749	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0

i	X_i	B_i
11	137922	0
12	123175	1
13	8630	0
14	114386	0
15	14863	1
16	133015	1
17	106065	1
18	45870	0
19	137171	1
20	48060	0

PSEUDORANDOM NUMBER GENERATION USING A BLOCK CIPHER

- For any **block of plaintext**, a symmetric block cipher produces an **output block that is apparently random**.

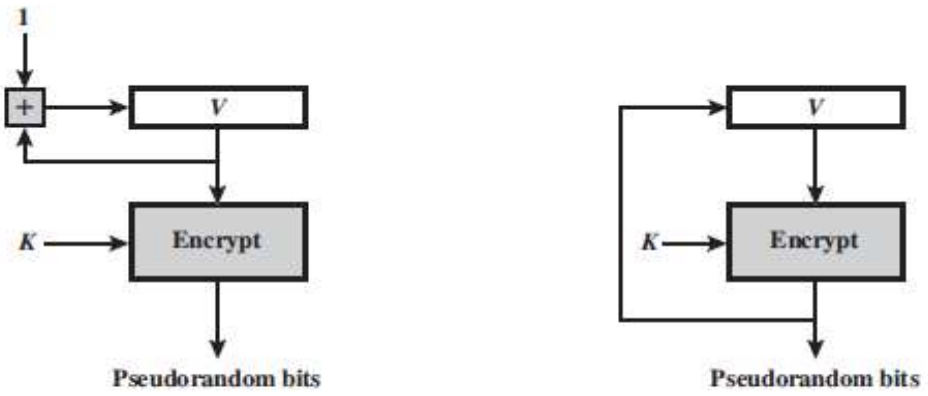
- That is, there are no patterns or regularities in the ciphertext that provide information that can be used to deduce the plaintext.

- Thus, a symmetric block cipher is a good candidate for building a pseudorandom number generator.

  Block ciphers: DES, AES etc.

- Two approaches that uses block ciphers to build PRNG are :
  * CTR mode
  * OFB mode

PRNG Using Block Cipher Modes of Operation



(a) CTR mode (b) OFB mode

PRNG Mechanisms Based on Block Ciphers

- In each case, the **seed** consists of two parts: **the encryption key** value and a **value *V*** that will be updated after each block of pseudorandom numbers is generated.

- In the **CTR case**, the value of ***V* is incremented by 1** after each encryption. In the case of **OFB**, the **value of *V* is updated to equal** the value of the preceding PRNG block.

- In both cases, pseudorandom bits are produced one block at a time

PRNG Using Block Cipher Modes: Algorithms

The CTR algorithm for PRNG can be summarized as follows.

```
while (len (temp) < requested_number_of_bits) do
        V = (V + 1) mod 2^128.
        output_block = E(Key, V)
        temp = temp || ouput_block
```

The OFB algorithm can be summarized as follows.

```
while (len (temp) < requested_number_of_bits) do
        V = E(Key, V)
        temp = temp || V
```

Experiment

- A random bit sequence of 256 bits was obtained from random.org, which uses three radios tuned between stations to pick up atmospheric noise.

Key:	`cfb0ef3108d49cc4562d5810b0a9af60`
V:	`4c89af496176b728ed1e2ea8ba27f5a4`

- The total number of one bits in the 256-bit seed is 124, or a fraction of 0.48, which is reassuringly close to the ideal of 0.5.

Experiment

Example Results for PRNG Using OFB

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
5e17b22b14677a4d66890f87565eae64	0.51	0.52
fd18284ac82251dfb3aa62c326cd46cc	0.47	0.54
c8e545198a758ef5dd86b41946389bd5	0.50	0.44
fe7bae0e23019542962e2c52d215a2e3	0.47	0.48
14fdf5ec99469598ae0379472803accd	0.49	0.52
6aeca972e5a3ef17bd1a1b775fc8b929	0.57	0.48
f7e97badf359d128f00d9b4ae323db64	0.55	0.45

Example Results for PRNG Using CTR

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
60809669a3e092a01b463472fdcae420	0.41	0.41
d4e6e170b46b0573eedf88ee39bff33d	0.59	0.45
5f8fcfc5deca18ea246785d7fadc76f8	0.59	0.52
90e63ed27bb07868c753545bdd57ee28	0.53	0.52
0125856fdf4a17f747c7833695c52235	0.50	0.47
f4be2d179b0f2548fd748c8fc7c81990	0.51	0.48
1151fc48f90eebac658a3911515c3c66	0.47	0.45

- The results indicate that the output is split roughly equally between zero and one bits.

- The third column shows the fraction of bits that match between adjacent blocks. If this number differs substantially from 0.5, that suggests a correlation between blocks, which could be a security weakness. The results suggest no correlation.

ANSI X9.17 PRNG

- .

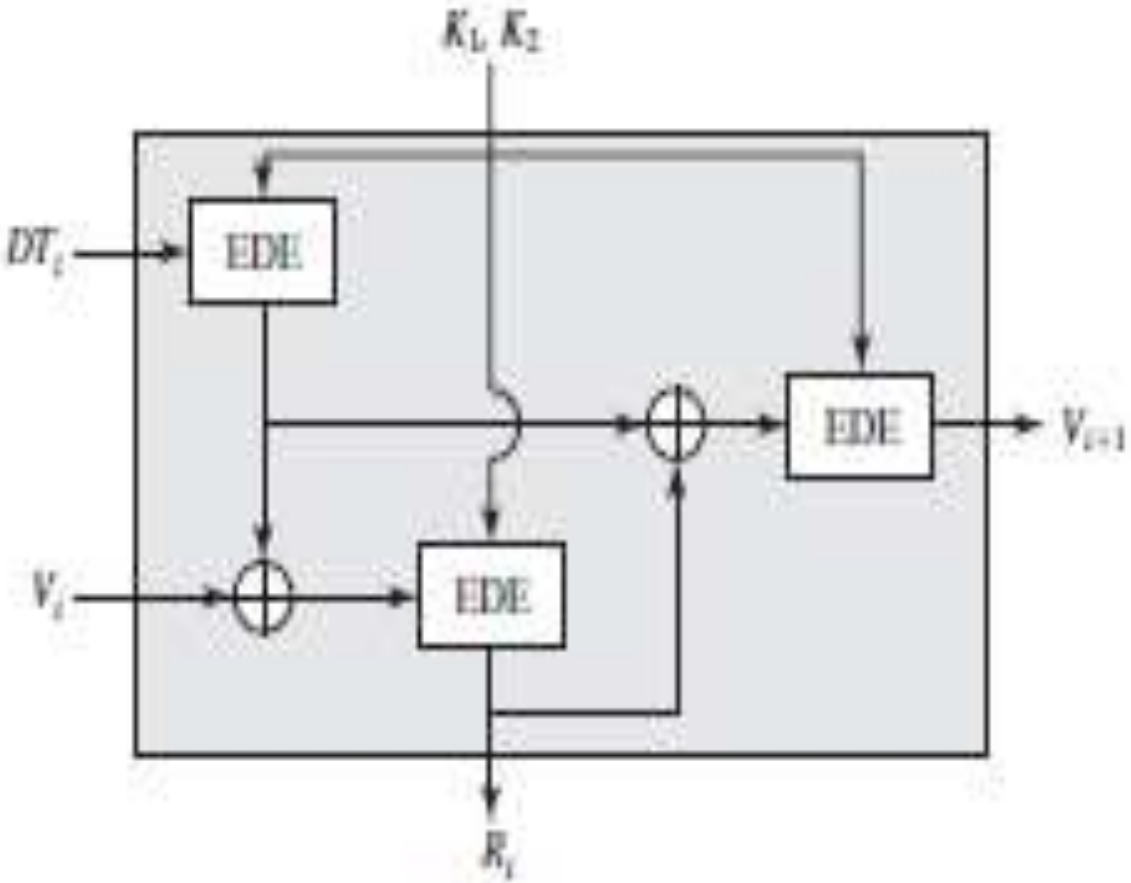
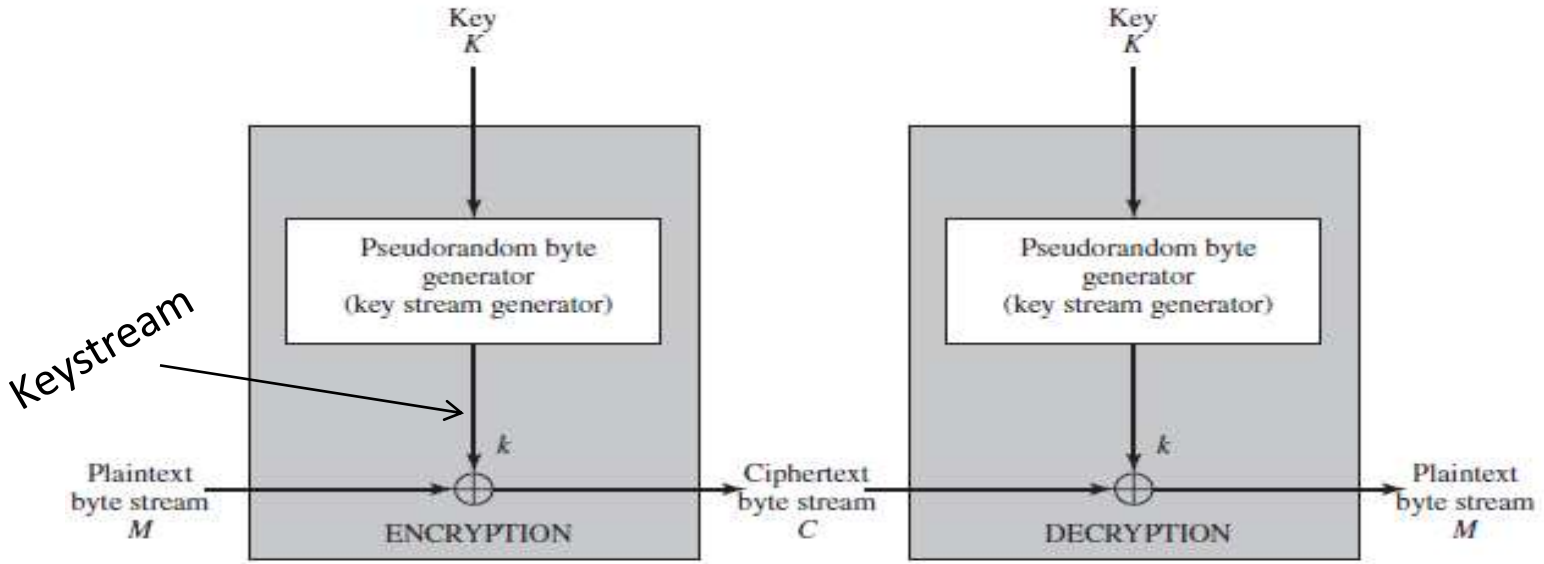


Figure 7.4 ANSI X9.17 Pseudorandom Number Generator

STREAM CIPHERS

- A typical stream cipher encrypts plaintext one byte at a time.
- Figure 7.5 is a representative diagram of stream cipher structure.
- In this structure, a key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random. The output of the generator, called a **keystream**, is combined one byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation.



Stream Cipher Diagram

PRNG using STREAM CIPHERS

For example, if the next byte generated by the generator is 01101100 and the next plaintext byte is 11001100, then the resulting ciphertext byte is

```
  11001100  plaintext
⊕ 01101100  key stream
  10100000  ciphertext
```

```
  10100000  ciphertext
⊕ 01101100  key stream
  11001100  plaintext
```

STREAM CIPHERS

- The stream cipher is similar to the one-time pad.
- The difference is that a **one-time pad uses a genuine random number stream**, whereas a **stream cipher uses a pseudorandom number stream**.
- The following important design considerations for a stream cipher:
  - The encryption sequence should have a large period. (To avoid stream of bits that eventually repeats)
  - The keystream should approximate the properties of a true random number stream as close as possible. (approximately equal number of 1s and 0s.)
  - Key must withstand bruteforce attacks.

Streamcipher or Blockcipher ?

- **Stream cipher**

  - Typically **faster** and use far **less code** than do block ciphers.

  - If two plaintexts are encrypted with the same key using a stream cipher, then **cryptanalysis** is often quite *simple*.

  - For applications that require encryption/decryption of a stream of data, such as over a data communications channel or a browser/Web link, a stream cipher might be the better alternative.

- **Block cipher:**

  - Advantage of a block cipher is that you can **reuse keys**.

  - For applications that deal with blocks of data, such as file transfer, e-mail, and database, block ciphers may be more appropriate.

However, either type of cipher can be used in virtually any application.

Pseudorandom Number Generation using RC4

- RC4 is a stream cipher designed in 1987 by Ron Rivest for RSA Security.
- It is a variable key size stream cipher with byte-oriented operations.

- The algorithm is based on the use of a random permutation.

- RC4 is used in the Secure Sockets Layer/Transport Layer Security (SSL/TLS) standards that have been defined for communication between Web browsers and servers. It is also used in the Wired Equivalent Privacy (WEP) protocol and the newer WiFi Protected Access (WPA) protocol that are part of the IEEE 802.11 wireless LAN standard.

- A variable- length key of from 1 to 256 bytes (8 to 2048 bits) is used to initialize a 256-byte state vector S, with elements . At all times, contains a permutation of all 8-bit numbers from 0 through 255. For encryption and decryption, a byte (see Figure 7.5) is generated from S by selecting one of the 255 entries in a systematic fashion. As each value of is generated, the entries in S are once again permuted.

- RC4 steps:
  - **Initialization of S**
  - **Initial Permutation of S**
  - **Stream Generation**

Pseudorandom Number Generation using RC4

- **RC4 steps:**

Step 1: Initialization of S :

To begin, the entries of are set equal to the values from 0 through 255 in ascending order; that is, . A temporary vector, T, is also created. If the length of the key is 256 bytes, then is transferred to T. Otherwise, for a key of length *keylen* bytes, the first *keylen* elements of T are copied from K, and then K is repeated as many times as necessary to fill out T.

These preliminary operations can be summarized as

/* **Initialization** */
for i = 0 **to** 255 **do**
S[i] = i;
T[i] = K[i **mod** keylen];

Step 2: Initial Permutation of S

```
/* Initial Permutation of S */
j = 0;
for i = 0 to 255 do
    j = (j + S[i] + T[i]) mod 256;
Swap (S[i], S[j]);
```

Pseudorandom Number Generation using RC4

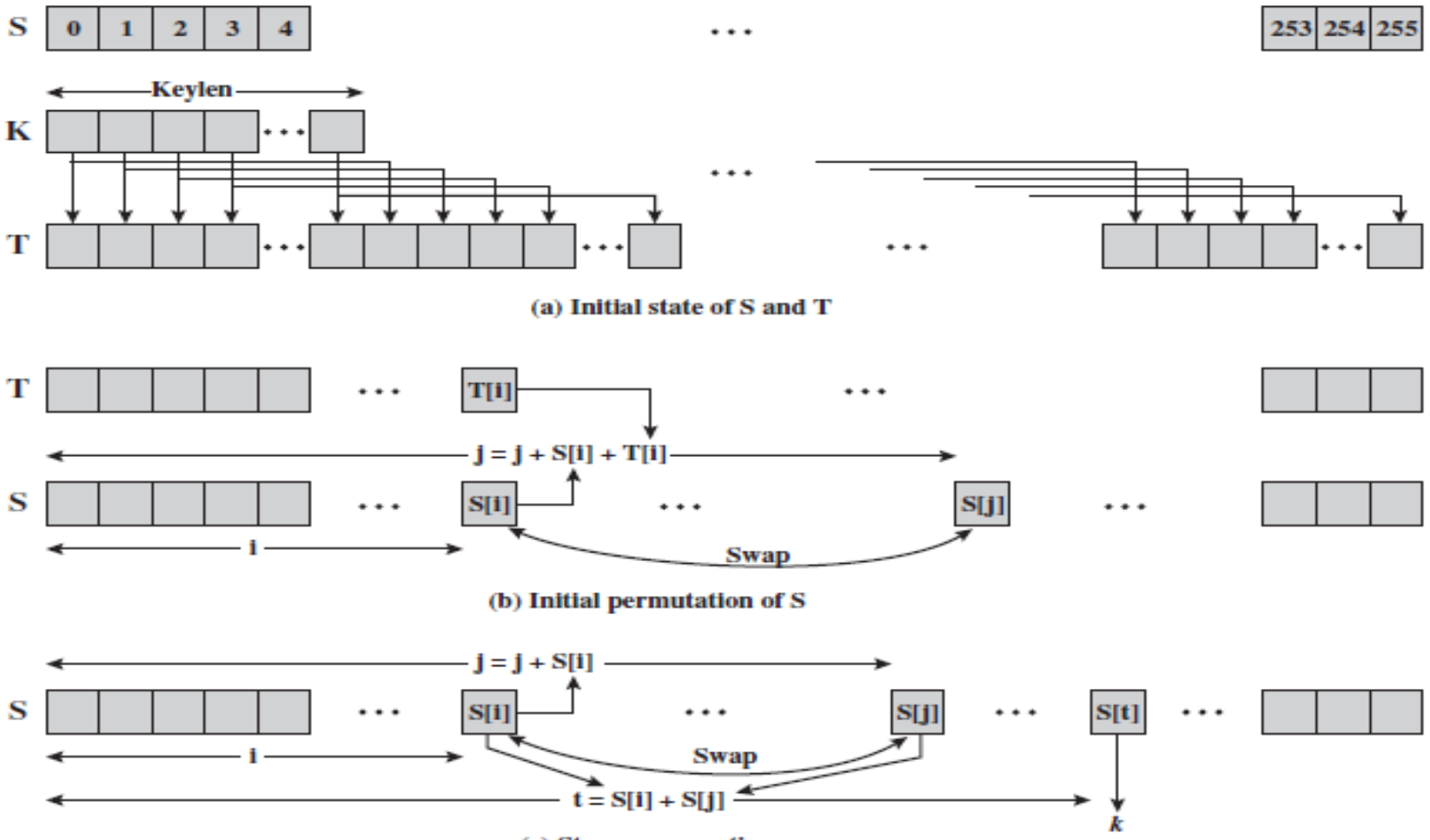
Step 3: Stream Generation:

- Once the S vector is initialized, the input key is no longer used. Stream generation involves cycling through all the elements of s[i] , and for eachs[i] , swapping with another byte in S according to a scheme dictated by the current configuration of S.
- After is reached, the process continues, starting over again at :

```
/* Stream Generation */
i, j = 0;
  while (true)
     i = (i + 1) mod 256;
     j = (j + S[i]) mod 256;
 Swap (S[i], S[j]);
    t = (S[i] + S[j]) mod 256;
    k = S[t];
```

- To encrypt, XOR the value with the next byte of plaintext. To decrypt,XOR the value with the next byte of ciphertext.

Pseudorandom Number Generation using RC4
Strength of RC4



RC4

- RC4 steps:
  - **Initialization of S**
  - **Initial Permutation of S**
  - **Stream Generation**

Instead of 256 bytes, we will use 8x3 bits i.e state vector *S* is 8x3 bits.

We will operate on 3-bits of plaintext at a time Since *S* can take 0 to 7, which can be represented as 3 bits.

Links to refer the examples of RC4 :

- **https://www.youtube.com/watch?v=1UP56WM4ook (Good Example RC4)**

- https://sandilands.info/sgordon/teaching/reports/rc4-example.pdf (Good Example RC4)

Module 4
MESSAGE AUTHENTICATION CODES

Message authentication

- Message authentication is a mechanism or service used to **verify the integrity of a message**.

- Message authentication assures that **data received are exactly as sent** by (i.e., contain no modification, insertion, deletion, or replay) and that the purported identity of the sender is valid.

MESSAGE AUTHENTICATION REQUIREMENTS

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

1. **Disclosure:** Release of message contents to any person or process not possessing the appropriate cryptographic key.
2. **Traffic analysis:** Discovery of the pattern of traffic between parties.
3. **Masquerade:** Insertion of messages into the network from a fraudulent source. This includes the creation of messages by an opponent that are purported to come from an authorized entity.
4. **Content modification:** Changes to the contents of a message, including insertion, deletion, transposition, and modification.
5. **Sequence modification:** Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
6. **Timing modification:** Delay or replay of messages.
7. **Source repudiation:** Denial of transmission of message by source.
8. **Destination repudiation:** Denial of receipt of message by destination.

Measures

Attack	Measure
Disclosure	message confidentiality
Traffic analysis	message confidentiality
Masquerade	message authentication
Content modification	message authentication
Sequence modification	message authentication
Timing modification	message authentication
Source repudiation	Digital signatures
Destination repudiation	Digital signatures

Message authentication is a procedure to verify that received messages come from the alleged source and have not been altered.

MESSAGE AUTHENTICATION FUNCTIONS

Types of functions that may be used to produce an authenticator. These may be grouped into *three classes*.

- **Message encryption:** The ciphertext of the entire message serves as its authenticator.

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

  M || H(M)

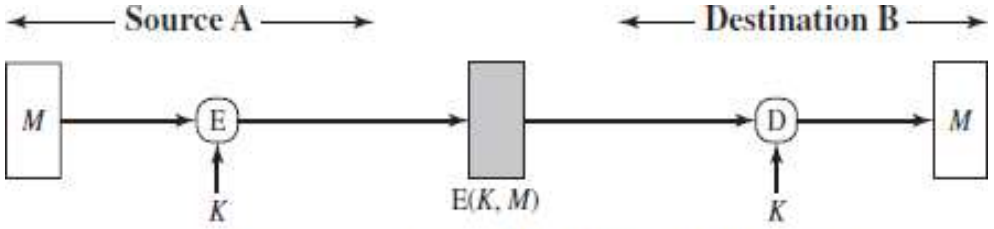
- **Message authentication code (MAC):** A function of the **message and a secret key** that produces a fixed-length value that serves as the authenticator.

- MAC = H (M || Secret Key)

Message Encryption

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

SYMMETRIC ENCRYPTION



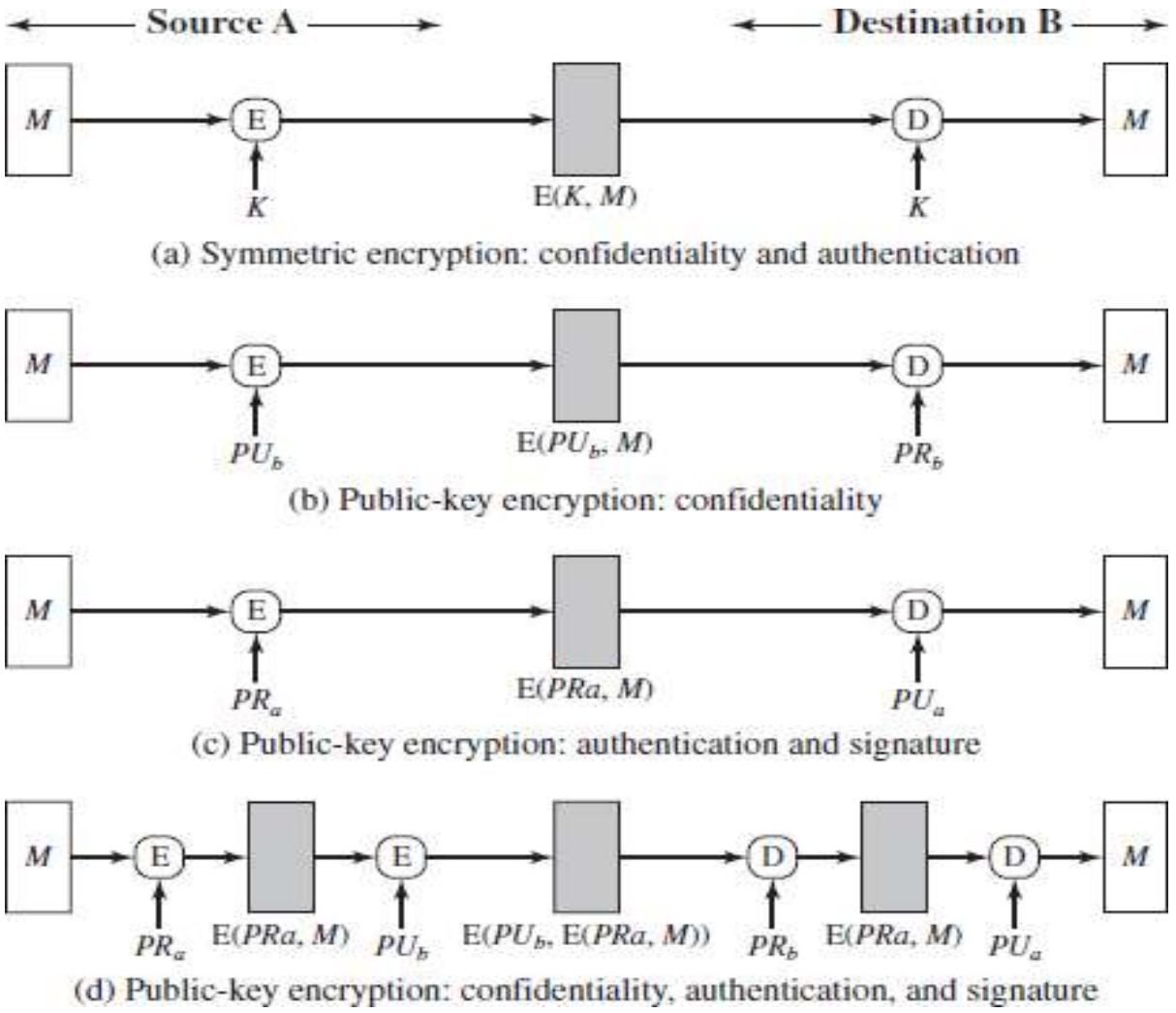
(a) Symmetric encryption: confidentiality and authentication

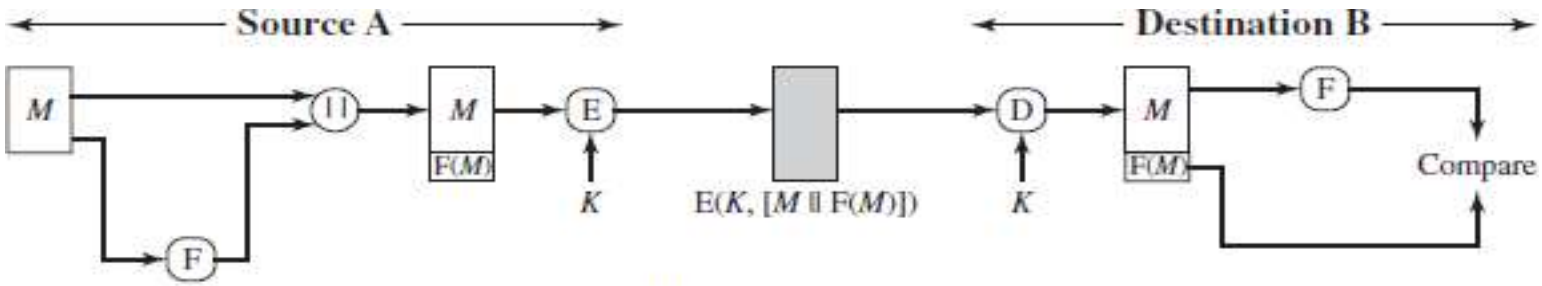
If no other party knows the key, then confidentiality is provided

5 users n(n-1)/2= 10

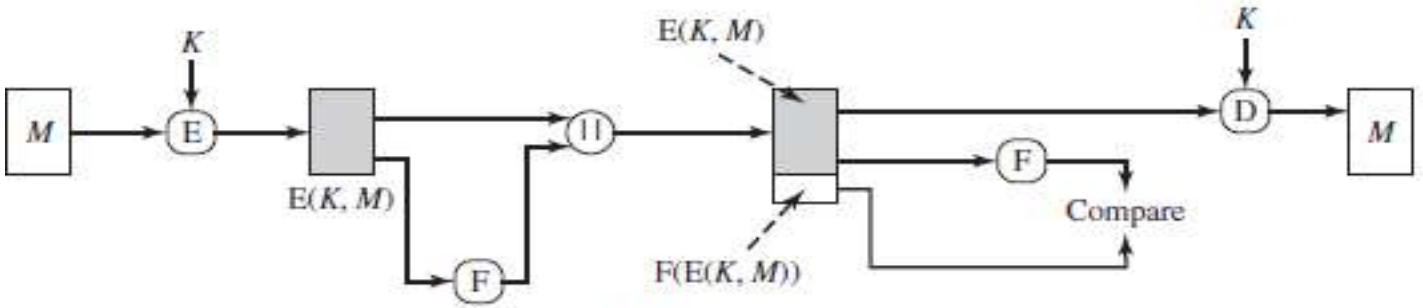
How many keys are required?

Basic Uses of Message Encryption





(a) Internal error control



(b) External error control

Figure 12.2 Internal and External Error Control

error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption function: CRC

Message Authentication Code (MAC)

- Used for authentication
- MAC- It is an algorithm/function
- MAC: A function of the message and a secret key that produces a fixed-length value that serves as the authenticator.
- Also known as –keyed hash function

$$MAC = MAC(K, M)$$

where

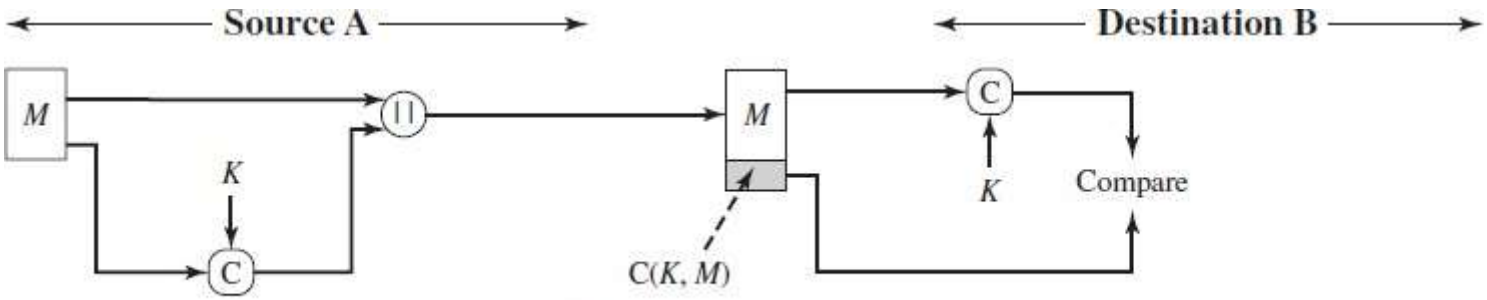
M = input message
C = MAC function
K = shared secret key
MAC = message authentication code

Message Authentication Code

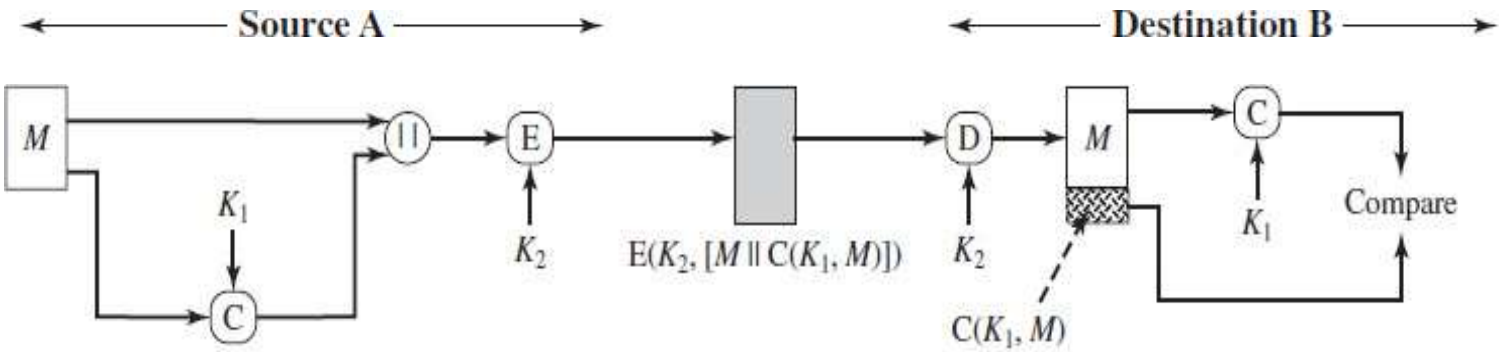
- I have a block of data, perhaps a video file.
- I want to send it to you and I would like you to be able to prove that it was unmodified in transit, and that I was the one who sent it to you.

$$MAC = H(key \,||\, message)$$

Basic uses of MAC



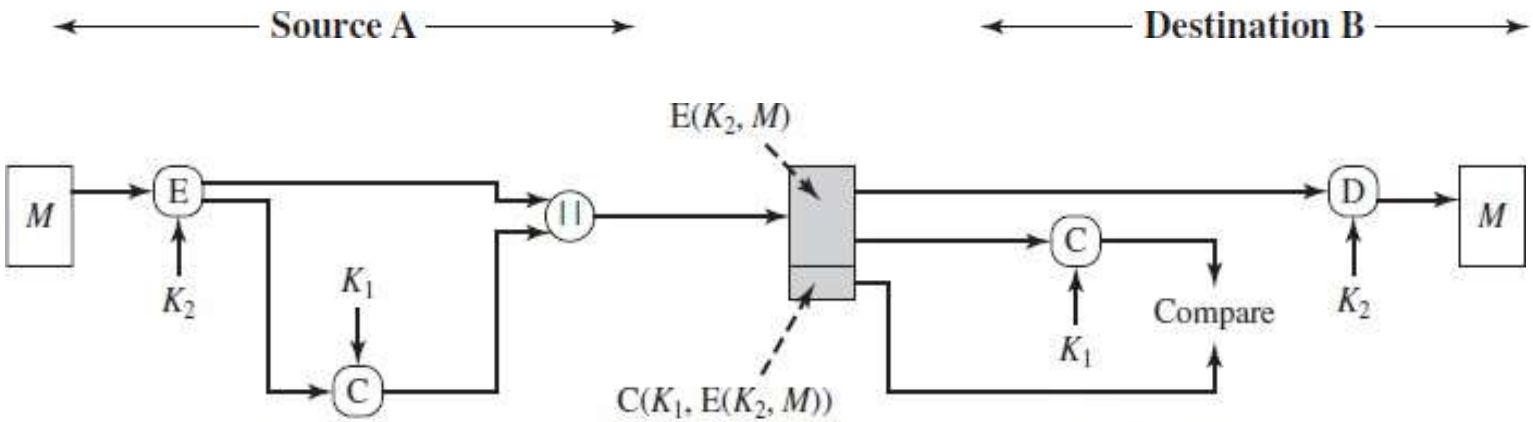
(a) Message authentication



(b) Message authentication and confidentiality; authentication tied to plaintext

C= MAC function

Basic uses of MAC



(c) Message authentication and confidentiality; authentication tied to ciphertext

Benefits of MAC

- The receiver is assured that the message has not been altered.
- The receiver is assured that the message is from the alleged sender.
- If the message includes a sequence number, then the receiver can be assured of the proper sequence.

Benefits of MAC

- There are a number of applications in which the same message is broadcast to a number of destinations.

  Example: notification to users that the network is now unavailable or an alarm signal in a military control center.

  It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication code.

- Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages. Authentication is carried out on a selective basis, messages being chosen at random for checking.

Benefits of MAC

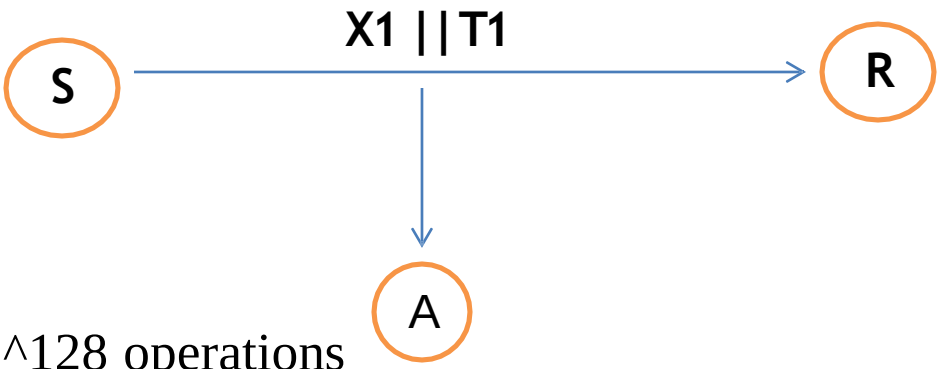
- Authentication of a computer program in plaintext is an attractive service. The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources. However, if a message authentication code were attached to the program, it could be checked whenever assurance was required of the integrity of the program.

Security of MAC

Brute Force Attack on Key

- Attacker knows $[x_1, T_1]$ where $T_1 = MAC(K, x_1)$
- Key size of k bits: brute force on key, 2^k
- But . . . many tags match T_1
- For keys that produce tag T_1, try again with $[x_2, T_2]$
- Effort to find K is approximately 2^k

X1-message
K- key (Ex: 128 bits)
T1-Tag



Brute force attack takes : 2^128 operations

Security of MAC

Brute Force Attack on MAC value

- For x_m, find T_m without knowing K
- Similar effort required as one-way/weak collision resistant property for hash functions
- For n bit MAC value length, effort is 2^n

Effort to break MAC: $\min(2^k, 2^n)$

X1-message
K- key (Ex: 128 bits)
MAC – 30 bits output

2^30 operations required

Real time MAC examples

- Data Authentication Algorithm (DAA): based on DES; considered insecure
- Cipher-Based Message Authentication Code (CMAC): mode of operation used with Triple-DES and AES
- OMAC, PMAC, UMAC, VMAC, ...

Note

- MD5
  - Produces a 128-bit hash
  - Collisions can be found in ~2^21 hashes
- SHA1
  - 160-bit hash
  - Collisions can be found in 2^61 hashes
- SHA2
  - Actually 4 different hash functions: SHA-224, SHA-256, SHA-384, SHA-512
  - Minor attacks, but still good
- SHA3
  - Just chosen as a new NIST standard
  - No known attacks

HMAC

- What's wrong with MAC?

  H(message1) == H(message2)

Collisions

- Then this is also true:

  H(key || message1) == H(key || message2)

HMAC: MAC function derived from cryptographic hashfunctions.

- HMAC solves this problem by using the following construction:

  HMAC = **H(key1 || H(key2 || message))**

$$\mathrm{HMAC}(K, M) = \mathrm{H}((K \oplus \mathrm{opad}) || \mathrm{H}((K \oplus \mathit{ipad}) || M))$$

HMAC

- HMAC = **H(key1 || H(key2 || message))**

$$\text{HMAC}(K, M) = \text{H}((K \oplus \text{opad}) || \text{H}((K \oplus \textit{ipad}) || M))$$

Ipad = 00110110
opad= 01011100

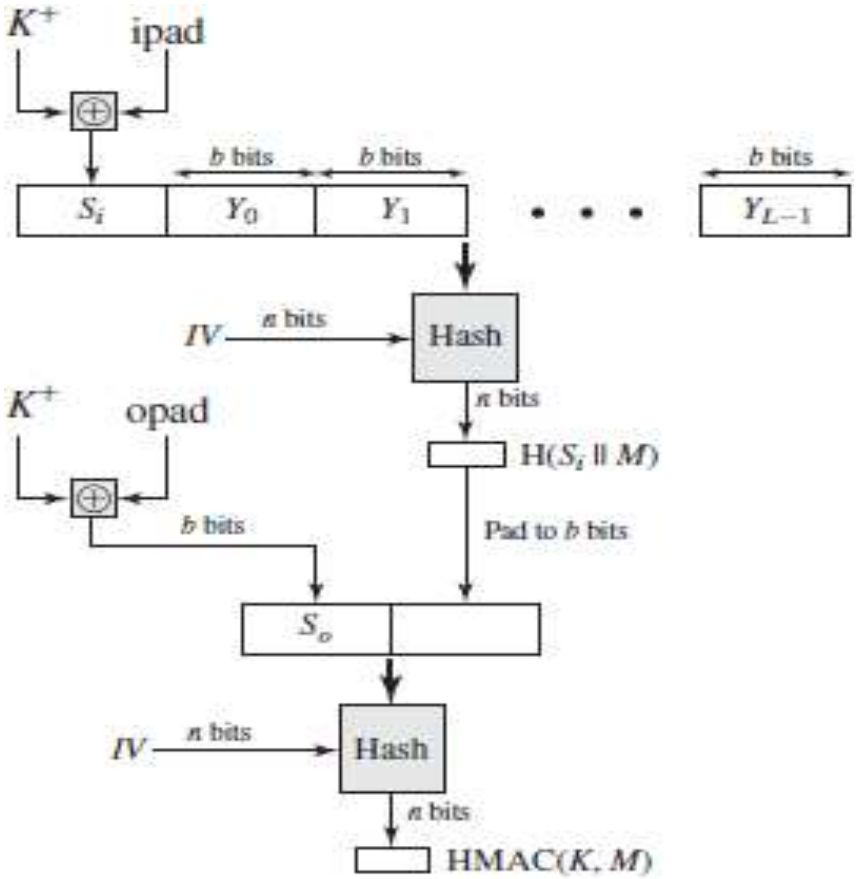


Figure 12.5 HMAC Structure

H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)

IV = initial value input to hash function

M = message input to HMAC (including the padding specified in the embedded hash function)

Y_i = i th block of $M, 0 \le i \le (L-1)$

L = number of blocks in M

b = number of bits in a block

n = length of hash code produced by embedded hash function

K = secret key; recommended length is $\ge n$; if key length is greater than b, the key is input to the hash function to produce an n-bit key

UNIT 4
DIGITAL SIGNATURES
KEY MANAGEMENT & KEY DISTRIBUTION

Digital Signatures

- A digital signature is an authentication mechanism that enables the creator of a message to attach a code that acts as a signature. Typically the signature is formed by taking the hash of the message and encrypting the message with the creator's private key. The signature guarantees the source and integrity of the message.

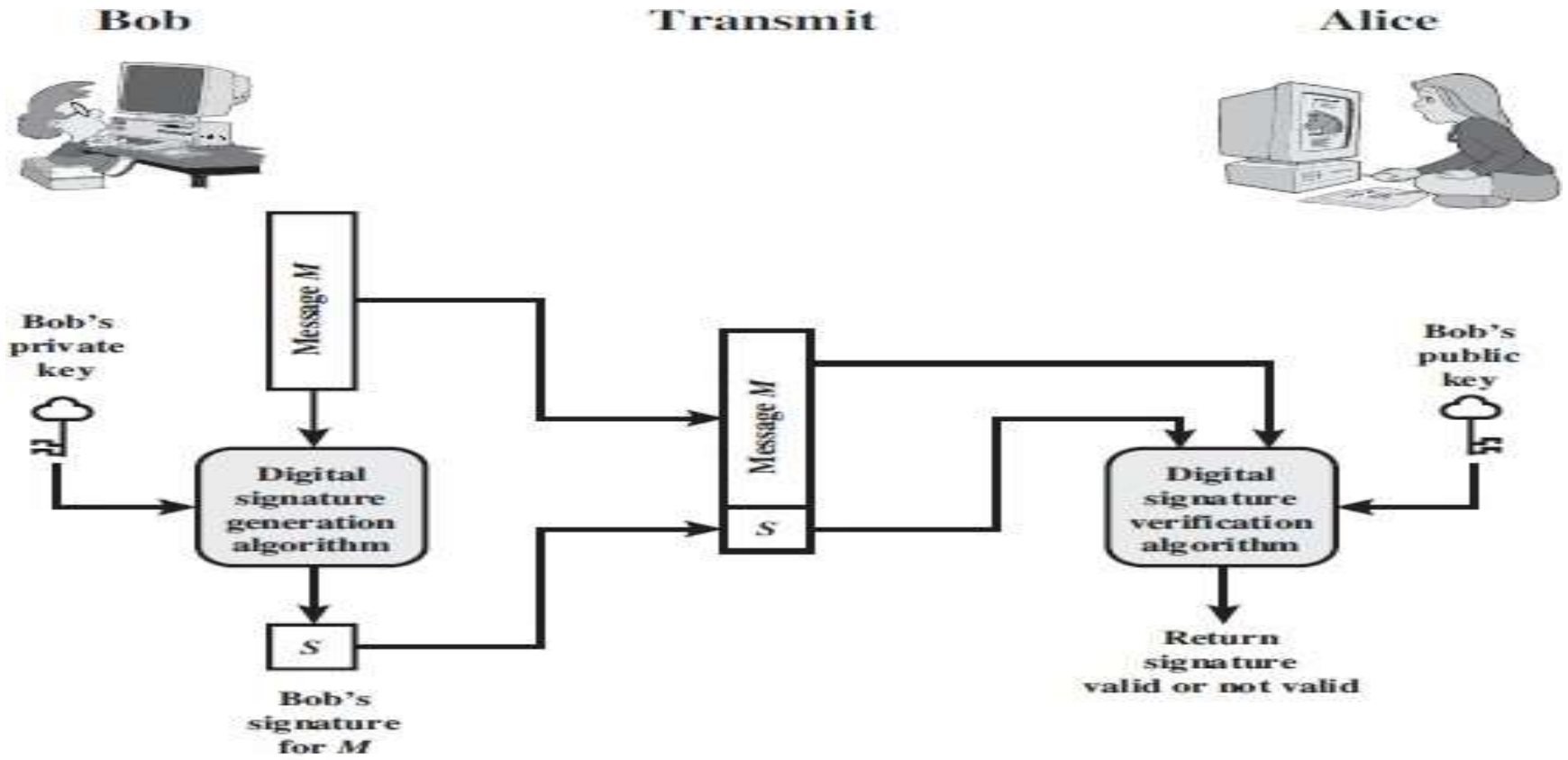
- The digital signature standard (DSS) is an NIST standard that uses the secure hash algorithm (SHA).

- The digital signature provides a set of security capabilities that would be difficult to implement in any other way.

Digital Signatures

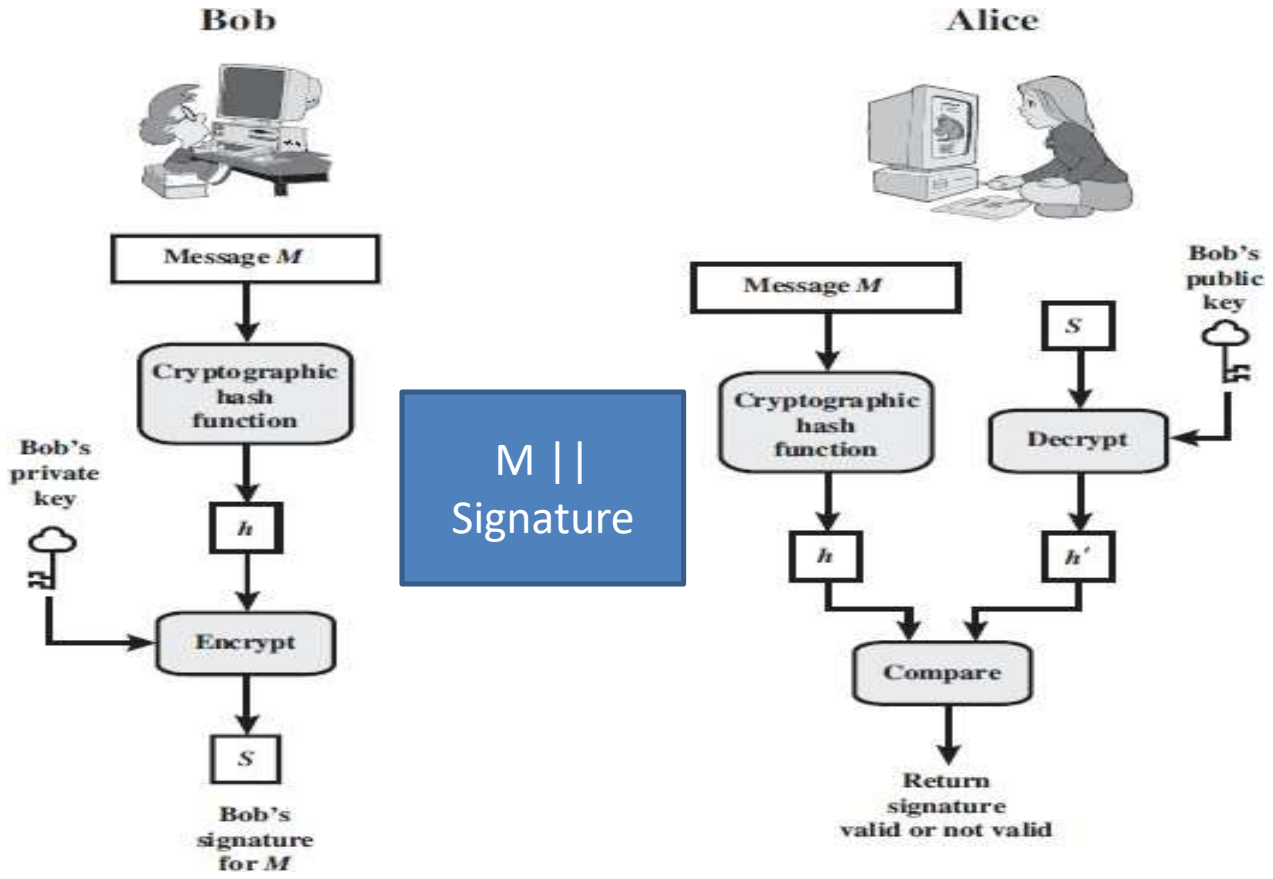
Figure 13.1(below figure) is a generic model of the process of making and using digital signatures.

- Bob can sign a message using a digital signature generation algorithm.
- The inputs to the algorithm are the message and Bob's private key. Any other user, say Alice, can verify the signature using a verification algorithm, whose inputs are the message, the signature, and Bob's public key.



Generic Model of Digital Signature Process

Simplified Depiction of Essential Elements of Digital Signature Process



In simplified terms, the essence of the digital signature mechanism is shown in Figure 13.2. (Above figure)

The **digital signature** must have the following **properties:**

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

Thus, the digital signature function includes the authentication function.

Attacks and Forgeries

[GOLD88] lists the following types of attacks, in order of increasing severity. Here A denotes the user whose signature method is being attacked, and C denotes the attacker.

- **Key-only attack:** C only knows A's public key.
- **Known message attack:** C is given access to a set of messages and their signatures.
- **Generic chosen message attack:** C chooses a list of messages before attempting to breaks A's signature scheme, independent of A's public key. C then obtains from A valid signatures for the chosen messages. The attack is generic, because it does not depend on A's public key; the same attack is used against everyone.
- **Directed chosen message attack:** Similar to the generic attack, except that the list of messages to be signed is chosen after C knows A's public key but before any signatures are seen.
- **Adaptive chosen message attack:** C is allowed to use A as an "oracle." This means the A may request signatures of messages that depend on previously obtained message–signature pairs.

[GOLD88] then defines success at breaking a signature scheme as an outcome in which C can do any of the following with a non-negligible probability:

- **Total break:** C determines A's private key.
- **Universal forgery:** C finds an efficient signing algorithm that provides an equivalent way of constructing signatures on arbitrary messages.
- **Selective forgery:** C forges a signature for a particular message chosen by C.
- **Existential forgery:** C forges a signature for at least one message. C has no control over the message. Consequently, this forgery may only be a minor nuisance to A

Digital Signature Requirements

On the basis of the properties and attacks just discussed, we can formulate the following requirements for a digital signature.

- The signature must be a bit pattern that depends on the message being signed.
- The signature must use some information unique to the sender to prevent both forgery and denial.
- It must be relatively easy to produce the digital signature.
- It must be relatively easy to recognize and verify the digital signature.
- It must be computationally infeasible to forge a digital signature, either by constructing a new message for an existing digital signature or by constructing a fraudulent digital signature for a given message.
- It must be practical to retain a copy of the digital signature in storage.

A secure hash function, embedded in a scheme such as that of Figure 13.2, provides a basis for satisfying these requirements. However, care must be taken in the design of the details of the scheme.

Direct Digital Signature :

Self Study

Digital Signature Standard

The National Institute of Standards and Technology (NIST) has published Federal Information Processing Standard FIPS 186, known as the Digital Signature Standard (DSS). The DSS makes use of the Secure Hash Algorithm (SHA) and presents a new digital signature technique, the **Digital Signature Algorithm (DSA)**.

The DSS Approach

The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique.

Figure 13.3 contrasts the DSS approach for generating digital signatures to that used with RSA.

RSA approach, the message to be signed is input to a hash function that produces a secure hash code of fixed length. This hash code is then encrypted using the sender's private key to form the signature. Both the message and the signature are then transmitted. The recipient takes the message and produces a hash code. The recipient also decrypts the signature using the sender's public key. If the calculated hash code matches the decrypted signature, the signature is accepted as valid. Because only the sender knows the private key, only the sender could have produced a valid signature.

DSS approach also makes use of a hash function. The hash code is provided as input to a signature function along with a random number generated for this particular signature. The signature function also depends on the sender's private key PRa and a set of parameters known to a group of communicating principals. We can consider this set to constitute a global public key (PU_G) The result is a signature consisting of two components, labeled *s* and *r*.

Digital Signature Standard

At the receiving end, the hash code of the incoming message is generated. This plus the signature is input to a verification function. The verification function also depends on the global public key as well as the sender's public key PUa, which is paired with the sender's private key. The output of the verification function is a value that is equal to the signature component if the signature is valid. The signature function is such that only the sender, with knowledge of the private key, could have produced the valid signature.

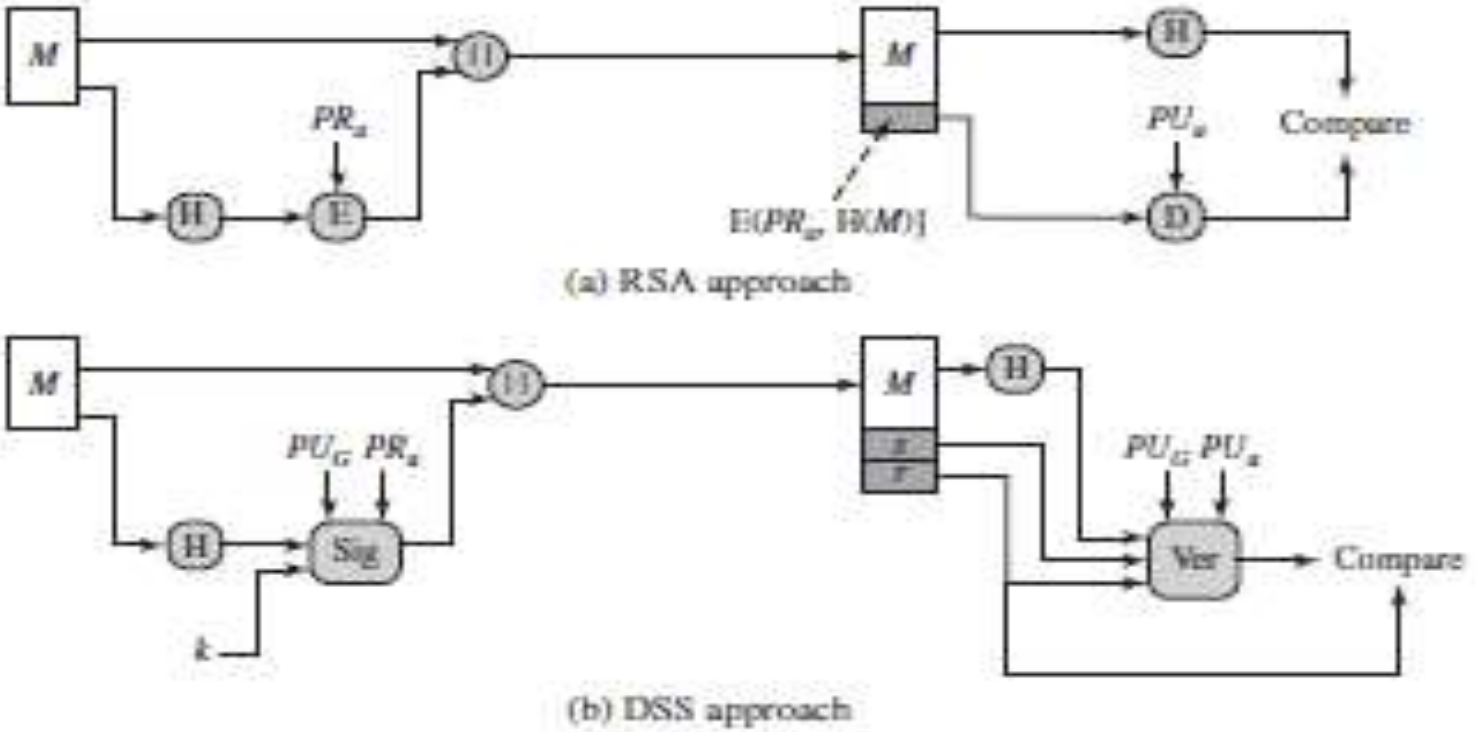


Figure 13.3 Two Approaches to Digital Signatures

Digital Signature Algorithm

Global Public-Key Components

p prime number where $2^{L-1} < p < 2^L$ for $512 \le L \le 1024$ and L a multiple of 64; i.e., bit length of between 512 and 1024 bits in increments of 64 bits

q prime divisor of $(p-1)$, where $2^{159} < q < 2^{160}$; i.e., bit length of 160 bits

$g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < (p-1)$ such that $h^{(p-1)/q} \bmod p > 1$

User's Private Key

x random or pseudorandom integer with $0 < x < q$

User's Public Key

$y = g^x \bmod p$

User's Per-Message Secret Number

k = random or pseudorandom integer with $0 < k < q$

Signing

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

$s = [k^{-1}(H(M) + xr)] \bmod q$

Signature = (r, s)

Verifying

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

$u_1 = [H(M')w] \bmod q$

$u_2 = (r')w \bmod q$

$v = [(g^{u_1} y^{u_2}) \bmod p] \bmod q$

TEST: $v = r'$

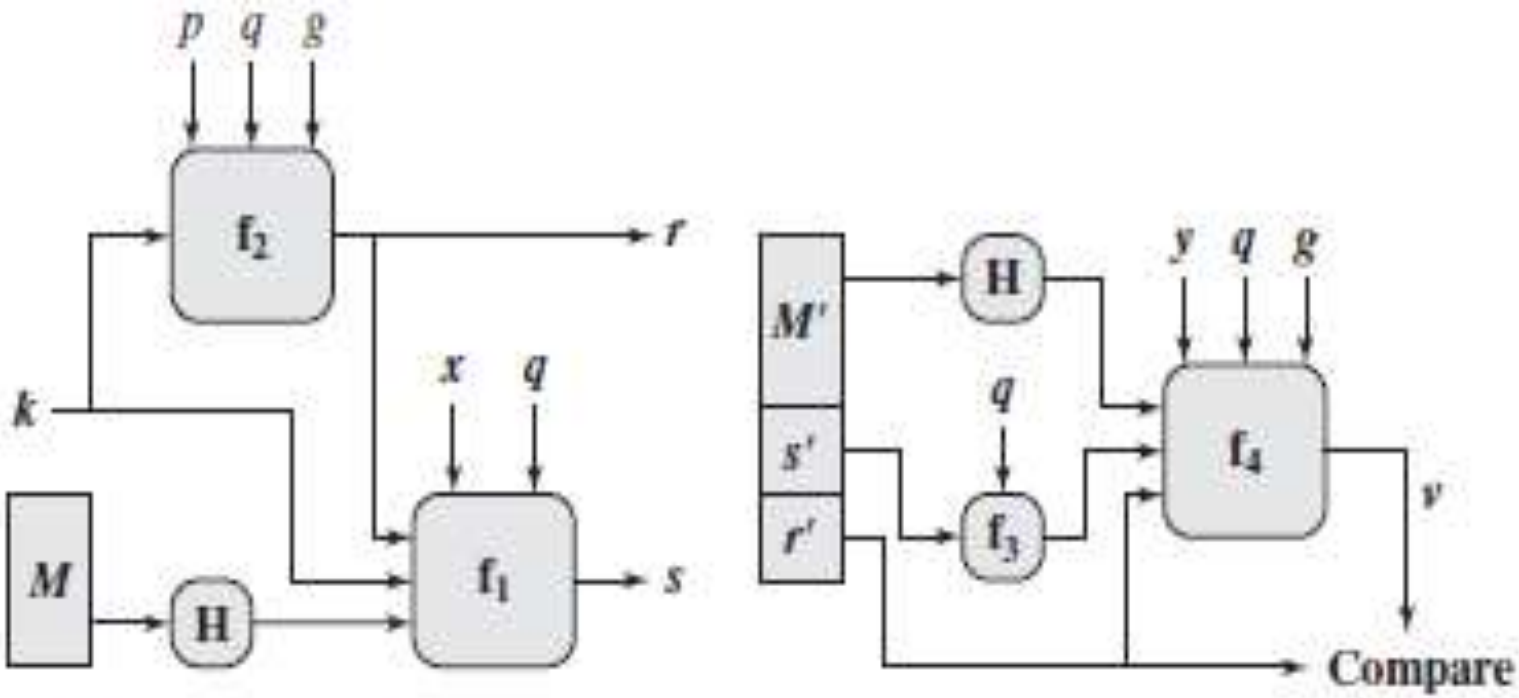
M = message to be signed

$H(M)$ = hash of M using SHA-1

M', r', s' = received versions of M, r, s

Figure 13.4 The Digital Signature Algorithm (DSA)

Digital Signature Algorithm



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

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

(a) Signing

$w = f_5(s', q) = (s')^{-1} \bmod q$

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

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

(b) Verifying

Figure 13.5 DSS Signing and Verifying

Cryptography and Network Security Chapter 14

Fifth Edition

by William Stallings

Lecture slides by Lawrie Brown

Chapter 14 – Key Management and Distribution

No Singhalese, whether man or woman, would venture out of the house without a bunch of keys in his hand, for without such a talisman he would fear that some devil might take advantage of his weak state to slip into his body.

—The Golden Bough, Sir James George Frazer

Key Management and Distribution

- topics of cryptographic key management / key distribution are complex
  - cryptographic, protocol, & management issues
- symmetric schemes require both parties to share a common secret key
- public key schemes require parties to acquire valid public keys
- have concerns with doing both

Road Map

- symmetric key distribution using symmetric encryption
- symmetric key distribution using public-key encryption
- distribution of public keys
  - announcement, directory, authrority, CA
- X.509 authentication and certificates
- public key infrastructure (PKIX)

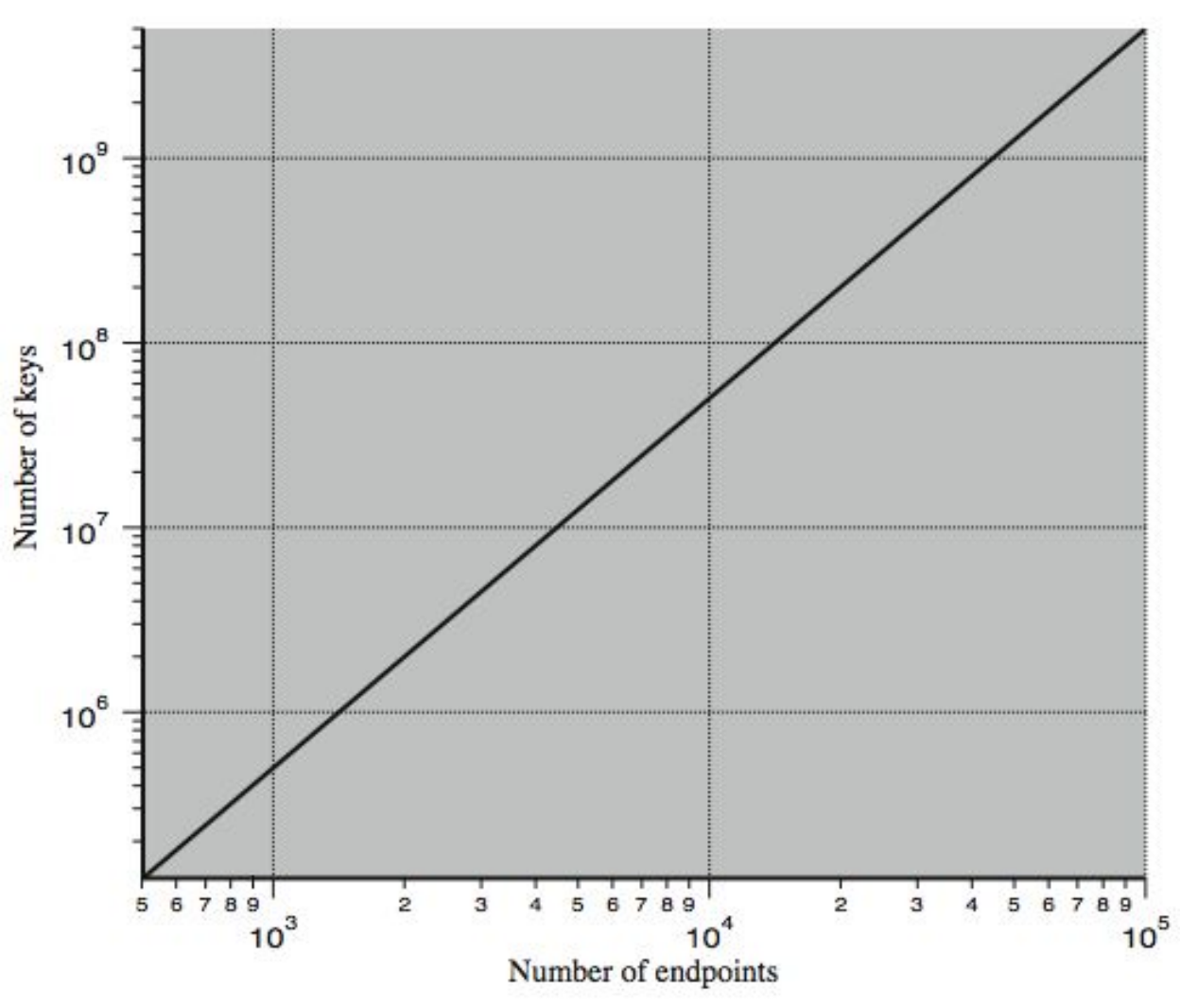
Key Distribution

- symmetric schemes require both parties to share a common secret key
- issue is how to securely distribute this key
- whilst protecting it from others
- frequent key changes can be desirable
- often secure system failure due to a break in the key distribution scheme

Key Distribution

- given parties A and B have various **key distribution** alternatives:
  1. A can select key and physically deliver to B
  2. third party can select & deliver key to A & B
  3. if A & B have communicated previously can use previous key to encrypt a new key
  4. if A & B have secure communications with a third party C, C can relay key between A & B

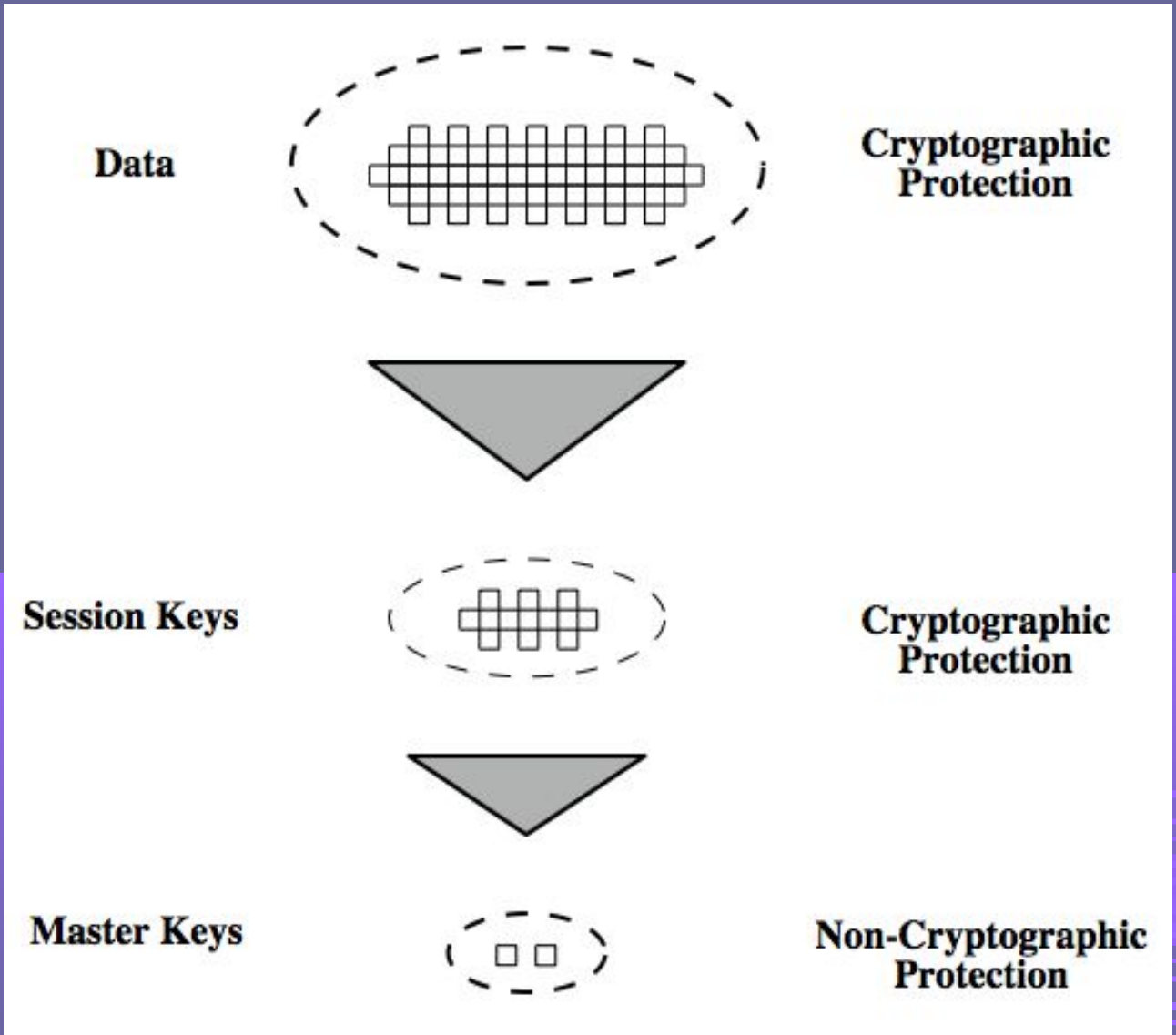
Key Distribution Task



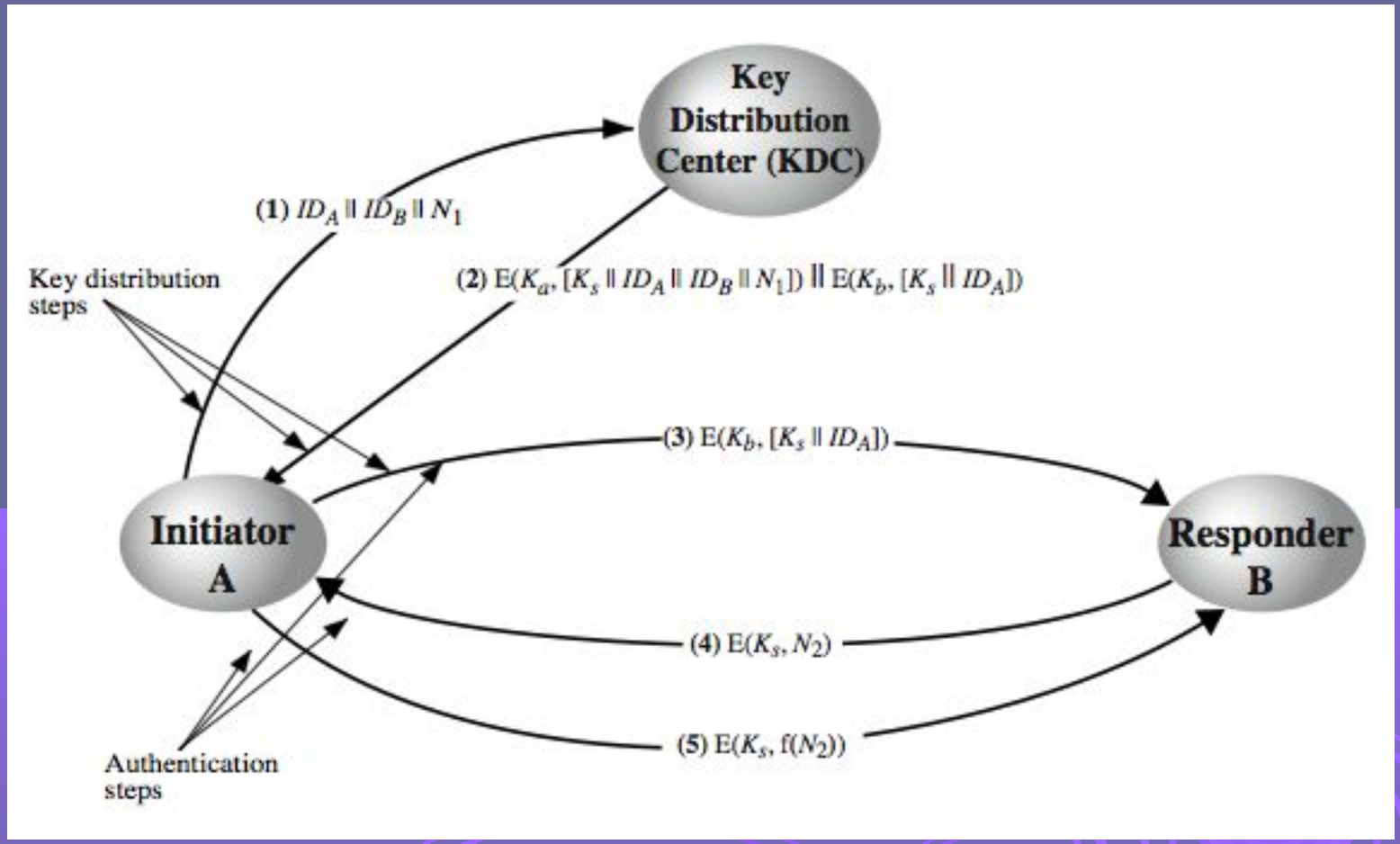
Key Hierarchy

- typically have a hierarchy of keys
- session key
  - temporary key
  - used for encryption of data between users
  - for one logical session then discarded
- master key
  - used to encrypt session keys
  - shared by user & key distribution center

Key Hierarchy



Key Distribution Scenario



Key Distribution Issues

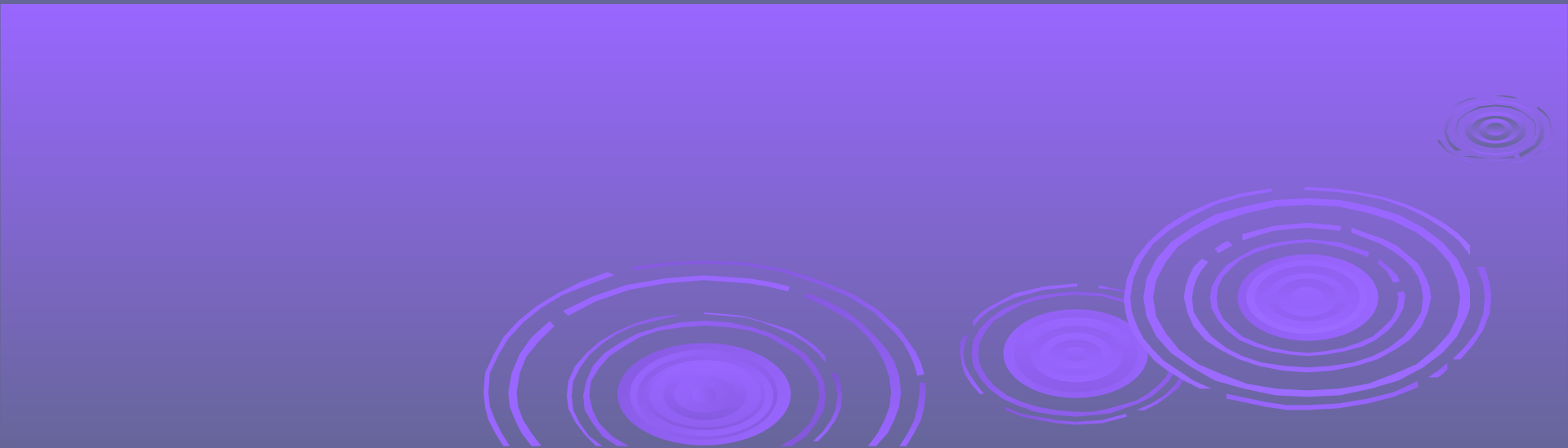
- hierarchies of KDC's required for large networks, but must trust each other
- session key lifetimes should be limited for greater security
- use of automatic key distribution on behalf of users, but must trust system
- use of decentralized key distribution
- controlling key usage

Road Map

- symmetric key distribution using symmetric encryption
- symmetric key distribution using public-key encryption
- distribution of public keys
  - announcement, directory, authrority, CA
- X.509 authentication and certificates
- public key infrastructure (PKIX)

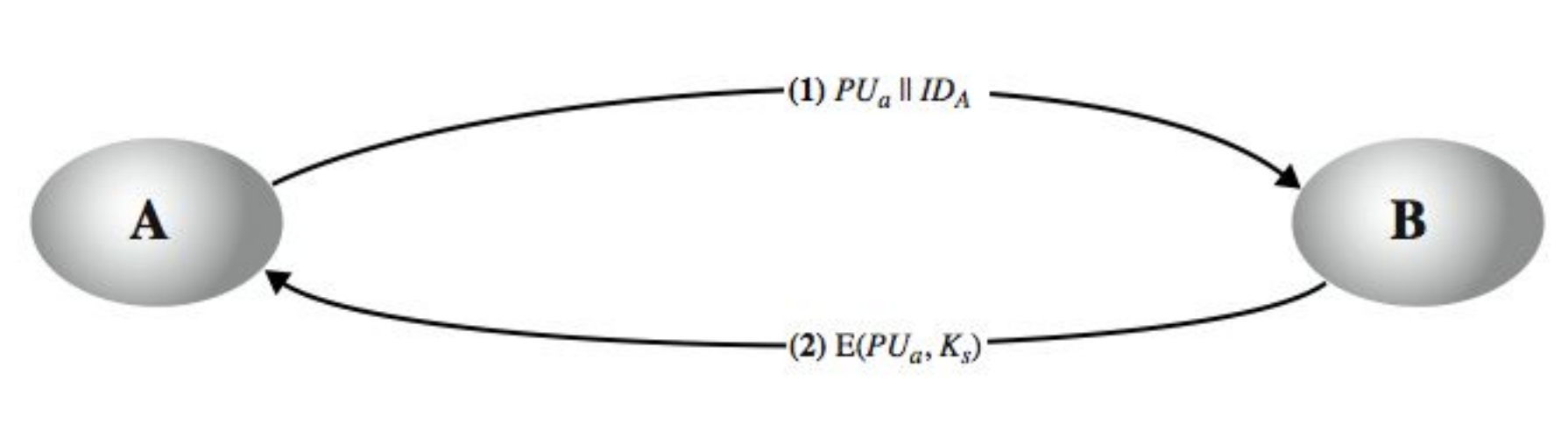
Symmetric Key Distribution Using Public Keys

- public key cryptosystems are inefficient
  - so almost never use for direct data encryption
  - rather use to encrypt secret keys for distribution



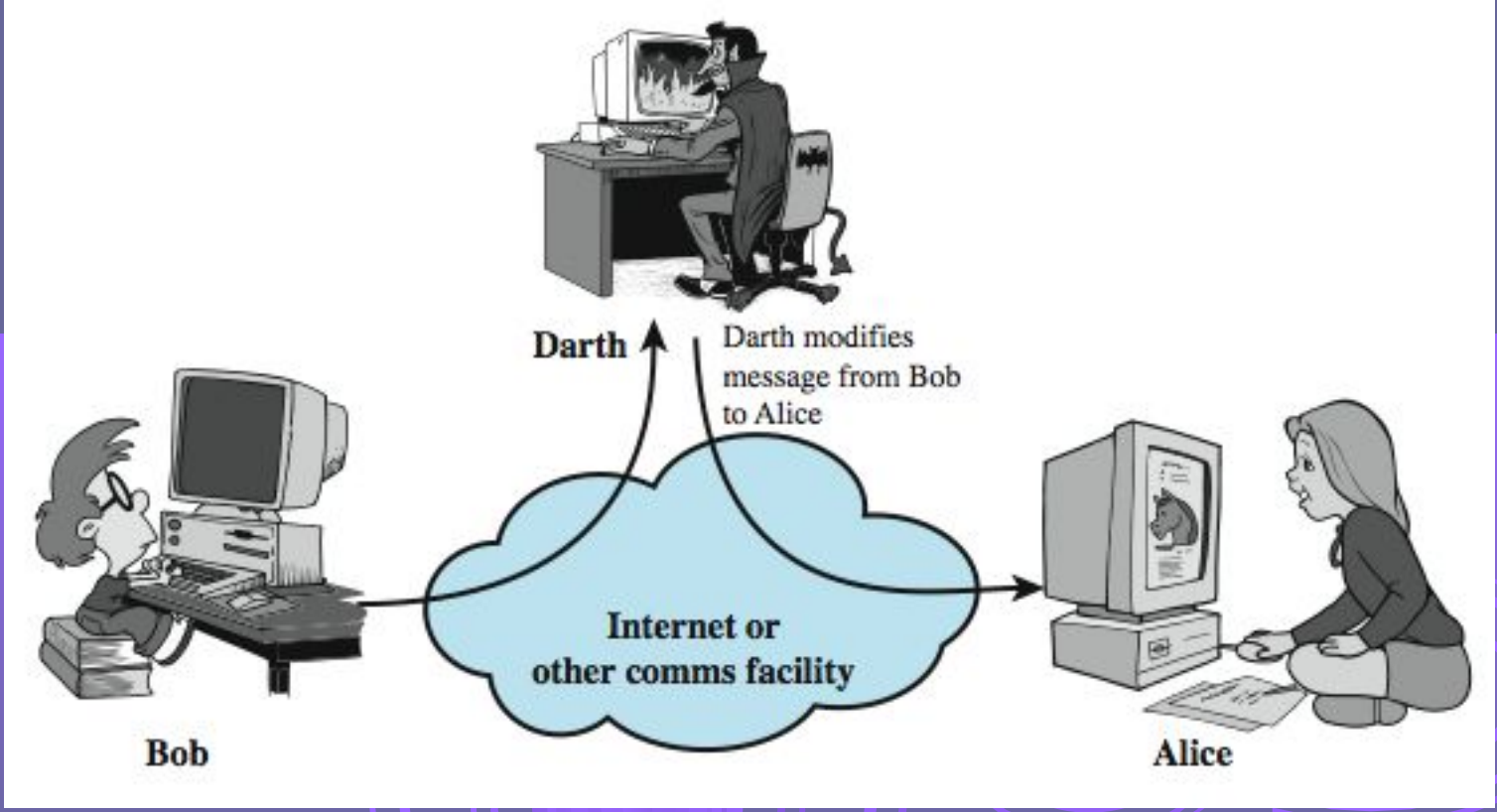
Simple Secret Key Distribution

- Merkle proposed this very simple scheme
  - allows secure communications
  - no keys before/after exist

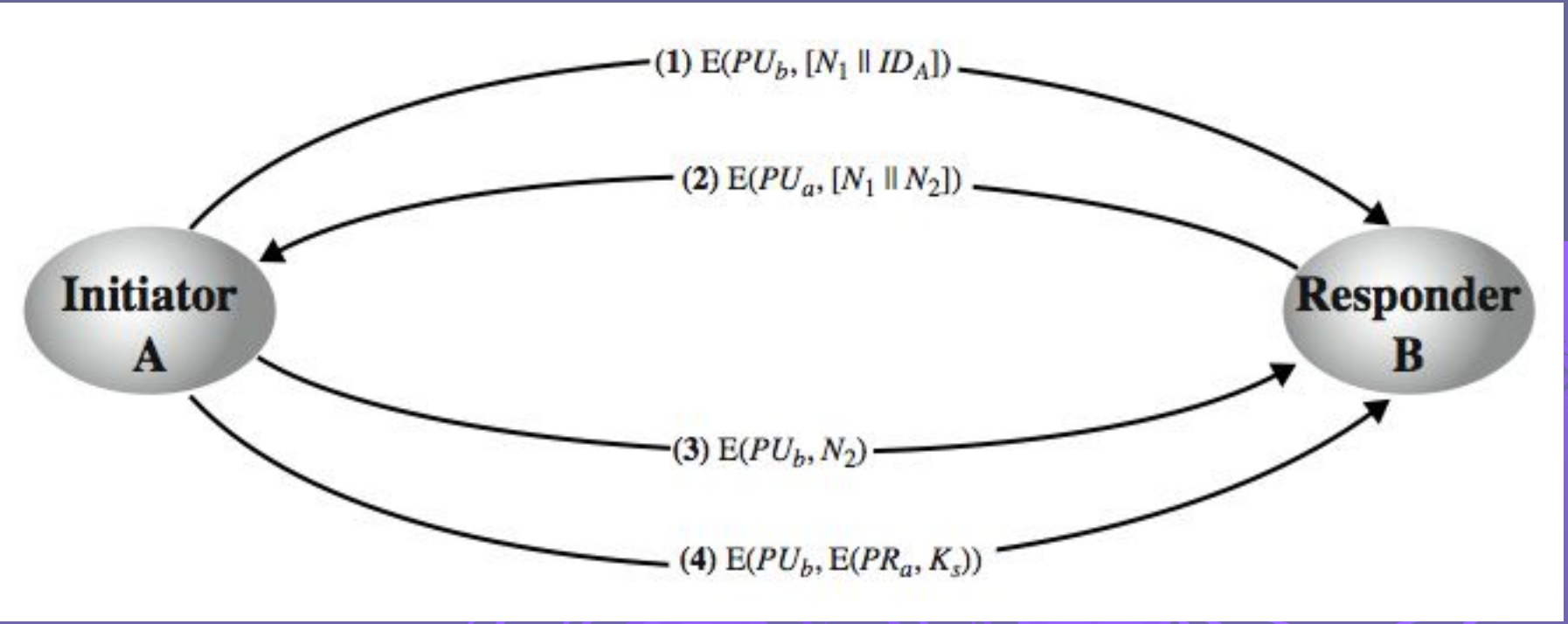


Man-in-the-Middle Attack

- this very simple scheme is vulnerable to an active man-in-the-middle attack



Secret Key Distribution with Confidentiality and Authentication



Hybrid Key Distribution

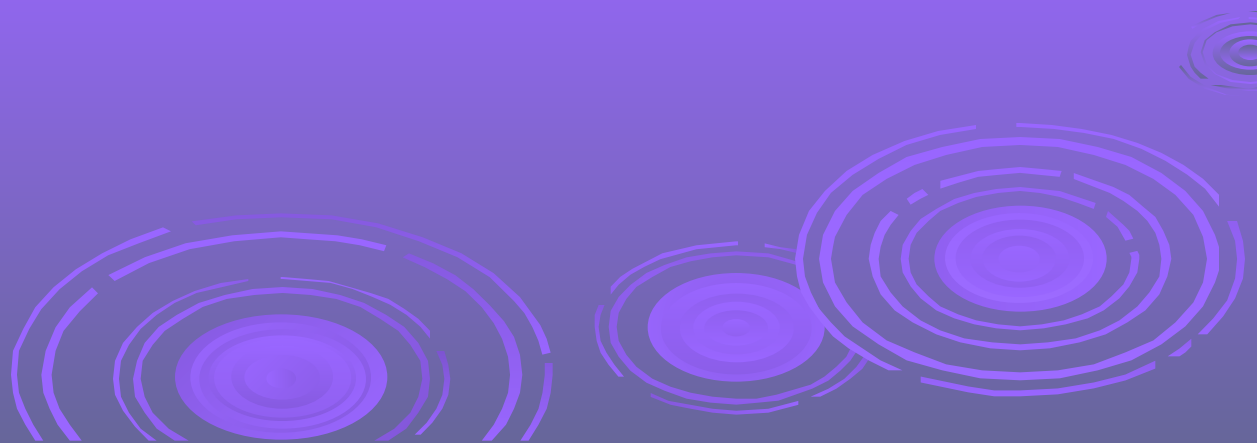
- retain use of private-key KDC
- shares secret master key with each user
- distributes session key using master key
- public-key used to distribute master keys
  - especially useful with widely distributed users
- rationale
  - performance
  - backward compatibility

Road Map

- symmetric key distribution using symmetric encryption
- symmetric key distribution using public-key encryption
- distribution of public keys
  - announcement, directory, authority, CA
- X.509 authentication and certificates
- public key infrastructure (PKIX)

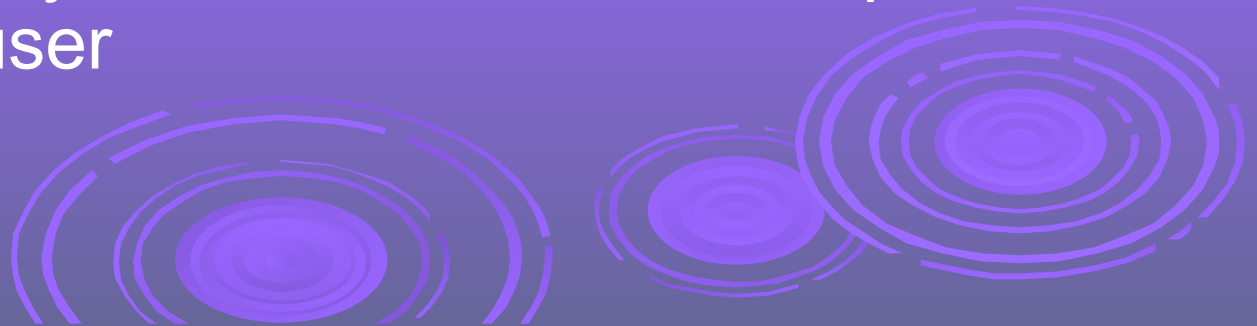
Distribution of Public Keys

- can be considered as using one of:
  - public announcement
  - publicly available directory
  - public-key authority
  - public-key certificates



Public Announcement

- users distribute public keys to recipients or broadcast to community at large
  - eg. append PGP keys to email messages or post to news groups or email list
- major weakness is forgery
  - anyone can create a key claiming to be someone else and broadcast it
  - until forgery is discovered can masquerade as claimed user



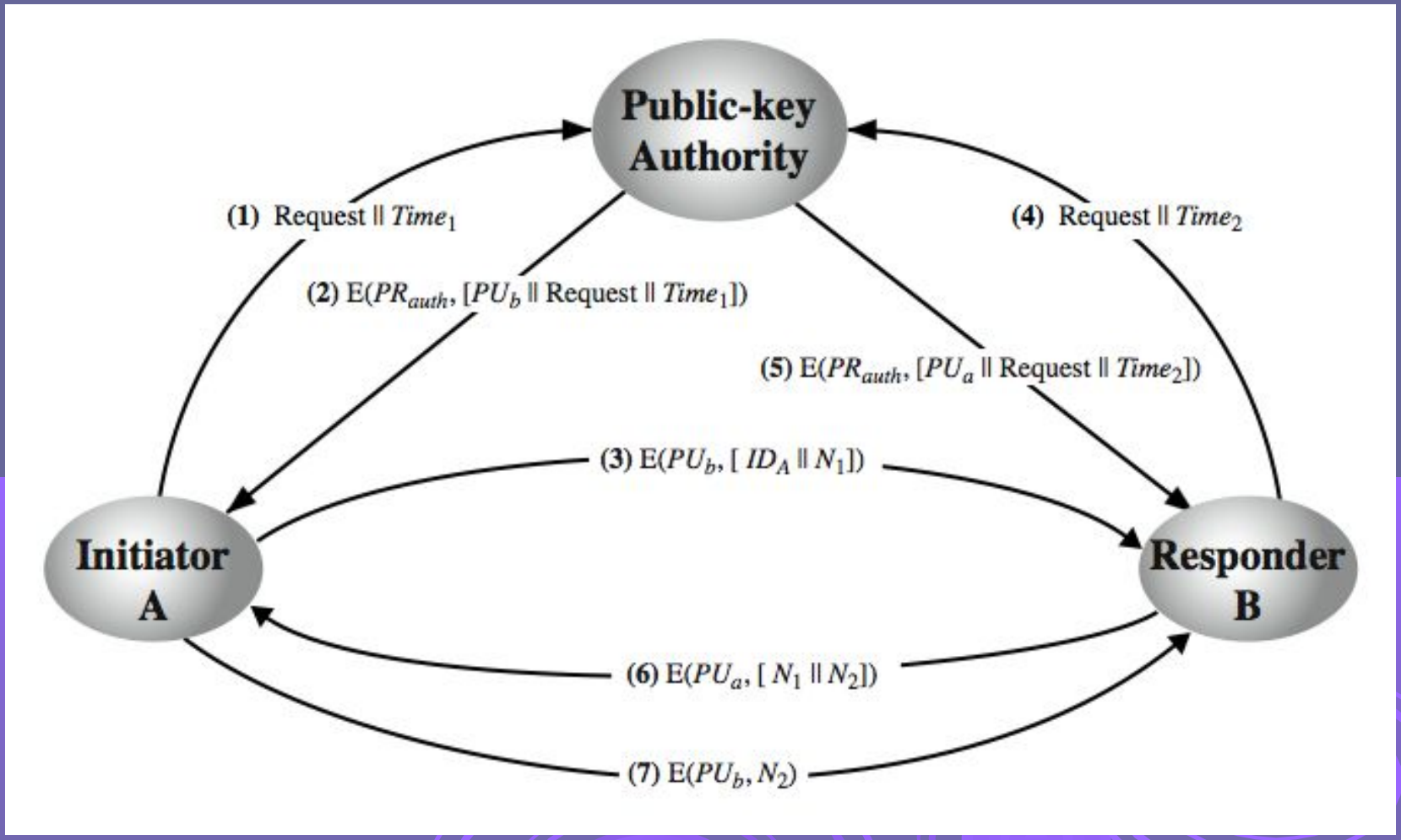
Publicly Available Directory

- can obtain greater security by registering keys with a public directory
- directory must be trusted with properties:
  - contains {name,public-key} entries
  - participants register securely with directory
  - participants can replace key at any time
  - directory is periodically published
  - directory can be accessed electronically
- still vulnerable to tampering or forgery

Public-Key Authority

- improve security by tightening control over distribution of keys from directory
- has properties of directory
- and requires users to know public key for the directory
- then users interact with directory to obtain any desired public key securely
  - does require real-time access to directory when keys are needed
  - may be vulnerable to tampering

Public-Key Authority



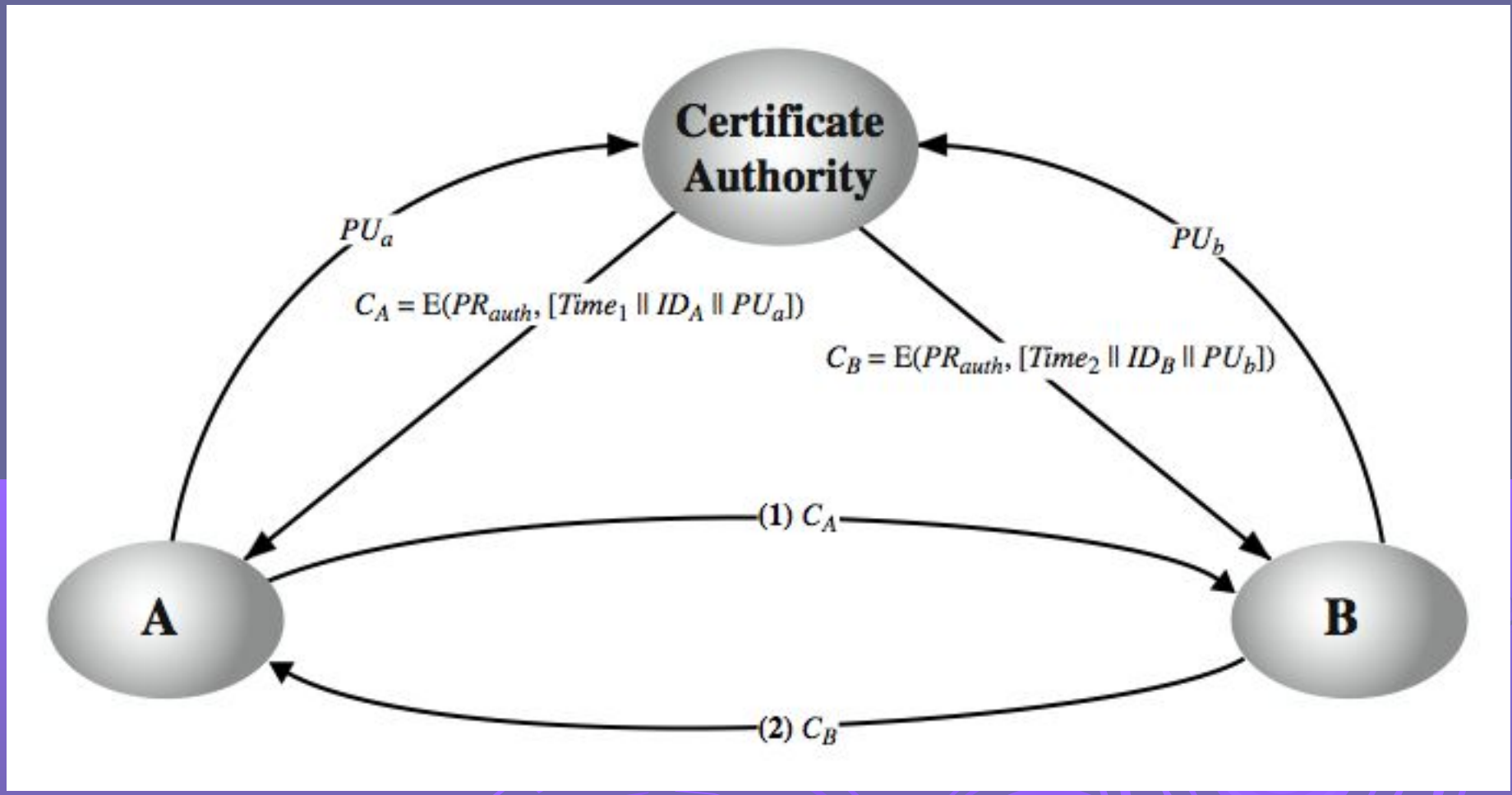
Road Map

- symmetric key distribution using symmetric encryption
- symmetric key distribution using public-key encryption
- distribution of public keys
  - announcement, directory, authrority, CA
- X.509 authentication and certificates
- public key infrastructure (PKIX)

Public-Key Certificates

- certificates allow key exchange without real-time access to public-key authority
- a certificate binds **identity** to **public key**
  - usually with other info such as period of validity, rights of use etc
- with all contents **signed** by a trusted Public-Key or Certificate Authority (CA)
- can be verified by anyone who knows the public-key authorities public-key

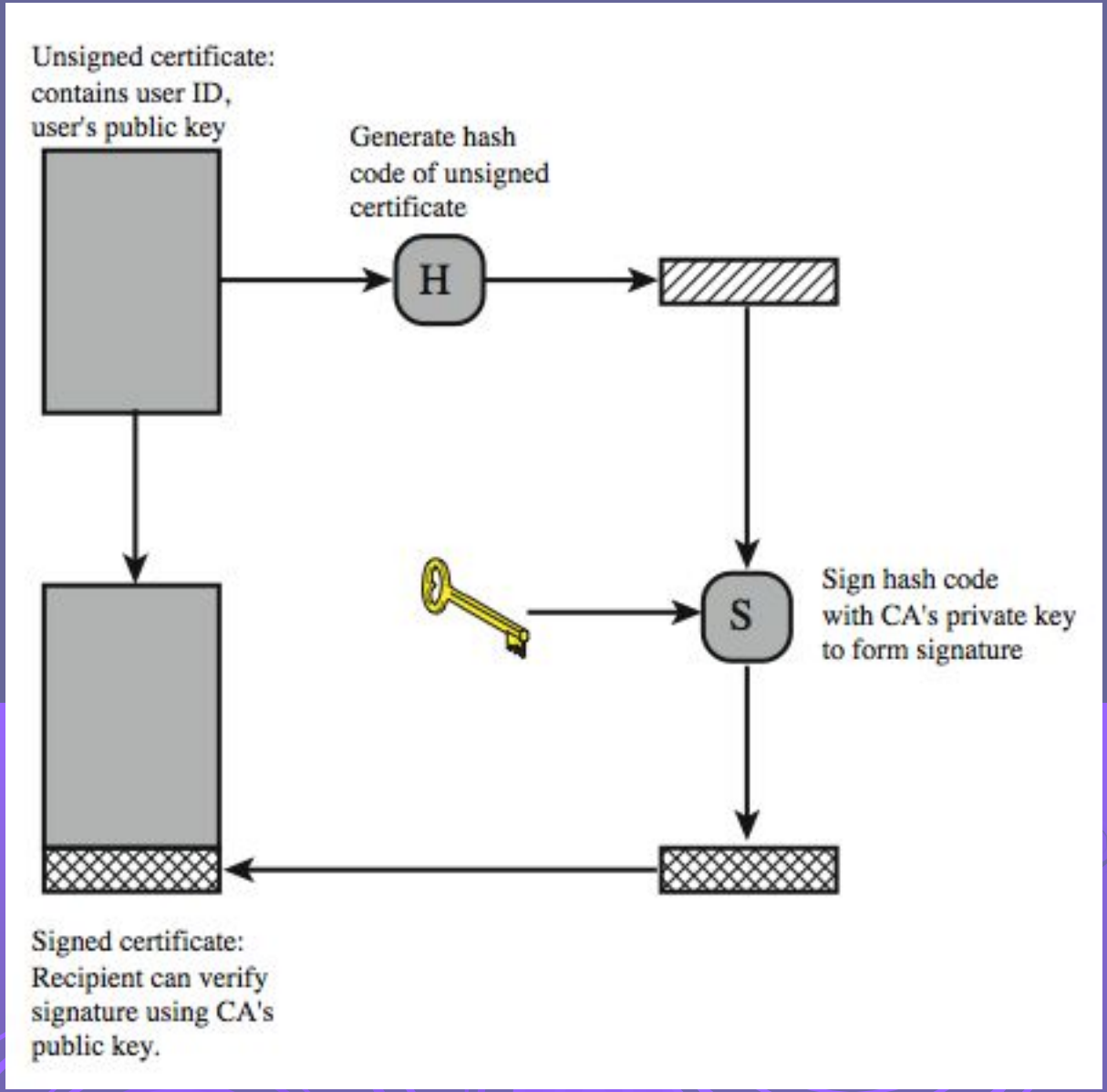
Public-Key Certificates



X.509 Authentication Service

- part of CCITT X.500 directory service standards
  - distributed servers maintaining user info database
- defines framework for authentication services
  - directory may store public-key certificates
  - with public key of user signed by certification authority
- also defines authentication protocols
- uses public-key crypto & digital signatures
  - algorithms not standardised, but RSA recommended
- X.509 certificates are widely used
  - have 3 versions

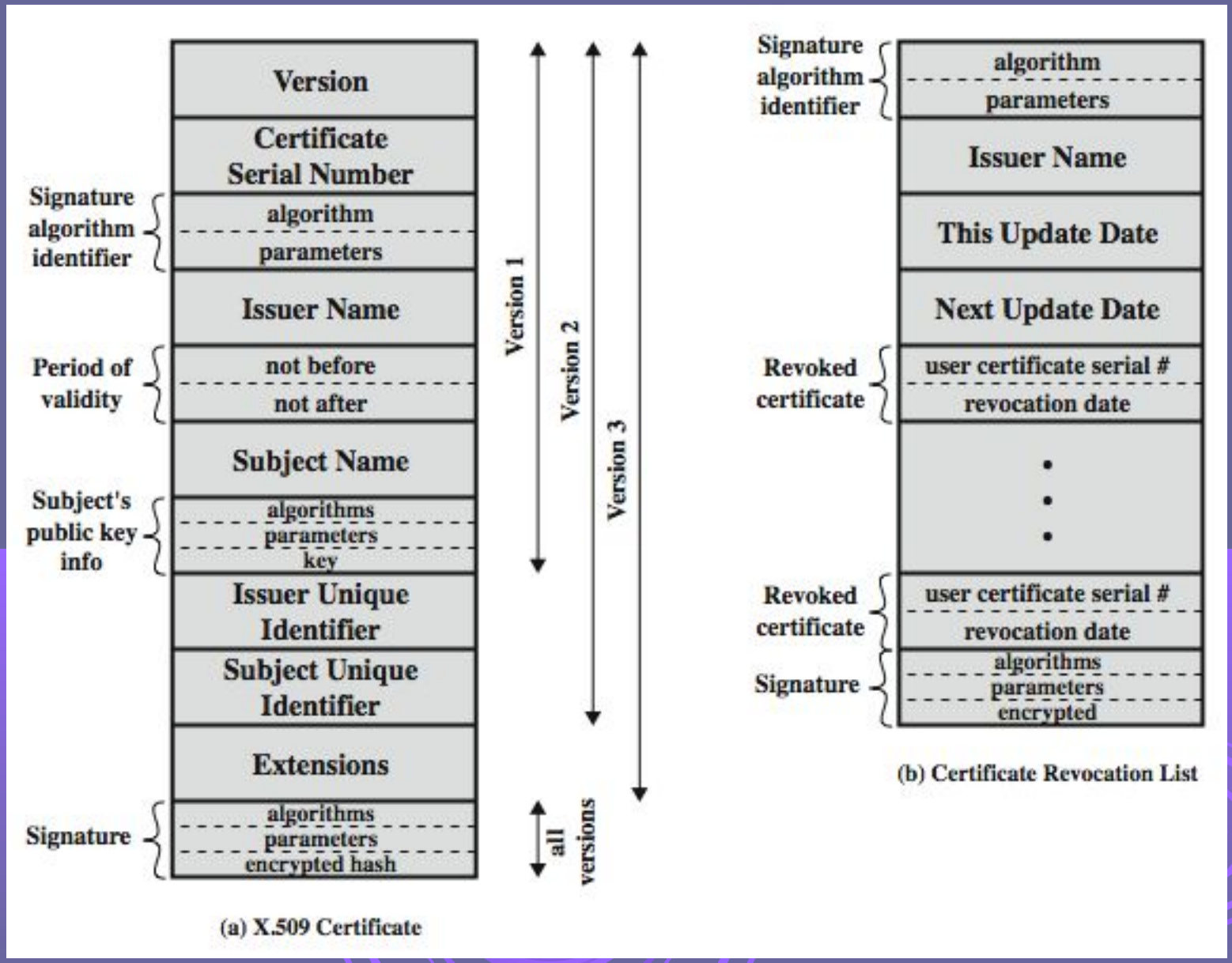
X.509 Certificate Use



X.509 Certificates

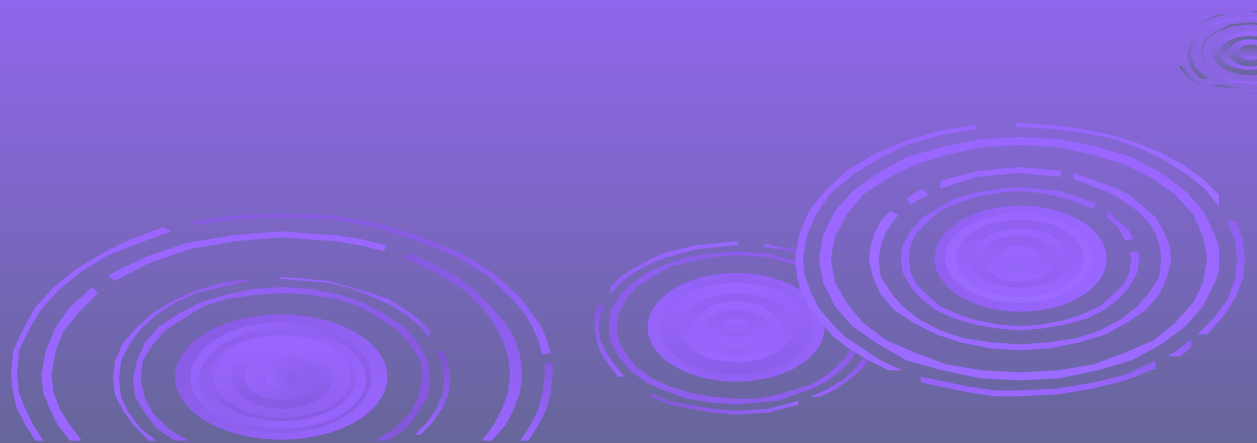
- issued by a Certification Authority (CA), containing:
  - version V (1, 2, or 3)
  - serial number SN (unique within CA) identifying certificate
  - signature algorithm identifier AI
  - issuer X.500 name CA)
  - period of validity TA (from - to dates)
  - subject X.500 name A (name of owner)
  - subject public-key info Ap (algorithm, parameters, key)
  - issuer unique identifier (v2+)
  - subject unique identifier (v2+)
  - extension fields (v3)
  - signature (of hash of all fields in certificate)
- notation `CA<<A>>` denotes certificate for A signed by CA

X.509 Certificates



Obtaining a Certificate

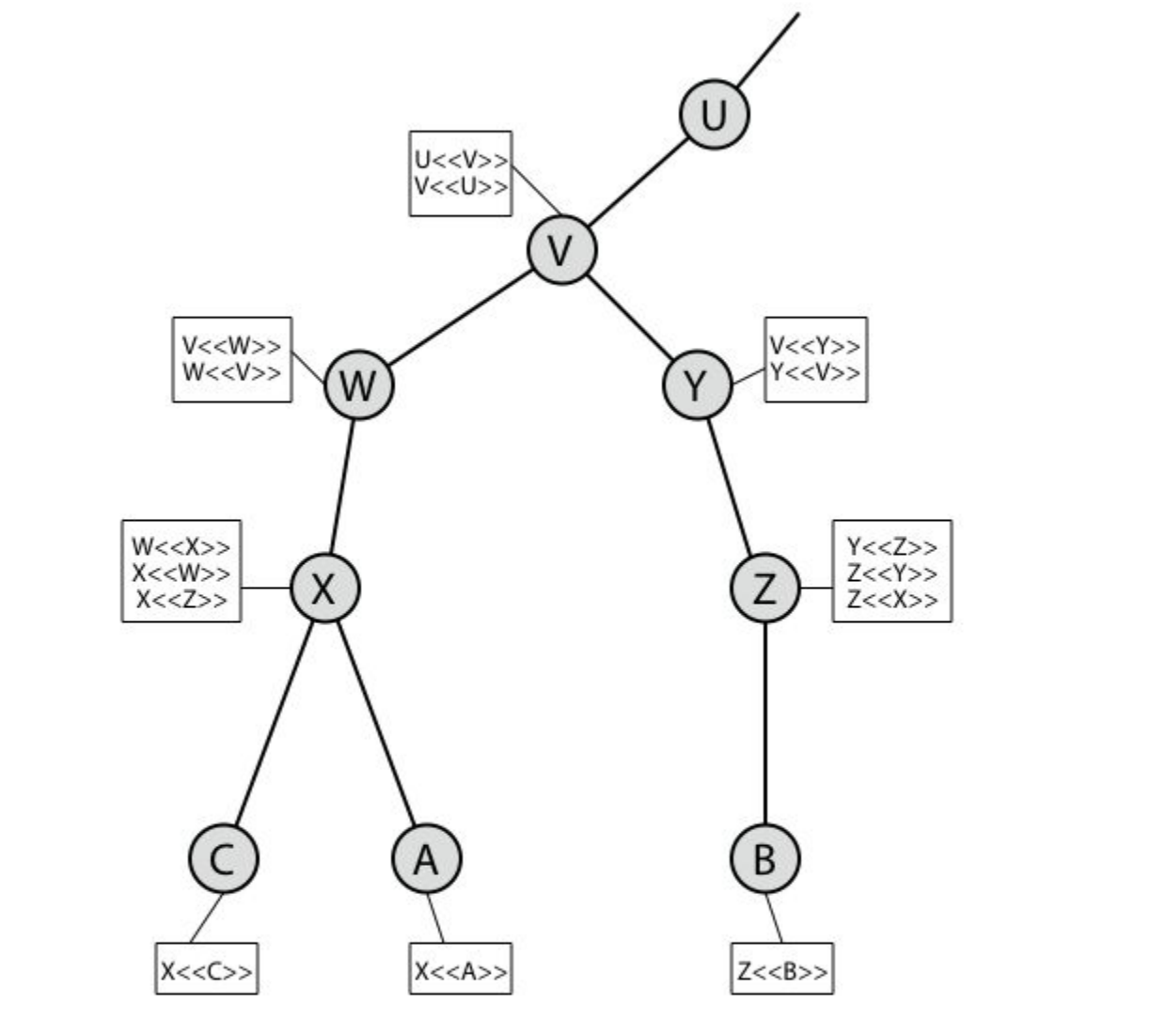
- any user with access to CA can get any certificate from it
- only the CA can modify a certificate
- because cannot be forged, certificates can be placed in a public directory



CA Hierarchy

- if both users share a common CA then they are assumed to know its public key
- otherwise CA's must form a hierarchy
- use certificates linking members of hierarchy to validate other CA's
  - each CA has certificates for clients (forward) and parent (backward)
- each client trusts parents certificates
- enable verification of any certificate from one CA by users of all other CAs in hierarchy

CA Hierarchy Use



Certificate Revocation

- certificates have a period of validity
- may need to revoke before expiry, eg:
  1. user's private key is compromised
  2. user is no longer certified by this CA
  3. CA's certificate is compromised
- CA's maintain list of revoked certificates
  - the Certificate Revocation List (CRL)
- users should check certificates with CA's CRL

X.509 Version 3

- has been recognised that additional information is needed in a certificate
  - email/URL, policy details, usage constraints
- rather than explicitly naming new fields defined a general extension method
- extensions consist of:
  - extension identifier
  - criticality indicator
  - extension value

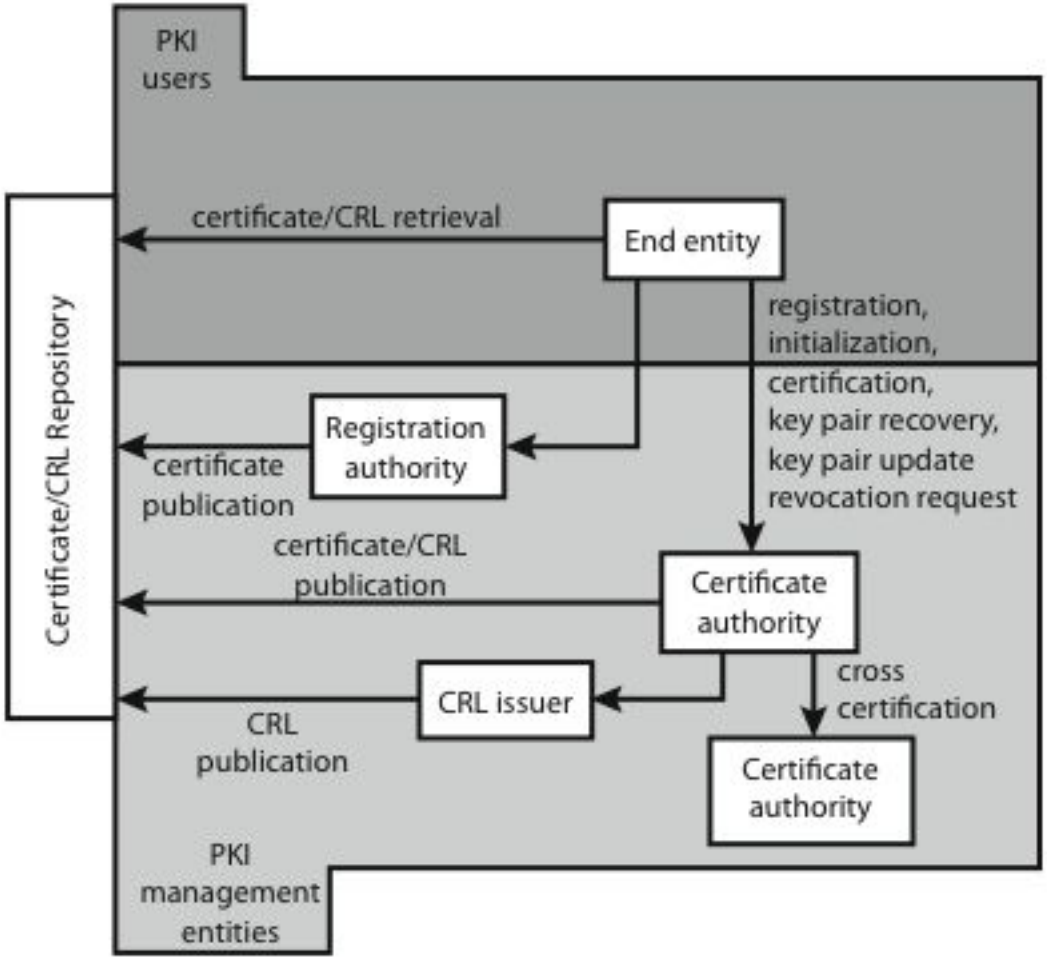
Certificate Extensions

- key and policy information
  - convey info about subject & issuer keys, plus indicators of certificate policy
- certificate subject and issuer attributes
  - support alternative names, in alternative formats for certificate subject and/or issuer
- certificate path constraints
  - allow constraints on use of certificates by other CA's

Road Map

- symmetric key distribution using symmetric encryption
- symmetric key distribution using public-key encryption
- distribution of public keys
  - announcement, directory, authrority, CA
- X.509 authentication and certificates
- public key infrastructure (PKIX)

Public Key Infrastructure



PKIX Management

- functions:
  - registration
  - initialization
  - certification
  - key pair recovery
  - key pair update
  - revocation request
  - cross certification
- protocols: CMP, CMC

Summary

- have considered:
  - symmetric key distribution using symmetric encryption
  - symmetric key distribution using public-key encryption
  - distribution of public keys
    - announcement, directory, authrority, CA
  - X.509 authentication and certificates
  - public key infrastructure (PKIX)