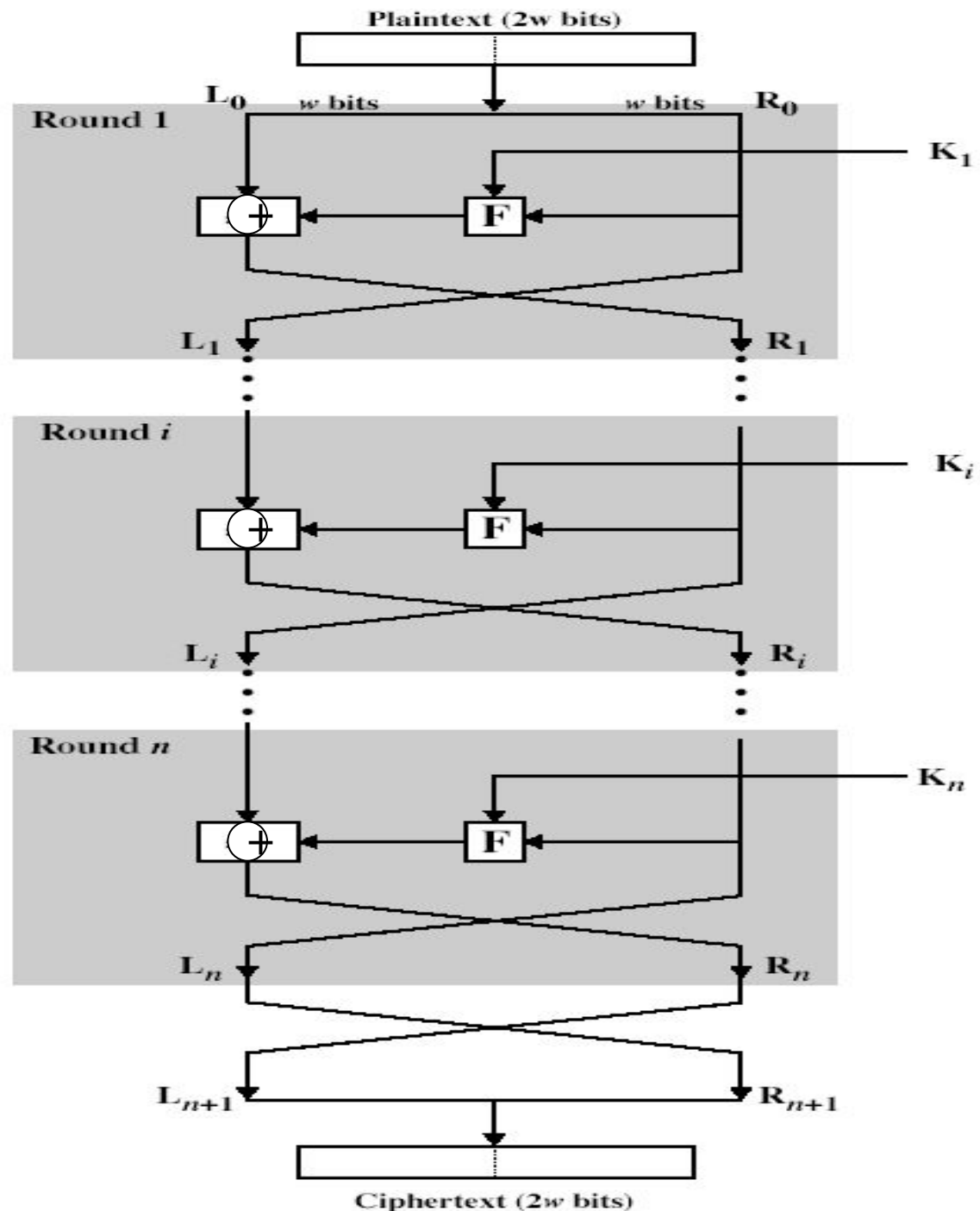


- Blowfish
- RC4 stream cipher
- Diffie-Hellman Key Exchange

Feistel cipher

- Key
 - Key length
 - subkey generation
- block
 - block length
 - two halves of block
- no. of round
- encryption algorithm
 - S-box
 - XOR





Key features not found in DES

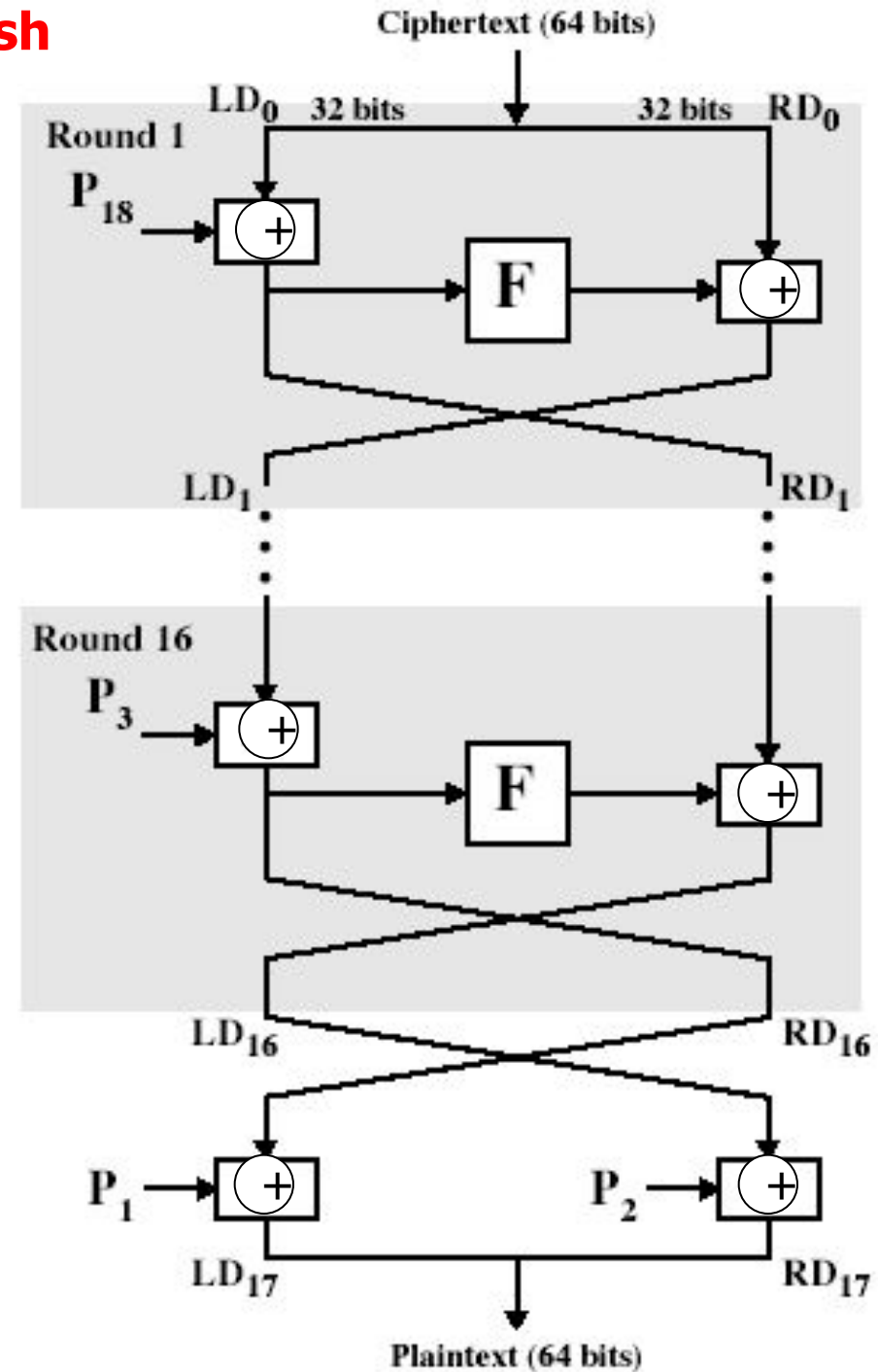
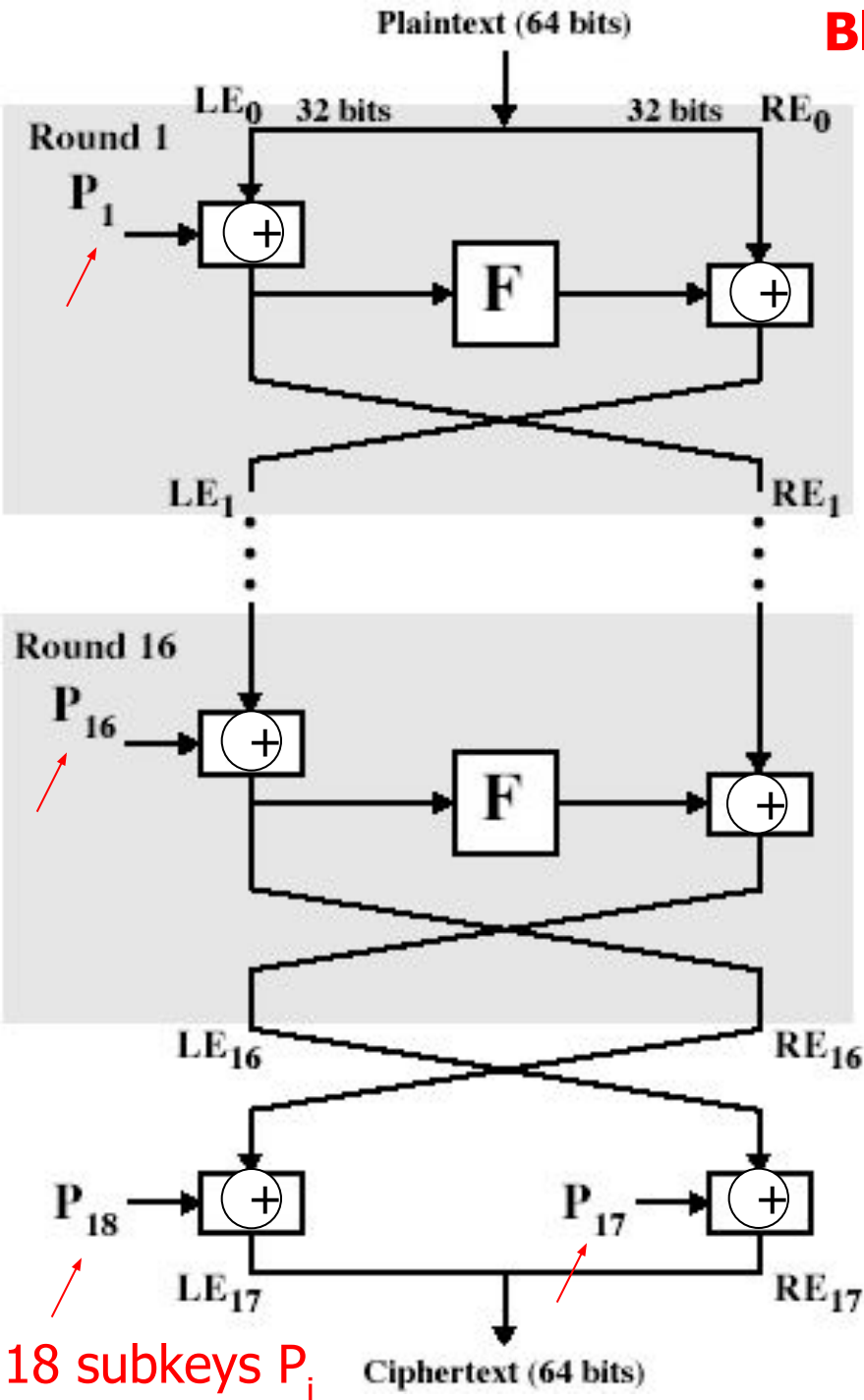
- Key
 - Key length
 - subkey generation
 - block
 - block length
 - two halves of block
 - no. of round
 - encryption algorithm
 - S-box
 - XOR
- Variable length key : Blowfish, RC5
- Complex subkey generation process: Blowfish
- Variable plain/ciphertext block length: RC5
- Operate on both halves each round: Blowfish, RC5
- Variable no. of round: RC5
- Key-dependent S-box: Blowfish
- Data/key-dependent **rotation**: RC5
- Mixed operation: more than one arithmetic and Boolean operations



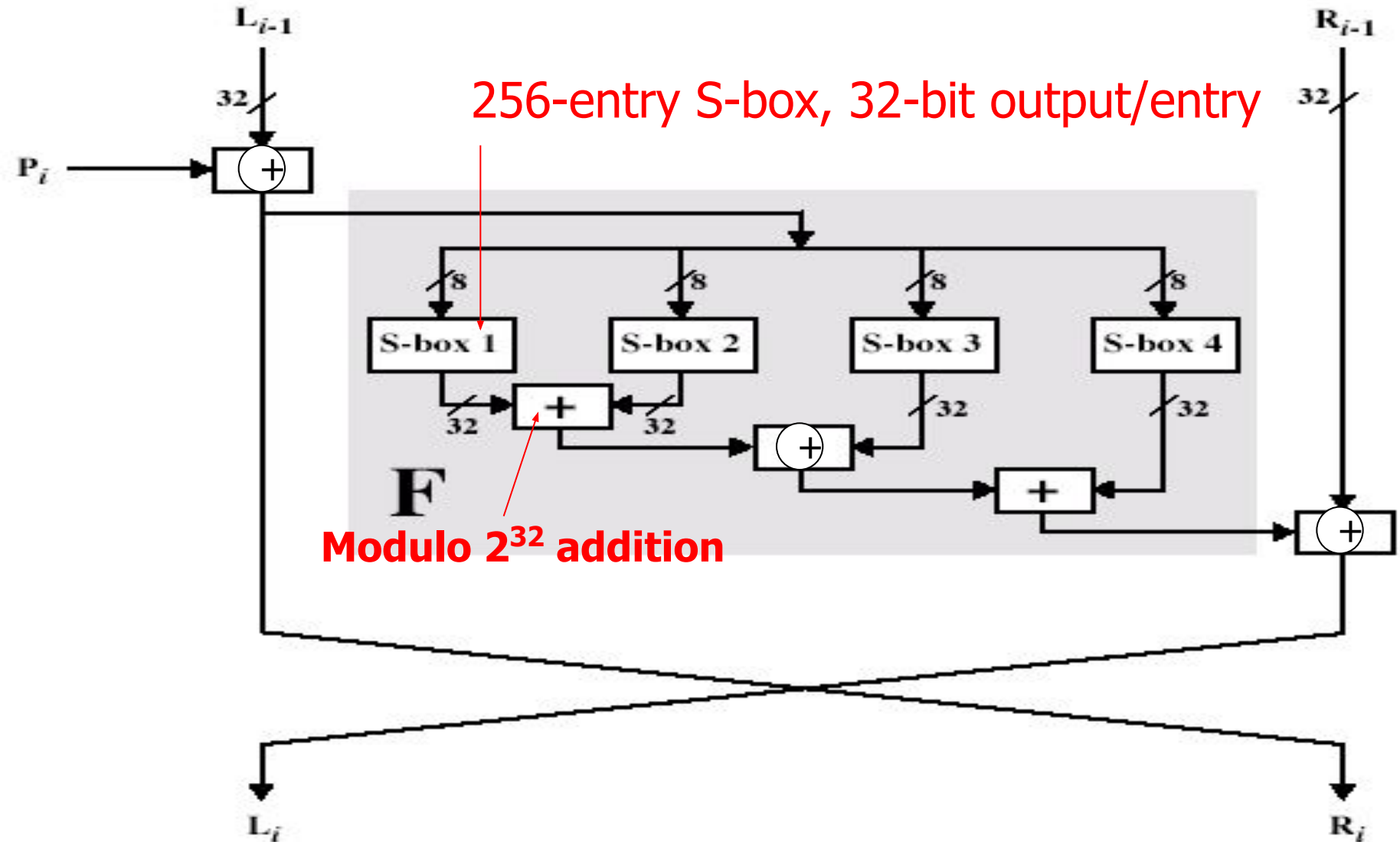
Blowfish

- a symmetric block cipher designed by Bruce Schneier in 1993/94
- Characteristics
 - 64-bit block cipher
 - Variable length key (32 bits to 448 bits)
 - Complex subkey generation
 - Key-dependent S-boxes
 - Simple operations – fast implementation
 - Modulo 2^{32} addition
 - Bitwise XOR

Blowfish



Blowfish single round



Subkey and S-box generation

- uses a 32 to 448 bit key
 - 1 word = 32 bits, 1 to 14 words

$$K_1, K_2, K_3, K_4, \dots, K_j \quad 1 \leq j \leq 14$$

- to generate
 - 18 32-bit subkeys stored in P-array

$$P_1, P_2, P_3, P_4, \dots, P_{18}$$

- four 256 entry S-boxes, 1 word in each entry

$$\begin{array}{cccccc} S_{1,0'} & S_{1,1'} & S_{1,2'} & S_{1,3'} & \dots & S_{1,255} \\ S_{2,0'} & S_{2,1'} & S_{2,2'} & S_{2,3'} & \dots & S_{2,255} \\ S_{3,0'} & S_{3,1'} & S_{3,2'} & S_{3,3'} & \dots & S_{3,255} \\ S_{4,0'} & S_{4,1'} & S_{4,2'} & S_{4,3'} & \dots & S_{4,255} \end{array} \quad \left. \vphantom{\begin{array}{cccccc} S_{1,0'} & S_{1,1'} & S_{1,2'} & S_{1,3'} & \dots & S_{1,255} \\ S_{2,0'} & S_{2,1'} & S_{2,2'} & S_{2,3'} & \dots & S_{2,255} \\ S_{3,0'} & S_{3,1'} & S_{3,2'} & S_{3,3'} & \dots & S_{3,255} \\ S_{4,0'} & S_{4,1'} & S_{4,2'} & S_{4,3'} & \dots & S_{4,255} \end{array}} \right\} \text{Totally 1024 words}$$

Subkey and S-box generation

1. **initialize** P-array and then 4 S-boxes using π

用 π 的小數點依序填入 P 和 S 陣列

$$P_1 = 243F6A88 \quad P_2 = 85A308D3 \quad \dots \quad S_{4,255} = 3AC372E6$$

2. XOR P-array with K-array (reuse as needed)

$$\begin{array}{cccccccccccc} P_1, & P_2, & P_3, & P_4, & \dots & P_{14}, & P_{15}, & P_{16}, & P_{17}, & P_{18} \\ \oplus & \oplus & \oplus & \oplus & & \oplus & \oplus & \oplus & \oplus & \oplus \\ K_1, & K_2, & K_3, & K_4, & \dots, & K_{14}, & K_1, & K_2, & K_3, & K_4 \end{array}$$

(Suppose input key is 14 words)

Subkey and S-box generation

3. loop repeatedly encrypting data using current P & S and replace successive pairs of P then S values

$P_1, P_2 = E_{P,S} [0]$ 用現在P,S值當參數的 Blowfish, 加密 0
更新 P_1, P_2

$P_3, P_4 = E_{P,S} [P_1 || P_2]$

...

$S_{1,0}, S_{1,1} = E_{P,S} [P_{17} || P_{18}]$

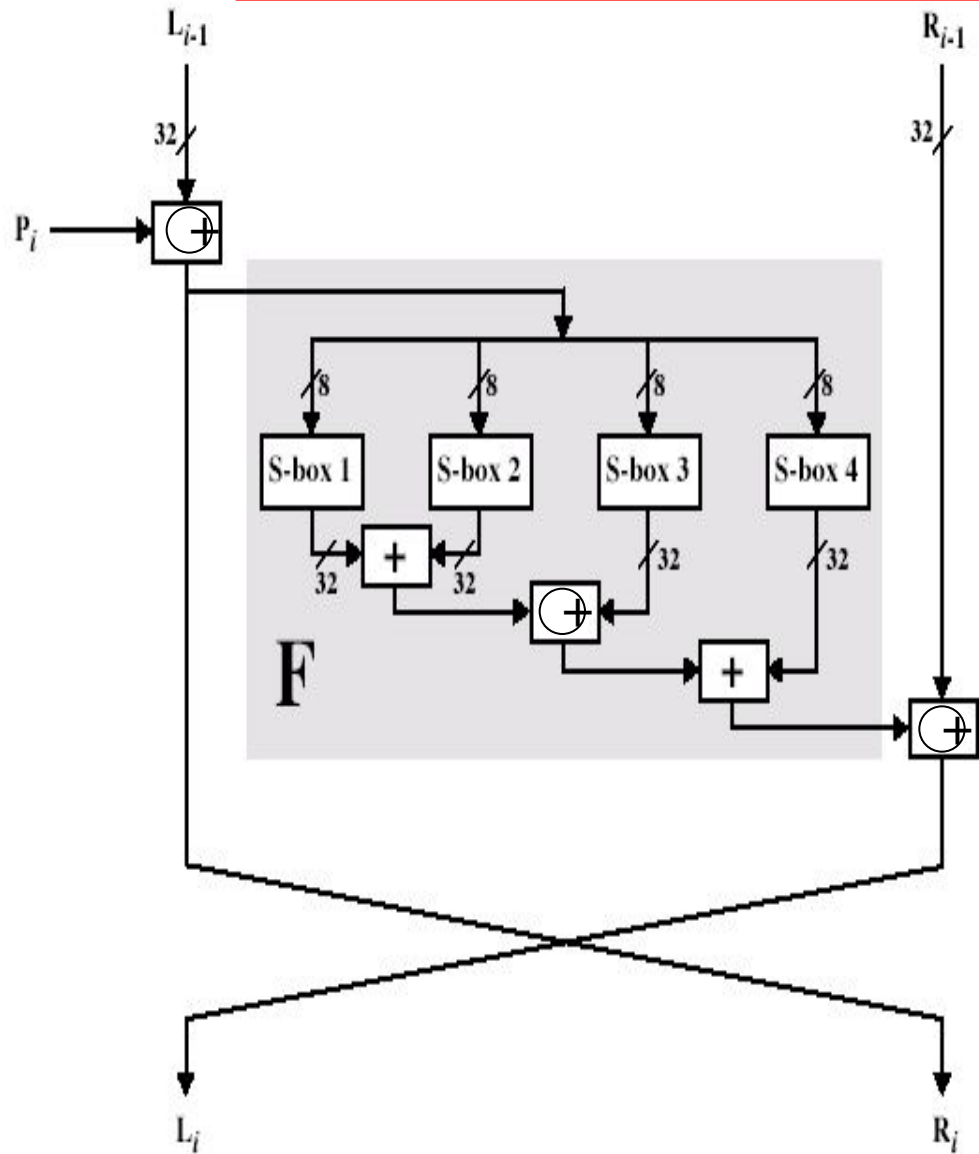
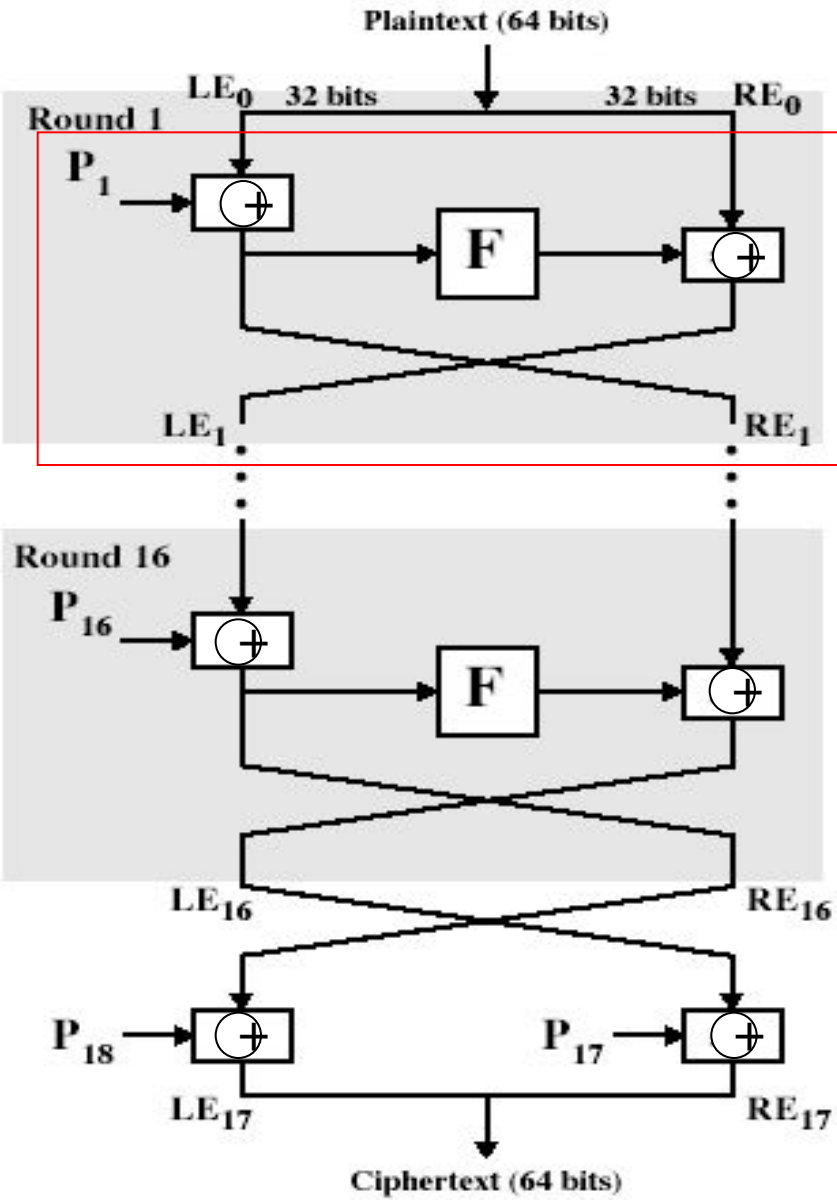
...

$S_{4,254}, S_{4,255} = E_{P,S} [S_{4,252} || S_{4,253}]$

Totally 521 executions of Blowfish encryption
=> not suitable for frequent key changes

Blowfish Encryption

+ and \oplus do not commute





Discussion

- key dependent S-boxes and subkeys, generated using cipher itself, makes analysis very difficult
- changing both halves in each round increases security (c.f. Feistel cipher)
- brute-force key search is not practical (maximally 448 bits)



Discussion (cont.)

- fast

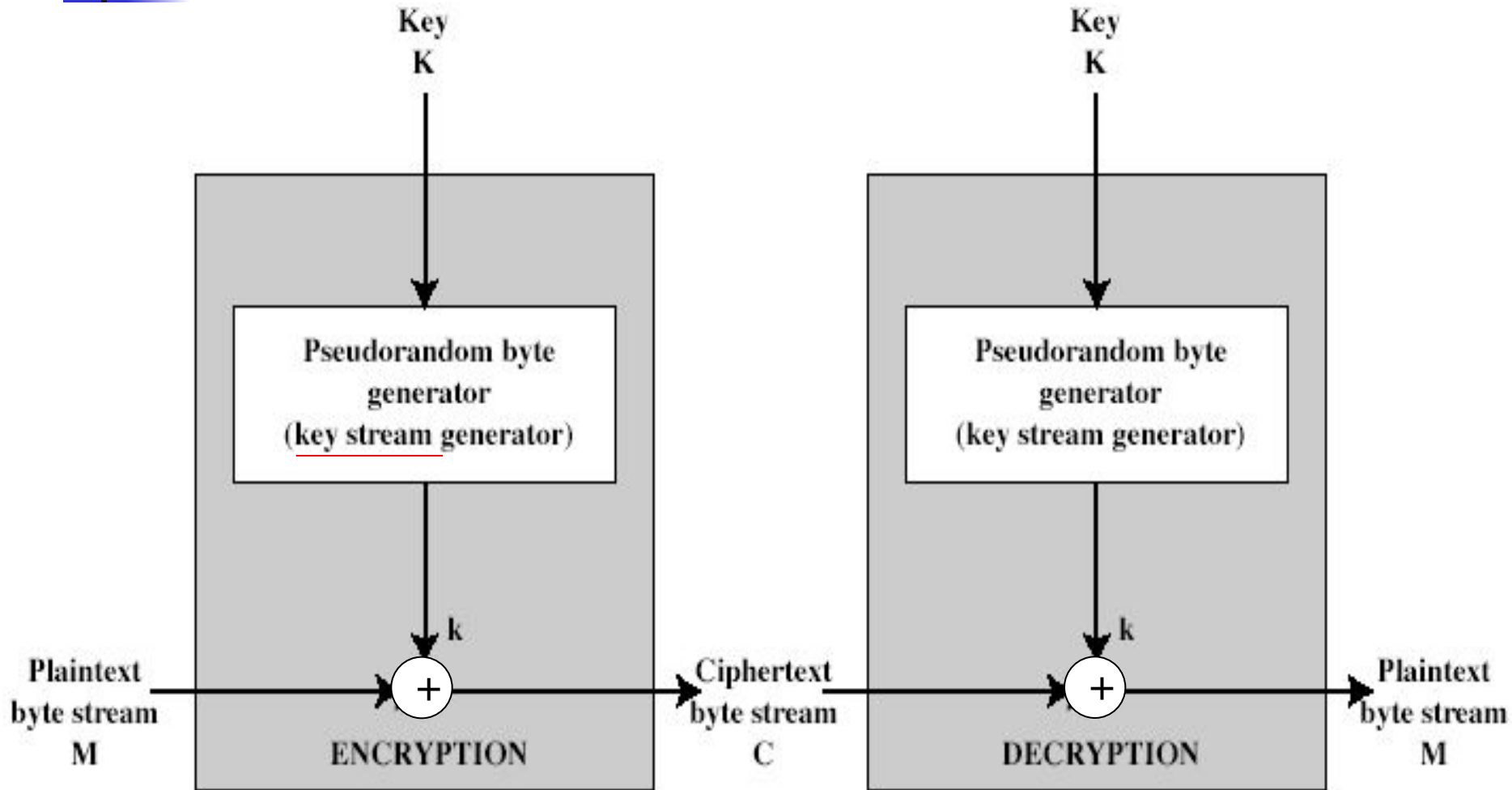
Algorithm	Clock cycles per round	# of rounds	# of clock cycles per byte encrypted
Blowfish	9	16	18
RC5	12	16	23
DES	18	16	45
IDEA	50	8	50
Triple-DES	18	48	108



Outline

- Characteristics of advanced symmetric block cipher
- Triple DES
- Blowfish
- RC5
- RC4 stream cipher

Stream cipher diagram



Recall: One-time pad in Chap. 2



Stream Cipher Properties

- some design considerations are:
 - A pseudorandom number generator produces a **deterministic stream that eventually repeats**
 - The period should be long
 - Keystream should **approximate a true random stream**
 - Ex. Approximately equal number of 1s and 0s
 - The key needs to be sufficiently long
 - Ex. 128 bits or longer key to avoid brute-force attack



Advantage of stream cipher

- Faster than block ciphers

Table 6.2 Speed Comparisons of Symmetric Ciphers on a Pentium II

Cipher	Key Length	Speed (Mbps)
DES	56	9
3DES	168	3
RC2	variable	0.9
RC4	variable	45



Disadvantage of stream cipher

- never reuse stream key

Ciphertext 1 = plaintext 1 \oplus keystream

Ciphertext 2 = plaintext 2 \oplus keystream

Ciphertext 1 \oplus Ciphertext 2
= (plaintext 1 \oplus keystream) \oplus (plaintext 2 \oplus keystream)
= plaintext 1 \oplus plaintext 2

If plaintexts are text string, credit card no., or other streams with known properties, then cryptanalysis may be successful

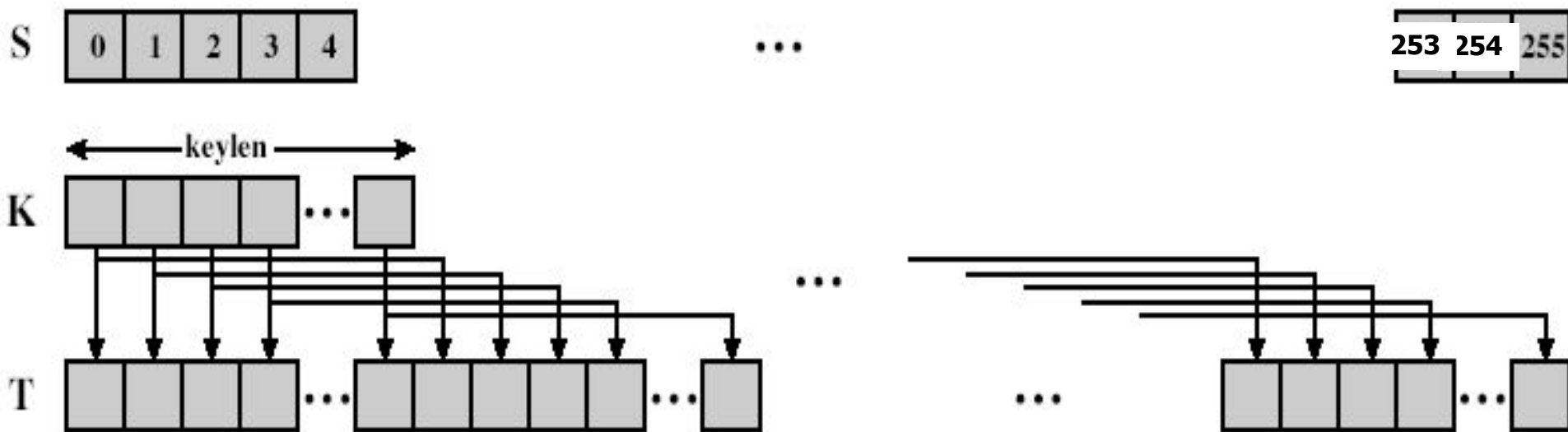


RC4

- Designed by Ron Rivest in 1987, owned by RSA DSI
- variable key size, **byte-oriented** stream cipher
- widely used (web SSL/TLS, wireless WEP)
 - SSL: secure sockets layer
 - TLS: transport layer security
 - WEP: wired equivalent privacy
- Main steps:
 - key forms random permutation of all 8-bit values (**state vector** $S[0], S[1], \dots, S[255]$)
 - uses that permutation to scramble input info processed a byte at a time

RC4 Key Schedule

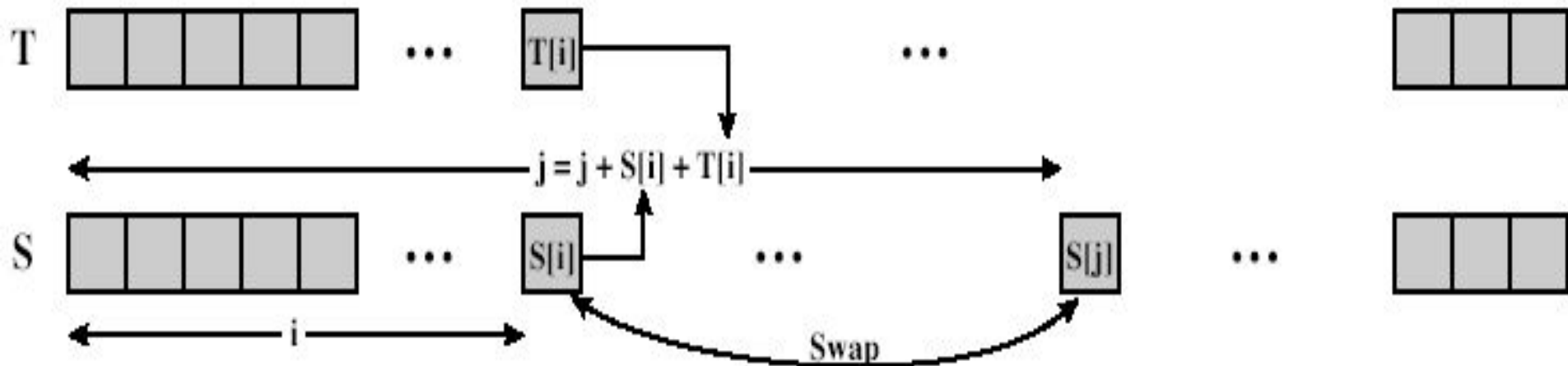
- starts with an array S of numbers: 0...255
- S forms **internal state** of the cipher
- given a key K of length **keylen** bytes (1 to 256 bytes)



(a) Initial state of S and T

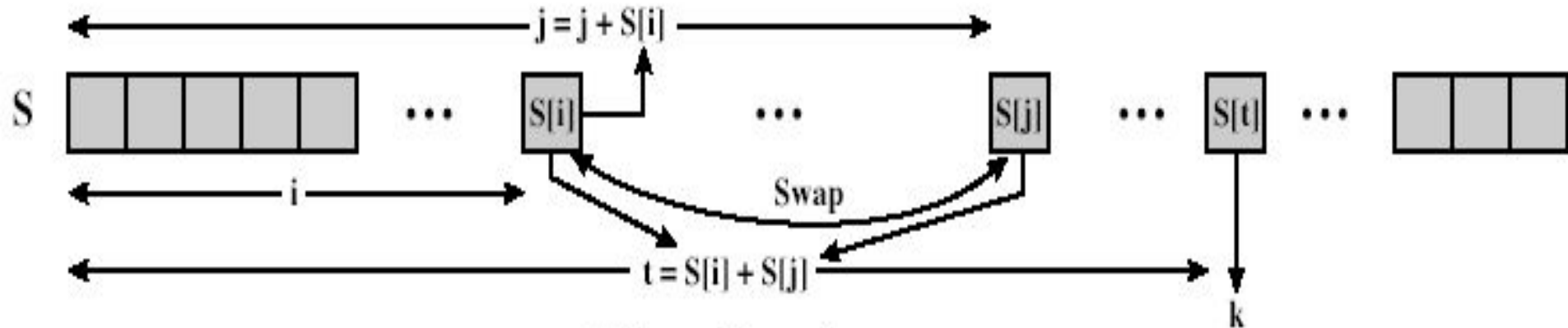
RC4 key scheduling and stream generation

$j=0$, for $i=0$ to 255



(b) Initial permutation of S

$i=0, j=0$ While(1){ $i=i+1 \bmod 256, \dots$ }



(c) Stream Generation

Plaintext \oplus **k** = ciphertext



RC4 Encryption

- encryption continues shuffling array values
- sum of shuffled pair selects "stream key" value
- XOR with next byte of message to en/decrypt

$i = j = 0$

for each message byte M_i

$i = (i + 1) \pmod{256}$

$j = (j + S[i]) \pmod{256}$

swap($S[i]$, $S[j]$)

$t = (S[i] + S[j]) \pmod{256}$

$C_i = M_i \text{ XOR } S[t]$




RC4 Security

- The period of RC4 $> 10^{100}$
- claimed secure against known attacks
 - have some analyses, none practical
- result is very non-linear
- since RC4 is a stream cipher, must **never reuse a key**
- have a concern with WEP, but due to key handling rather than RC4 itself

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
- The Diffie-Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms.
- The discrete logarithm is defined in the following way
a primitive root of a prime number p as one whose powers modulo p generate all the integers from 1 to $p-1$

- 
- That is, if a is a primitive root of the prime number p , then the numbers

$$a \bmod p, a^2 \bmod p, \dots, a^{p-1} \bmod p$$

are distinct and consist of the integers from 1 through $p - 1$ in some permutation.

For any integer b and a primitive root a of prime number p , we can find a unique exponent i such that

$$b \equiv a^i \pmod{p} \quad \text{where } 0 \leq i \leq (p - 1)$$

The exponent i is referred to as the **discrete logarithm** of b for the base a , mod p . We express this value as $\text{dlog}_{a,p}(b)$.



The Algorithm

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	$Y_A = \alpha^{X_A} \bmod q$

User B Key Generation

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

Calculation of Secret Key by User A

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

Calculation of Secret Key by User B

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

The Algorithm

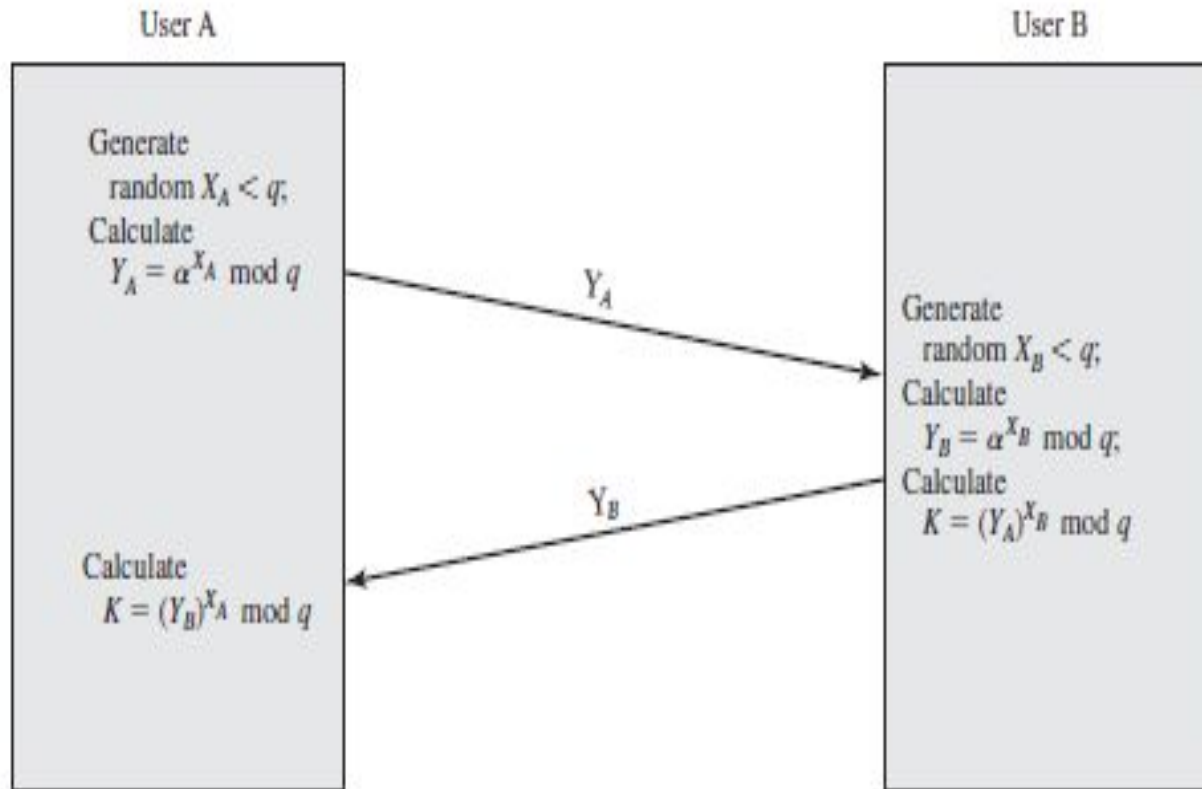
$$\begin{aligned} K &= (Y_B)^{X_A} \bmod q \\ &= (\alpha^{X_B} \bmod q)^{X_A} \bmod q \\ &= (\alpha^{X_B})^{X_A} \bmod q && \text{by the rules of modular arithmetic} \\ &= \alpha^{X_B X_A} \bmod q \\ &= (\alpha^{X_A})^{X_B} \bmod q \\ &= (\alpha^{X_A} \bmod q)^{X_B} \bmod q \\ &= (Y_A)^{X_B} \bmod q \end{aligned}$$

The result is that the two sides have exchanged a secret value. Furthermore, because X_A and X_B are private, an adversary only has the following ingredients to work with: q , α , Y_A , and Y_B . Thus, the adversary is forced to take a discrete logarithm to determine the key. For example, to determine the private key of user B, an adversary must compute

$$X_B = \text{dlog}_{\alpha, q}(Y_B)$$

The adversary can then calculate the key K in the same manner as user B calculates it.

Key Exchange Protocols



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 $K2 = (Y_A)^{X_{D2}} \bmod q$.
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 $K2 = (Y_{D2})^{X_A} \bmod q$.

At this point, Bob and Alice think that they share a secret key, but instead Bob and Darth share secret key $K1$ and Alice and Darth share secret key $K2$. 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 $E(K1, M)$ or $E(K1, M')$, 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.