Block Cipher and Stream Cipher Operation

Double-DES?

- Could use 2 DES encrypts/decrypts on each block
  - $C = E_{K2}(E_{K1}(P))$
  - $P = D_{K1}(D_{K2}(C))$
- Issue of reduction to single stage
  - It would be possible to find a key K3 such that

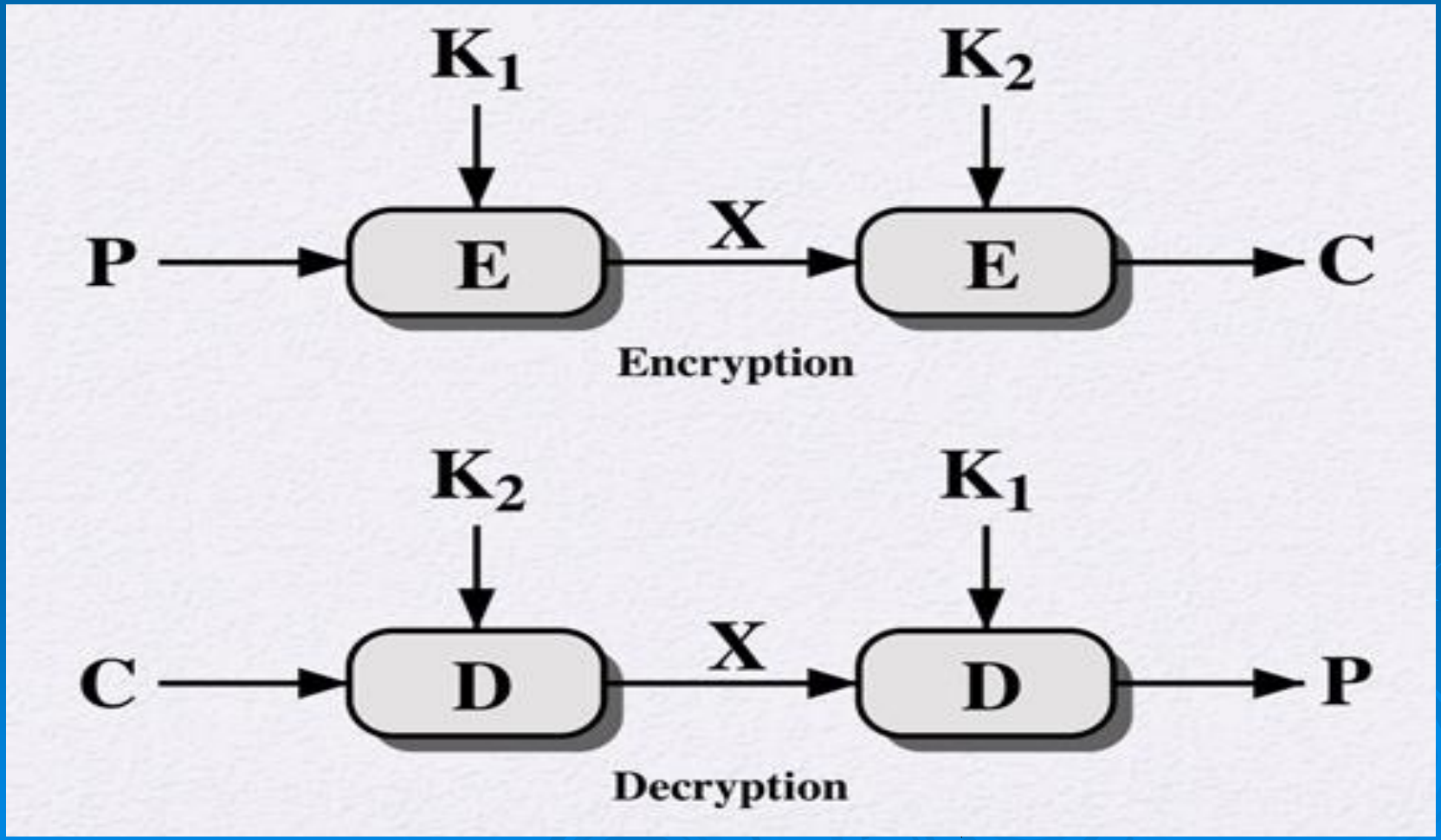
$$E_{K2}(E_{K1}(P)) = E_{K3}(P)$$

Double-DES?

- And have “meet-in-the-middle” attack
  - works whenever use a cipher twice
  - It is based on the observation that, if we have

    $C = E_{K2}(E_{K1}(P))$
  - Then $X = E_{K1}(P) = D_{K2}(C)$
  - attack by encrypting P with all keys and store
  - then decrypt C with keys and match X value
  - can show takes $O(2^{56})$ steps

Double DES



Meet-in-the-Middle Attack

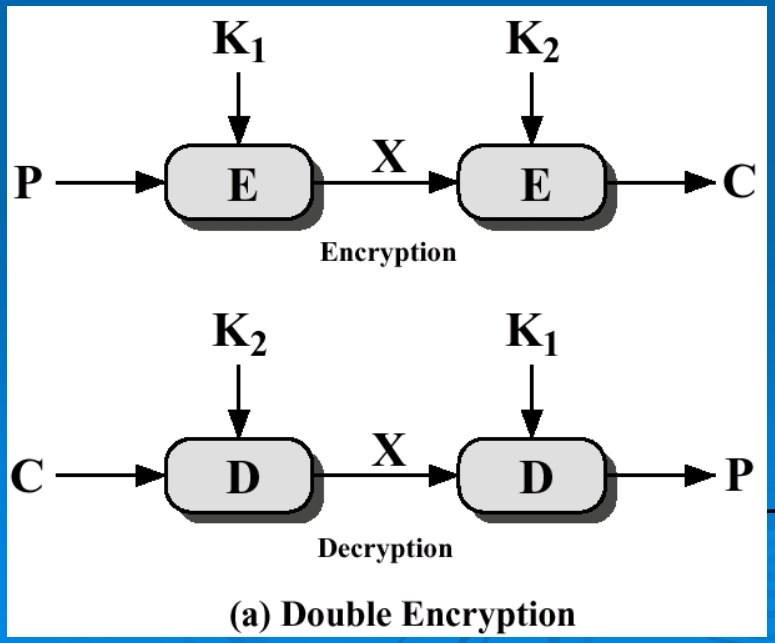
- Given a known plaintext-ciphertext pair, proceed as follows:
  - Encrypt P for all possible values of K1
  - Store results in table and sort by value of X
  - Decrypt C for all possible values of K2
    - During each decryption, check table for match. If find one, test two keys against another known plaintext-ciphertext pair

Meet-in-the-Middle Attack

- Analysis:
  - For any given plaintext P, there are 2^{64} possible ciphertexts produced by Double DES.
  - But Double DES effectively has 112 bit key, so there are 2^{112} possible keys.
  - On average then, for a given plaintext, the number of different 112 bit keys that will produce a given ciphertext is $2^{112}/2^{64}=2^{48}$
  - Thus, first (P,C) pair will produce about 2^{48} false alarms
  - Second (P,C) pair, however, reduces false alarm rate to $2^{48-64} = 2^{-16}$. So for two (P,C) pairs, the probability that correct key is determined is $1–2^{-16}$.
- Bottom line: a known plaintext attack will succeed against Double DES with an effort on order of 2^{56}, not much more than the 2^{55} required to crack single DES

Why Not Double DES?

- That is, why not just use $C=E_{K1}[E_{K2}[P]]$?
  - Proven that it's NOT same as $C=E_{K3}[P]$
- Susceptible to *Meet-in-the-Middle Attack*
  - Described by Diffie & Hellman in 1977
  - Based on observation that if $C= E_{K2}[E_{K1}[P]]$, then $X=E_{K1}[P]=D_{K2}[C]$



(a) Double Encryption

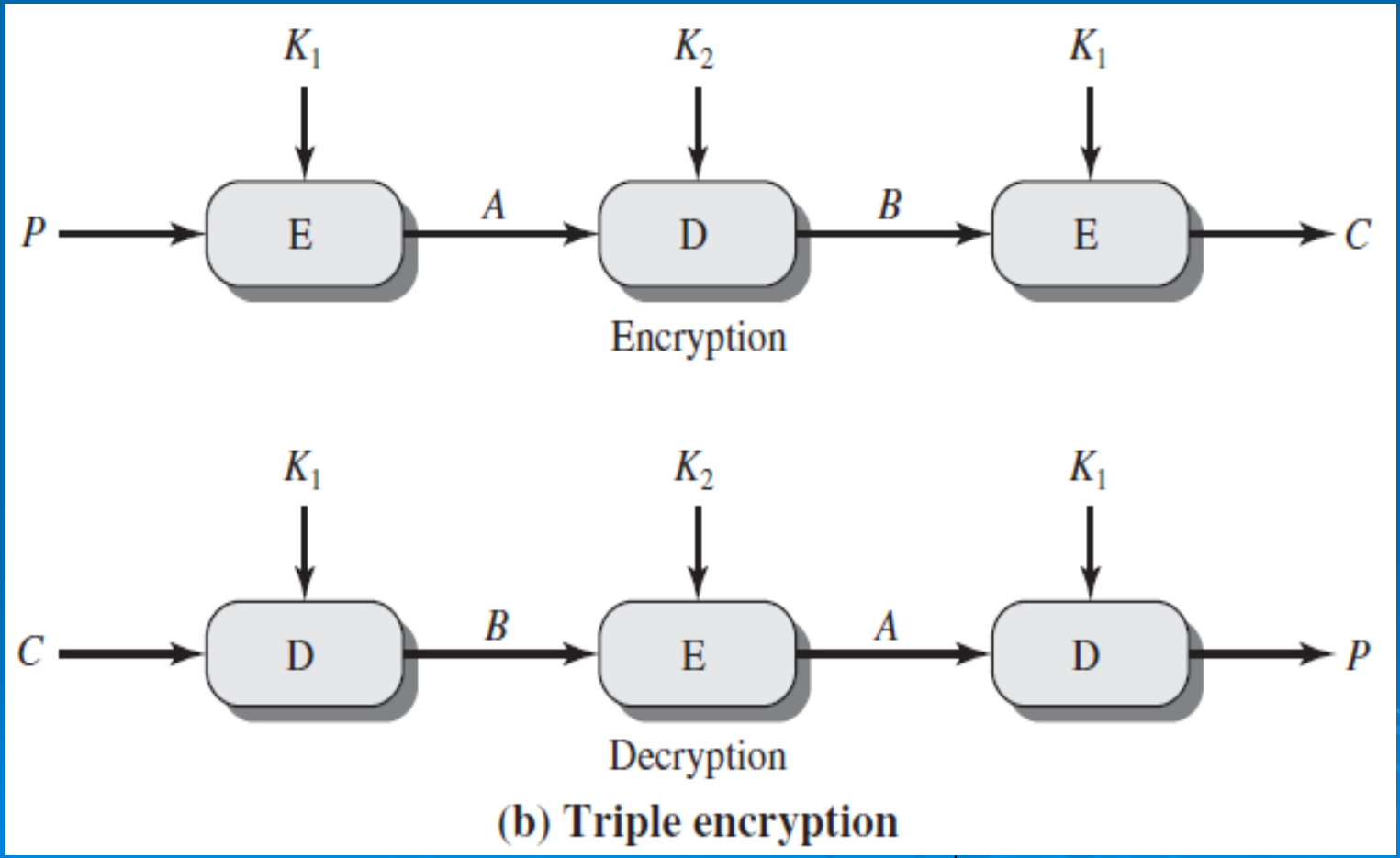
Why Triple-DES?

- Why not Double-DES?
  - NOT same as some other single-DES use, but have
- Meet-in-the-middle attack
  - works whenever use a cipher twice
  - since $X = E_{K1}[P] = D_{K2}[C]$
  - attack by encrypting P with all keys and store
  - then decrypt C with keys and match X value
  - can show takes $O(2^{56})$ steps

Triple DES

- A replacement for DES was needed
  - theoretical attacks that can break it
  - demonstrated exhaustive key search attacks
- AES is a new cipher alternative
- Before AES alternative
  - use multiple encryptions with DES
- Triple-DES is the chosen form

Triple DES



(b) Triple encryption

Triple-DES with Two-Keys

- Hence must use 3 encryptions
  - would seem to need 3 distinct keys
  - Key of 56 X 3 = 168 bits seems too long
- But can use 2 keys with E-D-E sequence
  - $C = E_{K1}[D_{K2}[E_{K1}[P]]]$
  - No cryptographic significance to the use of D in the second step
- Standardized in ANSI X9.17 & ISO8732
- No current known practical attacks
  - some are now adopting Triple-DES with three keys for greater security

Triple-DES with Three-Keys

- Although are no practical attacks on two-key Triple-DES have some indications
- Can use Triple-DES with Three-Keys to avoid even these
  - $C = E_{K3}[D_{K2}[E_{K1}[P]]]$
- Has been adopted by some Internet applications

Blowfish

- A symmetric block cipher designed by Bruce Schneier in 1993/94
- Characteristics
  - fast implementation on 32-bit CPUs, 18 clock cycles per byte
  - compact in use of memory, less than 5KB
  - simple structure for analysis/implementation
  - variable security by varying key size
    - Allows tuning for speed/security tradeoff

Blowfish Key Schedule

- uses a 32 to 448 bit key
- used to generate
  - 18 32-bit subkeys stored in P-array: P1 to P18
  - S-boxes stored in $S_{i,j}$,
    - $i=1..4$
    - $j=0..255$

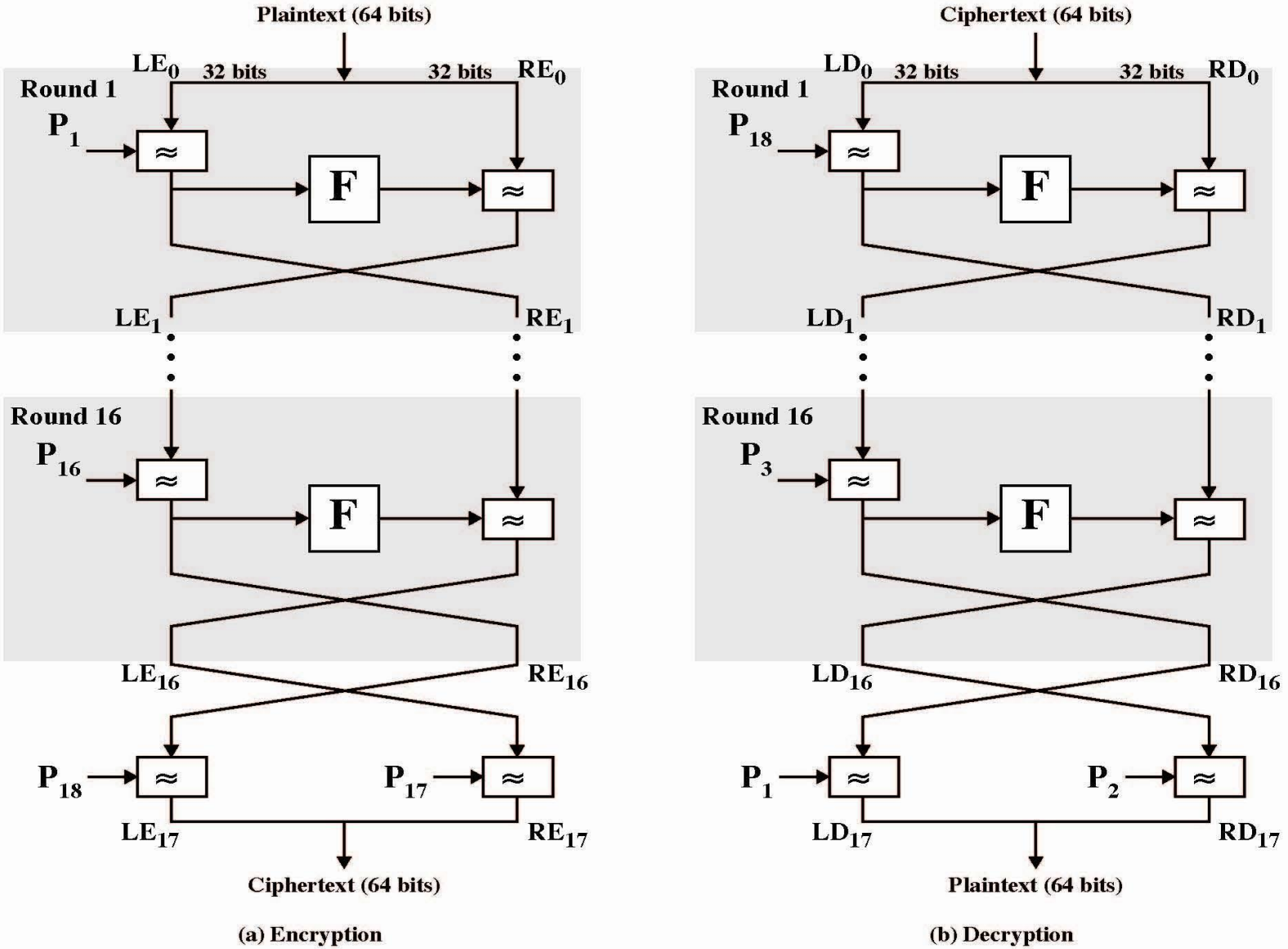


Figure 6.3 Blowfish Encryption and Decryption

Blowfish Encryption

- Uses two primitives: addition & XOR
- Data is divided into two 32-bit halves L_0 & R_0

for i = 1 to 16 do

$R_i = L_{i-1}$ XOR P_i;

L_i = F[R_i] XOR R_{i-1};

$L_{17} = R_{16}$ XOR P_{18};

$R_{17} = L_{16}$ XOR i_{17};

- Where

F[a,b,c,d] = (($S_{1,a}$ + $S_{2,b}$) XOR $S_{3,c}$) + $S_{4,a}$

Break 32-bit R_i into (a,b,c,d)

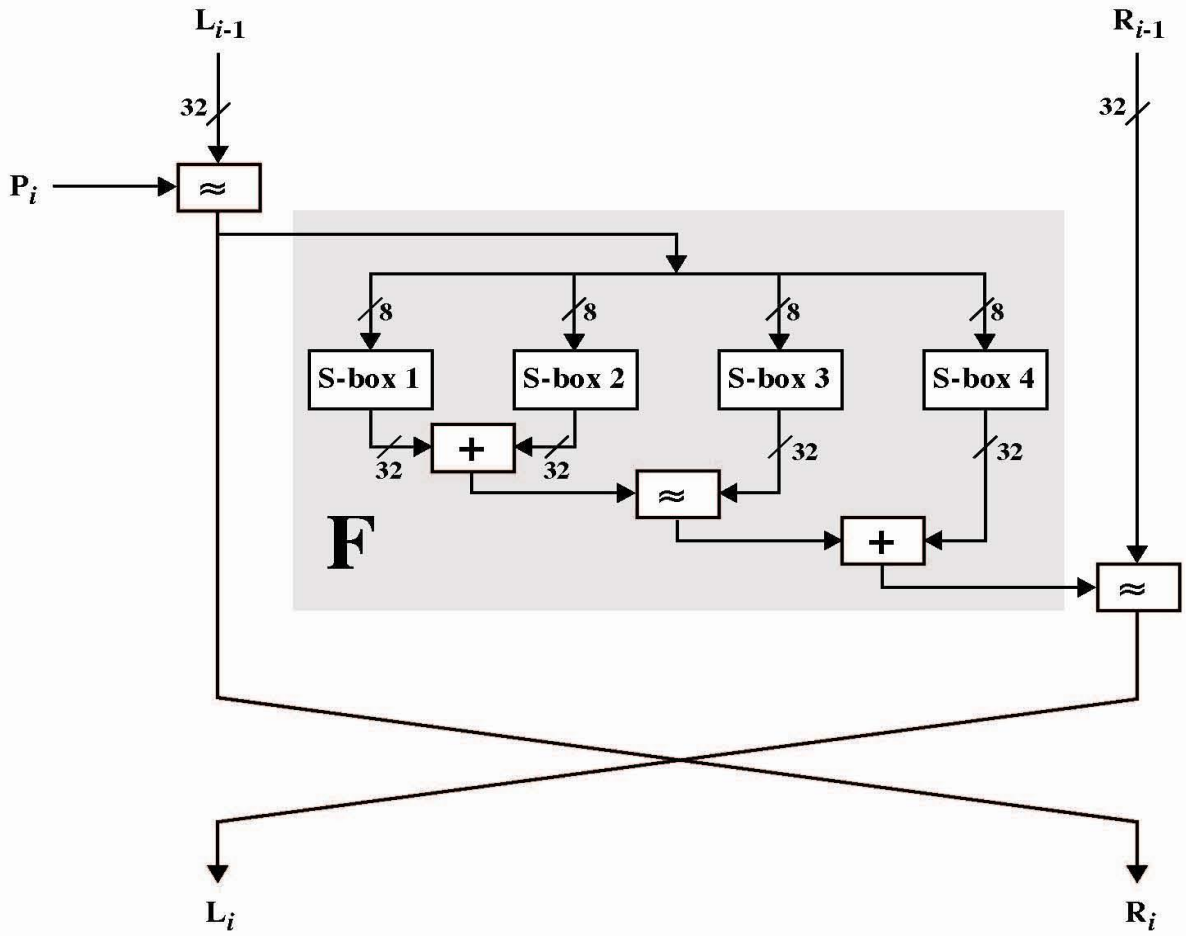


Figure 6.4 Detail of Single Blowfish Round

RC5

- Can vary key size / input data size /rounds
- Very clean and simple design
- Easy implementation on various CPUs
- Yet still regarded as secure
  - Vary parameters to achieve trade-offs

RC5 Ciphers

- RC5 is a family of ciphers RC5-w/r/b
  - w = word size in bits (16/32/64) data=2w
  - r = number of rounds (0..255)
  - b = number of bytes in key (0..255)
- nominal version is RC5-32/12/16
  - ie 32-bit words so encrypts 64-bit data blocks
  - using 12 rounds
  - with 16 bytes (128-bit) secret key

RC5 Key Expansion

- RC5 uses 2r+2 subkey words (w-bits)
  - Two subkeys for each round
  - 2 subkeys for additional operations
- subkeys are stored in array `S[i]`, i=0..t-1
- Key expansion: fill in pseudo-random bits to the original key K
- Certain amount of *one-wayness*
  - Difficult to determine K from S

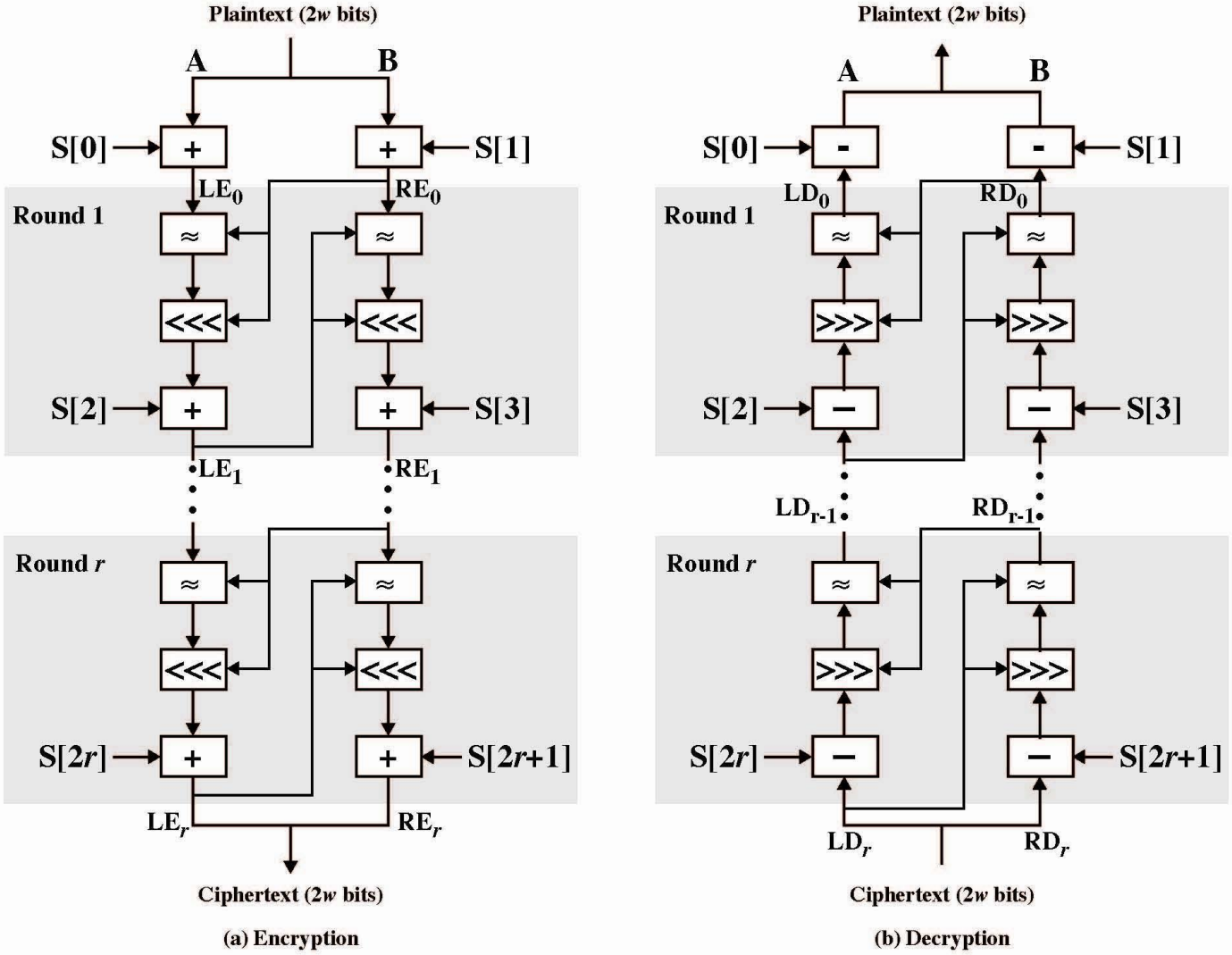


Figure 6.6 RC5 Encryption and Decryption

RC5 Encryption

- split input into two halves A & B

  $L_0 = A + S[0];$

  $R_0 = B + S[1];$

  for i = 1 to r do

  $L_i = ((L_{i-1} \text{ XOR } R_{i-1}) <<< R_{i-1}) + S[2 \times i];$

  $R_i = ((R_{i-1} \text{ XOR } L_i) <<< L_i) + S[2 \times i + 1];$
- each round is like 2 DES rounds
- note rotation is main source of non-linearity
- need reasonable number of rounds (eg 12-16)
- Striking features: simplicity, data-dependent rotations

RC5 Modes

- RFC2040 defines 4 modes used by RC5
  - RC5 Block Cipher, is ECB mode
  - RC5-CBC, input length is a multiples of 2w
  - RC5-CBC-PAD, any length CBC with padding
    - Output can be longer than input
  - RC5-CTS, CBC with padding
    - Output has same length than input

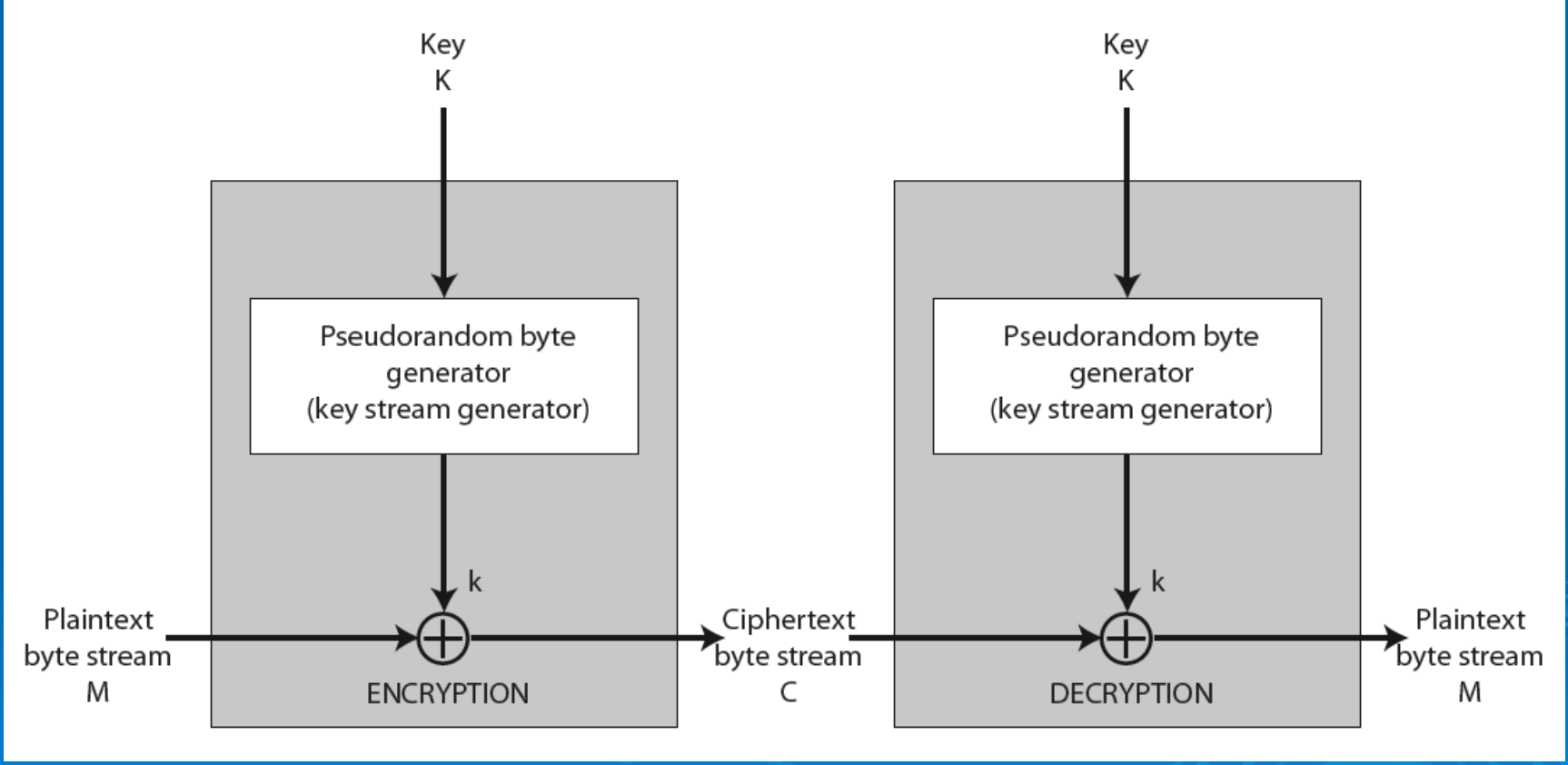
Advanced Block Cipher Characteristics

- Features seen in modern block ciphers are:
  - variable key length / block size / no rounds
  - mixed operators
    - data/key dependent rotation
    - key dependent S-boxes
  - more complex key scheduling
    - Lengthy key generation, simple encryption rounds
  - operation of full data in each round

Stream Ciphers

- Process message bit by bit (as a stream)
- Have a pseudo random **keystream**
- Combined (XOR) with plaintext bit by bit
- Randomness of **stream key** completely destroys statistically properties in message
  - $C_i = M_i \text{ XOR } StreamKey_i$
- But must never reuse stream key
  - otherwise can recover messages (cf book cipher)

Stream Cipher Structure



Stream Cipher Properties

- Some design considerations are:
  - long period with no repetitions
  - statistically random
  - depends on large enough key
  - large linear complexity
- Properly designed, can be as secure as a block cipher with same size key
- But usually simpler & faster

Block/Stream Ciphers

- Stream ciphers
  - For applications that require encryt/decryt of a stream of data
  - Examples: data communication channel, browser/web link
- Block ciphers
  - For applications dealing with blocks of data
  - Examples: file transfer, e-mail, database
- Either type can be used in virtually any application

RC4 Stream Cipher

- A proprietary cipher owned by RSA DSI
- Another Ron Rivest design, simple but effective
- Variable key size, byte-oriented stream cipher
- Widely used (web SSL/TLS, wireless WEP)
- Key forms random permutation of all 8-bit values
- 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
- Use key to well and truly shuffle
- S forms **internal state** of the cipher

```
for i = 0 to 255 do
   S[i] = i
   T[i] = K[i mod keylen])
j = 0
for i = 0 to 255 do
   j = (j + S[i] + T[i]) (mod 256)
   swap (S[i], S[j])
```

RC4 Encryption

- Encryption continues shuffling array values
- SSum of shuffled pair selects "stream key" value from permutation
- XOR S[t] with next byte of message to en/decrypt

```
i = j = 0
for each message byte Mi
    i = (i + 1) (mod 256)
    j = (j + S[i]) (mod 256)
    swap(S[i], S[j])
    t = (S[i] + S[j]) (mod 256)
    Ci = Mi XOR S[t]
```

RC4 Security

- 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