Encipherment Using Modern Symmetric-Key Ciphers

Dr. B C Dhara
Department of Information Technology
Jadavpur University

Objectives

- How modern standard ciphers, such as DES or AES, can be used to encipher long messages
- Discuss five modes of operation in modern block ciphers
- Which mode of operation creates stream ciphers out of the underlying block ciphers
- Discuss the security issues and the error propagation of different modes of operation
- Discuss two stream ciphers used for real-time processing of data

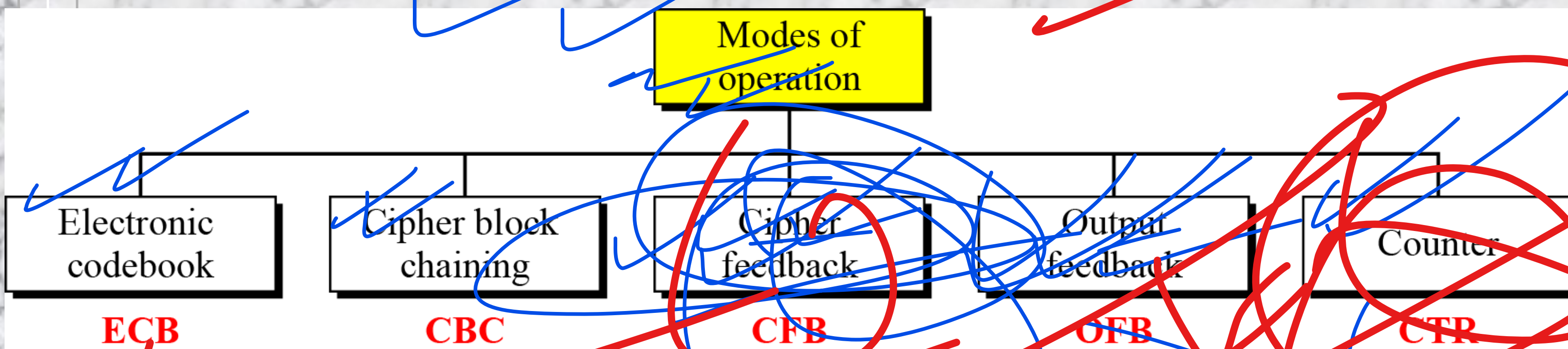
USE OF MODERN BLOCK CIPHERS

- *Symmetric-key encipherment can be done using modern block ciphers*
- *Modern ciphers like DES and AES are designed to encipher and decipher a block of text of fixed size*
- *A text to be enciphered is of variable size*
- ***Modes of operations** have been devised to encipher text of any size employing DES or AES*

Topics to be discussed

Electronic Codebook (ECB) Mode
Cipher Block Chaining (CBC) Mode
Cipher Feedback (CFB) Mode
Output Feedback (OFB) Mode
Counter (CTR) Mode

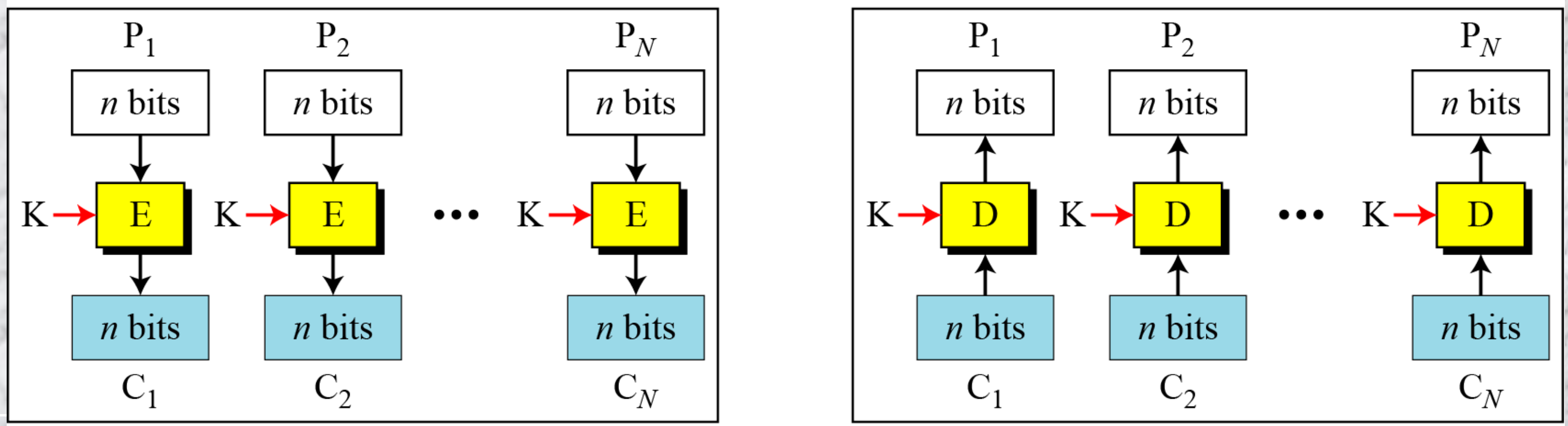
Modes of operation



Electronic Codebook (ECB) Mode

- This is the simplest mode
- The plaintext is divided into N blocks
  - Block size is n bits; ➔ total size of text is N*n bits
  - If needed then text padding is used
- Encryption: $C_i = E_K\ (P_i)$ Decryption: $P_i = D_K\ (C_i)$

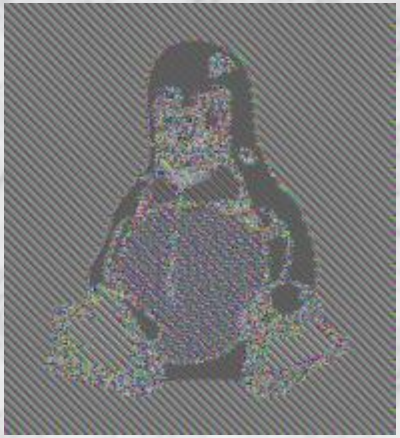
E: Encryption D: Decryption
P_i: Plaintext block *i* C_i: Ciphertext block *i*
K: Secret key



Encryption Decryption

Electronic Codebook (ECB) Mode (contd…)

- Same plaintext value will always result in the same ciphertext value
- ECB can leave plaintext data patterns in the ciphertext



- a bitmap image which uses large areas of uniform color
- Security can be improved by inclusion of random padding bits in each block

Pseudocode for ECB mode

- Block independency creates opportunities to exchange some ciphertext blocks without knowing the key.

Algorithm 8.1 *Encryption for ECB mode*

```
ECB_Encryption (K, Plaintext blocks)
{
    for (i = 1 to N)
    {
        C_i ← E_K (P_i)
    }
    return Ciphertext blocks
}
```

Error Propagation

- A single bit error in transmission can create errors in several in the corresponding block. However, the error does not have any effect on the other blocks

independent block.

Ciphertext Stealing

- Padding added (if required) to the last block
  - The size of the ciphertext would be greater size, disadvantage
- Ciphertext stealing is a technique for encrypting plaintext using a block cipher, without padding the message to a multiple of the block size, so the ciphertext is the same size as the plaintext

Ciphertext Stealing (contd…)

- The processing of all but the last two blocks is unchanged
- Change the processing of the last two blocks of the message
- A portion of the *second*-last block's ciphertext is "stolen" to pad the last plaintext block
- The padded final block is then encrypted as usual.

Ciphertext Stealing (contd…)

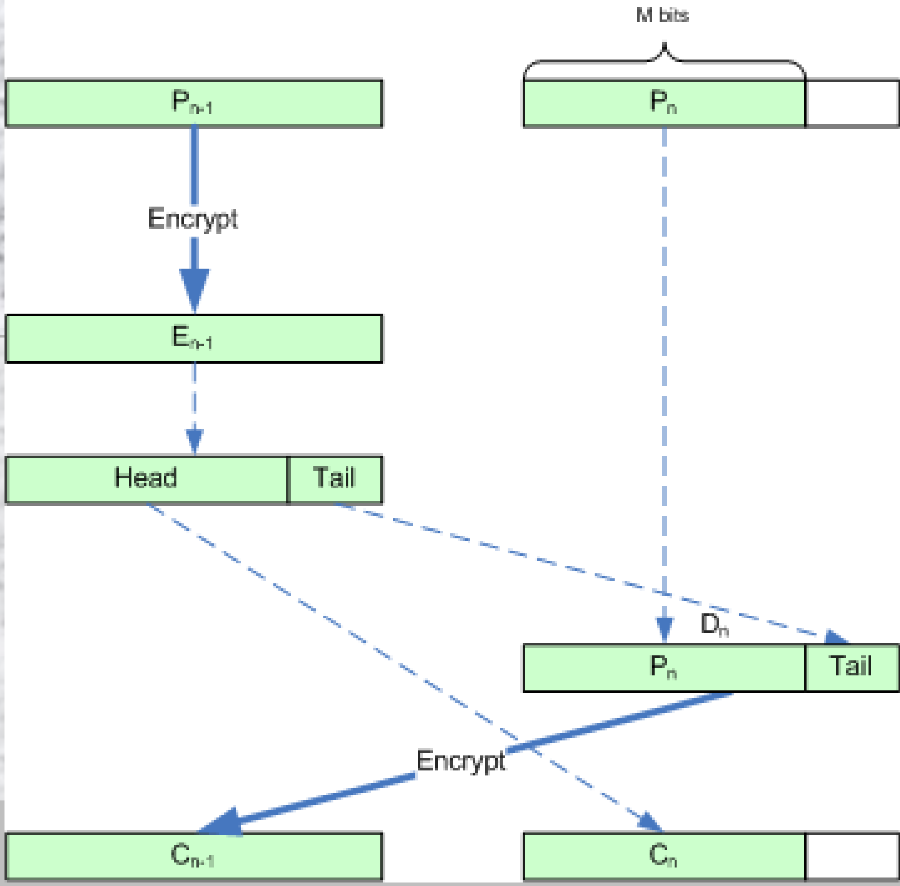
- The final ciphertext consists
  - Ciphertext of all blocks but last two blocks
  - For the last two blocks,
    - The partial penultimate block (with the "stolen" portion omitted) plus the full final block
- The size of the ciphertext is same size as the original plaintext
- Decryption requires decrypting the final block first
  - then restoring the stolen ciphertext to the penultimate block, which can then be decrypted as usual

- This step describes how to handle the last two blocks called P_{n-1} and P_n, of the plaintext and generates two ciphertext C_{n-1} and C_n
  - The length of P_{n-1} equals the block size of the cipher in bits, B
  - The length of the last block, P_n, is M bits
  - K is the key that is in use
  - M can range from 1 to B, inclusive, so P_n could possibly be a complete block

- Following functions and operators are used:
  - Head (data, *a*): returns the first *a* bits of the 'data' string
  - Tail (data, *a*): returns the last *a* bits of the 'data' string
  - Encrypt (*K*, data): use the underlying block cipher in encrypt mode on the 'data' string using the key *K*
  - Decrypt (*K*, data): use the underlying block cipher in decrypt mode on the 'data' string using the key *K*
  - XOR: Bitwise Exclusive-OR
  - ||: Concatenation operator
  - 0^a: a string of *a* 0 bits

Ciphertext Stealing in ECB

- Ciphertext stealing for ECB mode requires the plaintext to be longer than one block
- **ECB encryption steps**



1. E_{n-1} = Encrypt (K, P_{n-1})
2. C_n = Head (E_{n-1}, M). Select the first M bits of E_{n-1} to create C_n
3. $D_n = P_n$ || Tail (E_{n-1}, $B-M$). Pad P_n with the low order bits from E_{n-1}.
4. C_{n-1} = Encrypt (K, D_n), Encrypt D_n to create C_{n-1}
   - The first M bits, this is equivalent to what happen in ECB mode
   - The last $B-M$ bits, this is the second time that these data have been encrypted under this key (It was already encrypted in the production of E_{n-1} in step 2).

Ciphertext Stealing in ECB (contd…)

- Decryption

1. D_n = Decrypt (K, C_{n-1})
2. $E_{n-1} = C_n$ || Tail (D_n, $B-M$). Pad C_n with the extracted ciphertext in the tail end of D_n
3. P_n = Head (D_n, M). Select the first M bits of D_n to create P_n
4. P_{n-1} = Decrypt (K, E_{n-1}), Decrypt E_{n-1} to create P_{n-1}

- Error propagation:

1. In ECB a bit error in transmission has no effect on other block
2. A bit error in the transmission of C_{n-1} would result in the block-wide corruption of both P_{n-1} and P_n
3. A bit error in the transmission of C_n would result in the block-wide corruption of P_{n-1}.

Applications of ECB

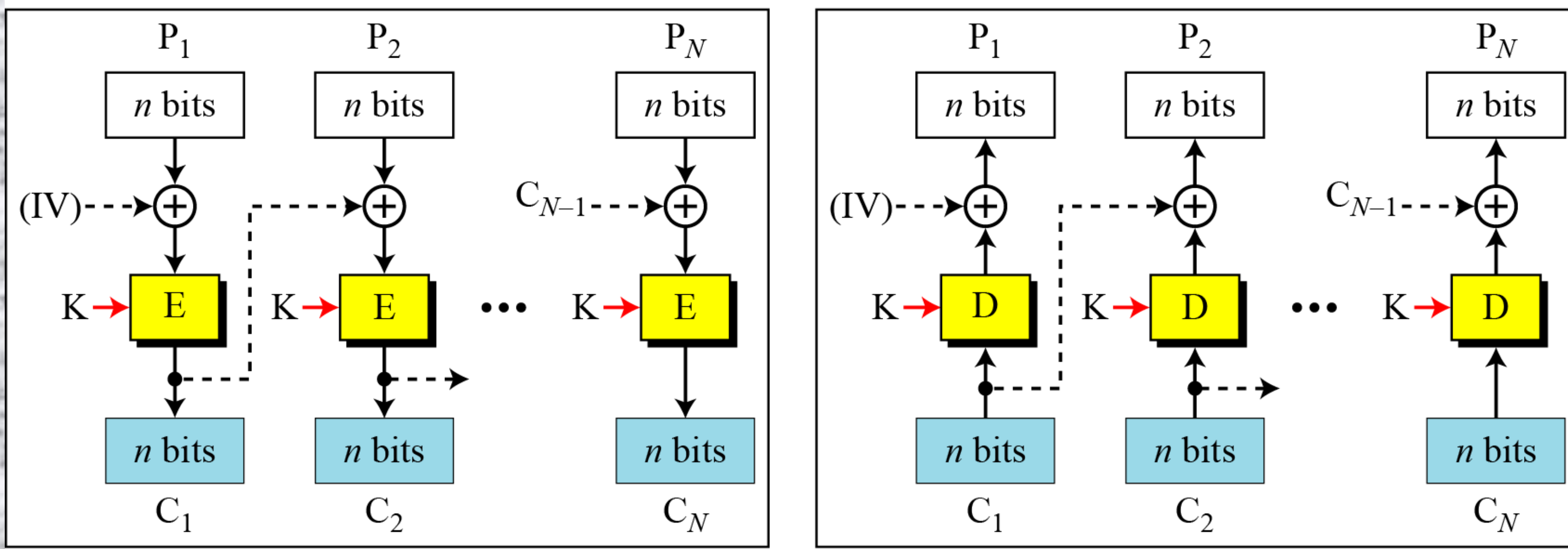
- Use of ECB is not recommended if the message length is more than one block
- Area where independency of the ciphertext block is useful like, database application, where records need to be encrypted before stored or decrypted before retrieved
- Parallel processing possible

Cipher block chaining mode (CBC)

- *In CBC mode, each plaintext block is XOR-ed with*
- *th ... ted*
- *For first block, there is a initialization vector (IV)*

Encryption:	**Decryption:**
$C_0 = IV$	$C_0 = IV$
$C_i = E_K(P_i \oplus C_{i-1})$	$P_i = D_K(C_i) \oplus C_{i-1}$

E: Encryption D : Decryption
P_i: Plaintext block i C_i : Ciphertext block i
K: Secret key IV: Initial vector (C_0)



Encryption Decryption

CBC mode

- It can be proved that each plaintext block is recovered exactly
  - Because encryption and decryption are inverses of each other

$$P_i = D_K(C_i) \oplus C_{i-1} = D_K(E_K(P_i \oplus C_{i-1})) \oplus C_{i-1} = P_i \oplus C_{i-1} \oplus C_{i-1} = P_i$$

- *The initialization vector (IV) should be known by the sender and the receiver*
  - *Keeping the vector as secret is not necessary*
    - *It should be kept safe from change*

Security issues

- Exactly same plaintext block within same message are enciphered into different ciphertext blocks
  - The patterns at block level is not preserved
  - If two messages are equal and their ciphertexts are also same only when their IV are same

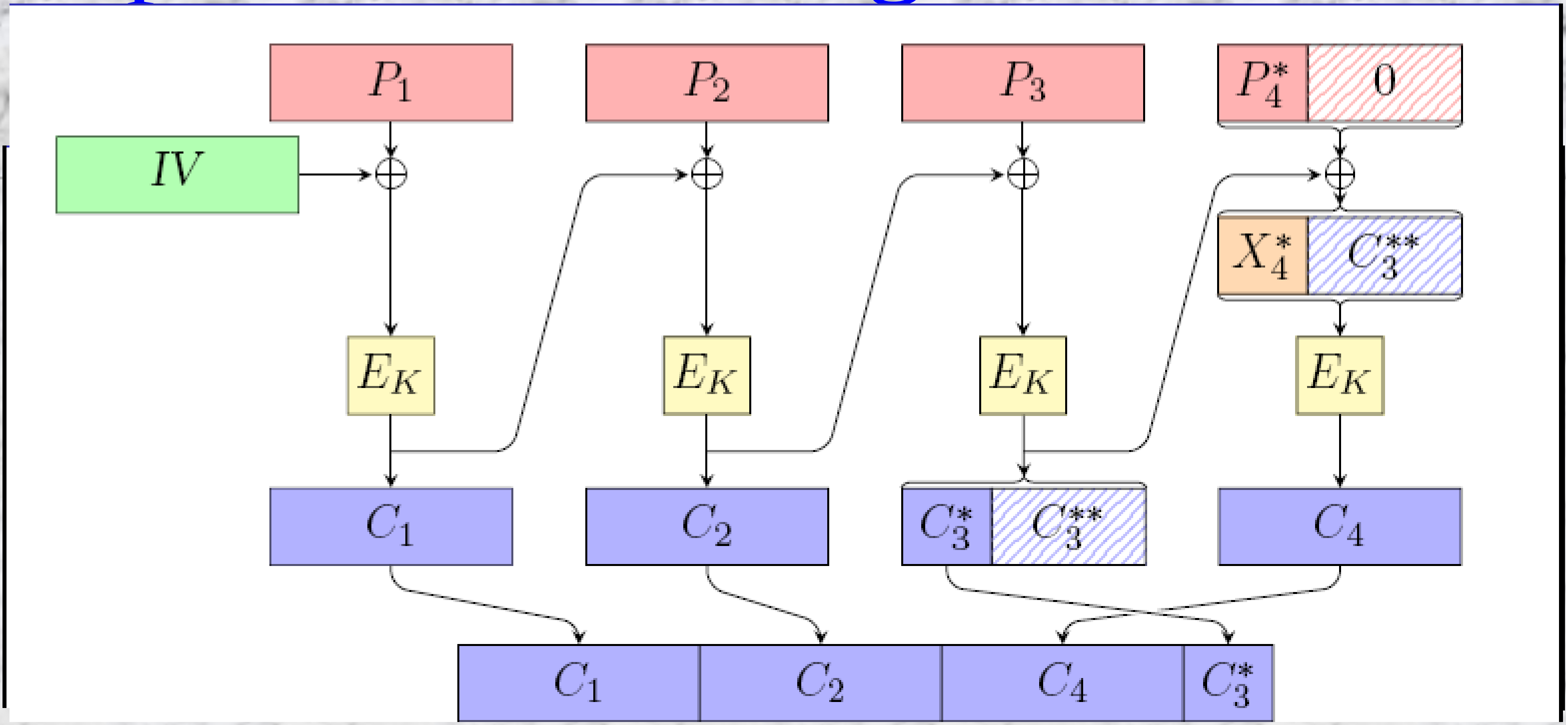
Error Propagation

- In CBC mode, a single bit error in ciphertext block C_j during transmission may create error in most bits in plaintext block P_j during decryption
- Single bit error in P_{j+1} (the bit at the same location)
- P_{j+2} to P_N are not affected by this single bit error

Pseudocode for CBC mode

```
CBC_Encryption (IV, K, Plaintext blocks)
{
    C0 ← IV
    for (i = 1 to N)
    {
        Temp ← Pi ⊕ Ci−1
        Ci ← EK (Temp)
    }
    return Ciphertext blocks
}
```

Ciphertext Stealing



1. $X_{n-1} = P_{n-1}$ XOR C_{n-2} // the behavior of standard CBC mode
2. E_{n-1} = Encrypt (K, X_{n-1}). //the behavior of standard CBC mode
3. C_n = Head (E_{n-1}, M)// Select the first M bits of E_{n-1} to create C_n
4. $P = P_n \,||\, 0^{B-M}$ //Pad P_n with zeros to create P of length B
5. $D_n = E_{n-1}$ XOR P
6. C_{n-1} = Encrypt (K, D_n)

Applications of CBC mode

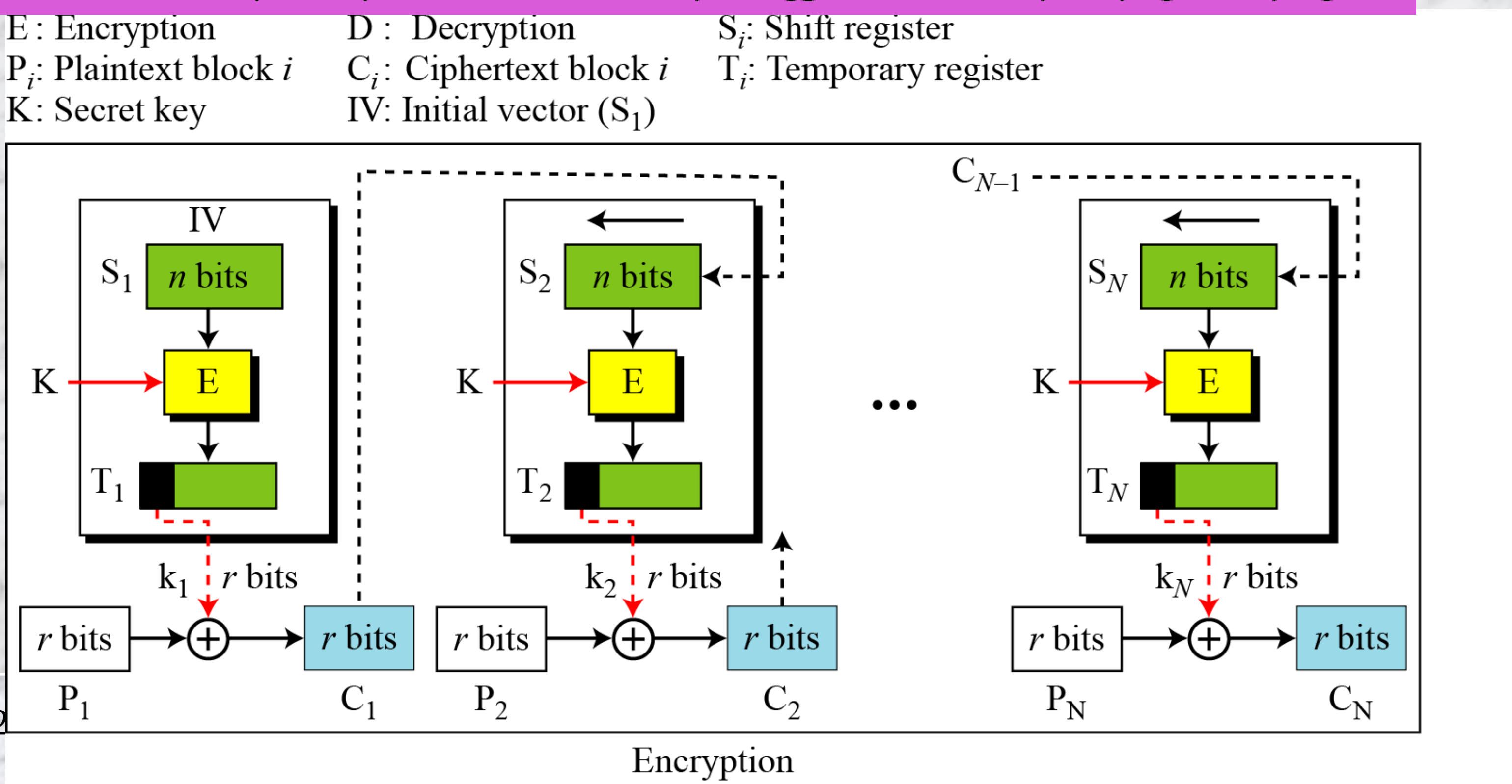
- CBC mode can be used to encipher messages
- Due to chaining mechanism, parallel processing is not possible
- Cannot be used to encrypt or decrypt random access files

Cipher Feedback (CFB) Mode

- *In some situations, we need to use DES or AES as secure ciphers, but the plaintext or ciphertext block sizes are to be smaller*
- U ved t

Encryption: $C_i = P_i \oplus \text{SelectLeft}_r\{E_K[\text{ShiftLeft}_r(S_{i-1}) \mid C_{i-1})]\}$

Decryption: $P_i = C_i \oplus \text{SelectLeft}_r\{E_K[\text{ShiftLeft}_r(S_{i-1}) \mid C_{i-1})]\}$

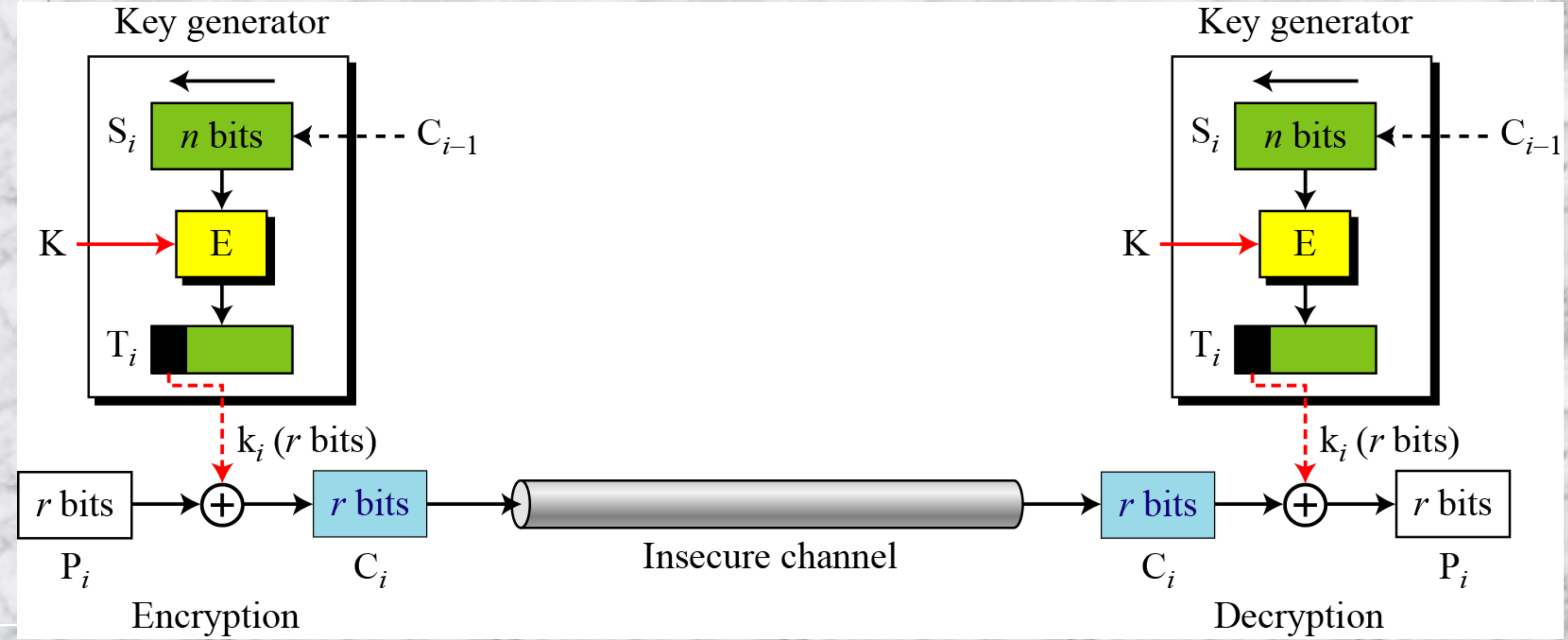


Cipher Feedback (CFB) Mode (contd…)

- No padding is required because the size of the block, *r*, is normally chosen to fit the data unit to be encrypted
- CFB is less efficient than CBC or ECB
  - As it applied encryption function for small block size

CFB as a Stream Cipher

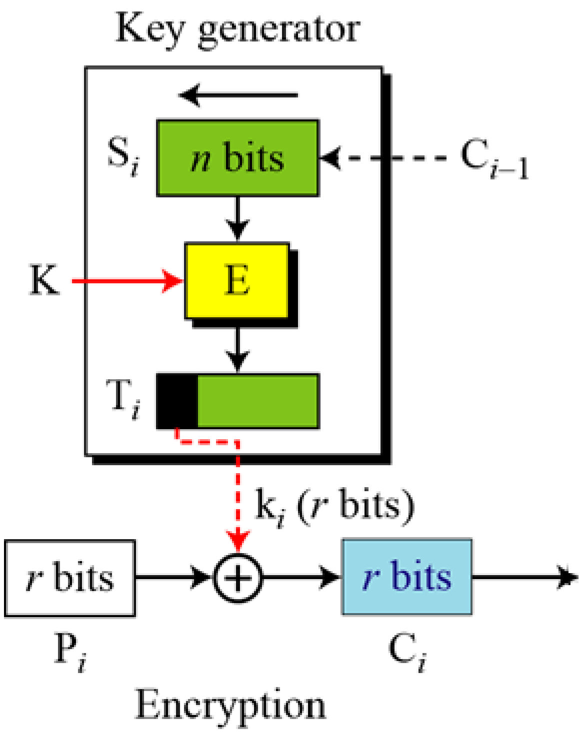
- This is non-synchronous stream cipher, key stream dependent on ciphertext



Pseudocode for CFB

Algorithm 8.3 *Encryption algorithm for CFB*

```
CFB_Encryption (IV, K, r)
{
   i ← 1
   while (more blocks to encrypt)
   {
   input (P_i)
   if (i = 1)
      S ← IV
   else
      {
      Temp ← shiftLeft_r (S)
      S ← concatenate (Temp, C_{i−1})
      }
   T ← E_K(S)
   k_i ← selectLeft_r (T)
   C_i ← P_i ⊕ k_i
   output (C_i)
   i ← i + 1
   }
}
```



Security issues

- Block level pattern is not preserved
- An attacker can add some ciphertext block at the end of ciphertext stream

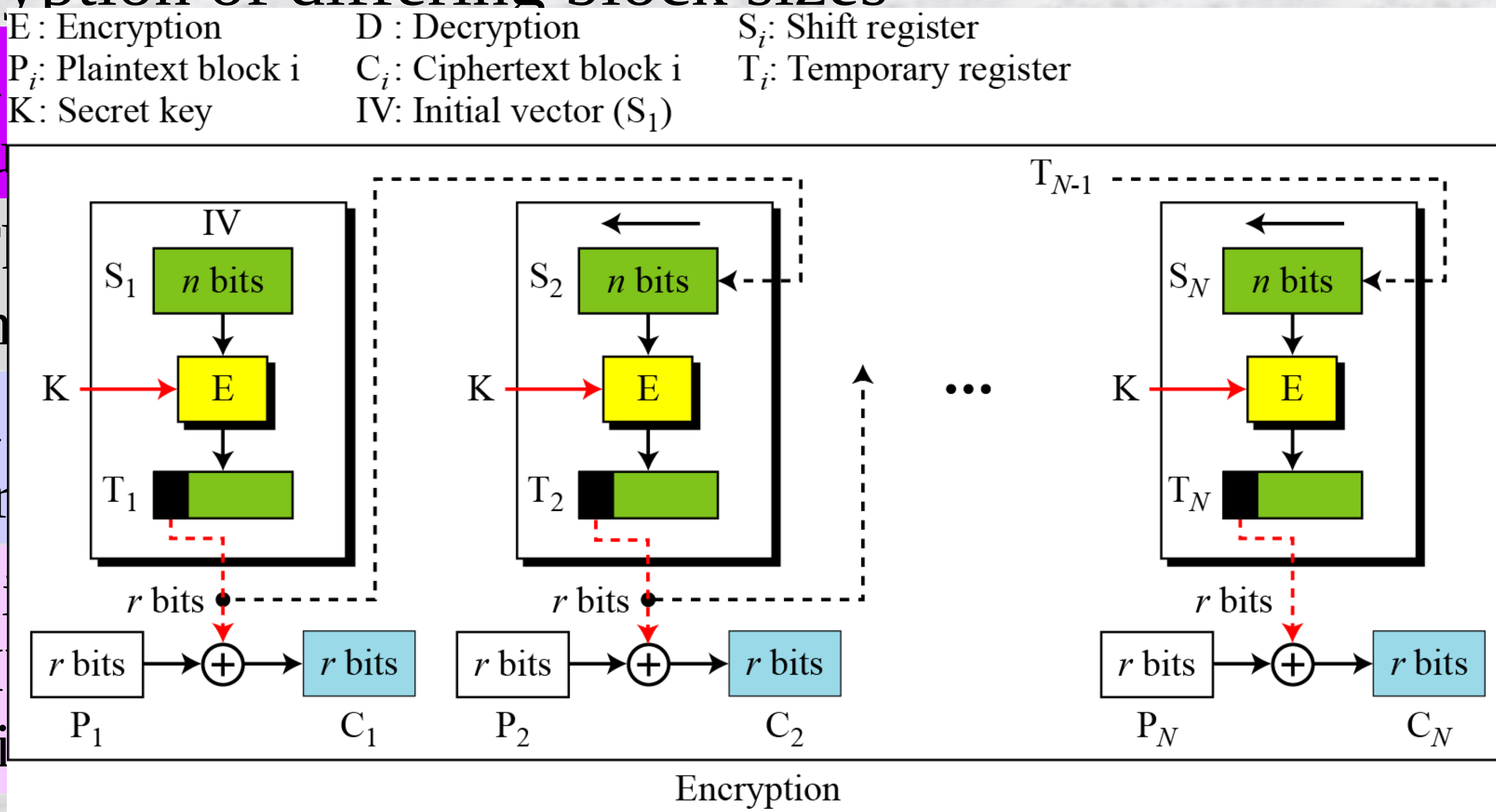
Error propagation

$1 \to 1$ (bit

- A single bit error in ciphertext block Cj during transmission creates a single bit error (at the same position) in plaintext block Pj
- As long as bits of Cj present in the shift register, most of the bits of the subsequent blocks are in error (with 50% probability)

Output Feedback (OFB) Mode

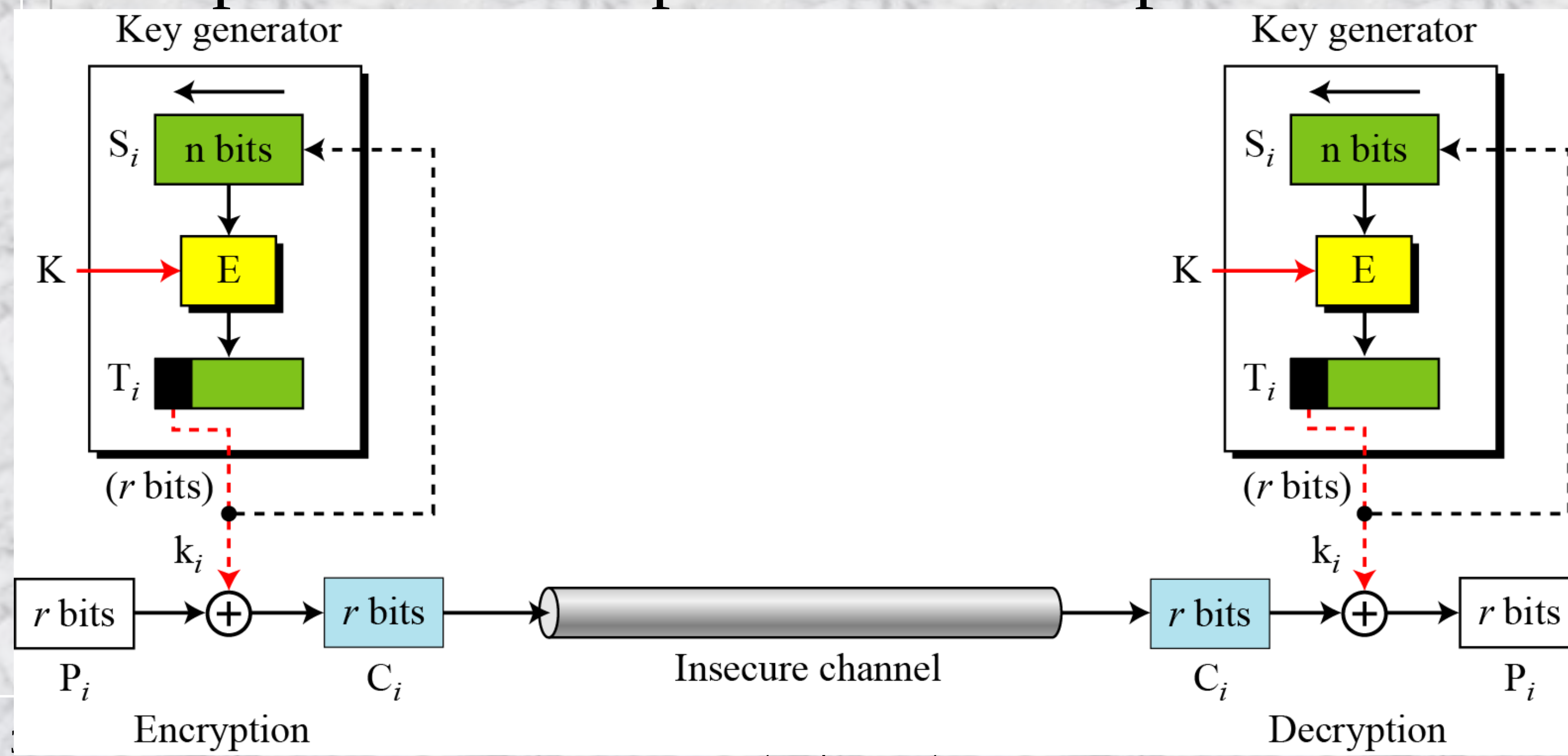
- It has some similarities to the CFB mode in that it permits encryption of differing block sizes



It ... k
fu...
Th... both
th...
It ...
er...
Li... ctor
(I...
di...

OFB as a Stream Cipher

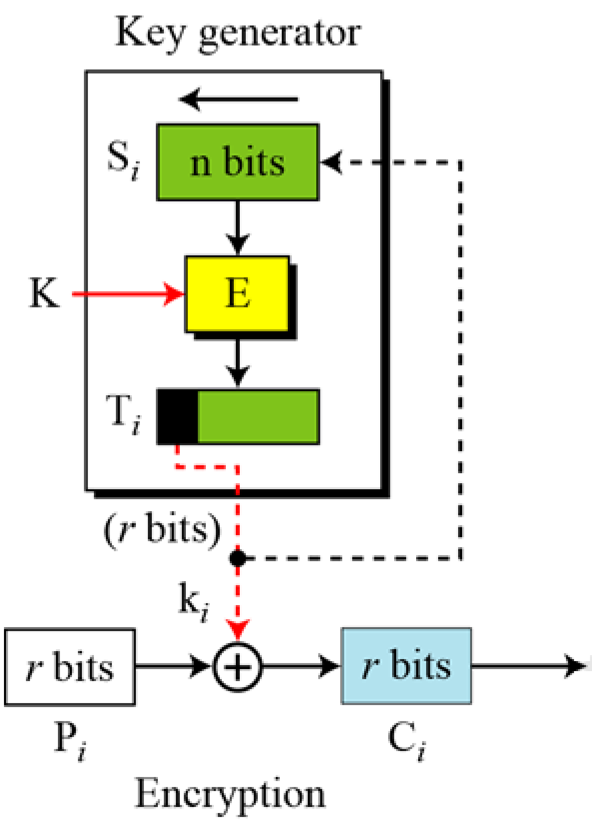
- This is synchronous stream cipher as key stream independent from plaintext and ciphertext



Pseudocode for OFB mode

Algorithm 8.4 *Encryption algorithm for OFB*

OFB_Encryption (IV, K, r)
{
    $i \leftarrow 1$
    while (more blocks to encrypt)
    {
        input (P_i)
        if ($i = 1$) $S \leftarrow IV$
        else
        {
            Temp $\leftarrow$ **shiftLeft**$_r$ (S)
            S $\leftarrow$ **concatenate** (Temp, k_{i-1})
        }
        $T \leftarrow E_K(S)$
        $k_i \leftarrow$ **selectLeft**$_r$ (T)
        $C_i \leftarrow P_i \oplus k_i$
        output (C_i)
        $i \leftarrow i + 1$
    }
}

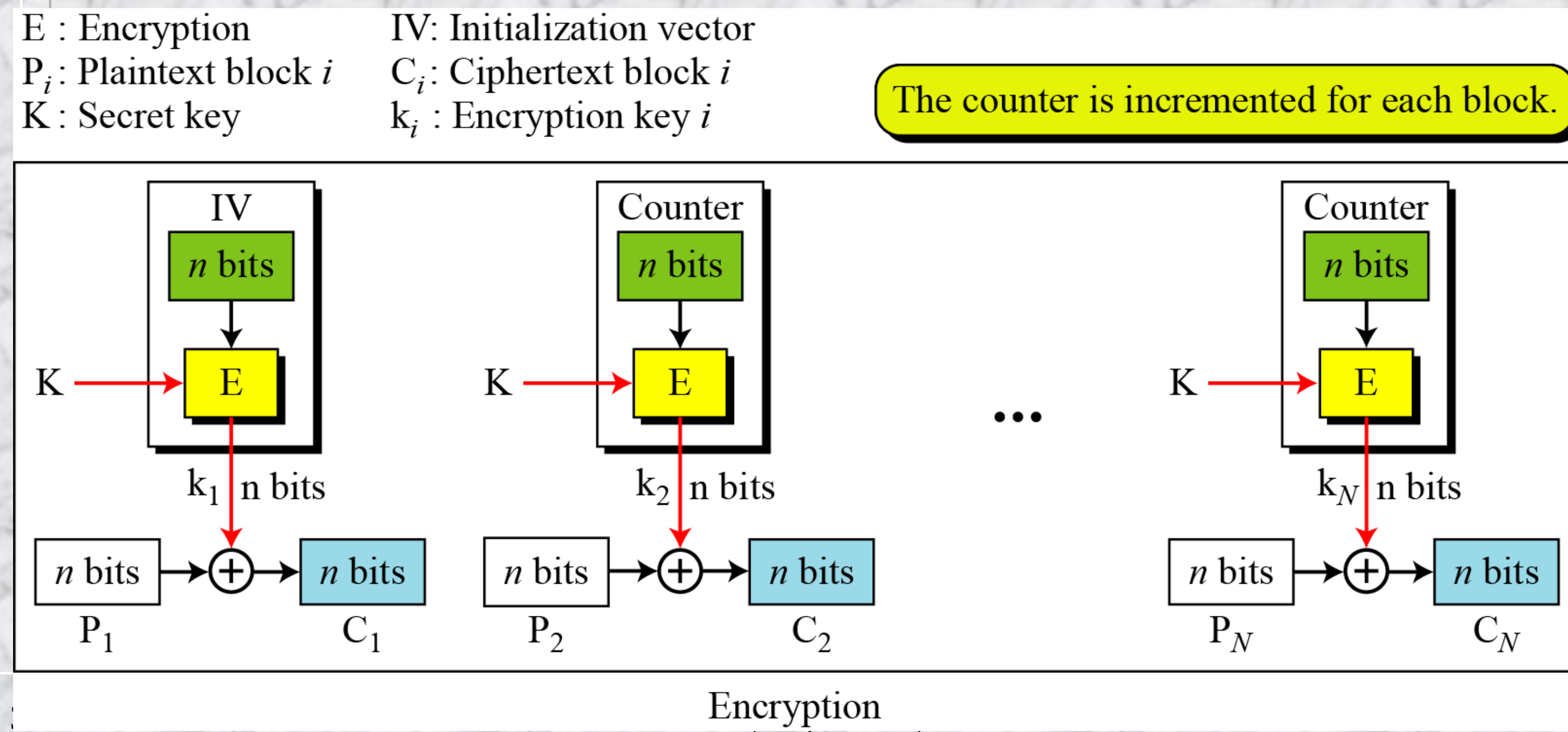


Output Feedback (OFB) Mode (contd…)

- Security issues:
  - Patterns at the block level are not preserved
  - Any change in the ciphertext affects the correspoing plaintext only
- Error propagation:
  - A single error in the ciphertext affects only the corresponding bit the plaintext

Counter (CTR) mode

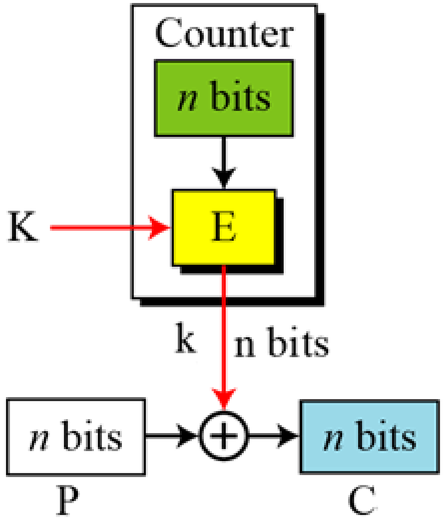
- *In the counter (CTR) mode, there is no feedback*



Pseudocode for CTR mode

Algorithm 8.5 *Encryption algorithm for CTR*

CTR_Encryption (IV, K, Plaintext blocks)
{
    Counter $\leftarrow$ IV
    for (i = 1 to N)
    {
        Counter $\leftarrow$ (Counter + i − 1) mod 2^N
        $k_i \leftarrow E_K$ (Counter)
        $C_i \leftarrow P_i \oplus k_i$
    }
    return Ciphertext blocks
}



Counter (CTR) mode (contd...)

- Security issues:
  - Same as that of OFB mode
- Error propagation:
  - A single error in the ciphertext affects only the corresponding bit in the plaintext

Comparison of Different Modes

Table 8.1 *Summary of operation modes*

Operation Mode	*Description*	*Type of Result*	*Data Unit Size*
ECB	Each n-bit block is encrypted independently with the same cipher key.	Block cipher	n
CBC	Same as ECB, but each block is first exclusive-ored with the previous ciphertext.	Block cipher	n
CFB	Each r-bit block is exclusive-ored with an r-bit key, which is part of previous cipher text	Stream cipher	$r \leq n$
OFB	Same as CFB, but the shift register is updated by the previous r-bit key.	Stream cipher	$r \leq n$
CTR	Same as OFB, but a counter is used instead of a shift register.	Stream cipher	n

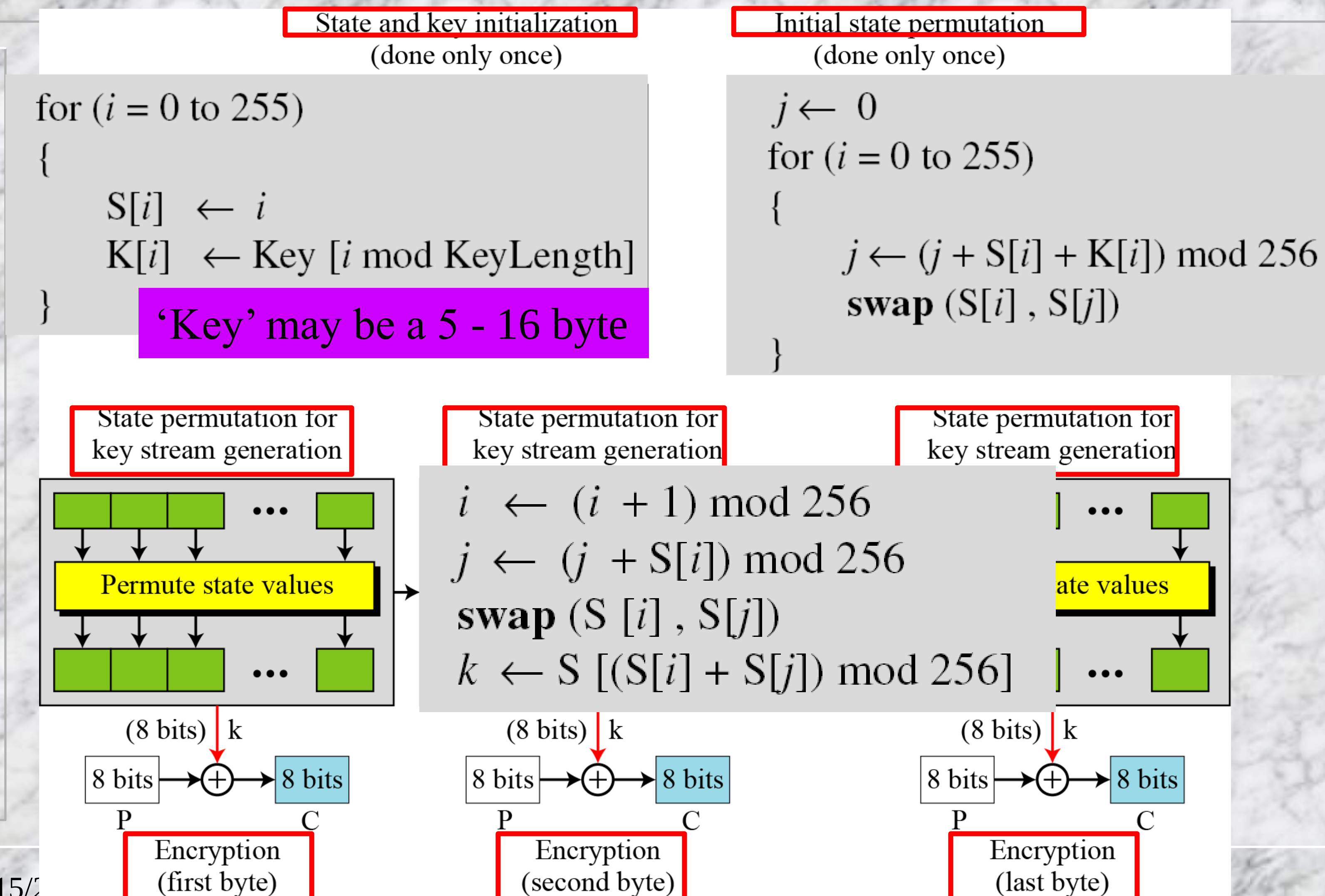
Use of Stream Ciphers

- *Five modes of operations enable the use of block ciphers for encipherment of messages or files in large units and small units*
- *Sometimes pure stream are needed for enciphering small units of data such as characters or bits*
- *Stream ciphers are more efficient for real-time processing*
- *We discuss two stream ciphers:* RC4, A5/1

RC_4

- *RC4 is a byte-oriented stream cipher in which a byte (8 bits) of a plaintext is exclusive-ORed with a byte of key to produce a byte of a ciphertext*
- *RC_4 is based on the concept of a state*
  – *State is a byte, stored in an array S[0...255]*
  – *Randomly one byte is selected to serve as the key for encryption*

Idea of RC_4 stream cipher



Pseudocode for RC_4

Algorithm 8.6 *Encryption algorithm for RC4*

```
RC4_Encryption (K)
{
    // Creation of initial state and key bytes
    for (i = 0 to 255)
    {
        S[i] ← i
        K[i] ← Key [i mod KeyLength]
    }
    // Permuting state bytes based on values of key bytes
    j ← 0
    for (i = 0 to 255)
    {
        j ← (j + S[i] + K[i]) mod 256
        swap (S[i] , S[j])
    }
    // Continuously permuting state bytes, generating keys, and encrypting
    i ← 0
    j ← 0
    while (more byte to encrypt)
    {
        i ← (i + 1) mod 256
        j ← (j + S[i]) mod 256
        swap (S [i] , S[j])
        k ← S [(S[i] + S[j]) mod 256]
        // Key is ready, encrypt
        input P
        C ← P ⊕ k
        output C
    }
}
```

Randomness of key stream

A secret key with all bytes set to 0

The key stream for 20 values of *k* is (222, 24, 137, 65, 163, 55, 93, 58, 138, 6, 30, 103, 87, 110, 146, 109, 199, 26, 127, 163)

The secret key is of five bytes of (15, 202, 33, 6, 8)

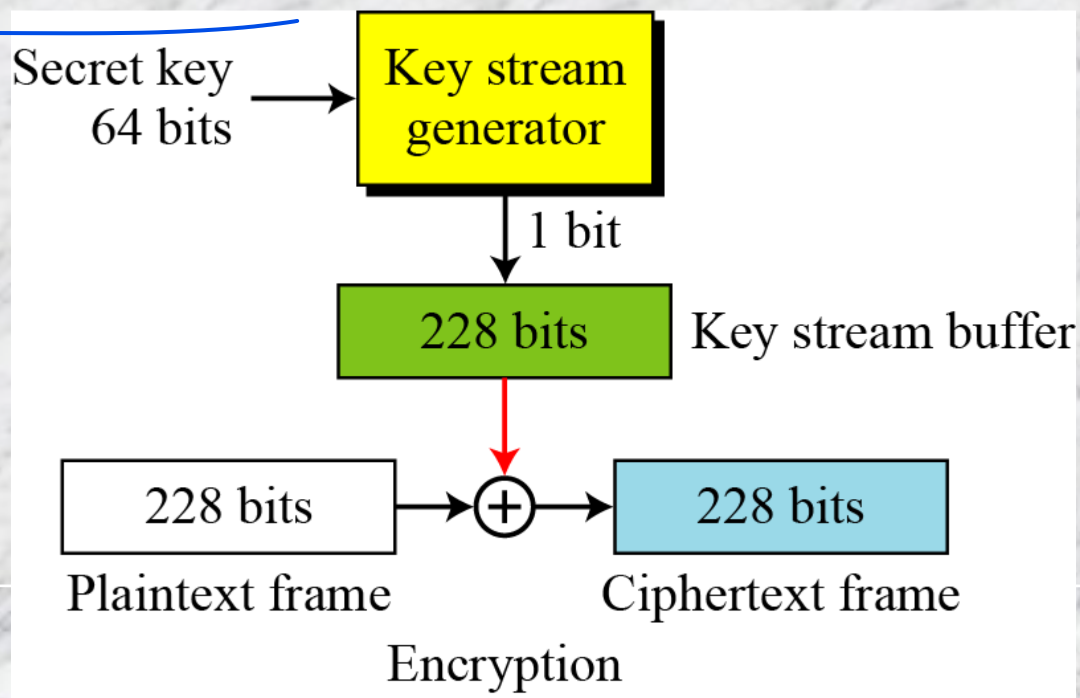
The key stream for 20 values is (248, 184, 102, 54, 212, 237, 186, 133, 51, 238, 108, 106, 103, 214, 39, 242, 30, 34, 144, 49)

Security issues

- Cipher is secure if key size is at least 128 bits
- Some reported attacks for smaller key (size less than 5 bytes)
- It is recommended that different keys should be used for different sessions

A5/1

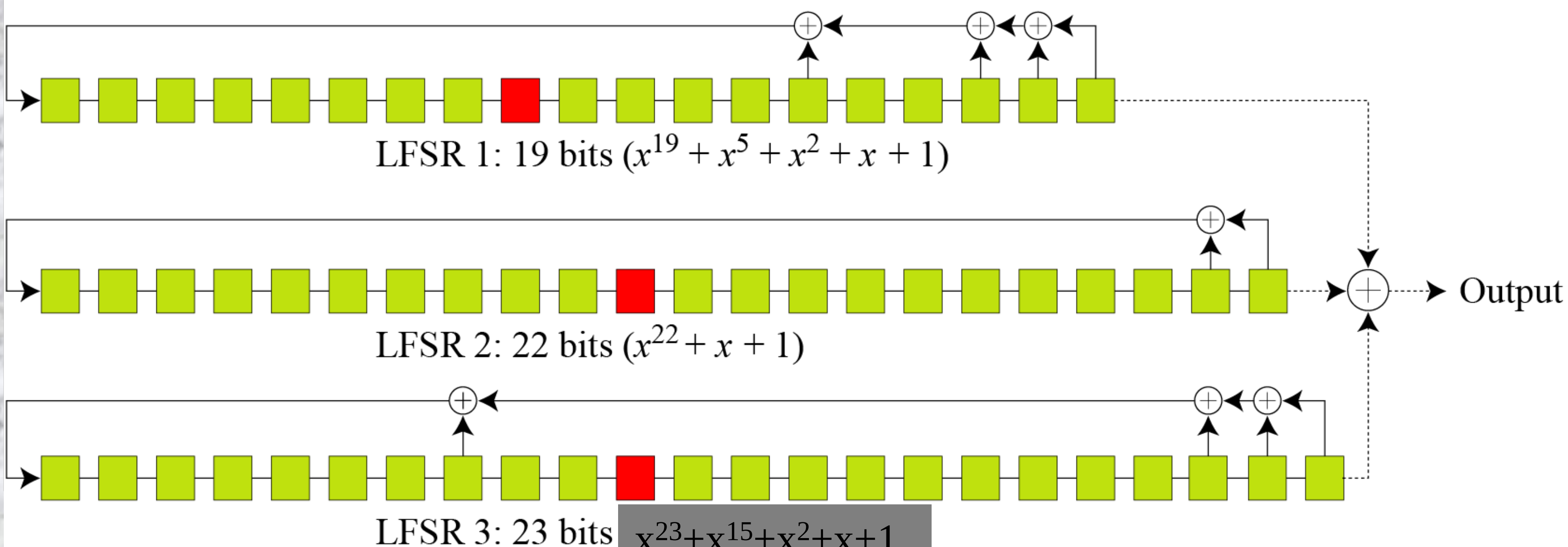
- A5/1 is a stream cipher which uses LFSR to generates bit stream
- Key stream is generated using 64 bits
- Key stream is stored in a 228-bit buffer and XOR-ed with a 228-bit stream



Key Generator

- A5/1 uses three LFSRs with 19, 22, and 23 bits
- Red one is the clock bit

Note: The three red boxes are used in the majority function



$x^{23}+x^{15}+x^2+x+1$

Key Generator (contd…)

- Initialization: 64-bit secret key, 22 bit frame number are used
  1. *set all bits in three LFSRs to 0*
  2. *Mixing with key value and clocking means shifting LFSR*

```
for (i = 0 to 63)
  {
     Exclusive-or K[i] with the leftmost bit in all three registers.
     Clock all three LFSRs
  }
```

  3. *Mix with 22 bit frame number*

```
for (i = 0 to 21)
  {
     Exclusive-or FrameNumber [i] with the leftmost bit in all three registers.
     Clock all three LFSRs
  }
```

Key Generator (contd…)

4. Clock key generator 100 cycle, using majority function

```
for (i = 0 to 99)
  {
    Clock the whole generator based on the majority function.
  }
```

2. Key stream bits
   a. Key generator creates one bit in each click
   b. Input to the majority function are the clicking bits
   c. It clicking bit matched with output of majority function, the LFSR will be clocked; otherwise not
   d. At a time two or three LFSRs will be clocked

Example

At a point of time the clocking bits are 1, 0, and 1. Which LFSR is clocked (shifted)?

Solution

The result of Majority (1, 0, 1) = 1. LFSR1 and LAFS3 are shifted, but LFSR2 is not.

Encryption/Decryption

- *The bit streams created from the key generator are buffered to form a 228-bit key*
- *XOR-ed with the plaintext frame to create the ciphertext frame*
- *Encryption/decryption is done one frame at a time.*

Key management

- *Alice and Bob need to share a secret key between themselves to securely communicate using a symmetric-key cipher. If there are n entities in the community, $n(n-1)/2$ keys are needed*