STREAM CIPHERS

Stream Ciphers Design Issues

- A typical stream cipher encrypts plaintext one byte (or bit) at a time
- A key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random, called a **keystream**
- This is combined one byte at a time with the plaintext stream using XOR operation.

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

Encryption

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

Decryption

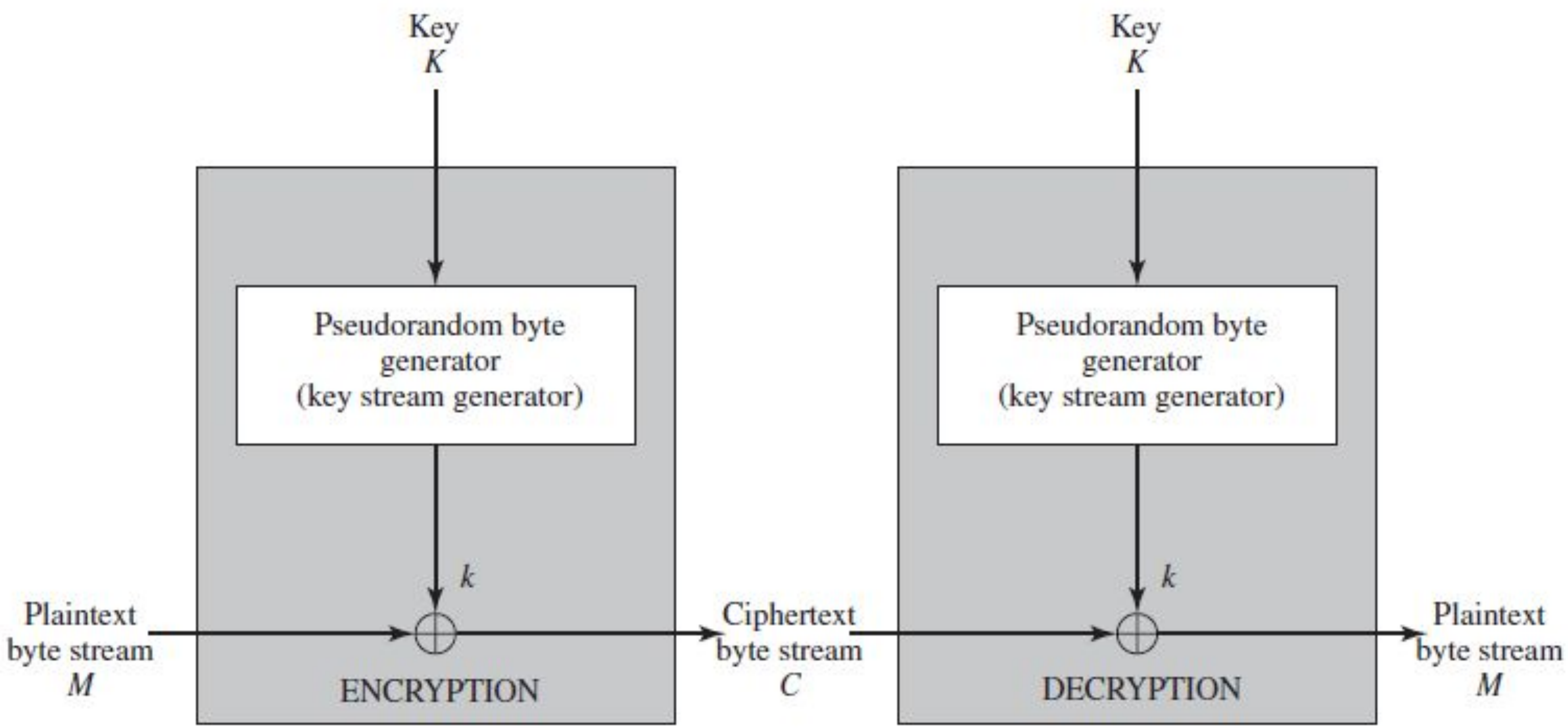


Figure 7.7 Stream Cipher Diagram

Important design considerations for a stream cipher

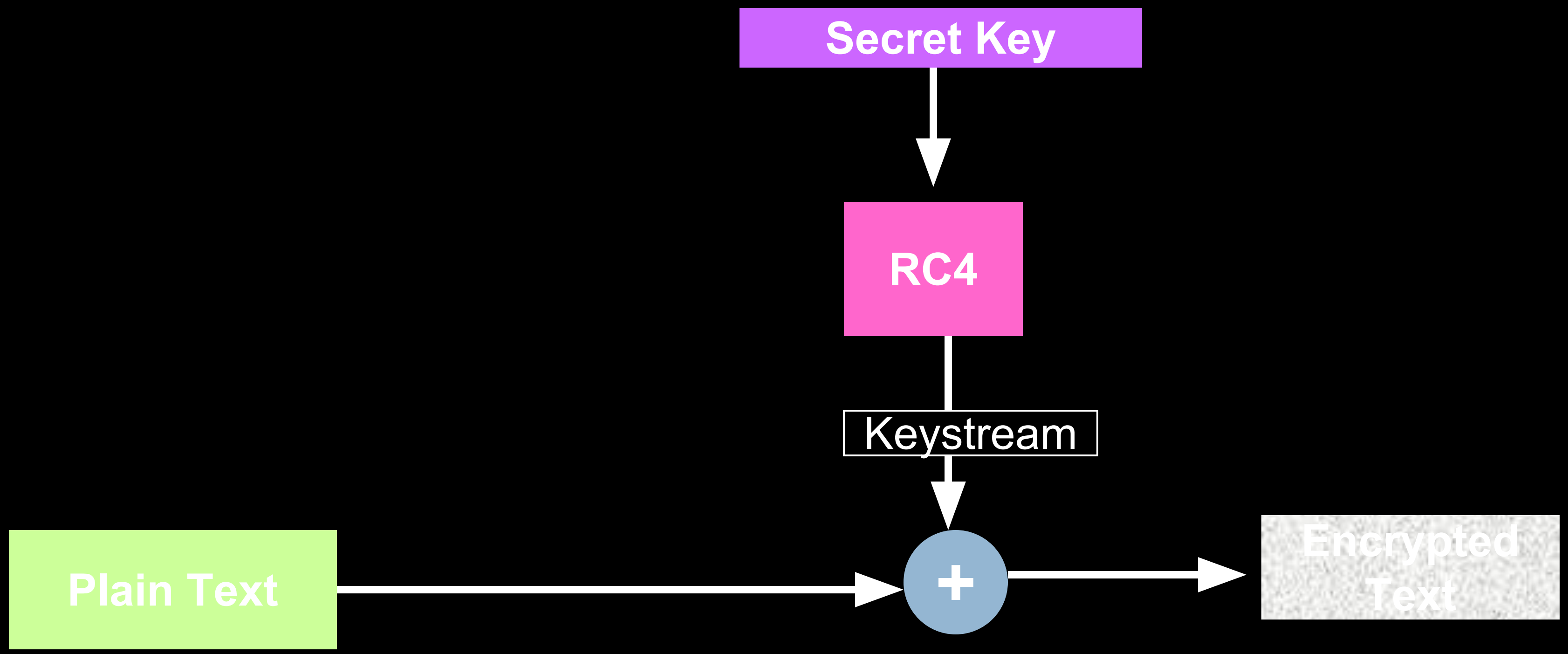
- The encryption sequence should have a large period.
- The keystream should approximate the properties of a true random number stream.
- The output of the pseudorandom number generator is conditioned on the value of the input key.
  - To guard against brute-force attacks, the key needs to be sufficiently long

RC4

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.
- Variable key length of from 1 byte to 256 bytes
- RC4 is used in the SSL/TLS standards, WEP and WPA protocols.

RC4 Block Diagram



Cryptographically very strong and easy to implement

Discuss the stream cipher RC4 in detail (4)

- **RC4** is a stream cipher and variable length key algorithm.
- This algorithm encrypts one byte at a time. A key input is pseudorandom bit generator that produces a stream 8-bit number that is unpredictable without knowledge of input key
- The output of the generator is called key-stream, is combined one byte at a time with the plaintext stream cipher using X-OR operation.

Example:

```
RC4 Encryption

10011000 ? 01010000 = 11001000


RC4 Decryption

11001000 ? 01010000 = 10011000
```

RC4 ...Inside

- Consists of 2 parts:
  - Key Scheduling Algorithm (KSA)
  - Pseudo-Random Generation Algorithm (PRGA)

- KSA
  - Generate State array
- PRGA on the KSA
  - Generate keystream
  - XOR keystream with the data to generated encrypted stream

RC4 Algorithm

1. Initialization of vectors S and T
2. Initial Permutation of S
3. Stream Generation

- **Key-GenerationAlgorithm** – A variable-length key from 1 to 256 byte is used to initialize a 256-byte state vector S, with elements S[0] to S[255].
- For encryption and decryption, a byte k is generated from S by selecting one of the 255 entries in a systematic fashion, then the entries in S are permuted again.

1. Initialization of S and T

1.1 S is set with values from 0 to 255

S[]=[0,1,………… 255]

1.2 The temp vector T is initialized with the key K. K is written repeatedly until it fills up, with 256 bytes

- **Key-Scheduling Algorithm:**
- **Initialization**: The entries of S are set equal to the values from 0 to 255 in ascending order, a temporary vectorT,iscreated. If the length of the key k is 256 bytes, then k is assigned to T. Otherwise, for a key with length(k-len) bytes, the first k-len elements of T as copied from K and then K is repeated as many times as necessary to fill T. The idea is illustrated as follow:

Initialization of vectors S and T

```
for i = 0 to 255 do
S[i] = i;
T[i] = K[i mod keylen];
```

[S], S is set equal to the values from 0 to 255
S[0]=0, S[1]=1,…, S[255]=255
[T], A temporary vector
[K], Array of bytes of secret key
|K| = Keylen, Length of (K)

2. Initial Permutation of S

2.1 Starting from S[0] till S[255] , each S[i] is swapped with another byte in S according to:

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

- Use T to produce initial permutation of S
- The only operation on S is a swap; S still contains number from 0 to 255

3. 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];
```

The PRGA

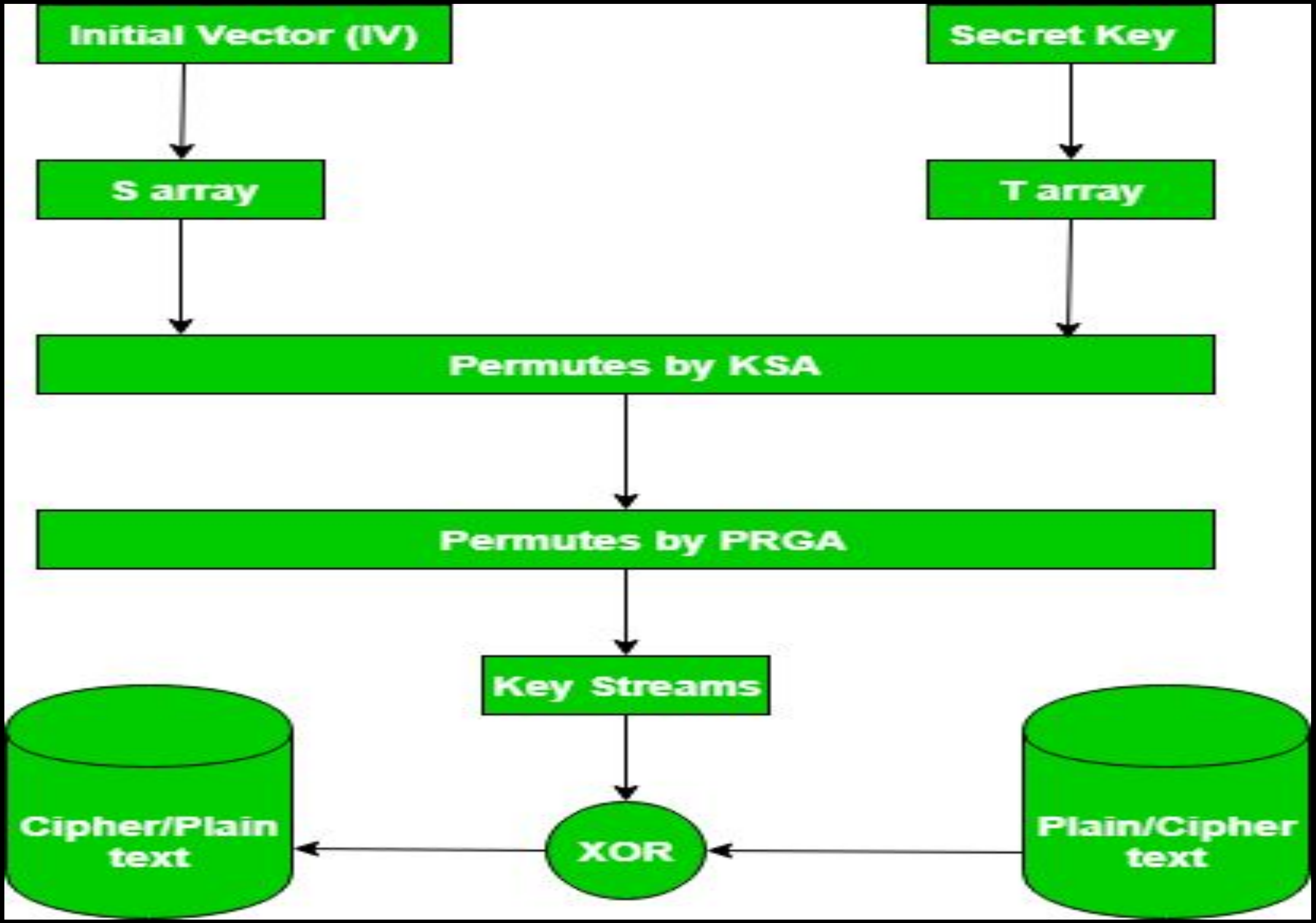
- Generate key stream *k* , one by one
- XOR S[k] with next byte of message to encrypt/decrypt

```
i = 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[i] + S[j]) (mod 256);
    C_i = M_i XOR S[k];
```

Sum of shuffled pair selects "stream key" value from permutation

RC4 Encryption and Decryption

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


Initial Vector (IV)
Secret Key
S array
T array
Permutes by KSA
Permutes by PRGA
Key Streams
Cipher/Plain text
XOR
Plain/Cipher text

- <u>Simplified RC4</u>
- Example Lets consider the stream cipher RC4, but instead of the full 256 bytes, we will use 8 x 3-bits. That is, the state vector S is 8 x 3-bits. We will operate on 3-bits of plaintext at a time since S can take the values 0 to 7, which can be represented as 3 bits.

- Assume we use a 4 x 3-bit key of K = [1 2 3 6]. And a plaintext P = [1 2 2 2]
- The first step is to generate the stream. Initialize the state vector S and temporary vector T. S is initialized so the S[i] = i, and T is initialized so it is the key K (repeated as necessary).

- S = [0 1 2 3 4 5 6 7] T = [1 2 3 6 1 2 3 6]

Now perform the initial permutation on S.

```
j = 0;
for i = 0 to 7 do
            j = (j + S[i] + T[i]) mod 8
            Swap(S[i],S[j]);
end
```

For i = 0:

$$j = (0 + 0 + 1) \bmod 8 = 1$$

Swap(S[0],S[1]);

S = [1 0 2 3 4 5 6 7]

For i = 1:
j = 3
Swap(S[1],S[3])
S = [1 3 2 0 4 5 6 7];

For i = 2:
j = 0
Swap(S[2],S[0]);
S = [2 3 1 0 4 5 6 7];

For i = 3:
j = 6;
Swap(S[3],S[6])
S = [2 3 1 6 4 5 0 7];

For i = 4:
j = 3
Swap(S[4],S[3])
S = [2 3 1 4 6 5 0 7];

For i = 5:
j = 2
Swap(S[5],S[2]);
S = [2 3 5 4 6 1 0 7];

For i = 6:
j = 5;
Swap(S[6],S[4])
S = [2 3 5 4 0 1 6 7];

For i = 7:
j = 2;
Swap(S[7],S[2])
S = [2 3 7 4 0 1 6 5];

Hence, our initial permutation of **S** = [2 3 7 4 0 1 6 5];

Now we generate 3-bits at a time, k, that we XOR with each 3-bits of plaintext to produce t
ciphertext. The 3-bits k is generated by:

```
i, j = 0;
while (true) {
      i = (i + 1) mod 8;
      j = (j + S[i]) mod 8;
      Swap (S[i], S[j]);
      t = (S[i] + S[j]) mod 8;
      k = S[t]; }
```

The first iteration:
S = [2 3 7 4 0 1 6 5]
i = (0 + 1) mod 8 = 1
j = (0 + S[1]) mod 8 = 3
Swap(S[1],S[3])
S = [2 4 7 3 0 1 6 5]
t = (S[1] + S[3]) mod 8 = 7
k = S[7] = 5

Remember, **P** = [1 2 2 2]

So our first 3-bits of ciphertext is obtained by: k XOR P
5 XOR 1 = 101 XOR 001 = 100 = 4