CHAPTER 5

Introduction to Modern Symmetric-Key Ciphers

(Solution to Odd-Numbered Problems)

Review Questions

1. The traditional symmetric-key ciphers are character-oriented ciphers. The modern symmetric-key ciphers are bit-oriented ciphers.

3. A transposition is definitely a permutation of bits. A substitution of bits can be thought of a permutation if we add decoding and encoding to the operation.

5. A P-box (permutation box) transposes bits. We have three types of P-boxes in modern block ciphers: straight P-boxes, expansion P-boxes, and compression P-boxes. A straight P-box is invertible; the other two are not.

7. A product cipher is a complex cipher combining substitution, permutation, and other components discussed in this chapter. We discussed two classes of product ciphers: Feistel and non-Feistel ciphers.

9. A Feistel block cipher uses both invertible and noninvertible components. A non-Feistel block cipher uses only invertible components.

11. In a synchronous stream cipher the key stream is independent of the plaintext or ciphertext. In a nonsynchronous stream cipher the key stream is somehow dependent on the plaintext or ciphertext.

Exercises

13. The order of the group is 10! = 3,626,800. The key size is $\lceil \log_2(10!) \rceil$ = 22 bits. Note that a key of 22 bits can select 2^{22} = 4,194,304 different permutations, but only 3,626,800 of them are used here.

15.

a. $\text{CircularLeftShift}_3$(**100**11011) $\rightarrow$ 11011**100**

b. $\text{CircularRightShift}_3$(11011**100**) $\rightarrow$ **100**11011

c. The original word in Part *a* and the result of Part *b* are the same, which shows that circular left shift and circular right shift operations are inverses of each other.

17. We show the complement of A with $\overline{A}$.

a. (01001101) ⊕ (01001101) = (00000000) → (A ⊕ A = All 0s)

b. (01001101) ⊕ (10110010) = (11111111) → (A ⊕ $\overline{A}$ = All 1s)

c. (01001101) ⊕ (00000000) = (01001101) → (A ⊕ All 0s = A)

d. (01001101) ⊕ (11111111) = (10110010) → (A ⊕ All 1s = $\overline{A}$)

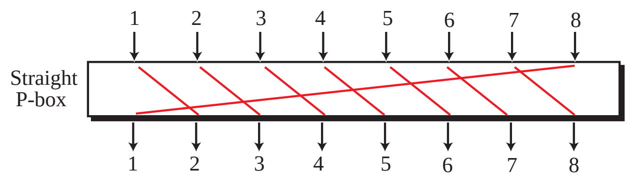
19. Using eight bits for each character, |M| = 8 × 2000 = 16,000 bits. Therefore, we have

|M| + |Pad| = 0 mod 64 → |Pad| = – |M| mod 64 → |Pad| = – 16,000 mod 64 = 0

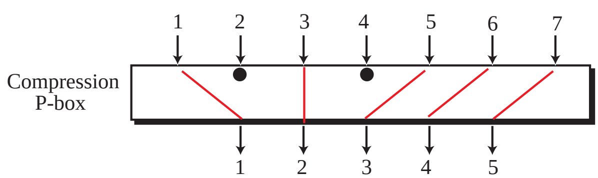
This means no padding is needed. The message is divided into 250 blocks.

21. The permutation table is [1 3 5].

23. See the figure below:



25. It is a compression P-box with 7 inputs and 5 outputs as shown below:



27. We use the following procedure:

a. We first find the input/output relation based on the given S-box:

Input:	00	01	10	11
Output:	01	11	10	00

b. We then find the inverse input/output relation (sorted on input):

Input:	00	01	10	11
Output:	11	00	10	01

c. Now we create the table for the inverse S-box (the left row defines the first input bit, the first column defines the second input bit, and the entries define the output):

	0	1
0	11	00
1	10	01

29. The characteristic polynomial is $x^4 + x^3 + x^2 + 1$ or $(11101)_2$ or $(1D)_{16}$. The polynomial is not primitive (see Appendix G). The maximum period is then less than $2^4 - 1$ or less than 15.

31. To have a maximum period of 32, the characteristic polynomial should be of degree 6 because $2^5 - 1 = 31 < 32$. However, if the characteristic polynomial is primitive, the maximum period is $2^6 - 1 = 63$. But the problem says that the maximum period is 32, therefore, the characteristic polynomial is not primitive. In other words, we have an LFSR of degree 6, with 6 cells (6-bit register) whose characteristic polynomial is not primitive.

33.

Input: **1**011	→	Left rotate	→	Output: 110
Input: **0**110	→	Right rotate	→	Output: 011

35.

```
combine (rightWord [1 … m], leftWord [1 … m], m)
{
        i ← 1
        while (i ≤ m)
        {
                word[i] ← rigthWord[i]
                word[i + m] ← leftWord[i]
                i ← i + 1
        }
        return (word[1 … n])                    // n = 2m
}
```

37.

```
circularShift (shift, word [1 … n], n, k)
{
        i ← 1
        while (i ≤ k)
        {
                if (shift = left)
                        word ← circularShiftLeft(word, n)
                else
                        word ← circularShiftRight(word, n)
                i ← i + 1
        }
        return (word[1 … n])
}
circularShiftLeft (word [1 … n], n)
{
        temp ← word[n]
        j ← n − 1
        while (j ≥ 0)
        {
                word[j +1] ← word[j]
                j ← j − 1
        }
        word[1] ← temp
        return (word[0 … n])
}
circularShiftRight (word [0 … n], n)
{
        temp ← word[1]
        j ← 1
        while (j ≤ n)
        {
                word [j −1] ← word[j]
                j ← j + 1
        }
        word[n] ← temp
        return (word[0 … n])
}
```

39. The table can be designed in many different ways. We assume, we have a linear table of n cells, in which each cell contains a value of m bits. The input defines the cell number, the contents of the cell defines the output. With this configuration, the routine looks like the one shown below:

```
S-box (inputBits [1 … n], Table [1 … n], n, m)
{
        index ← binaryToDecimal (inputBits)
        value ← Table [index]
        outputBits ← decimalToBinary (value)
        return (outputBits [0 … m])
}
```

41.

```
FeistelRound (inputBits [1 … n], roundKey[1 … n], n)
{
        (tempLeft, tempRight) ← split (word, n)
        tempLeft ← tempLeft ⊕ function (tempRight, roundKey)
        (tempLeft, tempRight) ← swap (tempLeft, tempRight)
        outputBits ← combine(tempLeft, tempRight)
        return (outputBits [1 … n])
}
```