

CHAPTER 2

INTRODUCTION TO NUMBER THEORY

2.1 Divisibility and The Division Algorithm

- Divisibility
- The Division Algorithm

2.2 The Euclidean Algorithm

- Greatest Common Divisor
- Finding the Greatest Common Divisor

2.3 Modular Arithmetic

- The Modulus
- Properties of Congruences
- Modular Arithmetic Operations
- Properties of Modular Arithmetic
- Euclidean Algorithm Revisited
- The Extended Euclidean Algorithm

2.4 Prime Numbers

2.5 Fermat's and Euler's Theorems

- Fermat's Theorem
- Euler's Totient Function
- Euler's Theorem

2.6 Testing for Primality

- Miller–Rabin Algorithm
- A Deterministic Primality Algorithm
- Distribution of Primes

2.7 The Chinese Remainder Theorem

2.8 Discrete Logarithms

- The Powers of an Integer, Modulo n
- Logarithms for Modular Arithmetic
- Calculation of Discrete Logarithms

2.9 Key Terms, Review Questions, and Problems

Appendix 2A The Meaning of Mod

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Understand the concept of divisibility and the division algorithm.
- ◆ Understand how to use the Euclidean algorithm to find the greatest common divisor.
- ◆ Present an overview of the concepts of modular arithmetic.
- ◆ Explain the operation of the extended Euclidean algorithm.
- ◆ Discuss key concepts relating to prime numbers.
- ◆ Understand Fermat's theorem.
- ◆ Understand Euler's theorem.
- ◆ Define Euler's totient function.
- ◆ Make a presentation on the topic of testing for primality.
- ◆ Explain the Chinese remainder theorem.
- ◆ Define discrete logarithms.

Number theory is pervasive in cryptographic algorithms. This chapter provides sufficient breadth and depth of coverage of relevant number theory topics for understanding the wide range of applications in cryptography. The reader familiar with these topics can safely skip this chapter.

The first three sections introduce basic concepts from number theory that are needed for understanding finite fields; these include divisibility, the Euclidean algorithm, and modular arithmetic. The reader may study these sections now or wait until ready to tackle Chapter 5 on finite fields.

Sections 2.4 through 2.8 discuss aspects of number theory related to prime numbers and discrete logarithms. These topics are fundamental to the design of asymmetric (public-key) cryptographic algorithms. The reader may study these sections now or wait until ready to read Part Three.

The concepts and techniques of number theory are quite abstract, and it is often difficult to grasp them intuitively without examples. Accordingly, this chapter includes a number of examples, each of which is highlighted in a shaded box.

2.1 DIVISIBILITY AND THE DIVISION ALGORITHM

Divisibility

We say that a nonzero b **divides** a if $a = mb$ for some m , where a , b , and m are integers. That is, b divides a if there is no remainder on division. The notation $b|a$ is commonly used to mean b divides a . Also, if $b|a$, we say that b is a **divisor** of a .

The positive divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

$13 \mid 182$; $-5 \mid 30$; $17 \mid 289$; $-3 \mid 33$; $17 \mid 0$

Subsequently, we will need some simple properties of divisibility for integers, which are as follows:

- If $a \mid 1$, then $a = \pm 1$.
- If $a \mid b$ and $b \mid a$, then $a = \pm b$.
- Any $b \neq 0$ divides 0.
- If $a \mid b$ and $b \mid c$, then $a \mid c$:

$$11 \mid 66 \text{ and } 66 \mid 198 \Rightarrow 11 \mid 198$$

- If $b \mid g$ and $b \mid h$, then $b \mid (mg + nh)$ for arbitrary integers m and n .

To see this last point, note that

- If $b \mid g$, then g is of the form $g = b \times g_1$ for some integer g_1 .
- If $b \mid h$, then h is of the form $h = b \times h_1$ for some integer h_1 .

So

$$mg + nh = mbg_1 + nbh_1 = b \times (mg_1 + nh_1)$$

and therefore b divides $mg + nh$.

$$b = 7; g = 14; h = 63; m = 3; n = 2$$

$$7 \mid 14 \text{ and } 7 \mid 63.$$

$$\text{To show } 7 \mid (3 \times 14 + 2 \times 63),$$

$$\text{we have } (3 \times 14 + 2 \times 63) = 7(3 \times 2 + 2 \times 9),$$

$$\text{and it is obvious that } 7 \mid (7(3 \times 2 + 2 \times 9)).$$

The Division Algorithm

Given any positive integer n and any nonnegative integer a , if we divide a by n , we get an integer quotient q and an integer remainder r that obey the following relationship:

$$a = qn + r \quad 0 \leq r < n; q = \lfloor a/n \rfloor \quad (2.1)$$

where $\lfloor x \rfloor$ is the largest integer less than or equal to x . Equation (2.1) is referred to as the division algorithm.¹

¹Equation (2.1) expresses a theorem rather than an algorithm, but by tradition, this is referred to as the division algorithm.

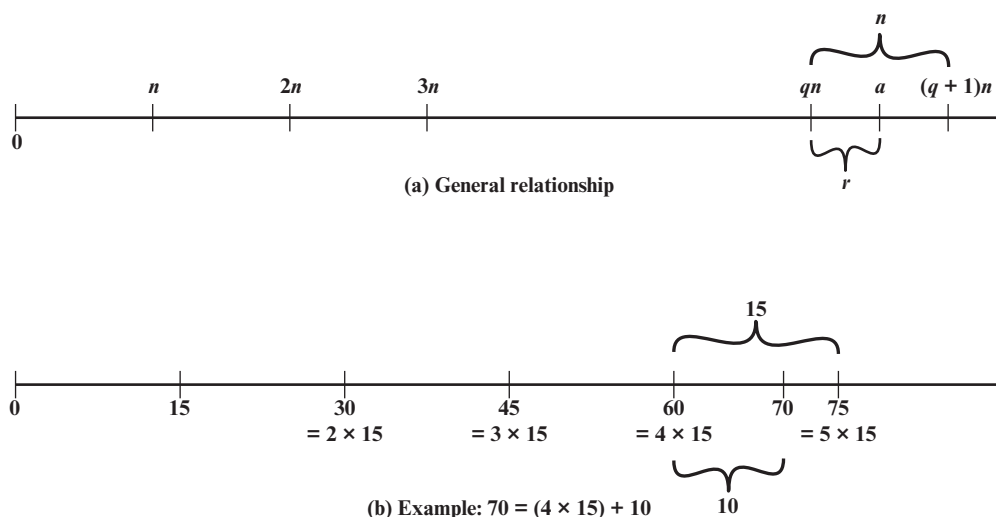


Figure 2.1 The Relationship $a = qn + r$; $0 \leq r < n$

Figure 2.1a demonstrates that, given a and positive n , it is always possible to find q and r that satisfy the preceding relationship. Represent the integers on the number line; a will fall somewhere on that line (positive a is shown, a similar demonstration can be made for negative a). Starting at 0, proceed to n , $2n$, up to qn , such that $qn \leq a$ and $(q + 1)n > a$. The distance from qn to a is r , and we have found the unique values of q and r . The remainder r is often referred to as a **residue**.

$$\begin{array}{llll} a = 11; & n = 7; & 11 = 1 \times 7 + 4; & r = 4 \quad q = 1 \\ a = -11; & n = 7; & -11 = (-2) \times 7 + 3; & r = 3 \quad q = -2 \end{array}$$

Figure 2.1b provides another example.

2.2 THE EUCLIDEAN ALGORITHM

One of the basic techniques of number theory is the Euclidean algorithm, which is a simple procedure for determining the greatest common divisor of two positive integers. First, we need a simple definition: Two integers are **relatively prime** if and only if their only common positive integer factor is 1.

Greatest Common Divisor

Recall that nonzero b is defined to be a divisor of a if $a = mb$ for some m , where a , b , and m are integers. We will use the notation $\gcd(a, b)$ to mean the **greatest common divisor** of a and b . The greatest common divisor of a and b is the largest integer that divides both a and b . We also define $\gcd(0, 0) = 0$.

More formally, the positive integer c is said to be the greatest common divisor of a and b if

1. c is a divisor of a and of b .
2. any divisor of a and b is a divisor of c .

An equivalent definition is the following:

$$\gcd(a, b) = \max[k, \text{such that } k|a \text{ and } k|b]$$

Because we require that the greatest common divisor be positive, $\gcd(a, b) = \gcd(a, -b) = \gcd(-a, b) = \gcd(-a, -b)$. In general, $\gcd(a, b) = \gcd(|a|, |b|)$.

$$\gcd(60, 24) = \gcd(60, -24) = 12$$

Also, because all nonzero integers divide 0, we have $\gcd(a, 0) = |a|$.

We stated that two integers a and b are relatively prime if and only if their only common positive integer factor is 1. This is equivalent to saying that a and b are relatively prime if $\gcd(a, b) = 1$.

8 and 15 are relatively prime because the positive divisors of 8 are 1, 2, 4, and 8, and the positive divisors of 15 are 1, 3, 5, and 15. So 1 is the only integer on both lists.

Finding the Greatest Common Divisor

We now describe an algorithm credited to Euclid for easily finding the greatest common divisor of two integers (Figure 2.2). This algorithm has broad significance in cryptography. The explanation of the algorithm can be broken down into the following points:

1. Suppose we wish to determine the greatest common divisor d of the integers a and b ; that is determine $d = \gcd(a, b)$. Because $\gcd(|a|, |b|) = \gcd(a, b)$, there is no harm in assuming $a \geq b > 0$.
2. Dividing a by b and applying the division algorithm, we can state:

$$a = q_1b + r_1 \quad 0 \leq r_1 < b \quad (2.2)$$

3. First consider the case in which $r_1 = 0$. Therefore b divides a and clearly no larger number divides both b and a , because that number would be larger than b . So we have $d = \gcd(a, b) = b$.
4. The other possibility from Equation (2.2) is $r_1 \neq 0$. For this case, we can state that $d|r_1$. This is due to the basic properties of divisibility: the relations $d|a$ and $d|b$ together imply that $d|(a - q_1b)$, which is the same as $d|r_1$.
5. Before proceeding with the Euclidian algorithm, we need to answer the question: What is the $\gcd(b, r_1)$? We know that $d|b$ and $d|r_1$. Now take any arbitrary integer c that divides both b and r_1 . Therefore, $c|(q_1b + r_1) = a$. Because c divides both a and b , we must have $c \leq d$, which is the greatest common divisor of a and b . Therefore $d \leq \gcd(b, r_1)$.

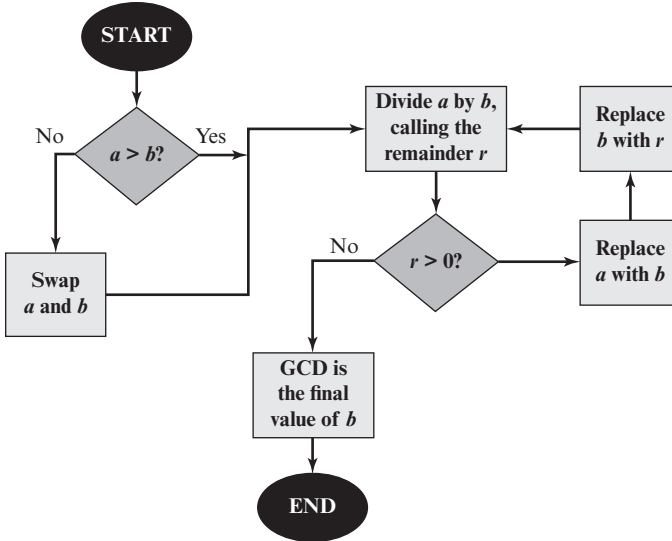


Figure 2.2 Euclidean Algorithm

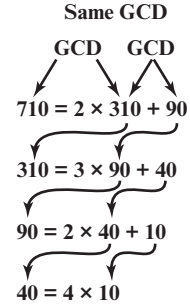


Figure 2.3 Euclidean Algorithm Example: gcd(710, 310)

Let us now return to Equation (2.2) and assume that $r_1 \neq 0$. Because $b > r_1$, we can divide b by r_1 and apply the division algorithm to obtain:

$$b = q_2 r_1 + r_2 \quad 0 \leq r_2 < r_1$$

As before, if $r_2 = 0$, then $d = r_1$ and if $r_2 \neq 0$, then $d = \gcd(r_1, r_2)$. Note that the remainders form a descending series of nonnegative values and so must terminate when the remainder is zero. This happens, say, at the $(n + 1)$ th stage where r_{n-1} is divided by r_n . The result is the following system of equations:

$$\left. \begin{array}{ll} a = q_1 b + r_1 & 0 < r_1 < b \\ b = q_2 r_1 + r_2 & 0 < r_2 < r_1 \\ r_1 = q_3 r_2 + r_3 & 0 < r_3 < r_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ r_{n-2} = q_n r_{n-1} + r_n & 0 < r_n < r_{n-1} \\ r_{n-1} = q_{n+1} r_n + 0 & \\ d = \gcd(a, b) = r_n & \end{array} \right\} \quad (2.3)$$

At each iteration, we have $d = \gcd(r_i, r_{i+1})$ until finally $d = \gcd(r_n, 0) = r_n$. Thus, we can find the greatest common divisor of two integers by repetitive application of the division algorithm. This scheme is known as the Euclidean algorithm. Figure 2.3 illustrates a simple example.

We have essentially argued from the top down that the final result is the $\gcd(a, b)$. We can also argue from the bottom up. The first step is to show that r_n divides a and b . It follows from the last division in Equation (2.3) that r_n divides r_{n-1} . The next to last division shows that r_n divides r_{n-2} because it divides both

terms on the right. Successively, one sees that r_n divides all r_i 's and finally a and b . It remains to show that r_n is the largest divisor that divides a and b . If we take any arbitrary integer that divides a and b , it must also divide r_1 , as explained previously. We can follow the sequence of equations in Equation (2.3) down and show that c must divide all r_i 's. Therefore c must divide r_n , so that $r_n = \gcd(a, b)$.

Let us now look at an example with relatively large numbers to see the power of this algorithm:

To find $d = \gcd(a, b) = \gcd(1160718174, 316258250)$		
$a = q_1b + r_1$	$1160718174 = 3 \times 316258250 + 211943424$	$d = \gcd(316258250, 211943424)$
$b = q_2r_1 + r_2$	$316258250 = 1 \times 211943424 + 104314826$	$d = \gcd(211943424, 104314826)$
$r_1 = q_3r_2 + r_3$	$211943424 = 2 \times 104314826 + 3313772$	$d = \gcd(104314826, 3313772)$
$r_2 = q_4r_3 + r_4$	$104314826 = 31 \times 3313772 + 1587894$	$d = \gcd(3313772, 1587894)$
$r_3 = q_5r_4 + r_5$	$3313772 = 2 \times 1587894 + 137984$	$d = \gcd(1587894, 137984)$
$r_4 = q_6r_5 + r_6$	$1587894 = 11 \times 137984 + 70070$	$d = \gcd(137984, 70070)$
$r_5 = q_7r_6 + r_7$	$137984 = 1 \times 70070 + 67914$	$d = \gcd(70070, 67914)$
$r_6 = q_8r_7 + r_8$	$70070 = 1 \times 67914 + 2156$	$d = \gcd(67914, 2156)$
$r_7 = q_9r_8 + r_9$	$67914 = 31 \times 2156 + 1078$	$d = \gcd(2156, 1078)$
$r_8 = q_{10}r_9 + r_{10}$	$2156 = 2 \times 1078 + 0$	$d = \gcd(1078, 0) = 1078$
Therefore, $d = \gcd(1160718174, 316258250) = 1078$		

In this example, we begin by dividing 1160718174 by 316258250, which gives 3 with a remainder of 211943424. Next we take 316258250 and divide it by 211943424. The process continues until we get a remainder of 0, yielding a result of 1078.

It will be helpful in what follows to recast the above computation in tabular form. For every step of the iteration, we have $r_{i-2} = q_i r_{i-1} + r_i$, where r_{i-2} is the dividend, r_{i-1} is the divisor, q_i is the quotient, and r_i is the remainder. Table 2.1 summarizes the results.

Table 2.1 Euclidean Algorithm Example

Dividend	Divisor	Quotient	Remainder
$a = 1160718174$	$b = 316258250$	$q_1 = 3$	$r_1 = 211943424$
$b = 316258250$	$r_1 = 211943424$	$q_2 = 1$	$r_2 = 104314826$
$r_1 = 211943424$	$r_2 = 104314826$	$q_3 = 2$	$r_3 = 3313772$
$r_2 = 104314826$	$r_3 = 3313772$	$q_4 = 31$	$r_4 = 1587894$
$r_3 = 3313772$	$r_4 = 1587894$	$q_5 = 2$	$r_5 = 137984$
$r_4 = 1587894$	$r_5 = 137984$	$q_6 = 11$	$r_6 = 70070$
$r_5 = 137984$	$r_6 = 70070$	$q_7 = 1$	$r_7 = 67914$
$r_6 = 70070$	$r_7 = 67914$	$q_8 = 1$	$r_8 = 2156$
$r_7 = 67914$	$r_8 = 2156$	$q_9 = 31$	$r_9 = 1078$
$r_8 = 2156$	$r_9 = 1078$	$q_{10} = 2$	$r_{10} = 0$

2.3 MODULAR ARITHMETIC

The Modulus

If a is an integer and n is a positive integer, we define $a \bmod n$ to be the remainder when a is divided by n . The integer n is called the **modulus**. Thus, for any integer a , we can rewrite Equation (2.1) as follows:

$$\begin{aligned} a &= qn + r & 0 \leq r < n; q &= \lfloor a/n \rfloor \\ a &= \lfloor a/n \rfloor \times n + (a \bmod n) \end{aligned}$$

$$11 \bmod 7 = 4; \quad -11 \bmod 7 = 3$$

Two integers a and b are said to be **congruent modulo n** , if $(a \bmod n) = (b \bmod n)$. This is written as $a \equiv b \pmod{n}$.²

$$73 \equiv 4 \pmod{23}; \quad 21 \equiv -9 \pmod{10}$$

Note that if $a \equiv 0 \pmod{n}$, then $n|a$.

Properties of Congruences

Congruences have the following properties:

1. $a \equiv b \pmod{n}$ if $n|(a - b)$.
2. $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$.
3. $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ imply $a \equiv c \pmod{n}$.

To demonstrate the first point, if $n|(a - b)$, then $(a - b) = kn$ for some k . So we can write $a = b + kn$. Therefore, $(a \bmod n) = (\text{remainder when } b + kn \text{ is divided by } n) = (\text{remainder when } b \text{ is divided by } n) = (b \bmod n)$.

$$\begin{array}{lll} 23 \equiv 8 \pmod{5} & \text{because} & 23 - 8 = 15 = 5 \times 3 \\ -11 \equiv 5 \pmod{8} & \text{because} & -11 - 5 = -16 = 8 \times (-2) \\ 81 \equiv 0 \pmod{27} & \text{because} & 81 - 0 = 81 = 27 \times 3 \end{array}$$

The remaining points are as easily proved.

²We have just used the operator *mod* in two different ways: first as a **binary operator** that produces a remainder, as in the expression $a \bmod b$; second as a **congruence relation** that shows the equivalence of two integers, as in the expression $a \equiv b \pmod{n}$. See Appendix 2A for a discussion.

Modular Arithmetic Operations

Note that, by definition (Figure 2.1), the $(\text{mod } n)$ operator maps all integers into the set of integers $\{0, 1, \dots, (n - 1)\}$. This suggests the question: Can we perform arithmetic operations within the confines of this set? It turns out that we can; this technique is known as **modular arithmetic**.

Modular arithmetic exhibits the following properties:

1. $[(a \text{ mod } n) + (b \text{ mod } n)] \text{ mod } n = (a + b) \text{ mod } n$
2. $[(a \text{ mod } n) - (b \text{ mod } n)] \text{ mod } n = (a - b) \text{ mod } n$
3. $[(a \text{ mod } n) \times (b \text{ mod } n)] \text{ mod } n = (a \times b) \text{ mod } n$

We demonstrate the first property. Define $(a \text{ mod } n) = r_a$ and $(b \text{ mod } n) = r_b$. Then we can write $a = r_a + jn$ for some integer j and $b = r_b + kn$ for some integer k . Then

$$\begin{aligned}(a + b) \text{ mod } n &= (r_a + jn + r_b + kn) \text{ mod } n \\ &= (r_a + r_b + (k + j)n) \text{ mod } n \\ &= (r_a + r_b) \text{ mod } n \\ &= [(a \text{ mod } n) + (b \text{ mod } n)] \text{ mod } n\end{aligned}$$

The remaining properties are proven as easily. Here are examples of the three properties:

$$\begin{aligned}11 \text{ mod } 8 &= 3; 15 \text{ mod } 8 = 7 \\ [(11 \text{ mod } 8) + (15 \text{ mod } 8)] \text{ mod } 8 &= 10 \text{ mod } 8 = 2 \\ (11 + 15) \text{ mod } 8 &= 26 \text{ mod } 8 = 2 \\ [(11 \text{ mod } 8) - (15 \text{ mod } 8)] \text{ mod } 8 &= -4 \text{ mod } 8 = 4 \\ (11 - 15) \text{ mod } 8 &= -4 \text{ mod } 8 = 4 \\ [(11 \text{ mod } 8) \times (15 \text{ mod } 8)] \text{ mod } 8 &= 21 \text{ mod } 8 = 5 \\ (11 \times 15) \text{ mod } 8 &= 165 \text{ mod } 8 = 5\end{aligned}$$

Exponentiation is performed by repeated multiplication, as in ordinary arithmetic.

To find $11^7 \text{ mod } 13$, we can proceed as follows:

$$\begin{aligned}11^2 &= 121 \equiv 4 \pmod{13} \\ 11^4 &= (11^2)^2 \equiv 4^2 \equiv 3 \pmod{13} \\ 11^7 &= 11 \times 11^2 \times 11^4 \\ 11^7 &\equiv 11 \times 4 \times 3 \equiv 132 \equiv 2 \pmod{13}\end{aligned}$$

Thus, the rules for ordinary arithmetic involving addition, subtraction, and multiplication carry over into modular arithmetic.

Table 2.2 provides an illustration of modular addition and multiplication modulo 8. Looking at addition, the results are straightforward, and there is a regular pattern to the matrix. Both matrices are symmetric about the main diagonal in conformance to the commutative property of addition and multiplication. As in ordinary addition, there is an additive inverse, or negative, to each integer in modular arithmetic. In this case, the negative of an integer x is the integer y such that $(x + y) \bmod 8 = 0$. To find the additive inverse of an integer in the left-hand column, scan across the corresponding row of the matrix to find the value 0; the integer at the top of that column is the additive inverse; thus, $(2 + 6) \bmod 8 = 0$. Similarly, the entries in the multiplication table are straightforward. In modular arithmetic mod 8, the multiplicative inverse of x is the integer y such that $(x \times y) \bmod 8 = 1 \bmod 8$. Now, to find the multiplicative inverse of an integer from the multiplication table, scan across the matrix in the row for that integer to find the value 1; the integer at the top of that column is the multiplicative inverse; thus, $(3 \times 3) \bmod 8 = 1$. Note that not all integers mod 8 have a multiplicative inverse; more about that later.

Properties of Modular Arithmetic

Define the set Z_n as the set of nonnegative integers less than n :

$$Z_n = \{0, 1, \dots, (n - 1)\}$$

Table 2.2 Arithmetic Modulo 8

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

(a) Addition modulo 8

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

(b) Multiplication modulo 8

w	$-w$	w^{-1}
0	0	—
1	7	1
2	6	—
3	5	3
4	4	—
5	3	5
6	2	—
7	1	7

(c) Additive and multiplicative inverse modulo 8

This is referred to as the **set of residues**, or **residue classes** $(\text{mod } n)$. To be more precise, each integer in Z_n represents a residue class. We can label the residue classes $(\text{mod } n)$ as $[0], [1], [2], \dots, [n-1]$, where

$$[r] = \{a: a \text{ is an integer, } a \equiv r \pmod{n}\}$$

The residue classes $(\text{mod } 4)$ are

$$[0] = \{ \dots, -16, -12, -8, -4, 0, 4, 8, 12, 16, \dots \}$$

$$[1] = \{ \dots, -15, -11, -7, -3, 1, 5, 9, 13, 17, \dots \}$$

$$[2] = \{ \dots, -14, -10, -6, -2, 2, 6, 10, 14, 18, \dots \}$$

$$[3] = \{ \dots, -13, -9, -5, -1, 3, 7, 11, 15, 19, \dots \}$$

Of all the integers in a residue class, the smallest nonnegative integer is the one used to represent the residue class. Finding the smallest nonnegative integer to which k is congruent modulo n is called **reducing k modulo n** .

If we perform modular arithmetic within Z_n , the properties shown in Table 2.3 hold for integers in Z_n . We show in the next section that this implies that Z_n is a commutative ring with a multiplicative identity element.

There is one peculiarity of modular arithmetic that sets it apart from ordinary arithmetic. First, observe that (as in ordinary arithmetic) we can write the following:

$$\text{if } (a + b) \equiv (a + c) \pmod{n} \text{ then } b \equiv c \pmod{n} \quad (2.4)$$

$$(5 + 23) \equiv (5 + 7) \pmod{8}; 23 \equiv 7 \pmod{8}$$

Equation (2.4) is consistent with the existence of an additive inverse. Adding the additive inverse of a to both sides of Equation (2.4), we have

$$\begin{aligned} ((-a) + a + b) &\equiv ((-a) + a + c) \pmod{n} \\ b &\equiv c \pmod{n} \end{aligned}$$

Table 2.3 Properties of Modular Arithmetic for Integers in Z_n

Property	Expression
Commutative Laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative Laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive Law	$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 \times w) \bmod n = w \bmod n$
Additive Inverse $(-w)$	For each $w \in Z_n$, there exists a z such that $w + z \equiv 0 \pmod{n}$

However, the following statement is true only with the attached condition:

if $(a \times b) \equiv (a \times c) \pmod{n}$ then $b \equiv c \pmod{n}$ if a is relatively prime to n (2.5)

Recall that two integers are **relatively prime** if their only common positive integer factor is 1. Similar to the case of Equation (2.4), we can say that Equation (2.5) is consistent with the existence of a multiplicative inverse. Applying the multiplicative inverse of a to both sides of Equation (2.5), we have

$$\begin{aligned} ((a^{-1})ab) &\equiv ((a^{-1})ac) \pmod{n} \\ b &\equiv c \pmod{n} \end{aligned}$$

To see this, consider an example in which the condition of Equation (2.5) does not hold. The integers 6 and 8 are not relatively prime, since they have the common factor 2. We have the following:

$$\begin{aligned} 6 \times 3 &= 18 \equiv 2 \pmod{8} \\ 6 \times 7 &= 42 \equiv 2 \pmod{8} \end{aligned}$$

Yet $3 \not\equiv 7 \pmod{8}$.

The reason for this strange result is that for any general modulus n , a multiplier a that is applied in turn to the integers 0 through $(n - 1)$ will fail to produce a complete set of residues if a and n have any factors in common.

With $a = 6$ and $n = 8$,

Z_8	0	1	2	3	4	5	6	7
Multiply by 6	0	6	12	18	24	30	36	42
Residues	0	6	4	2	0	6	4	2

Because we do not have a complete set of residues when multiplying by 6, more than one integer in Z_8 maps into the same residue. Specifically, $6 \times 0 \pmod{8} = 6 \times 4 \pmod{8}$; $6 \times 1 \pmod{8} = 6 \times 5 \pmod{8}$; and so on. Because this is a many-to-one mapping, there is not a unique inverse to the multiply operation.

However, if we take $a = 5$ and $n = 8$, whose only common factor is 1,

Z_8	0	1	2	3	4	5	6	7
Multiply by 5	0	5	10	15	20	25	30	35
Residues	0	5	2	7	4	1	6	3

The line of residues contains all the integers in Z_8 , in a different order.

In general, an integer has a multiplicative inverse in Z_n if and only if that integer is relatively prime to n . Table 2.2c shows that the integers 1, 3, 5, and 7 have a multiplicative inverse in Z_8 ; but 2, 4, and 6 do not.

Euclidean Algorithm Revisited

The Euclidean algorithm can be based on the following theorem: For any integers a, b , with $a \geq b \geq 0$,

$$\gcd(a, b) = \gcd(b, a \bmod b) \quad (2.6)$$

$$\gcd(55, 22) = \gcd(22, 55 \bmod 22) = \gcd(22, 11) = 11$$

To see that Equation (2.6) works, let $d = \gcd(a, b)$. Then, by the definition of \gcd , $d|a$ and $d|b$. For any positive integer b , we can express a as

$$\begin{aligned} a &= kb + r \equiv r \pmod{b} \\ a \bmod b &= r \end{aligned}$$

with k, r integers. Therefore, $(a \bmod b) = a - kb$ for some integer k . But because $d|b$, it also divides kb . We also have $d|a$. Therefore, $d|(a \bmod b)$. This shows that d is a common divisor of b and $(a \bmod b)$. Conversely, if d is a common divisor of b and $(a \bmod b)$, then $d|kb$ and thus $d|[kb + (a \bmod b)]$, which is equivalent to $d|a$. Thus, the set of common divisors of a and b is equal to the set of common divisors of b and $(a \bmod b)$. Therefore, the \gcd of one pair is the same as the \gcd of the other pair, proving the theorem.

Equation (2.6) can be used repetitively to determine the greatest common divisor.

$$\begin{aligned} \gcd(18, 12) &= \gcd(12, 6) = \gcd(6, 0) = 6 \\ \gcd(11, 10) &= \gcd(10, 1) = \gcd(1, 0) = 1 \end{aligned}$$

This is the same scheme shown in Equation (2.3), which can be rewritten in the following way.

Euclidean Algorithm	
Calculate	Which satisfies
$r_1 = a \bmod b$	$a = q_1b + r_1$
$r_2 = b \bmod r_1$	$b = q_2r_1 + r_2$
$r_3 = r_1 \bmod r_2$	$r_1 = q_3r_2 + r_3$
\vdots	\vdots
$r_n = r_{n-2} \bmod r_{n-1}$	$r_{n-2} = q_nr_{n-1} + r_n$
$r_{n+1} = r_{n-1} \bmod r_n = 0$	$r_{n-1} = q_{n+1}r_n + 0$ $d = \gcd(a, b) = r_n$

We can define the Euclidean algorithm concisely as the following recursive function.

```

Euclid(a,b)
  if (b=0) then return a;
  else return Euclid(b, a mod b);

```

The Extended Euclidean Algorithm

We now proceed to look at an extension to the Euclidean algorithm that will be important for later computations in the area of finite fields and in encryption algorithms, such as RSA. For given integers a and b , the extended Euclidean algorithm not only calculates the greatest common divisor d but also two additional integers x and y that satisfy the following equation.

$$ax + by = d = \gcd(a, b) \quad (2.7)$$

It should be clear that x and y will have opposite signs. Before examining the algorithm, let us look at some of the values of x and y when $a = 42$ and $b = 30$. Note that $\gcd(42, 30) = 6$. Here is a partial table of values³ for $42x + 30y$.

x	-3	-2	-1	0	1	2	3
y							
-3	-216	-174	-132	-90	-48	-6	36
-2	-186	-144	-102	-60	-18	24	66
-1	-156	-114	-72	-30	12	54	96
0	-126	-84	-42	0	42	84	126
1	-96	-54	-12	30	72	114	156
2	-66	-24	18	60	102	144	186
3	-36	6	48	90	132	174	216

Observe that all of the entries are divisible by 6. This is not surprising, because both 42 and 30 are divisible by 6, so every number of the form $42x + 30y = 6(7x + 5y)$ is a multiple of 6. Note also that $\gcd(42, 30) = 6$ appears in the table. In general, it can be shown that for given integers a and b , the smallest positive value of $ax + by$ is equal to $\gcd(a, b)$.

Now let us show how to extend the Euclidean algorithm to determine (x, y, d) given a and b . We again go through the sequence of divisions indicated in Equation (2.3), and we assume that at each step i we can find integers x_i and y_i that satisfy $r_i = ax_i + by_i$. We end up with the following sequence.

$$\begin{array}{ll}
 a = q_1b + r_1 & r_1 = ax_1 + by_1 \\
 b = q_2r_1 + r_2 & r_2 = ax_2 + by_2 \\
 r_1 = q_3r_2 + r_3 & r_3 = ax_3 + by_3 \\
 \vdots & \vdots \\
 r_{n-2} = q_nr_{n-1} + r_n & r_n = ax_n + by_n \\
 r_{n-1} = q_{n+1}r_n + 0 &
 \end{array}$$

³This example is taken from [SILV06].

Now, observe that we can rearrange terms to write

$$r_i = r_{i-2} - r_{i-1}q_i \quad (2.8)$$

Also, in rows $i - 1$ and $i - 2$, we find the values

$$r_{i-2} = ax_{i-2} + by_{i-2} \quad \text{and} \quad r_{i-1} = ax_{i-1} + by_{i-1}$$

Substituting into Equation (2.8), we have

$$\begin{aligned} r_i &= (ax_{i-2} + by_{i-2}) - (ax_{i-1} + by_{i-1})q_i \\ &= a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1}) \end{aligned}$$

But we have already assumed that $r_i = ax_i + by_i$. Therefore,

$$x_i = x_{i-2} - q_i x_{i-1} \quad \text{and} \quad y_i = y_{i-2} - q_i y_{i-1}$$

We now summarize the calculations:

Extended Euclidean Algorithm			
Calculate	Which satisfies	Calculate	Which satisfies
$r_{-1} = a$		$x_{-1} = 1; y_{-1} = 0$	$a = ax_{-1} + by_{-1}$
$r_0 = b$		$x_0 = 0; y_0 = 1$	$b = ax_0 + by_0$
$r_1 = a \bmod b$ $q_1 = \lfloor a/b \rfloor$	$a = q_1 b + r_1$	$x_1 = x_{-1} - q_1 x_0 = 1$ $y_1 = y_{-1} - q_1 y_0 = -q_1$	$r_1 = ax_1 + by_1$
$r_2 = b \bmod r_1$ $q_2 = \lfloor b/r_1 \rfloor$	$b = q_2 r_1 + r_2$	$x_2 = x_0 - q_2 x_1$ $y_2 = y_0 - q_2 y_1$	$r_2 = ax_2 + by_2$
$r_3 = r_1 \bmod r_2$ $q_3 = \lfloor r_1/r_2 \rfloor$	$r_1 = q_3 r_2 + r_3$	$x_3 = x_1 - q_3 x_2$ $y_3 = y_1 - q_3 y_2$	$r_3 = ax_3 + by_3$
\vdots	\vdots	\vdots	\vdots
$r_n = r_{n-2} \bmod r_{n-1}$ $q_n = \lfloor r_{n-2}/r_{n-1} \rfloor$	$r_{n-2} = q_n r_{n-1} + r_n$	$x_n = x_{n-2} - q_n x_{n-1}$ $y_n = y_{n-2} - q_n y_{n-1}$	$r_n = ax_n + by_n$
$r_{n+1} = r_{n-1} \bmod r_n = 0$ $q_{n+1} = \lfloor r_{n-1}/r_n \rfloor$	$r_{n-1} = q_{n+1} r_n + 0$		$d = \gcd(a, b) = r_n$ $x = x_n; y = y_n$

We need to make several additional comments here. In each row, we calculate a new remainder r_i based on the remainders of the previous two rows, namely r_{i-1} and r_{i-2} . To start the algorithm, we need values for r_0 and r_{-1} , which are just a and b . It is then straightforward to determine the required values for x_{-1}, y_{-1}, x_0 , and y_0 .

We know from the original Euclidean algorithm that the process ends with a remainder of zero and that the greatest common divisor of a and b is $d = \gcd(a, b) = r_n$. But we also have determined that $d = r_n = ax_n + by_n$. Therefore, in Equation (2.7), $x = x_n$ and $y = y_n$.

As an example, let us use $a = 1759$ and $b = 550$ and solve for $1759x + 550y = \gcd(1759, 550)$. The results are shown in Table 2.4. Thus, we have $1759 \times (-111) + 550 \times 355 = -195249 + 195250 = 1$.

Table 2.4 Extended Euclidean Algorithm Example

i	r_i	q_i	x_i	y_i
-1	1759		1	0
0	550		0	1
1	109	3	1	-3
2	5	5	-5	16
3	4	21	106	-339
4	1	1	-111	355
5	0	4		

Result: $d = 1$; $x = -111$; $y = 355$

2.4 PRIME NUMBERS⁴

A central concern of number theory is the study of prime numbers. Indeed, whole books have been written on the subject (e.g., [CRAN01], [RIBE96]). In this section, we provide an overview relevant to the concerns of this book.

An integer $p > 1$ is a prime number if and only if its only divisors⁵ are ± 1 and $\pm p$. **Prime numbers** play a critical role in number theory and in the techniques discussed in this chapter. Table 2.5 shows the primes less than 2000. Note the way the primes are distributed. In particular, note the number of primes in each range of 100 numbers.

Any integer $a > 1$ can be factored in a unique way as

$$a = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_t^{a_t} \quad (2.9)$$

where $p_1 < p_2 < \cdots < p_t$ are prime numbers and where each a_i is a positive integer. This is known as the fundamental theorem of arithmetic; a proof can be found in any text on number theory.

$$\begin{aligned} 91 &= 7 \times 13 \\ 3600 &= 2^4 \times 3^2 \times 5^2 \\ 11011 &= 7 \times 11^2 \times 13 \end{aligned}$$

It is useful for what follows to express this another way. If P is the set of all prime numbers, then any positive integer a can be written uniquely in the following form:

$$a = \prod_{p \in P} p^{a_p} \quad \text{where each } a_p \geq 0$$

⁴In this section, unless otherwise noted, we deal only with the nonnegative integers. The use of negative integers would introduce no essential differences.

⁵Recall from Section 2.1 that integer a is said to be a divisor of integer b if there is no remainder on division. Equivalently, we say that a divides b .

Table 2.5 Primes Under 2000

2	101	211	307	401	503	601	701	809	907	1009	1103	1201	1301	1409	1511	1601	1709	1801	1901
3	103	223	311	409	509	607	709	811	911	1013	1109	1213	1303	1423	1523	1607	1721	1811	1907
5	107	227	313	419	521	613	719	821	919	1019	1117	1217	1307	1427	1531	1609	1723	1823	1913
7	109	229	317	421	523	617	727	823	929	1021	1123	1223	1319	1429	1543	1613	1733	1831	1931
11	113	233	331	431	541	619	733	827	937	1031	1129	1229	1321	1433	1549	1619	1741	1847	1933
13	127	239	337	433	547	631	739	829	941	1033	1151	1231	1327	1439	1553	1621	1747	1861	1949
17	131	241	347	439	557	641	743	839	947	1039	1153	1237	1361	1447	1559	1627	1753	1867	1951
19	137	251	349	443	563	643	751	853	953	1049	1163	1249	1367	1451	1567	1637	1759	1871	1973
23	139	257	353	449	569	647	757	857	967	1051	1171	1259	1373	1453	1571	1657	1777	1873	1979
29	149	263	359	457	571	653	761	859	971	1061	1181	1277	1381	1459	1579	1663	1783	1877	1987
31	151	269	367	461	577	659	769	863	977	1063	1187	1279	1399	1471	1583	1667	1787	1879	1993
37	157	271	373	463	587	661	773	877	983	1069	1193	1283		1481	1597	1669	1789	1889	1997
41	163	277	379	467	593	673	787	881	991	1087		1289		1483		1693			1999
43	167	281	383	479	599	677	797	883	997	1091		1291		1487		1697			
47	173	283	389	487		683		887		1093		1297		1489		1699			
53	179	293	397	491		691				1097				1493					
59	181			499										1499					
61	191																		
67	193																		
71	197																		
73	199																		
79																			
83																			
89																			
97																			

The right-hand side is the product over all possible prime numbers p ; for any particular value of a , most of the exponents a_p will be 0.

The value of any given positive integer can be specified by simply listing all the nonzero exponents in the foregoing formulation.

The integer 12 is represented by $\{a_2 = 2, a_3 = 1\}$.
 The integer 18 is represented by $\{a_2 = 1, a_3 = 2\}$.
 The integer 91 is represented by $\{a_7 = 1, a_{13} = 1\}$.

Multiplication of two numbers is equivalent to adding the corresponding exponents. Given $a = \prod_{p \in \mathbf{P}} p^{a_p}$, $b = \prod_{p \in \mathbf{P}} p^{b_p}$. Define $k = ab$. We know that the integer k can be expressed as the product of powers of primes: $k = \prod_{p \in \mathbf{P}} p^{k_p}$. It follows that $k_p = a_p + b_p$ for all $p \in \mathbf{P}$.

$$\begin{aligned} k &= 12 \times 18 = (2^2 \times 3) \times (2 \times 3^2) = 216 \\ k_2 &= 2 + 1 = 3; k_3 = 1 + 2 = 3 \\ 216 &= 2^3 \times 3^3 = 8 \times 27 \end{aligned}$$

What does it mean, in terms of the prime factors of a and b , to say that a divides b ? Any integer of the form p^n can be divided only by an integer that is of a lesser or equal power of the same prime number, p^j with $j \leq n$. Thus, we can say the following.

Given

$$a = \prod_{p \in \mathbf{P}} p^{a_p}, b = \prod_{p \in \mathbf{P}} p^{b_p}$$

If $a|b$, then $a_p \leq b_p$ for all p .

$$\begin{aligned} a &= 12; b = 36; 12|36 \\ 12 &= 2^2 \times 3; 36 = 2^2 \times 3^2 \\ a_2 &= 2 = b_2 \\ a_3 &= 1 \leq 2 = b_3 \\ \text{Thus, the inequality } a_p &\leq b_p \text{ is satisfied for all prime numbers.} \end{aligned}$$

It is easy to determine the greatest common divisor of two positive integers if we express each integer as the product of primes.

$$\begin{aligned}
300 &= 2^2 \times 3^1 \times 5^2 \\
18 &= 2^1 \times 3^2 \\
\gcd(18, 300) &= 2^1 \times 3^1 \times 5^0 = 6
\end{aligned}$$

The following relationship always holds:

$$\text{If } k = \gcd(a, b), \text{ then } k_p = \min(a_p, b_p) \text{ for all } p.$$

Determining the prime factors of a large number is no easy task, so the preceding relationship does not directly lead to a practical method of calculating the greatest common divisor.

2.5 FERMAT'S AND EULER'S THEOREMS

Two theorems that play important roles in public-key cryptography are Fermat's theorem and Euler's theorem.

Fermat's Theorem⁶

Fermat's theorem states the following: If p is prime and a is a positive integer not divisible by p , then

$$a^{p-1} \equiv 1 \pmod{p} \quad (2.10)$$

Proof: Consider the set of positive integers less than p : $\{1, 2, \dots, p-1\}$ and multiply each element by a , modulo p , to get the set $X = \{a \bmod p, 2a \bmod p, \dots, (p-1)a \bmod p\}$. None of the elements of X is equal to zero because p does not divide a . Furthermore, no two of the integers in X are equal. To see this, assume that $ja \equiv ka \pmod{p}$, where $1 \leq j < k \leq p-1$. Because a is relatively prime⁷ to p , we can eliminate a from both sides of the equation [see Equation (2.3)] resulting in $j \equiv k \pmod{p}$. This last equality is impossible, because j and k are both positive integers less than p . Therefore, we know that the $(p-1)$ elements of X are all positive integers with no two elements equal. We can conclude the X consists of the set of integers $\{1, 2, \dots, p-1\}$ in some order. Multiplying the numbers in both sets (p and X) and taking the result mod p yields

$$\begin{aligned}
a \times 2a \times \dots \times (p-1)a &\equiv [(1 \times 2 \times \dots \times (p-1)) \pmod{p}] \\
a^{p-1}(p-1)! &\equiv (p-1)! \pmod{p}
\end{aligned}$$

We can cancel the $(p-1)!$ term because it is relatively prime to p [see Equation (2.5)]. This yields Equation (2.10), which completes the proof.

⁶This is sometimes referred to as Fermat's little theorem.

⁷Recall from Section 2.2 that two numbers are relatively prime if they have no prime factors in common; that is, their only common divisor is 1. This is equivalent to saying that two numbers are relatively prime if their greatest common divisor is 1.

$$\begin{aligned}
a &= 7, p = 19 \\
7^2 &= 49 \equiv 11 \pmod{19} \\
7^4 &\equiv 121 \equiv 7 \pmod{19} \\
7^8 &\equiv 49 \equiv 11 \pmod{19} \\
7^{16} &\equiv 121 \equiv 7 \pmod{19} \\
a^{p-1} &= 7^{18} = 7^{16} \times 7^2 \equiv 7 \times 11 \equiv 1 \pmod{19}
\end{aligned}$$

An alternative form of Fermat's theorem is also useful: If p is prime and a is a positive integer, then

$$a^p \equiv a \pmod{p} \quad (2.11)$$

Note that the first form of the theorem [Equation (2.10)] requires that a be relatively prime to p , but this form does not.

$$\begin{aligned}
p = 5, a = 3 & \quad a^p = 3^5 = 243 \equiv 3 \pmod{5} = a \pmod{p} \\
p = 5, a = 10 & \quad a^p = 10^5 = 100000 \equiv 10 \pmod{5} \equiv 0 \pmod{5} = a \pmod{p}
\end{aligned}$$

Euler's Totient Function

Before presenting Euler's theorem, we need to introduce an important quantity in number theory, referred to as **Euler's totient function**. This function, written $\phi(n)$, is defined as the number of positive integers less than n and relatively prime to n . By convention, $\phi(1) = 1$.

Determine $\phi(37)$ and $\phi(35)$.

Because 37 is prime, all of the positive integers from 1 through 36 are relatively prime to 37. Thus $\phi(37) = 36$.

To determine $\phi(35)$, we list all of the positive integers less than 35 that are relatively prime to it:

$$\begin{aligned}
&1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18 \\
&19, 22, 23, 24, 26, 27, 29, 31, 32, 33, 34
\end{aligned}$$

There are 24 numbers on the list, so $\phi(35) = 24$.

Table 2.6 lists the first 30 values of $\phi(n)$. The value $\phi(1)$ is without meaning but is defined to have the value 1.

It should be clear that, for a prime number p ,

$$\phi(p) = p - 1$$

Now suppose that we have two prime numbers p and q with $p \neq q$. Then we can show that, for $n = pq$,

Table 2.6 Some Values of Euler's Totient Function $\phi(n)$

n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$
1	1	11	10	21	12
2	1	12	4	22	10
3	2	13	12	23	22
4	2	14	6	24	8
5	4	15	8	25	20
6	2	16	8	26	12
7	6	17	16	27	18
8	4	18	6	28	12
9	6	19	18	29	28
10	4	20	8	30	8

$$\phi(n) = \phi(pq) = \phi(p) \times \phi(q) = (p - 1) \times (q - 1)$$

To see that $\phi(n) = \phi(p) \times \phi(q)$, consider that the set of positive integers less than n is the set $\{1, \dots, (pq - 1)\}$. The integers in this set that are not relatively prime to n are the set $\{p, 2p, \dots, (q - 1)p\}$ and the set $\{q, 2q, \dots, (p - 1)q\}$. To see this, consider that any integer that divides n must divide either of the prime numbers p or q . Therefore, any integer that does not contain either p or q as a factor is relatively prime to n . Further note that the two sets just listed are non-overlapping: Because p and q are prime, we can state that none of the integers in the first set can be written as a multiple of q , and none of the integers in the second set can be written as a multiple of p . Thus the total number of unique integers in the two sets is $(q - 1) + (p - 1)$. Accordingly,

$$\begin{aligned}
 \phi(n) &= (pq - 1) - [(q - 1) + (p - 1)] \\
 &= pq - (p + q) + 1 \\
 &= (p - 1) \times (q - 1) \\
 &= \phi(p) \times \phi(q)
 \end{aligned}$$

$$\begin{aligned}
 \phi(21) &= \phi(3) \times \phi(7) = (3 - 1) \times (7 - 1) = 2 \times 6 = 12 \\
 &\text{where the 12 integers are } \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}.
 \end{aligned}$$

Euler's Theorem

Euler's theorem states that for every a and n that are relatively prime:

$$a^{\phi(n)} \equiv 1 \pmod{n} \quad (2.12)$$

Proof: Equation (2.12) is true if n is prime, because in that case, $\phi(n) = (n - 1)$ and Fermat's theorem holds. However, it also holds for any integer n . Recall that

$\phi(n)$ is the number of positive integers less than n that are relatively prime to n . Consider the set of such integers, labeled as

$$R = \{x_1, x_2, \dots, x_{\phi(n)}\}$$

That is, each element x_i of R is a unique positive integer less than n with $\gcd(x_i, n) = 1$. Now multiply each element by a , modulo n :

$$S = \{(ax_1 \bmod n), (ax_2 \bmod n), \dots, (ax_{\phi(n)} \bmod n)\}$$

The set S is a permutation⁸ of R , by the following line of reasoning:

1. Because a is relatively prime to n and x_i is relatively prime to n , ax_i must also be relatively prime to n . Thus, all the members of S are integers that are less than n and that are relatively prime to n .
2. There are no duplicates in S . Refer to Equation (2.5). If $ax_i \bmod n = ax_j \bmod n$, then $x_i = x_j$.

Therefore,

$$\begin{aligned} \prod_{i=1}^{\phi(n)} (ax_i \bmod n) &= \prod_{i=1}^{\phi(n)} x_i \\ \prod_{i=1}^{\phi(n)} ax_i &\equiv \prod_{i=1}^{\phi(n)} x_i \pmod{n} \\ a^{\phi(n)} \times \left[\prod_{i=1}^{\phi(n)} x_i \right] &\equiv \prod_{i=1}^{\phi(n)} x_i \pmod{n} \\ a^{\phi(n)} &\equiv 1 \pmod{n} \end{aligned}$$

which completes the proof. This is the same line of reasoning applied to the proof of Fermat's theorem.

$a = 3; n = 10; \phi(10) = 4; \quad a^{\phi(n)} = 3^4 = 81 = 1 \pmod{10} = 1 \pmod{n}$ $a = 2; n = 11; \phi(11) = 10; \quad a^{\phi(n)} = 2^{10} = 1024 = 1 \pmod{11} = 1 \pmod{n}$
--

As is the case for Fermat's theorem, an alternative form of the theorem is also useful:

$$a^{\phi(n)+1} \equiv a \pmod{n} \tag{2.13}$$

Again, similar to the case with Fermat's theorem, the first form of Euler's theorem [Equation (2.12)] requires that a be relatively prime to n , but this form does not.

⁸A permutation of a finite set of elements S is an ordered sequence of all the elements of S , with each element appearing exactly once.

2.6 TESTING FOR PRIMALITY

For many cryptographic algorithms, it is necessary to select one or more very large prime numbers at random. Thus, we are faced with the task of determining whether a given large number is prime. There is no simple yet efficient means of accomplishing this task.

In this section, we present one attractive and popular algorithm. You may be surprised to learn that this algorithm yields a number that is not necessarily a prime. However, the algorithm can yield a number that is almost certainly a prime. This will be explained presently. We also make reference to a deterministic algorithm for finding primes. The section closes with a discussion concerning the distribution of primes.

Miller–Rabin Algorithm⁹

The algorithm due to Miller and Rabin [MILL75, RABI80] is typically used to test a large number for primality. Before explaining the algorithm, we need some background. First, any positive odd integer $n \geq 3$ can be expressed as

$$n - 1 = 2^k q \quad \text{with } k > 0, q \text{ odd}$$

To see this, note that $n - 1$ is an even integer. Then, divide $(n - 1)$ by 2 until the result is an odd number q , for a total of k divisions. If n is expressed as a binary number, then the result is achieved by shifting the number to the right until the rightmost digit is a 1, for a total of k shifts. We now develop two properties of prime numbers that we will need.

TWO PROPERTIES OF PRIME NUMBERS The **first property** is stated as follows: If p is prime and a is a positive integer less than p , then $a^2 \bmod p = 1$ if and only if either $a \bmod p = 1$ or $a \bmod p = -1 \bmod p = p - 1$. By the rules of modular arithmetic $(a \bmod p)(a \bmod p) = a^2 \bmod p$. Thus, if either $a \bmod p = 1$ or $a \bmod p = -1$, then $a^2 \bmod p = 1$. Conversely, if $a^2 \bmod p = 1$, then $(a \bmod p)^2 = 1$, which is true only for $a \bmod p = 1$ or $a \bmod p = -1$.

The **second property** is stated as follows: Let p be a prime number greater than 2. We can then write $p - 1 = 2^k q$ with $k > 0$, q odd. Let a be any integer in the range $1 < a < p - 1$. Then one of the two following conditions is true.

1. a^q is congruent to 1 modulo p . That is, $a^q \bmod p = 1$, or equivalently, $a^q \equiv 1 \pmod{p}$.
2. One of the numbers $a^q, a^{2q}, a^{4q}, \dots, a^{2^{k-1}q}$ is congruent to -1 modulo p . That is, there is some number j in the range $(1 \leq j \leq k)$ such that $a^{2^{j-1}q} \bmod p = -1 \bmod p = p - 1$ or equivalently, $a^{2^{j-1}q} \equiv -1 \pmod{p}$.

Proof: Fermat's theorem [Equation (2.10)] states that $a^{n-1} \equiv 1 \pmod{n}$ if n is prime. We have $p - 1 = 2^k q$. Thus, we know that $a^{p-1} \bmod p = a^{2^k q} \bmod p = 1$. Thus, if we look at the sequence of numbers

$$a^q \bmod p, a^{2q} \bmod p, a^{4q} \bmod p, \dots, a^{2^{k-1}q} \bmod p, a^{2^k q} \bmod p \quad (2.14)$$

⁹Also referred to in the literature as the Rabin–Miller algorithm, or the Rabin–Miller test, or the Miller–Rabin test.

we know that the last number in the list has value 1. Further, each number in the list is the square of the previous number. Therefore, one of the following possibilities must be true.

1. The first number on the list, and therefore all subsequent numbers on the list, equals 1.
2. Some number on the list does not equal 1, but its square mod p does equal 1. By virtue of the first property of prime numbers defined above, we know that the only number that satisfies this condition is $p - 1$. So, in this case, the list contains an element equal to $p - 1$.

This completes the proof.

DETAILS OF THE ALGORITHM These considerations lead to the conclusion that, if n is prime, then either the first element in the list of residues, or remainders, $(a^q, a^{2q}, \dots, a^{2^{k-1}q}, a^{2^kq})$ modulo n equals 1; or some element in the list equals $(n - 1)$; otherwise n is composite (i.e., not a prime). On the other hand, if the condition is met, that does not necessarily mean that n is prime. For example, if $n = 2047 = 23 \times 89$, then $n - 1 = 2 \times 1023$. We compute $2^{1023} \bmod 2047 = 1$, so that 2047 meets the condition but is not prime.

We can use the preceding property to devise a test for primality. The procedure **TEST** takes a candidate integer n as input and returns the result *composite* if n is definitely not a prime, and the result *inconclusive* if n may or may not be a prime.

TEST (n)

1. Find integers k, q , with $k > 0, q$ odd, so that $(n - 1 = 2^k q)$;
2. Select a random integer $a, 1 < a < n - 1$;
3. **if** $a^q \bmod n = 1$ **then** return("inconclusive");
4. **for** $j = 0$ **to** $k - 1$ **do**
5. **if** $a^{2^j q} \bmod n = n - 1$ **then** return("inconclusive");
6. return("composite");

Let us apply the test to the prime number $n = 29$. We have $(n - 1) = 28 = 2^2(7) = 2^k q$. First, let us try $a = 10$. We compute $10^7 \bmod 29 = 17$, which is neither 1 nor 28, so we continue the test. The next calculation finds that $(10^7)^2 \bmod 29 = 28$, and the test returns *inconclusive* (i.e., 29 may be prime). Let's try again with $a = 2$. We have the following calculations: $2^7 \bmod 29 = 12$; $2^{14} \bmod 29 = 28$; and the test again returns *inconclusive*. If we perform the test for all integers a in the range 1 through 28, we get the same *inconclusive* result, which is compatible with n being a prime number.

Now let us apply the test to the composite number $n = 13 \times 17 = 221$. Then $(n - 1) = 220 = 2^2(55) = 2^k q$. Let us try $a = 5$. Then we have $5^{55} \bmod 221 = 112$, which is neither 1 nor $220(5^{55})^2 \bmod 221 = 168$. Because we have used all values of j (i.e., $j = 0$ and $j = 1$) in line 4 of the **TEST** algorithm, the test returns *composite*, indicating that 221 is definitely a composite number. But suppose we had selected $a = 21$. Then we have $21^{55} \bmod 221 = 200$; $(21^{55})^2 \bmod 221 = 220$; and the test returns *inconclusive*, indicating that 221 may be prime. In fact, of the 218 integers from 2 through 219, four of these will return an *inconclusive* result, namely 21, 47, 174, and 200.

REPEATED USE OF THE MILLER–RABIN ALGORITHM How can we use the Miller–Rabin algorithm to determine with a high degree of confidence whether or not an integer is prime? It can be shown [KNUT98] that given an odd number n that is not prime and a randomly chosen integer, a with $1 < a < n - 1$, the probability that TEST will return *inconclusive* (i.e., fail to detect that n is not prime) is less than $1/4$. Thus, if t different values of a are chosen, the probability that all of them will pass TEST (return *inconclusive*) for n is less than $(1/4)^t$. For example, for $t = 10$, the probability that a nonprime number will pass all ten tests is less than 10^{-6} . Thus, for a sufficiently large value of t , we can be confident that n is prime if Miller’s test always returns *inconclusive*.

This gives us a basis for determining whether an odd integer n is prime with a reasonable degree of confidence. The procedure is as follows: Repeatedly invoke TEST (n) using randomly chosen values for a . If, at any point, TEST returns *composite*, then n is determined to be nonprime. If TEST continues to return *inconclusive* for t tests, then for a sufficiently large value of t , assume that n is prime.

A Deterministic Primality Algorithm

Prior to 2002, there was no known method of efficiently proving the primality of very large numbers. All of the algorithms in use, including the most popular (Miller–Rabin), produced a probabilistic result. In 2002 (announced in 2002, published in 2004), Agrawal, Kayal, and Saxena [AGRA04] developed a relatively simple deterministic algorithm that efficiently determines whether a given large number is a prime. The algorithm, known as the AKS algorithm, does not appear to be as efficient as the Miller–Rabin algorithm. Thus far, it has not supplanted this older, probabilistic technique.

Distribution of Primes

It is worth noting how many numbers are likely to be rejected before a prime number is found using the Miller–Rabin test, or any other test for primality. A result from number theory, known as the prime number theorem, states that the primes near n are spaced on the average one every $\ln(n)$ integers. Thus, on average, one would have to test on the order of $\ln(n)$ integers before a prime is found. Because all even integers can be immediately rejected, the correct figure is $0.5 \ln(n)$. For example, if a prime on the order of magnitude of 2^{200} were sought, then about $0.5 \ln(2^{200}) = 69$ trials would be needed to find a prime. However, this figure is just an average. In some places along the number line, primes are closely packed, and in other places there are large gaps.

The two consecutive odd integers 1,000,000,000,061 and 1,000,000,000,063 are both prime. On the other hand, $1001! + 2, 1001! + 3, \dots, 1001! + 1000, 1001! + 1001$ is a sequence of 1000 consecutive composite integers.

2.7 THE CHINESE REMAINDER THEOREM

One of the most useful results of number theory is the **Chinese remainder theorem** (CRT).¹⁰ In essence, the CRT says it is possible to reconstruct integers in a certain range from their residues modulo a set of pairwise relatively prime moduli.

The 10 integers in Z_{10} , that is the integers 0 through 9, can be reconstructed from their two residues modulo 2 and 5 (the relatively prime factors of 10). Say the known residues of a decimal digit x are $r_2 = 0$ and $r_5 = 3$; that is, $x \bmod 2 = 0$ and $x \bmod 5 = 3$. Therefore, x is an even integer in Z_{10} whose remainder, on division by 5, is 3. The unique solution is $x = 8$.

The CRT can be stated in several ways. We present here a formulation that is most useful from the point of view of this text. An alternative formulation is explored in Problem 2.33. Let

$$M = \prod_{i=1}^k m_i$$

where the m_i are pairwise relatively prime; that is, $\gcd(m_i, m_j) = 1$ for $1 \leq i, j \leq k$, and $i \neq j$. We can represent any integer A in Z_M by a k -tuple whose elements are in Z_{m_i} using the following correspondence:

$$A \leftrightarrow (a_1, a_2, \dots, a_k) \quad (2.15)$$

where $A \in Z_M$, $a_i \in Z_{m_i}$, and $a_i = A \bmod m_i$ for $1 \leq i \leq k$. The CRT makes two assertions.

1. The mapping of Equation (2.15) is a one-to-one correspondence (called a **bijection**) between Z_M and the Cartesian product $Z_{m_1} \times Z_{m_2} \times \dots \times Z_{m_k}$. That is, for every integer A such that $0 \leq A < M$, there is a unique k -tuple (a_1, a_2, \dots, a_k) with $0 \leq a_i < m_i$ that represents it, and for every such k -tuple (a_1, a_2, \dots, a_k) , there is a unique integer A in Z_M .
2. Operations performed on the elements of Z_M can be equivalently performed on the corresponding k -tuples by performing the operation independently in each coordinate position in the appropriate system.

Let us demonstrate the **first assertion**. The transformation from A to (a_1, a_2, \dots, a_k) , is obviously unique; that is, each a_i is uniquely calculated as $a_i = A \bmod m_i$. Computing A from (a_1, a_2, \dots, a_k) can be done as follows. Let

¹⁰The CRT is so called because it is believed to have been discovered by the Chinese mathematician Sun-Tsu in around 100 A.D.

$M_i = M/m_i$ for $1 \leq i \leq k$. Note that $M_i = m_1 \times m_2 \times \dots \times m_{i-1} \times m_{i+1} \times \dots \times m_k$, so that $M_i \equiv 0 \pmod{m_j}$ for all $j \neq i$. Then let

$$c_i = M_i \times (M_i^{-1} \pmod{m_i}) \quad \text{for } 1 \leq i \leq k \quad (2.16)$$

By the definition of M_i , it is relatively prime to m_i and therefore has a unique multiplicative inverse mod m_i . So Equation (2.16) is well defined and produces a unique value c_i . We can now compute

$$A \equiv \left(\sum_{i=1}^k a_i c_i \right) \pmod{M} \quad (2.17)$$

To show that the value of A produced by Equation (2.17) is correct, we must show that $a_i = A \pmod{m_i}$ for $1 \leq i \leq k$. Note that $c_j \equiv M_j \equiv 0 \pmod{m_i}$ if $j \neq i$, and that $c_i \equiv 1 \pmod{m_i}$. It follows that $a_i = A \pmod{m_i}$.

The **second assertion** of the CRT, concerning arithmetic operations, follows from the rules for modular arithmetic. That is, the second assertion can be stated as follows: If

$$\begin{aligned} A &\leftrightarrow (a_1, a_2, \dots, a_k) \\ B &\leftrightarrow (b_1, b_2, \dots, b_k) \end{aligned}$$

then

$$\begin{aligned} (A + B) \pmod{M} &\leftrightarrow ((a_1 + b_1) \pmod{m_1}, \dots, (a_k + b_k) \pmod{m_k}) \\ (A - B) \pmod{M} &\leftrightarrow ((a_1 - b_1) \pmod{m_1}, \dots, (a_k - b_k) \pmod{m_k}) \\ (A \times B) \pmod{M} &\leftrightarrow ((a_1 \times b_1) \pmod{m_1}, \dots, (a_k \times b_k) \pmod{m_k}) \end{aligned}$$

One of the useful features of the Chinese remainder theorem is that it provides a way to manipulate (potentially very large) numbers mod M in terms of tuples of smaller numbers. This can be useful when M is 150 digits or more. However, note that it is necessary to know beforehand the factorization of M .

To represent $973 \pmod{1813}$ as a pair of numbers mod 37 and 49, define

$$\begin{aligned} m_1 &= 37 \\ m_2 &= 49 \\ M &= 1813 \\ A &= 973 \end{aligned}$$

We also have $M_1 = 49$ and $M_2 = 37$. Using the extended Euclidean algorithm, we compute $M_1^{-1} = 34 \pmod{m_1}$ and $M_2^{-1} = 4 \pmod{m_2}$. (Note that we only need to compute each M_i and each M_i^{-1} once.) Taking residues modulo 37 and 49, our representation of 973 is $(11, 42)$, because $973 \pmod{37} = 11$ and $973 \pmod{49} = 42$.

Now suppose we want to add 678 to 973. What do we do to $(11, 42)$? First we compute $(678) \leftrightarrow (678 \pmod{37}, 678 \pmod{49}) = (12, 41)$. Then we add the tuples element-wise and reduce $(11 + 12 \pmod{37}, 42 + 41 \pmod{49}) = (23, 34)$. To verify that this has the correct effect, we compute

$$\begin{aligned}
(23, 34) &\leftrightarrow a_1 M_1 M_1^{-1} + a_2 M_2 M_2^{-1} \bmod M \\
&= [(23)(49)(34) + (34)(37)(4)] \bmod 1813 \\
&= 43350 \bmod 1813 \\
&= 1651
\end{aligned}$$

and check that it is equal to $(973 + 678) \bmod 1813 = 1651$. Remember that in the above derivation, M_i^{-1} is the multiplicative inverse of M_i modulo m_i and M_2^{-1} is the multiplicative inverse of M_2 modulo m_2 .

Suppose we want to multiply $1651 \pmod{1813}$ by 73. We multiply $(23, 34)$ by 73 and reduce to get $(23 \times 73 \bmod 37, 34 \times 73 \bmod 49) = (14, 32)$. It is easily verified that

$$\begin{aligned}
(14, 32) &\leftrightarrow [(14)(49)(34) + (32)(37)(4)] \bmod 1813 \\
&= 865 \\
&= 1651 \times 73 \bmod 1813
\end{aligned}$$

2.8 DISCRETE LOGARITHMS

Discrete logarithms are fundamental to a number of public-key algorithms, including Diffie–Hellman key exchange and the digital signature algorithm (DSA). This section provides a brief overview of discrete logarithms. For the interested reader, more detailed developments of this topic can be found in [ORE67] and [LEVE90].

The Powers of an Integer, Modulo n

Recall from Euler's theorem [Equation (2.12)] that, for every a and n that are relatively prime,

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where $\phi(n)$, Euler's totient function, is the number of positive integers less than n and relatively prime to n . Now consider the more general expression:

$$a^m \equiv 1 \pmod{n} \tag{2.18}$$

If a and n are relatively prime, then there is at least one integer m that satisfies Equation (2.18), namely, $m = \phi(n)$. The least positive exponent m for which Equation (2.18) holds is referred to in several ways:

- The order of $a \pmod{n}$
- The exponent to which a belongs \pmod{n}
- The length of the period generated by a

To see this last point, consider the powers of 7, modulo 19:

$$\begin{aligned}
 7^1 &\equiv 7 \pmod{19} \\
 7^2 &= 49 = 2 \times 19 + 11 \equiv 11 \pmod{19} \\
 7^3 &= 343 = 18 \times 19 + 1 \equiv 1 \pmod{19} \\
 7^4 &= 2401 = 126 \times 19 + 7 \equiv 7 \pmod{19} \\
 7^5 &= 16807 = 884 \times 19 + 11 \equiv 11 \pmod{19}
 \end{aligned}$$

There is no point in continuing because the sequence is repeating. This can be proven by noting that $7^3 \equiv 1 \pmod{19}$, and therefore, $7^{3+j} \equiv 7^3 7^j \equiv 7^j \pmod{19}$, and hence, any two powers of 7 whose exponents differ by 3 (or a multiple of 3) are congruent to each other (mod 19). In other words, the sequence is periodic, and the length of the period is the smallest positive exponent m such that $7^m \equiv 1 \pmod{19}$.

Table 2.7 shows all the powers of a , modulo 19 for all positive $a < 19$. The length of the sequence for each base value is indicated by shading. Note the following:

1. All sequences end in 1. This is consistent with the reasoning of the preceding few paragraphs.
2. The length of a sequence divides $\phi(19) = 18$. That is, an integral number of sequences occur in each row of the table.
3. Some of the sequences are of length 18. In this case, it is said that the base integer a generates (via powers) the set of nonzero integers modulo 19. Each such integer is called a primitive root of the modulus 19.

More generally, we can say that the highest possible exponent to which a number can belong (mod n) is $\phi(n)$. If a number is of this order, it is referred to as a **primitive root** of n . The importance of this notion is that if a is a primitive root of n , then its powers

$$a, a^2, \dots, a^{\phi(n)}$$

are distinct (mod n) and are all relatively prime to n . In particular, for a prime number p , if a is a primitive root of p , then

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

are distinct (mod p). For the prime number 19, its primitive roots are 2, 3, 10, 13, 14, and 15.

Not all integers have primitive roots. In fact, the only integers with primitive roots are those of the form 2, 4, p^α , and $2p^\alpha$, where p is any odd prime and α is a positive integer. The proof is not simple but can be found in many number theory books, including [ORE76].

Table 2.7 Powers of Integers, Modulo 19

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

Logarithms for Modular Arithmetic

With ordinary positive real numbers, the logarithm function is the inverse of exponentiation. An analogous function exists for modular arithmetic.

Let us briefly review the properties of ordinary logarithms. The logarithm of a number is defined to be the power to which some positive base (except 1) must be raised in order to equal the number. That is, for base x and for a value y ,

$$y = x^{\log_x(y)}$$

The properties of logarithms include

$$\log_x(1) = 0$$

$$\log_x(x) = 1$$

$$\log_x(yz) = \log_x(y) + \log_x(z) \quad (2.19)$$

$$\log_x(y^r) = r \times \log_x(y) \quad (2.20)$$

Consider a primitive root a for some prime number p (the argument can be developed for nonprimes as well). Then we know that the powers of a from

1 through $(p - 1)$ produce each integer from 1 through $(p - 1)$ exactly once. We also know that any integer b satisfies

$$b \equiv r \pmod{p} \text{ for some } r, \text{ where } 0 \leq r \leq (p - 1)$$

by the definition of modular arithmetic. It follows that for any integer b and a primitive root a of prime number p , we can find a unique exponent i such that

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

This exponent i is referred to as the **discrete logarithm** of the number b for the base $a \pmod{p}$. We denote this value as $\text{dlog}_{a,p}(b)$.¹¹

Note the following:

$$\text{dlog}_{a,p}(1) = 0 \text{ because } a^0 \bmod p = 1 \bmod p = 1 \quad (2.21)$$

$$\text{dlog}_{a,p}(a) = 1 \text{ because } a^1 \bmod p = a \quad (2.22)$$

Here is an example using a nonprime modulus, $n = 9$. Here $\phi(n) = 6$ and $a = 2$ is a primitive root. We compute the various powers of a and find

$$\begin{aligned} 2^0 &= 1 & 2^4 &\equiv 7 \pmod{9} \\ 2^1 &= 2 & 2^5 &\equiv 5 \pmod{9} \\ 2^2 &= 4 & 2^6 &\equiv 1 \pmod{9} \\ 2^3 &= 8 \end{aligned}$$

This gives us the following table of the numbers with given discrete logarithms (mod 9) for the root $a = 2$:

Logarithm	0	1	2	3	4	5
Number	1	2	4	8	7	5

To make it easy to obtain the discrete logarithms of a given number, we rearrange the table:

Number	1	2	4	5	7	8
Logarithm	0	1	2	5	4	3

Now consider

$$\begin{aligned} x &= a^{\text{dlog}_{a,p}(x)} \bmod p & y &= a^{\text{dlog}_{a,p}(y)} \bmod p \\ xy &= a^{\text{dlog}_{a,p}(xy)} \bmod p \end{aligned}$$

¹¹Many texts refer to the discrete logarithm as the **index**. There is no generally agreed notation for this concept, much less an agreed name.

Using the rules of modular multiplication,

$$\begin{aligned} xy \bmod p &= [(x \bmod p)(y \bmod p)] \bmod p \\ a^{\text{dlog}_{a,p}(xy)} \bmod p &= [(a^{\text{dlog}_{a,p}(x)} \bmod p)(a^{\text{dlog}_{a,p}(y)} \bmod p)] \bmod p \\ &= (a^{\text{dlog}_{a,p}(x) + \text{dlog}_{a,p}(y)}) \bmod p \end{aligned}$$

But now consider Euler's theorem, which states that, for every a and n that are relatively prime,

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Any positive integer z can be expressed in the form $z = q + k\phi(n)$, with $0 \leq q < \phi(n)$. Therefore, by Euler's theorem,

$$a^z \equiv a^q \pmod{n} \quad \text{if } z \equiv q \pmod{\phi(n)}$$

Applying this to the foregoing equality, we have

$$\text{dlog}_{a,p}(xy) \equiv [\text{dlog}_{a,p}(x) + \text{dlog}_{a,p}(y)] \pmod{\phi(p)}$$

and generalizing,

$$\text{dlog}_{a,p}(y^r) \equiv [r \times \text{dlog}_{a,p}(y)] \pmod{\phi(p)}$$

This demonstrates the analogy between true logarithms and discrete logarithms.

Keep in mind that unique discrete logarithms mod m to some base a exist only if a is a primitive root of m .

Table 2.8, which is directly derived from Table 2.7, shows the sets of discrete logarithms that can be defined for modulus 19.

Calculation of Discrete Logarithms

Consider the equation

$$y = g^x \bmod p$$

Given g , x , and p , it is a straightforward matter to calculate y . At the worst, we must perform x repeated multiplications, and algorithms exist for achieving greater efficiency (see Chapter 9).

However, given y , g , and p , it is, in general, very difficult to calculate x (take the discrete logarithm). The difficulty seems to be on the same order of magnitude as that of factoring primes required for RSA. At the time of this writing, the asymptotically fastest known algorithm for taking discrete logarithms modulo a prime number is on the order of [BETH91]:

$$e^{((\ln p)^{1/3}(\ln(\ln p))^{2/3})}$$

which is not feasible for large primes.

Table 2.8 Tables of Discrete Logarithms, Modulo 19

(a) Discrete logarithms to the base 2, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{2,19}(a)$	18	1	13	2	16	14	6	3	8	17	12	15	5	7	11	4	10	9

(b) Discrete logarithms to the base 3, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{3,19}(a)$	18	7	1	14	4	8	6	3	2	11	12	15	17	13	5	10	16	9

(c) Discrete logarithms to the base 10, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{10,19}(a)$	18	17	5	16	2	4	12	15	10	1	6	3	13	11	7	14	8	9

(d) Discrete logarithms to the base 13, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{13,19}(a)$	18	11	17	4	14	10	12	15	16	7	6	3	1	5	13	8	2	9

(e) Discrete logarithms to the base 14, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{14,19}(a)$	18	13	7	8	10	2	6	3	14	5	12	15	11	1	17	16	4	9

(f) Discrete logarithms to the base 15, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_{15,19}(a)$	18	5	11	10	8	16	12	15	4	13	6	3	7	17	1	2	14	9

2.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

bijection composite number commutative Chinese remainder theorem discrete logarithm divisor Euclidean algorithm	Euler's theorem Euler's totient function Fermat's theorem greatest common divisor identity element index modular arithmetic	modulus order prime number primitive root relatively prime residue
---	---	---

Review Questions

- 2.1 What does it mean to say that b is a divisor of a ?
- 2.2 What is the meaning of the expression a divides b ?
- 2.3 What is the difference between modular arithmetic and ordinary arithmetic?
- 2.4 What is a prime number?
- 2.5 What is Euler's totient function?
- 2.6 The Miller–Rabin test can determine if a number is not prime but cannot determine if a number is prime. How can such an algorithm be used to test for primality?
- 2.7 What is a primitive root of a number?
- 2.8 What is the difference between an index and a discrete logarithm?

Problems

- 2.1 Reformulate Equation (2.1), removing the restriction that a is a nonnegative integer. That is, let a be any integer.
- 2.2 Draw a figure similar to Figure 2.1 for $a < 0$.
- 2.3 For each of the following equations, find an integer x that satisfies the equation.
 - a. $4x \equiv 2 \pmod{3}$
 - b. $7x \equiv 4 \pmod{9}$
 - c. $5x \equiv 3 \pmod{11}$
- 2.4 In this text, we assume that the modulus is a positive integer. But the definition of the expression $a \bmod n$ also makes perfect sense if n is negative. Determine the following:
 - a. $7 \bmod 4$
 - b. $7 \bmod -4$
 - c. $-7 \bmod 4$
 - d. $-7 \bmod -4$
- 2.5 A modulus of 0 does not fit the definition but is defined by convention as follows: $a \bmod 0 = a$. With this definition in mind, what does the following expression mean: $a \equiv b \pmod{0}$?
- 2.6 In Section 2.3, we define the congruence relationship as follows: Two integers a and b are said to be congruent modulo n if $(a \bmod n) = (b \bmod n)$. We then proved that $a \equiv b \pmod{n}$ if $n \mid (a - b)$. Some texts on number theory use this latter relationship as the definition of congruence: Two integers a and b are said to be congruent modulo n if $n \mid (a - b)$. Using this latter definition as the starting point, prove that, if $(a \bmod n) = (b \bmod n)$, then n divides $(a - b)$.
- 2.7 What is the smallest positive integer that has exactly k divisors? Provide answers for values for $1 \leq k \leq 8$.
- 2.8 Prove the following:
 - a. $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$
 - b. $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ imply $a \equiv c \pmod{n}$
- 2.9 Prove the following:
 - a. $[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$
 - b. $[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$
- 2.10 Find the multiplicative inverse of each nonzero element in \mathbb{Z}_5 .
- 2.11 Show that an integer N is congruent modulo 9 to the sum of its decimal digits. For example, $723 \equiv 7 + 2 + 3 \equiv 12 \equiv 1 + 2 \equiv 3 \pmod{9}$. This is the basis for the familiar procedure of “casting out 9’s” when checking computations in arithmetic.

- 2.12** a. Determine $\gcd(72345, 43215)$
 b. Determine $\gcd(3486, 10292)$
- 2.13** The purpose of this problem is to set an upper bound on the number of iterations of the Euclidean algorithm.
- a. Suppose that $m = qn + r$ with $q > 0$ and $0 \leq r < n$. Show that $m/2 > r$.
- b. Let A_i be the value of A in the Euclidean algorithm after the i th iteration. Show that

$$A_{i+2} < \frac{A_i}{2}$$

- c. Show that if m, n , and N are integers with $(1 \leq m, n, \leq 2^N)$, then the Euclidean algorithm takes at most $2N$ steps to find $\gcd(m, n)$.
- 2.14** The Euclidean algorithm has been known for over 2000 years and has always been a favorite among number theorists. After these many years, there is now a potential competitor, invented by J. Stein in 1961. Stein's algorithm is as follows: Determine $\gcd(A, B)$ with $A, B \geq 1$.

STEP 1 Set $A_1 = A, B_1 = B, C_1 = 1$

STEP 2 For $n > 1$, (1) If $A_n = B_n$, stop. $\gcd(A, B) = A_n C_n$

(2) If A_n and B_n are both even, set $A_{n+1} = A_n/2, B_{n+1} = B_n/2,$
 $C_{n+1} = 2C_n$

(3) If A_n is even and B_n is odd, set $A_{n+1} = A_n/2, B_{n+1} = B_n,$
 $C_{n+1} = C_n$

(4) If A_n is odd and B_n is even, set $A_{n+1} = A_n, B_{n+1} = B_n/2,$
 $C_{n+1} = C_n$

(5) If A_n and B_n are both odd, set $A_{n+1} = |A_n - B_n|, B_{n+1} =$
 $\min(B_n, A_n), C_{n+1} = C_n$

Continue to step $n + 1$.

- a. To get a feel for the two algorithms, compute $\gcd(6150, 704)$ using both the Euclidean and Stein's algorithm.
- b. What is the apparent advantage of Stein's algorithm over the Euclidean algorithm?
- 2.15** a. Show that if Stein's algorithm does not stop before the n th step, then

$$C_{n+1} \times \gcd(A_{n+1}, B_{n+1}) = C_n \times \gcd(A_n, B_n)$$

- b. Show that if the algorithm does not stop before step $(n - 1)$, then

$$A_{n+2}B_{n+2} \leq \frac{A_nB_n}{2}$$

- c. Show that if $1 \leq A, B \leq 2^N$, then Stein's algorithm takes at most $4N$ steps to find $\gcd(m, n)$. Thus, Stein's algorithm works in roughly the same number of steps as the Euclidean algorithm.
- d. Demonstrate that Stein's algorithm does indeed return $\gcd(A, B)$.
- 2.16** Using the extended Euclidean algorithm, find the multiplicative inverse of
- a. $135 \bmod 61$
 b. $7465 \bmod 2464$
 c. $42828 \bmod 6407$
- 2.17** The purpose of this problem is to determine how many prime numbers there are. Suppose there are a total of n prime numbers, and we list these in order: $p_1 = 2 < p_2 = 3 < p_3 = 5 < \dots < p_n$.
- a. Define $X = 1 + p_1 p_2 \dots p_n$. That is, X is equal to one plus the product of all the primes. Can we find a prime number P_m that divides X ?
- b. What can you say about m ?
- c. Deduce that the total number of primes cannot be finite.
- d. Show that $P_{n+1} \leq 1 + p_1 p_2 \dots p_n$.

- 2.18** The purpose of this problem is to demonstrate that the probability that two random numbers are relatively prime is about 0.6.
- a. Let $P = \Pr[\gcd(a, b) = 1]$. Show that $P = \Pr[\gcd(a, b) = d] = P/d^2$. *Hint:* Consider the quantity $\gcd\left(\frac{a}{d}, \frac{b}{d}\right)$.
- b. The sum of the result of part (a) over all possible values of d is 1. That is $\sum_{d \geq 1} \Pr[\gcd(a, b) = d] = 1$. Use this equality to determine the value of P . *Hint:* Use the identity $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$.
- 2.19** Why is $\gcd(n, n+1) = 1$ for two consecutive integers n and $n+1$?
- 2.20** Using Fermat's theorem, find $4^{225} \bmod 13$.
- 2.21** Use Fermat's theorem to find a number a between 0 and 92 with a congruent to 7^{1013} modulo 93.
- 2.22** Use Fermat's theorem to find a number x between 0 and 37 with x^{73} congruent to 4 modulo 37. (You should not need to use any brute-force searching.)
- 2.23** Use Euler's theorem to find a number a between 0 and 9 such that a is congruent to 9^{101} modulo 10. (*Note:* This is the same as the last digit of the decimal expansion of 9^{100} .)
- 2.24** Use Euler's theorem to find a number x between 0 and 14 with x^{61} congruent to 7 modulo 15. (You should not need to use any brute-force searching.)
- 2.25** Notice in Table 2.6 that $\phi(n)$ is even for $n > 2$. This is true for all $n > 2$. Give a concise argument why this is so.
- 2.26** Prove the following: If p is prime, then $\phi(p^j) = p^j - p^{j-1}$. *Hint:* What numbers have a factor in common with p^j ?
- 2.27** It can be shown (see any book on number theory) that if $\gcd(m, n) = 1$ then $\phi(mn) = \phi(m)\phi(n)$. Using this property, the property developed in the preceding problem, and the property that $\phi(p) = p - 1$ for p prime, it is straightforward to determine the value of $\phi(n)$ for any n . Determine the following:
a. $\phi(29)$ b. $\phi(51)$ c. $\phi(455)$ d. $\phi(616)$
- 2.28** It can also be shown that for arbitrary positive integer a , $\phi(a)$ is given by

$$\phi(a) = \prod_{i=1}^t [p_i^{a_i-1}(p_i - 1)]$$

where a is given by Equation (2.9), namely: $a = P_1^{a_1} P_2^{a_2} \dots P_t^{a_t}$. Demonstrate this result.

- 2.29** Consider the function: $f(n)$ = number of elements in the set $\{a: 0 \leq a < n \text{ and } \gcd(a, n) = 1\}$. What is this function?
- 2.30** Although ancient Chinese mathematicians did good work coming up with their remainder theorem, they did not always get it right. They had a test for primality. The test said that n is prime if and only if n divides $(2^n - 2)$.
- a. Give an example that satisfies the condition using an odd prime.
- b. The condition is obviously true for $n = 2$. Prove that the condition is true if n is an odd prime (proving the **if** condition).
- c. Give an example of an odd n that is not prime and that does not satisfy the condition. You can do this with nonprime numbers up to a very large value. This misled the Chinese mathematicians into thinking that if the condition is true then n is prime.
- d. Unfortunately, the ancient Chinese never tried $n = 341$, which is nonprime ($341 = 11 \times 31$), yet 341 divides $2^{341} - 2$ without remainder. Demonstrate that $2341 \equiv 2 \pmod{341}$ (disproving the **only if** condition). *Hint:* It is not necessary to calculate 2^{341} ; play around with the congruences instead.

- 2.31** Show that, if n is an odd composite integer, then the Miller–Rabin test will return inconclusive for $a = 1$ and $a = (n - 1)$.
- 2.32** If n is composite and passes the Miller–Rabin test for the base a , then n is called a *strong pseudoprime to the base a* . Show that 2047 is a strong pseudoprime to the base 2.
- 2.33** A common formulation of the Chinese remainder theorem (CRT) is as follows: Let m_1, \dots, m_k be integers that are pairwise relatively prime for $1 \leq i, j \leq k$, and $i \neq j$. Define M to be the product of all the m_i 's. Let a_1, \dots, a_k be integers. Then the set of congruences:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

has a unique solution modulo M . Show that the theorem stated in this form is true.

- 2.34** The example used by Sun-Tsu to illustrate the CRT was

$$x \equiv 2 \pmod{3}; x \equiv 3 \pmod{5}; x \equiv 2 \pmod{7}$$

Solve for x .

- 2.35** Six professors begin courses on Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday, respectively, and announce their intentions of lecturing at intervals of 3, 2, 5, 6, 1, and 4 days, respectively. The regulations of the university forbid Sunday lectures (so that a Sunday lecture must be omitted). When first will all six professors find themselves compelled to omit a lecture? *Hint:* Use the CRT.
- 2.36** Find all primitive roots of 37.
- 2.37** Given 5 as a primitive root of 23, construct a table of discrete logarithms, and use it to solve the following congruences.
- $3x^5 \equiv 2 \pmod{23}$
 - $7x^{10} + 1 \equiv 0 \pmod{23}$
 - $5^x \equiv 6 \pmod{23}$

Programming Problems

- 2.1** Write a computer program that implements fast exponentiation (successive squaring) modulo n .
- 2.2** Write a computer program that implements the Miller–Rabin algorithm for a user-specified n . The program should allow the user two choices: (1) specify a possible witness a to test using the Witness procedure or (2) specify a number s of random witnesses for the Miller–Rabin test to check.

APPENDIX 2A THE MEANING OF MOD

The operator mod is used in this book and in the literature in two different ways: as a binary operator and as a congruence relation. This appendix explains the distinction and precisely defines the notation used in this book regarding parentheses. This notation is common but, unfortunately, not universal.

The Binary Operator mod

If a is an integer and n is a positive integer, we define $a \bmod n$ to be the remainder when a is divided by n . The integer n is called the **modulus**, and the remainder is called the **residue**. Thus, for any integer a , we can always write

$$a = \lfloor a/n \rfloor \times n + (a \bmod n)$$

Formally, we define the operator mod as

$$a \bmod n = a - \lfloor a/n \rfloor \times n \quad \text{for } n \neq 0$$

As a binary operation, mod takes two integer arguments and returns the remainder. For example, $7 \bmod 3 = 1$. The arguments may be integers, integer variables, or integer variable expressions. For example, all of the following are valid, with the obvious meanings:

$$\begin{aligned} &7 \bmod 3 \\ &7 \bmod m \\ &x \bmod 3 \\ &x \bmod m \\ &(x^2 + y + 1) \bmod (2m + n) \end{aligned}$$

where all of the variables are integers. In each case, the left-hand term is divided by the right-hand term, and the resulting value is the remainder. Note that if either the left- or right-hand argument is an expression, the expression is parenthesized. The operator mod is not inside parentheses.

In fact, the mod operation also works if the two arguments are arbitrary real numbers, not just integers. In this book, we are concerned only with the integer operation.

The Congruence Relation mod

As a congruence relation, mod expresses that two arguments have the same remainder with respect to a given modulus. For example, $7 \equiv 4 \pmod{3}$ expresses the fact that both 7 and 4 have a remainder of 1 when divided by 3. The following two expressions are equivalent:

$$a \equiv b \pmod{m} \quad \Leftrightarrow \quad a \bmod m = b \bmod m$$

Another way of expressing it is to say that the expression $a \equiv b \pmod{m}$ is the same as saying that $a - b$ is an integral multiple of m . Again, all the arguments may be integers, integer variables, or integer variable expressions. For example, all of the following are valid, with the obvious meanings:

$$\begin{aligned} &7 \equiv 4 \pmod{3} \\ &x \equiv y \pmod{m} \\ &(x^2 + y + 1) \equiv (a + 1) \pmod{[m + n]} \end{aligned}$$

where all of the variables are integers. Two conventions are used. The congruence sign is \equiv . The modulus for the relation is defined by placing the mod operator followed by the modulus in parentheses. ³⁸

ASCII Encoding

In ASCII encoding, each letter is converted to one byte. Look at the following examples:

A = 65 or 0b01000001

B = 66 or 0b01000010

C = 67 or 0b01000011

ABC = 0b01000001 0b01000010 0b01000011

Base64 Encoding Text

Base64, also known as privacy enhanced electronic mail (PEM), is the encoding that converts binary data into a textual format; it can be passed through communication channels where text can be handled in a safe environment. PEM is primarily used in the email encryption process. To use the functions included in the Base64 module, you will need to import the library in your code. Base64 offers a decode and encode module that both accepts input and provides output.

To break ASCII encoding into Base64-encoded text, each sequence of six bits encodes to a single character. The characters used can be seen in the following examples:

A-Z: 0-25

a-z: 26-51

0, 1, 2, ..., 9: 52-61

+, /: 62 and 63

Examine the 24 bits from the previous section:

0b01000001 0b01000010 0b01000011

Break the line into 6-bit groups:

0b010000 010100 001001 000011

When you convert the four groups to decimal, you will see that they are equal to the following:

16 20 9 3

You now convert the numbers to Base64:

Q U J D

Therefore, when you encode “ABC” to Base64, you should end up with QUJD, as shown here:

```
>>> import base64
>>> value = 'ABC'.encode()
>>> print(base64.b64encode(value))
b'QUJD'
```

In the previous example, we used Python to encode three bytes at a time. When performing Base64 encoding, the text is broken down into groups of three. In the event that the text cannot be broken down into groups of three, you will see the padding character, which is shown using the equal sign (=). If the example had four bytes, then the output would look like the following:

```
>>> value = 'ABCD'.encode()
>>> print(base64.b64encode(value))
b'QUJDRA=='

>>> value = 'ABCDE'.encode()
>>> print(base64.b64encode(value))
b'QUJDREU='

>>> value = 'ABCDEF'.encode()
>>> print(base64.b64encode(value))
b'QUJDREVG'
```

The preceding padding uses null bytes, which equals A. The capital A is the first character you have in Base64; it stands for six bits of zero (000000). You can prove that with the following:

```
>>> value1 = 'ABCD'.encode()
>>> value2 = 'ABCD\x00'.encode()
>>> value3 = 'ABCD\x00\x00'.encode()
>>> print(base64.b64encode(value1))
b'QUJDRA=='
>>> print(base64.b64encode(value2))
b'QUJDRAA='
>>> print(base64.b64encode(value3))
b'QUJDRAAA'
```

Once you start evaluating Base64, it may become confusing to tell Base64 and ASCII apart. One of the major differences between the two is the encoding process. When you encode text in ASCII, you first start with a text string, and it is converted into a sequence of bytes. When you encode Base64, you are starting with a sequence of bytes and converting the bytes to text.

Binary Data

We will now examine binary data. The file types on your computer are composed of binary data. ASCII data always begins with a first bit of zero. When you open a file using Python, you can convert binary data to Base64:

```
With open('file.exe','rb') as f:
    data = f.read()
    data.encode()
```

Decoding

In this section, we will take what you have learned using the `encode()` methods and get the inverse using the `decode()` method. Examine the following syntax:

```
>>> # encode ABCD
>>> value1 = 'ABCD'.encode()
>>> value1
b'ABCD'

>>> #decode ABCD
>>> value1.decode()
'ABCD'
```

Once you have the binary values, you can use the `base64` library to encode the values using `b64encode`. The inverse is the `b64decode`:

```
>>> myb64 = base64.b64encode(value1)
>>> myb64
b'QUJDRA=='

>>> my = base64.b64decode(myb64)
>> my
b'ABCD'

>>> print (my.decode())
ABCD
>>>
```

Historical Ciphers

Since the invention of writing, there has been the need for message secrecy. While most of the historical ciphers you will learn about in this chapter have been broken, it is important to understand their encryption scheme so that you have a better understanding of how to break them.

In fact, all historical codes used prior to 1980 have been broken except for the one the Native American code talkers used in World Wars I and II. During World War I, the Choctaw Indian language was used, and during World War II, the Navajo language was used. The US Marine Corps recruited Navajo men to serve as Marine Corps radio operators. Both languages served perfectly as they were unwritten and undecipherable due to their complexities.

Scytale of Sparta

One of the oldest cryptographic tools was the Spartan scytale, which was used to perform a transposition cipher. We will examine transposition ciphers in greater detail both later in this chapter and throughout the book. The scytale consisted of a cylinder with a strip of parchment; the parchment would wrap around the cylinder and then the message would be written lengthwise on the parchment. The key in this case would be the radius of the cylinder itself. If the parchment were wrapped around a cylinder of a different radius, the letters would not align in the same way, making the message unreadable.

Substitution Ciphers

The substitution cipher simply substitutes one letter in the alphabet for another based upon a cryptovvariable. The substitution involves shifting positions in the alphabet. This includes the Caesar cipher and ROT-13, which will be covered shortly. Examine the following example:

Plaintext: WE HOLD THESE TRUTHS TO BE SELF-EVIDENT, THAT ALL MEN ARE CREATED EQUAL.

Ciphertext: ZH KROG WKHVH WUXWKV WR EH VHOI-HYLGHQW, WKDW DOO PHQ DUH FUHDWHG HTXDO.

The Python syntax to both encrypt and decrypt a substitution cipher is presented next. This example shows the use of ROT-13:

```
key = 'abcdefghijklmnopqrstuvwxyz'

def enc_substitution(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()
```

```
def dec_substitution(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
        except ValueError:
            result += l

    return result

origtext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_substitution(13, origtext)
plaintext = dec_substitution(13, ciphertext)

print(origtext)
print(ciphertext)
print(plaintext)
```

Caesar Cipher

The Caesar cipher is one of the oldest recorded ciphers. De Vita Caesarum, Divus Iulius ("The Lives of the Caesars, the Deified Julius), commonly known as The Twelve Caesars, was written in approximately 121 CE. In The Twelve Caesars, it states that if someone has a message that they want to keep private, they can do so by changing the order of the letters so that the original word cannot be determined. When the recipient of the message receives it, the reader must substitute the letters so that they shift by four positions.

Simply put, the cipher shifted letters of the alphabet three places forward so that the letter A was replaced with the letter D, the letter B was replaced with E, and so on. Once the end of the alphabet was reached, the letters would start over:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

The Caesar cipher is an example of a mono-alphabet substitution. This type of substitution substitutes one character of the ciphertext from a character in plaintext. Other examples that include this type of substitution are Atbash, Affine, and the ROT-13 cipher. There are many flaws with this type of cipher, the most obvious of which is that the encryption and decryption methods are fixed and require no shared key. This would allow anyone who knew this method to read Caesar's encrypted messages with ease. Over the years, there have been several variations that include ROT-13, which shifts the letters 13

places instead of 3. We will explore how to encrypt and decrypt Caesar cipher and ROT-13 codes using Python.

For example, given that x is the current letter of the alphabet, the Caesar cipher function adds three for encryption and subtracts three for decryption. While this could be a variable shift, let's start with the original shift of 3:

$$\text{Enc}(x) = (x + 3) \% 26$$

$$\text{Dec}(x) = (x - 3) \% 26$$

These functions are the first use of modular arithmetic; there are other ways to get the same result, but this is the cleanest and fastest method. The encryption formula adds 3 to the numeric value of the number. If the value exceeds 26, which is the final position of Z, then the modular arithmetic wraps the value back to the beginning of the alphabet. While it is possible to get the ordinal (ord) of a number and convert it back to ASCII, the use of the key simplifies the alphabet indexing. You will learn how to use the `ord()` function when we explore the Vigenère cipher in the next section. In the following Python recipe, the `enc_caesar` function will access a variable index to encrypt the plaintext that is passed in.

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_caesar(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()

plaintext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_caesar(3, plaintext)
print (ciphertext)
```

The output of this should result in the following:

```
zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh fuhdwhg httxdo.
```

The reverse in this case is straightforward. Instead of adding, we subtract. The decryption would look like the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def dec_caesar(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
```

```
i = (key.index(l) - n) % 26
result += key[i]
except ValueError:
    result += l
return result

ciphertext = 'zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh
fuhdwhg httxdo.'
plaintext = dec_caesar(3, ciphertext)
print (plaintext)
```

ROT-13

Now that you understand the Caesar cipher, take a look at the ROT-13 cipher. The unique construction of the ROT-13 cipher allows you to encrypt and decrypt using the same method. The reason for this is that since ROT-13 moves the letter of the alphabet exactly halfway, when you run the process again, the letter goes back to its original value.

To see the code behind the cipher, take a look at the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_dec_ROT13(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result.lower()

plaintext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_dec_ROT13(13, plaintext)
print (ciphertext)

plaintext = enc_dec_ROT13(13, ciphertext)
print (plaintext)

jr ubyq gurfr gehguf gb or frys-rivrag, gung nyy zra ner pernrgq rdhny.
we hold these truths to be self-evident, that all men are created equal.
```

Whether we use a Caesar cipher or the ROT-13 variation, brute-forcing an attack would take at most 25 tries, and we could easily decipher the plaintext results when we see a language we understand. This will get more complex as we explore the other historical ciphers; the cryptanalysis requires frequency analysis and language detectors. We will focus on these concepts in upcoming chapters.

Atbash Cipher

The Atbash cipher is one of many substitution ciphers you will explore. Similar to ROT-13, the Atbash cipher is also its own inverse, which means you can encode and decode using the same key; this also means we need to have only one function to perform both the encryption and decryption processes. The original cipher was used to encode the Hebrew alphabets but, in reality, it can be modified to encode or decode any alphabet. The Atbash cipher is often thought to be a special case of the Alphine cipher that we will be exploring next.

The following is the Atbash key as it maps to the English alphabet:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

The Python code that implements the Atbash cipher is as follows:

```
def toAtBash(text):
    characters = list(text.upper())
    result = ""
    for character in characters:
        if character in code_dictionary:
            result += code_dictionary.get(character)
        else:
            result += character # preserve non-alpha chars found
    return result

alphabet = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
reverse_alphabet = list(reversed(alphabet))
code_dictionary = dict(zip(alphabet, reverse_alphabet))
plainText= "we hold these truths to be self-evident"
print(plainText)
cipherText = toAtBash(plainText)
print(cipherText)
cipherText = toAtBash(cipherText)
print(cipherText)

we hold these truths to be self-evident
DV SLOW GSVHV GIFGSH GL YV HVOU-VERWVMG
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT
```

Vigenère Cipher

The Vigenère cipher consists of using several Caesar ciphers in sequence with different shift values. To encipher, a table of alphabets can be used, termed a tabula recta, Vigenère square, or Vigenère table. It consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left

compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

Here's an example:

Keyword: **DECLARATION**

D	E	C	L	A	R	A	T	I	O	N
3	4	2	11	0	17	0	19	8	14	13

Plaintext: We hold these truths to be self-evident, that all men are created equal.

Ciphertext: zi jzlu tamgr wvwehj th js fhph-pvzdxvh, gkev llc mxv oeh gtpakew mehdp.

To create a numeric key such as the one shown, use the following syntax. You should see the output [3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:

```
def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i]) - 65
        keyArray.append(keyElement)
    return keyArray

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
print(key)
```

Once you have created the key, you can use it to create ciphertext. You should see the output dpnemvnmifrwgtpakewbsdxcn:

```
def shiftEnc(c, n):
    return chr(((ord(c) - ord('A') + n) % 26) + ord('a'))

def enc_vigenere(plaintext, key):
    secret = "".join([shiftEnc(plaintext[i], key[i % len(key)]) for i
in range(len(plaintext))])
    return secret

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
plaintext = 'ALL MEN ARE CREATED EQUAL'
ciphertext = enc_vigenere(plaintext, key)
print(ciphertext)
```

When you know the key, such as in this case, you can decrypt the Vigenère cipher with the following:

```
def shiftDec(c, n):
    c = c.upper()
    return chr((ord(c) - ord('A') - n) % 26 + ord('a'))

def dec_vigenere(ciphertext, key):
    plain = "".join([shiftDec(ciphertext[i], key[i % len(key)]) for i in
range(len(ciphertext))])
    return plain

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
decoded = dec_vigenere(ciphertext, key)
```

We will examine how to brute-force the Vigenère cipher in a later chapter. We will do this by creating a random key that will use the same encryption function, and then we will use frequency analysis to help find the appropriate key. For now, it is more important to understand how the Python code works with this cryptography scheme.

Playfair

The Playfair cipher was used by the Allied forces in World War II; it is the most common digraphic system, and was named after Lord Playfair of England. With this scheme, the sender and the receiver use a shared keyword. The keyword is then used to construct a table that consists of five rows and five columns; the shared word is then populated into the table followed by the rest of the alphabet. As the table is being built out, letters that already appear in the key are skipped. In addition, the letters I and J use the same letter. For this example, we will use the word DECLARATION, as shown here:

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

The Playfair table is read by looking at where the two letters of the blocks intersect. For example, if the first block, TH, were made into a rectangle, the letters at the other two corners of the rectangle would be OF. To see a graphical interpretation of this, examine the next table:

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

When you have two letters that fall in the same column such as WE, you will need to incorporate the next lower letter, and wrap to the top of the column if necessary. This would form a block around the letters ET. The block WE would be encrypted as ET. The same rule applies if letters fall in the same row.

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

Using the preceding table, examine the following plaintext and ciphertext:

Plaintext: He has obstructed the Administration of Justice by refusing his Assent to Laws for establishing Judiciary Powers

Ciphertext: flklyhhmitqafterfldeqrropondionrthazpogiawhvvntpmuorkxgouyluplriwnnyadypwkntwapodkcoysorkxazcrigdnzystem

To create a function in Python that encrypts plaintext using Playfair, type the following:

```
def Playfair_box_shift(i1, i2):
    r1 = i1/5
    r2 = i2/5
    c1 = i1 % 5
    c2 = i2 % 5
    out_r1 = r1
    out_c1 = c2
    out_r2 = r2
    out_c2 = c1
    if r1 == r2:
        out_c1 = (c1 + 1) % 5
        out_c2 = (c2 + 1) % 5
    elif c1 == c2:
        out_r1 = (r1 + 1) % 5
        out_r2 = (r2 + 1) % 5
    return out_r1*5 + out_c1, out_r2*5 + out_c2
```

```

def Playfair_enc(plain):
    random.shuffle(words)
    seed = "".join(words[:10]).replace('j','i')
    alpha = 'abcdefghijklmnopqrstuvwxyz'
    suffix = "".join( sorted( list( set(alpha) - set(seed) ) ) )
    seed_set = set()
    prefix = ""
    for letter in seed:
        if not letter in seed_set:
            seed_set.add(letter)
            prefix += letter
    key = prefix + suffix
    secret = ""
    for i in range(0,len(plain),2):
        chr1 = plain[i]
        chr2 = plain[i+1]
        if chr1 == chr2:
            chr2 = 'X'
        i1 = key.find(chr1.lower())
        i2 = key.find(chr2.lower())
        ci1, ci2 = Playfair_box_shift(i1, i2)
        secret += key[ci1] + key[ci2]
    return secret, key

```

As with the other historical ciphers presented in this chapter, the Playfair cipher can be cracked given enough text. Playfair has a weakness; it will decrypt to the same letter pattern in the plaintext for digraphs that are reciprocals of each other. Examine the next table. Notice how the letters DT and ER form a block. The letters ER (and their reverse RE) form a common digraph in the English language. Digraphs are often used for phonemes that cannot be represented using a single character, like the English *sh* in *ship* and *fish*. When using the English language, there are many words that contain these digraphs, such as *receiver*; notice how receiver contains both an RE and the reciprocal ER; these two letter combinations would encrypt to letter combinations that are easy to identify such as TD and DT. This weakness gives you additional foresight into the cryptographic scheme.

D	E	C	L	A
R	T	I	O	N
B	F	G	H	K
M	P	Q	S	U
V	W	X	Y	Z

Other digraphs in the English language include *sc*, *ng*, *ch*, *ck*, *gh*, *py*, *rh*, *sh*, *ti*, *th*, *wh*, *zh*, *ci*, *wr*, *qu*. Identifying nearby reversed digraphs in the ciphertext and matching the pattern to a list of known plaintext words containing the pattern

is an easy way to generate possible plaintext strings with which to begin constructing the key.

Another way to break the Playfair cipher is with a method called *shotgun hill climbing*. This starts with a random square of letters. Then minor changes are introduced (i.e., switching letters, or rows, or reflecting the entire square) to see if the candidate plaintext is more like standard plaintext than before the change. The minor changes are examined through frequency analysis and language detectors. For now, here is the Python that decrypts the ciphertext using Playfair with a known shared key:

```
def Playfair_box_shift_dec(i1, i2):
    r1 = i1/5
    r2 = i2/5
    c1 = i1 % 5
    c2 = i2 % 5
    out_r1 = r1
    out_c1 = c2
    out_r2 = r2
    out_c2 = c1
    if r1 == r2:
        out_c1 = (c1 - 1) % 5
        out_c2 = (c2 - 1) % 5
    elif c1 == c2:
        out_r1 = (r1 - 1) % 5
        out_r2 = (r2 - 1) % 5
    return out_r1*5 + out_c1, out_r2*5 + out_c2

def Playfair_dec(ciphertext, sharedkey):
    seed = "".join(sharedkey).replace('j','i')
    alpha = 'abcdefghijklmnopqrstuvwxyz'
    suffix = "".join( sorted( list( set(alpha) - set(seed) ) ) )
    seed_set = set()
    prefix = ""
    for letter in seed:
        if not letter in seed_set:
            seed_set.add(letter)
            prefix += letter
    key = prefix + suffix
    plaintext = ""
    for i in range(0,len(ciphertext),2):
        chr1 = ciphertext[i]
        chr2 = ciphertext[i+1]
        print chr1, chr2
        if chr1 == chr2:
            chr2 = 'X'
        i1 = key.find(chr1.lower())
        i2 = key.find(chr2.lower())
        ci1, ci2 = Playfair_box_shift_dec(i1, i2)
        plaintext += key[ci1] + key[ci2]
    return plaintext
```

Hill 2x2

The Hill 2x2 cipher is a polygraphic substitution cipher based on linear algebra. The inventor, Lester S. Hill, created the cipher in 1929. The cipher uses matrices and matrix multiplication to mix the plaintext to produce the ciphertext. To fully understand the Hill cipher, it helps to be familiar with a branch of mathematics known as number theory. The Hill cipher is often covered in depth in many textbooks on the number theory topic. The key used here is HILL, which corresponds to the numbers 7, 8, 11, and 11:

Key: 7, 8, 11, 11

Plaintext: SECRETMESSAG

Ciphertext: CIUBYTMUKGWO

To create a Python function that will create the Hill 2x2 encryption, type the following:

```
import sys
import numpy as np

def cipher_encryption(plain, key):

    # if message length is an odd number, place a zero at the end.
    len_chk = 0
    if len(plain) % 2 != 0:
        plain += "0"
        len_chk = 1

    # msg to matrices
    row = 2
    col = int(len(plain)/2)
    msg2d = np.zeros((row, col), dtype=int)

    itr1 = 0
    itr2 = 0
    for i in range(len(plain)):
        if i%2 == 0:
            msg2d[0][itr1] = int(ord(plain[i]) - 65)
            itr1 += 1
        else:
            msg2d[1][itr2] = int(ord(plain[i]) - 65)
            itr2 += 1

    # key to 2x2
    key2d = np.zeros((2,2), dtype=int)
    itr3 = 0
    for i in range(2):
```

```
        for j in range(2):
            key2d[i][j] = ord(key[itr3]) - 65
            itr3 += 1

    print (key2d)

    # checking validity of the key
    # finding determinant
    deter = key2d[0][0] * key2d[1][1] - key2d[0][1] * key2d[1][0]
    deter = deter % 26

    # finding multiplicative inverse
    for i in range(26):
        temp_inv = deter * i
        if temp_inv % 26 == 1:
            mul_inv = i
            break
        else:
            continue

    if mul_inv == -1:
        print("Invalid key")
        sys.exit()

    encryp_text = ""
    itr_count = int(len(plain)/2)
    if len_chk == 0:
        for i in range(itr_count):
            temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
            encryp_text += chr((temp1 % 26) + 65)
            temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
            encryp_text += chr((temp2 % 26) + 65)
        else:
            for i in range(itr_count-1):
                temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
                encryp_text += chr((temp1 % 26) + 65)
                temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
                encryp_text += chr((temp2 % 26) + 65)

    print("Encrypted text: {}".format(encryp_text))
    return encryp_text

def cipher_decryption(cipher, key):
```

```

# if message length is an odd number, place a zero at the end.
len_chk = 0
if len(cipher) % 2 != 0:
    cipher += "0"
    len_chk = 1

# msg to matrices
row = 2
col = int(len(cipher)/2)
msg2d = np.zeros((row, col), dtype=int)

itr1 = 0
itr2 = 0
for i in range(len(cipher)):
    if i%2 == 0:
        msg2d[0][itr1] = int(ord(cipher[i]) - 65)
        itr1 += 1
    else:
        msg2d[1][itr2] = int(ord(cipher[i]) - 65)
        itr2 += 1

# key to 2x2
key2d = np.zeros((2,2), dtype=int)
itr3 = 0
for i in range(2):
    for j in range(2):
        key2d[i][j] = ord(key[itr3]) - 65
        itr3 += 1

# finding determinant
deter = key2d[0][0] * key2d[1][1] - key2d[0][1] * key2d[1][0]
deter = deter % 26

# finding multiplicative inverse
for i in range(26):
    temp_inv = deter * i
    if temp_inv % 26 == 1:
        mul_inv = i
        break
    else:
        continue

# adjugate matrix
# swapping
key2d[0][0], key2d[1][1] = key2d[1][1], key2d[0][0]

#changing signs
key2d[0][1] *= -1
key2d[1][0] *= -1

```

```

key2d[0][1] = key2d[0][1] % 26
key2d[1][0] = key2d[1][0] % 26

# multiplying multiplicative inverse with adjugate matrix
for i in range(2):
    for j in range(2):
        key2d[i][j] *= mul_inv

# modulo
for i in range(2):
    for j in range(2):
        key2d[i][j] = key2d[i][j] % 26

# cipher to plaintext
decryp_text = ""
itr_count = int(len(cipher)/2)
if len_chk == 0:
    for i in range(itr_count):
        temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
        decryp_text += chr((temp1 % 26) + 65)
        temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
        decryp_text += chr((temp2 % 26) + 65)
    # for
else:
    for i in range(itr_count-1):
        temp1 = msg2d[0][i] * key2d[0][0] + msg2d[1][i] * key2d[0]
[1]
        decryp_text += chr((temp1 % 26) + 65)
        temp2 = msg2d[0][i] * key2d[1][0] + msg2d[1][i] * key2d[1]
[1]
        decryp_text += chr((temp2 % 26) + 65)
    # for
# if else

print("Decrypted text: {}".format(decryp_text))

plaintext = "Secret Message"
plaintext = plaintext.upper().replace(" ", "")
key = "hill"
key = key.upper().replace(" ", "")
ciphertext = cipher_encryption(plaintext, key)
cipher_decryption(ciphertext, key)

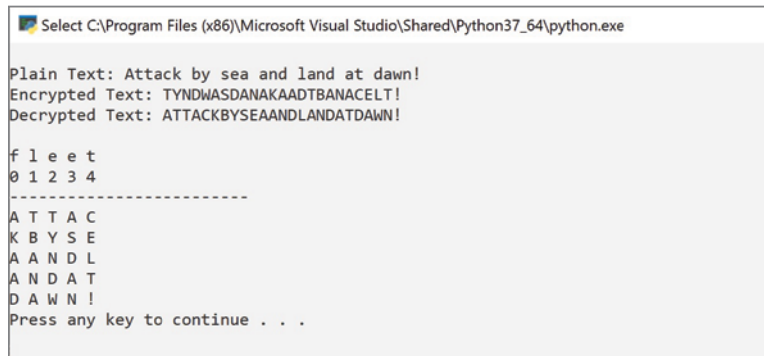
[[ 7  8]
 [11 11]]
Encrypted text: CIUBYTMUKGWO
Decrypted text: SECRETMESSAG

```

To examine the cryptanalysis of a Hill 2×2 cipher, you attack it by measuring the frequencies of all the digraphs that occur in the ciphertext. In standard English, the most common digraph is *th*, followed by *he*. If you know that the Hill cipher has been employed and the most common digraph is *kx*, followed by *vz*, you would guess that *kx* and *vz* correspond to *th* and *he*, respectively. Once you have a better understanding of frequency analysis, you will revisit this cipher and break it without a key.

Column Transposition

The column transposition cipher is based on a geometric arrangement. While relatively insecure as a standalone cipher, columnar transposition can be a powerful enhancement to other systems. A keyword is chosen as a permutation of *N* columns. The message is then written in a grid with *N* columns. Finally, in the order of the permutation, the columns are taken as the ciphertext. The keyword in this case is 11 characters long. The columns are read off in the letters' numerical order. The order is represented by the numbers under the keyword to give you a visual representation. See Figure 3.1.



```
Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe

Plain Text: Attack by sea and land at dawn!
Encrypted Text: TYNDWASDANAKAADTBANACELT!
Decrypted Text: ATTACKBYSEAANDLANDATDAWN!

f l e e t
0 1 2 3 4
-----
A T T A C
K B Y S E
A A N D L
A N D A T
D A W N !
Press any key to continue . . .
```

Figure 3.1: Column transposition table

The following code reworks the example to be a shorter (it is really long) and removes some debug and fairly self-evident comments in the code. Note that this code works only on Python 3:

```
def cipher_encryption(plain_text, key):

    keyword_num_list = keyword_num_assign(key)
    num_of_rows = int(len(plain_text) / len(key))

    # break message into grid for key
    arr = [[0] * len(key) for i in range(num_of_rows)]
    z = 0
```



```
for i in range(num_of_rows):
    for j in range(len(key)):
        arr[i][j] = plain_text[z]
        z += 1

num_loc = get_number_location(key, keyword_num_list)

cipher_text = ""
k = 0
for i in range(num_of_rows):
    if k == len(key):
        break
    else:
        d = int(num_loc[k])

        for j in range(num_of_rows):
            cipher_text += arr[j][d]
        k += 1
return cipher_text

def get_number_location(key, keyword_num_list):
    num_loc = ""
    for i in range(len(key) + 1):
        for j in range(len(key)):
            if keyword_num_list[j] == i:
                num_loc += str(j)
    return num_loc

def keyword_num_assign(key):
    alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    keyword_num_list = list(range(len(key)))
    init = 0
    for i in range(len(alpha)):
        for j in range(len(key)):
            if alpha[i] == key[j]:
                init += 1
                keyword_num_list[j] = init
    return keyword_num_list

def print_grid(plain_text, key):

    keyword_num_list = keyword_num_assign(key)

    for i in range(len(key)):
        print(key[i], end = " ", flush=True)

    print()
    for i in range(len(key)):
        print(str(keyword_num_list[i]), end=" ", flush=True)
    print()
```

```

print("-----")

# in case characters don't fit the entire grid perfectly.
extra_letters = len(plain_text) % len(key)

dummy_characters = len(key) - extra_letters

if extra_letters != 0:
    for i in range(dummy_characters):
        plain_text += "."

num_of_rows = int(len(plain_text) / len(key))

# Converting message into a grid
arr = [[0] * len(key) for i in range(num_of_rows)]
z = 0

for i in range(num_of_rows):
    for j in range(len(key)):
        arr[i][j] = plain_text[z]
        z += 1

for i in range(num_of_rows):
    for j in range(len(key)):
        print(arr[i][j], end=" ", flush=True)
    print()

def cipher_decryption(encrypted, key):

    keyword_num_list = keyword_num_assign(key)
    num_of_rows = int(len(encrypted) / len(key))

    num_loc = get_number_location(key, keyword_num_list)

    # Converting message into a grid
    arr = [[0] * len(key) for i in range(num_of_rows)]

    # decipher
    plain_text = ""
    k = 0
    itr = 0

    for i in range(len(encrypted)):
        d = 0
        if k == len(key):
            k = 0
        else:
            d: int = int(num_loc[k])
        for j in range(num_of_rows):
            arr[j][d] = encrypted[itr]

```

```

        itr += 1
    if itr == len(encrypted):
        break
    k += 1

print()

for i in range(num_of_rows):
    for j in range(len(key)):
        plain_text += str(arr[i][j])
    return plain_text

plain_text = "Attack by sea and land at dawn!"
key = "fleet"

msg = plain_text.replace(" ", "").upper()
msgkey = key.upper()

encrypted = cipher_encryption(msg, msgkey)
decrypted = cipher_decryption(encrypted, msgkey)

print ("Plain Text: " + plain_text)
print ("Encrypted Text: " + encrypted)
print ("Decrypted Text: " + decrypted)
print ()
print_grid(msg, key)

```

Affine Cipher

Next on our list is the Affine cipher. The Affine cipher is a mono-alphabetic substitution cipher. The difference with the Affine cipher is that each letter is mapped to a numeric equivalent, encrypted using a mathematical function, and then converted back to a letter. While using the mathematical function may sound difficult, the process is fundamentally a standard substitution cipher with a set of rules that govern which letters map to other letters. The mathematical function makes use of the modulo m , where m is the length of the alphabet. Each letter is mapped to a number as shown in the following grid where A = 0, B = 1, . . . , Y = 24, Z = 25:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

The encryption process of the Affine cipher uses modular arithmetic to transform letters into their corresponding integers and then converts the integer into

another number, which in turn is converted back into a letter. The encryption function for a single letter would look like the following:

$$\text{Encrypt}(x) = (ax + b) \bmod m$$

where x is the integer represented by the letter and m is the number of letters in the alphabet.

To decrypt the Affine cipher, you must find the inverse function. The first step is to convert the encrypted letter into an integer, use a mathematical function to convert the number to another number, and then convert the number back into a letter. The decryption process would look like the following:

$$\text{Decrypt}(x) = a^{-1}(x - b) \bmod m$$

where x is the integer represented by the letter, m is the number of letters in the alphabet, and a^{-1} is the modular multiplicative inverse of a modulo m . We will explore the multiplicative inverse of a modulo in more detail in the next chapter. If you find yourself having a hard time understanding the code, feel free to review the next chapter and come back to this cipher.

For now, you will see a working example of the Affine cipher in practice. Let's re-examine the formula for the encryption process: $E(x) = (ax + b)$. You can use the previous table to find the letter in the top row and the corresponding integer in the second row. You are left needing to know what the value is for both a and b . This is the key for the cipher. In this case, we will set the value of $a = 17$ and $b = 20$:

Plaintext	C	O	D	E	B	O	O	K
Value of x	02	14	03	04	01	14	14	10
$ax + b \% 26$	02	24	19	10	11	24	24	8
Encrypted	C	Y	T	K	L	Y	Y	I

Examine the letter E in the table. The letter maps to integer 04. You can use Python to validate the math using the following:

```
>>> print ((17 * 4 + 20) % 26)
10
```

The multiplicative inverse for the Affine cipher is $D(x) = 23 * (x - b) \% 26$. You will learn how it is derived in the next chapter. You should find that you are able to decode the cipher using the following table.

Encrypted	C	Y	T	K	L	Y	Y	I
Encrypted x	02	24	19	10	11	24	24	8
$23 * (x - b) \% 26$	02	14	03	04	01	14	14	10
Plaintext	C	O	D	E	B	O	O	K

To prove the previous encryption and decryption scheme, examine the decryption of the letter K. The letter maps to integer 04. You can use Python to validate the math using the following:

```
>>> print ((23 * (10 - 20) % 26))
4
```

This explanation should give you a bit of insight as you examine the following Python script for implementing the Affine cipher:

```
# Extended Euclidean Algorithm for finding modular inverse
# eg: modinv(7, 26) = 15
def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y

def modinv(a, m):
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None # modular inverse does not exist
    else:
        return x % m

# affine cipher encryption function
# returns the cipher text
def encrypt(text, key):
    '''
    E = (a * x + b) % 26
    '''
    return ''.join([ chr((( key[0]*(ord(t) - ord('A')) + key[1] ) % 26)
                      + ord('A')) for t in text.upper().replace(' ', '') ])

# affine cipher decryption function
# returns original text
def decrypt(cipher, key):
    '''
    D = (a^-1 * (x - b)) % 26
    '''
```

```

        return ''.join([ chr((( modinv(key[0], 26)*(ord(c) - ord('A') -
key[1]))
                        % 26) + ord('A')) for c in cipher ])

# Test the encrypt and decrypt functions
def main():
    # declaring text and key
    text = 'CODEBOOK'
    key = [17, 20]

    # calling encryption function
    encrypted_text = encrypt(text, key)

    print('Encrypted Text: {}'.format(encrypted_text ))

    # calling decryption function
    print('Decrypted Text: {}'.format
(decrypt(encrypted_text, key) ))

if __name__ == '__main__':
    main()

```

Summary

Maintaining password best practices will help mitigate against brute-force attacks on your data. If you are responsible for creating your own authentication system, it is highly recommended that you hash, salt, and/or stretch passwords . As you have learned, just encrypting the data is not enough. In fact, in the next chapter, you will learn how to use Python to crack historical ciphers. We will be building on the mathematical concepts that will enable you to determine the language and encryption scheme of several historical ciphers and determine methods toward their cryptanalysis.

6



Data Encryption Standard (DES)

Objectives

In this chapter, we discuss the Data Encryption Standard (DES), the modern symmetric-key block cipher. The following are our main objectives for this chapter:

- ☞ To review a short history of DES
- ☞ To define the basic structure of DES
- ☞ To describe the details of building elements of DES
- ☞ To describe the round keys generation process
- ☞ To analyze DES

The emphasis is on how DES uses a Feistel cipher to achieve confusion and diffusion of bits from the plaintext to the ciphertext.

6.1

INTRODUCTION

The **Data Encryption Standard (DES)** is a symmetric-key block cipher published by the **National Institute of Standards and Technology (NIST)**.

6.1.1 History

In 1973, NIST published a request for proposals for a national symmetric-key cryptosystem. A proposal from IBM, a modification of a project called Lucifer, was accepted as DES. DES was published in the *Federal Register* in March 1975 as a draft of the **Federal Information Processing Standard (FIPS)**.

After the publication, the draft was criticized severely for two reasons. First, critics questioned the small key length (only 56 bits), which could make the cipher vulnerable to brute-force attack. Second, critics were concerned about some hidden design behind the internal structure of DES. They were suspicious that some part of the structure (the S-boxes) may have some hidden trapdoor that would allow the **National Security Agency (NSA)** to decrypt the messages without the need for the key. Later IBM designers mentioned that the internal structure was designed to prevent differential cryptanalysis.

DES was finally published as FIPS 46 in the *Federal Register* in January 1977. NIST, however, defines DES as the standard for use in unclassified applications. DES has been the most widely used

symmetric-key block cipher since its publication. NIST later issued a new standard (FIPS 46-3) that recommends the use of triple DES (repeated DES cipher three times) for future applications. As we will see in Chapter 7, AES, the recent standard, is supposed to replace DES in the long run.

6.1.2 Overview

DES is a block cipher, as shown in Fig. 6.1.

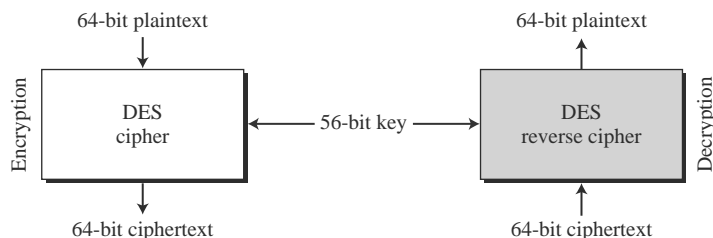


Fig. 6.1 Encryption and decryption with DES

At the encryption site, DES takes a 64-bit plaintext and creates a 64-bit ciphertext; at the decryption site, DES takes a 64-bit ciphertext and creates a 64-bit block of plaintext. The same 56-bit cipher key is used for both encryption and decryption.

6.2 DES STRUCTURE

Let us concentrate on encryption; later we will discuss decryption. The encryption process is made of two permutations (P-boxes), which we call initial and final permutations, and sixteen Feistel rounds. Each round uses a different 48-bit round key generated from the cipher key according to a predefined algorithm described later in the chapter. Figure 6.2 shows the elements of DES cipher at the encryption site.

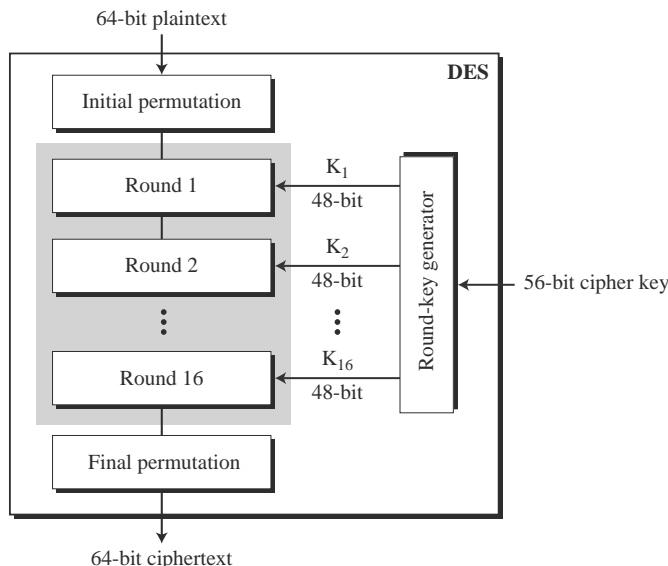


Fig. 6.2 General structure of DES

6.2.1 Initial and Final Permutations

Figure 6.3 shows the initial and final permutations (P-boxes). Each of these permutations takes a 64-bit input and permutes them according to a predefined rule. We have shown only a few input ports and the corresponding output ports. These permutations are keyless straight permutations that are the inverse of each other. For example, in the initial permutation, the 58th bit in the input becomes the first bit in the output. Similarly, in the final permutation, the first bit in the input becomes the 58th bit in the output. In other words, if the rounds between these two permutations do not exist, the 58th bit entering the initial permutation is the same as the 58th bit leaving the final permutation.

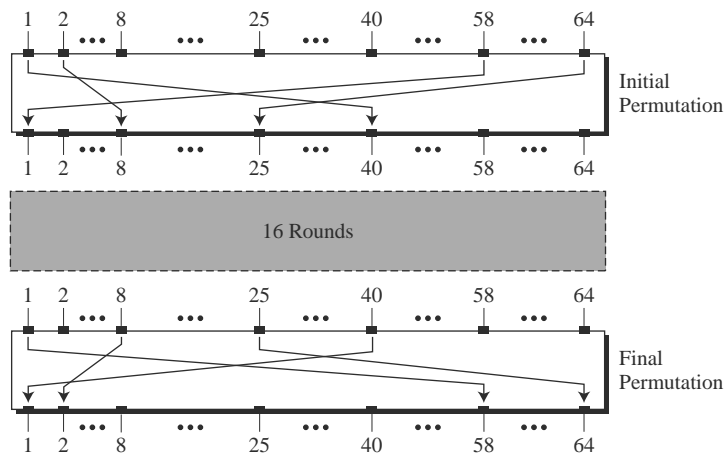


Fig. 6.3 Initial and final permutation steps in DES

The permutation rules for these P-boxes are shown in Table 6.1. Each side of the table can be thought of as a 64-element array. Note that, as with any permutation table we have discussed so far, the value of each element defines the input port number, and the order (index) of the element defines the output port number.

Table 6.1 Initial and final permutation tables

<i>Initial Permutation</i>	<i>Final Permutation</i>
58 50 42 34 26 18 10 02	40 08 48 16 56 24 64 32
60 52 44 36 28 20 12 04	39 07 47 15 55 23 63 31
62 54 46 38 30 22 14 06	38 06 46 14 54 22 62 30
64 56 48 40 32 24 16 08	37 05 45 13 53 21 61 29
57 49 41 33 25 17 09 01	36 04 44 12 52 20 60 28
59 51 43 35 27 19 11 03	35 03 43 11 51 19 59 27
61 53 45 37 29 21 13 05	34 02 42 10 50 18 58 26
63 55 47 39 31 23 15 07	33 01 41 09 49 17 57 25

These two permutations have no cryptography significance in DES. Both permutations are keyless and predetermined. The reason they are included in DES is not clear and has not been revealed by the DES designers. The guess is that DES was designed to be implemented in hardware (on chips) and that these two complex permutations may thwart a software simulation of the mechanism.

Example 6.1 Find the output of the initial permutation box when the input is given in hexadecimal as:

0x0002 0000 0000 0001

Solution The input has only two 1s (bit 15 and bit 64); the output must also have only two 1s (the nature of straight permutation). Using Table 6.1, we can find the output related to these two bits. Bit 15 in the input becomes bit 63 in the output. Bit 64 in the input becomes bit 25 in the output. So the output has only two 1s, bit 25 and bit 63. The result in hexadecimal is

0x0000 0080 0000 0002

Example 6.2 Prove that the initial and final permutations are the inverse of each other by finding the output of the final permutation if the input is

0x0000 0080 0000 0002

Solution Only bit 25 and bit 64 are 1s; the other bits are 0s. In the final permutation, bit 25 becomes bit 64 and bit 63 becomes bit 15. The result

0x0002 0000 0000 0001

The initial and final permutations are straight D-boxes that are inverses of each other and hence are permutations. They have no cryptography significance in DES.

6.2.2 Rounds

DES uses 16 rounds. Each round of DES is a Feistel cipher, as shown in Fig. 6.4.

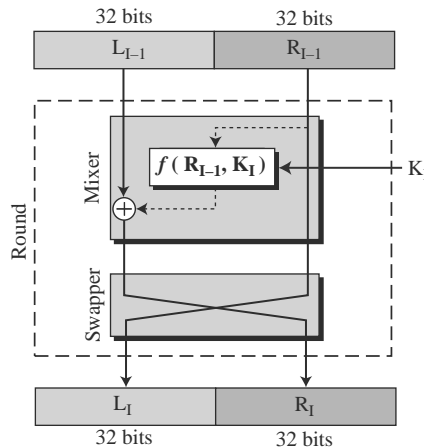


Fig. 6.4 A round in DES (encryption site)

The round takes L_{I-1} and R_{I-1} from previous round (or the initial permutation box) and creates L_I and R_I , which go to the next round (or final permutation box). As we discussed in Chapter 5, we can assume that each round has two cipher elements (mixer and swapper). Each of these elements is invertible. The swapper is obviously invertible. It swaps the left half of the text with the right half. The mixer is invertible because of the XOR operation. All noninvertible elements are collected inside the function $f(R_{I-1}, K_I)$.

DES Function

The heart of DES is the DES function. The DES function applies a 48-bit key to the rightmost 32 bits (R_{I-1}) to produce a 32-bit output. This function is made up of four sections: an expansion D-box, a whitener (that adds key), a group of S-boxes, and a straight D-box as shown in Fig. 6.5.

Expansion D-box Since R_{I-1} is a 32-bit input and K_I is a 48-bit key, we first need to expand R_{I-1} to 48 bits. R_{I-1} is divided into 8 4-bit sections. Each 4-bit section is then expanded to 6 bits. This expansion permutation follows a predetermined rule. For each section, input bits 1, 2, 3, and 4 are copied to output bits 2, 3, 4, and 5, respectively. Output bit 1 comes from bit 4 of the previous section; output bit 6 comes from bit 1 of the next section. If sections 1 and 8 can be considered adjacent sections, the same rule applies to bits 1 and 32. Fig. 6.6 shows the input and output in the expansion permutation.

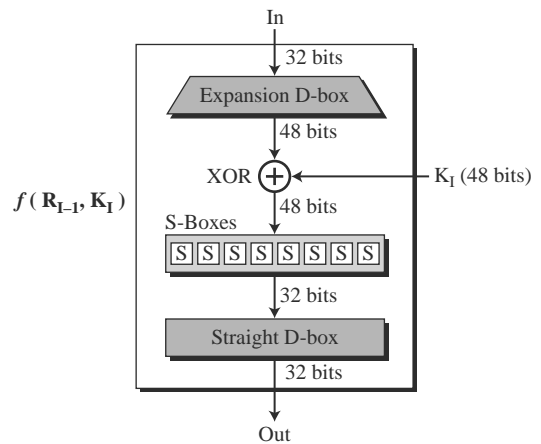


Fig. 6.5 DES function

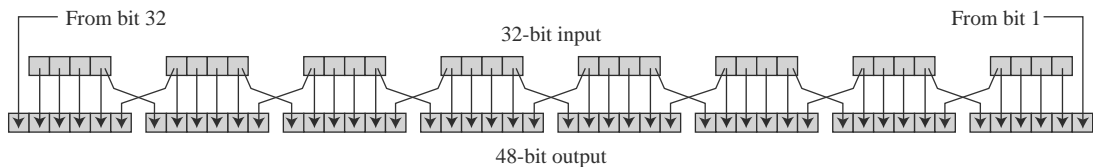


Fig. 6.6 Expansion permutation

Although the relationship between the input and output can be defined mathematically, DES uses Table 6.2 to define this D-box. Note that the number of output ports is 48, but the value range is only 1 to 32. Some of the inputs go to more than one output. For example, the value of input bit 5 becomes the value of output bits 6 and 8.

Table 6.2 Expansion D-box table

32	01	02	03	04	05
04	05	06	07	08	09
08	09	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	31	31	32	01

Whitener (XOR) After the expansion permutation, DES uses the XOR operation on the expanded right section and the round key. Note that both the right section and the key are 48-bits in length. Also note that the round key is used only in this operation.

S-Boxes The S-boxes do the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. See Fig. 6.7.

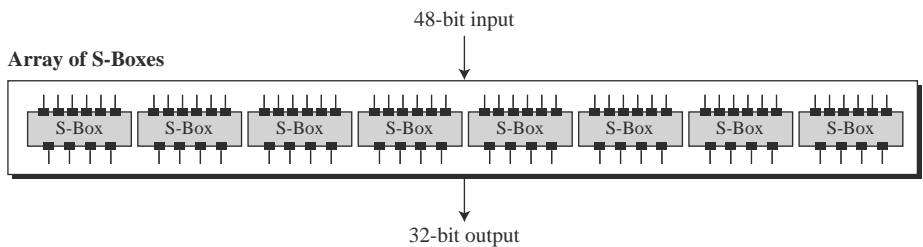


Fig. 6.7 S-boxes

The 48-bit data from the second operation is divided into eight 6-bit chunks, and each chunk is fed into a box. The result of each box is a 4-bit chunk; when these are combined the result is a 32-bit text. The substitution in each box follows a pre-determined rule based on a 4-row by 16-column table. The combination of bits 1 and 6 of the input defines one of four rows; the combination of bits 2 through 5 defines one of the sixteen columns as shown in Fig. 6.8. This will become clear in the examples.

Because each S-box has its own table, we need eight tables, as shown in Tables 6.3 to 6.10, to define the output of these boxes. The values of the inputs (row number and column number) and the values of the outputs are given as decimal numbers to save space. These need to be changed to binary.

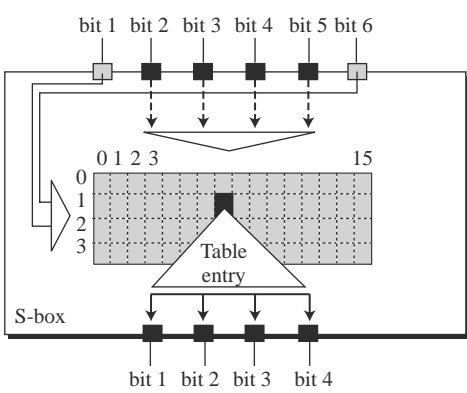


Fig. 6.8 S-box rule

Table 6.3 S-box 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	10	03	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Table 6.4 S-box 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	01	08	14	06	11	03	04	09	07	02	13	12	00	05	10
1	03	13	04	07	15	02	08	14	12	00	01	10	06	09	11	05
2	00	14	07	11	10	04	13	01	05	08	12	06	09	03	02	15
3	13	08	10	01	03	15	04	02	11	06	07	12	00	05	14	09

Table 6.5 S-box 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	00	09	14	06	03	15	05	01	13	12	07	11	04	02	08
1	13	07	00	09	03	04	06	10	02	08	05	14	12	11	15	01
2	13	06	04	09	08	15	03	00	11	01	02	12	05	10	14	07
3	01	10	13	00	06	09	08	07	04	15	14	03	11	05	02	12

Table 6.6 S-box 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	07	13	14	03	00	6	09	10	1	02	08	05	11	12	04	15
1	13	08	11	05	06	15	00	03	04	07	02	12	01	10	14	09
2	10	06	09	00	12	11	07	13	15	01	03	14	05	02	08	04
3	03	15	00	06	10	01	13	08	09	04	05	11	12	07	02	14

Table 6.7 S-box 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	12	04	01	07	10	11	06	08	05	03	15	13	00	14	09
1	14	11	02	12	04	07	13	01	05	00	15	10	03	09	08	06
2	04	02	01	11	10	13	07	08	15	09	12	05	06	03	00	14
3	11	08	12	07	01	14	02	13	06	15	00	09	10	04	05	03

Table 6.8 S-box 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	01	10	15	09	02	06	08	00	13	03	04	14	07	05	11
1	10	15	04	02	07	12	09	05	06	01	13	14	00	11	03	08
2	09	14	15	05	02	08	12	03	07	00	04	10	01	13	11	06
3	04	03	02	12	09	05	15	10	11	14	01	07	10	00	08	13

Table 6.9 S-box 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	11	2	14	15	00	08	13	03	12	09	07	05	10	06	01
1	13	00	11	07	04	09	01	10	14	03	05	12	02	15	08	06
2	01	04	11	13	12	03	07	14	10	15	06	08	00	05	09	02
3	06	11	13	08	01	04	10	07	09	05	00	15	14	02	03	12

Table 6.10 S-box 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	02	08	04	06	15	11	01	10	09	03	14	05	00	12	07
1	01	15	13	08	10	03	07	04	12	05	06	11	10	14	09	02
2	07	11	04	01	09	12	14	02	00	06	10	10	15	03	05	08
3	02	01	14	07	04	10	8	13	15	12	09	09	03	05	06	11

Example 6.3 The input to S-box 1 is 100011. What is the output?

Solution If we write the first and the sixth bits together, we get 11 in binary, which is 3 in decimal. The remaining bits are 0001 in binary, which is 1 in decimal. We look for the value in row 3, column 1, in Table 6.3 (S-box 1). The result is 12 in decimal, which in binary is 1100. So the input 100011 yields the output 1100.

Example 6.4 The input to S-box 8 is 000000. What is the output?

Solution If we write the first and the sixth bits together, we get 00 in binary, which is 0 in decimal. The remaining bits are 0000 in binary, which is 0 in decimal. We look for the value in row 0, column 0, in Table 6.10 (S-box 8). The result is 13 in decimal, which is 1101 in binary. So the input 000000 yields the output 1101.

Final Permutation The last operation in the DES function is a permutation with a 32-bit input and a 32-bit output. The input/output relationship for this operation is shown in Table 6.11 and follows the same general rule as previous tables. For example, the seventh bit of the input becomes the second bit of the output.

Table 6.11 *Straight permutation table*

16	07	20	21	29	12	28	17
01	15	23	26	05	18	31	10
02	08	24	14	32	27	03	09
19	13	30	06	22	11	04	25

6.2.3 Cipher and Reverse Cipher

Using mixers and swappers, we can create the cipher and reverse cipher, each having 16 rounds. The cipher is used at the encryption site; the reverse cipher is used at the decryption site. The whole idea is to make the cipher and the reverse cipher algorithms similar.

First Approach To achieve this goal, one approach is to make the last round (round 16) different from the others; it has only a mixer and no swapper. This is done in Figure 6.9.

Although the rounds are not aligned, the elements (mixer or swapper) are aligned. We proved in Chapter 5 that a mixer is a self-inverse; so is a swapper. The final and initial permutations are also inverses of each other. The left section of the plaintext at the encryption site, L_0 , is enciphered as L_{16} at the encryption site; L_{16} at the decryption is deciphered as L_0 at the decryption site. The situation is the same with R_0 and R_{16} .

A very important point we need to remember about the ciphers is that the round keys (K_1 to K_{16}) should be applied in the reverse order. At the encryption site, round 1 uses K_1 and round 16 uses K_{16} ; at the decryption site, round 1 uses K_{16} and round 16 uses K_1 .

In the first approach, there is no swapper in the last round.

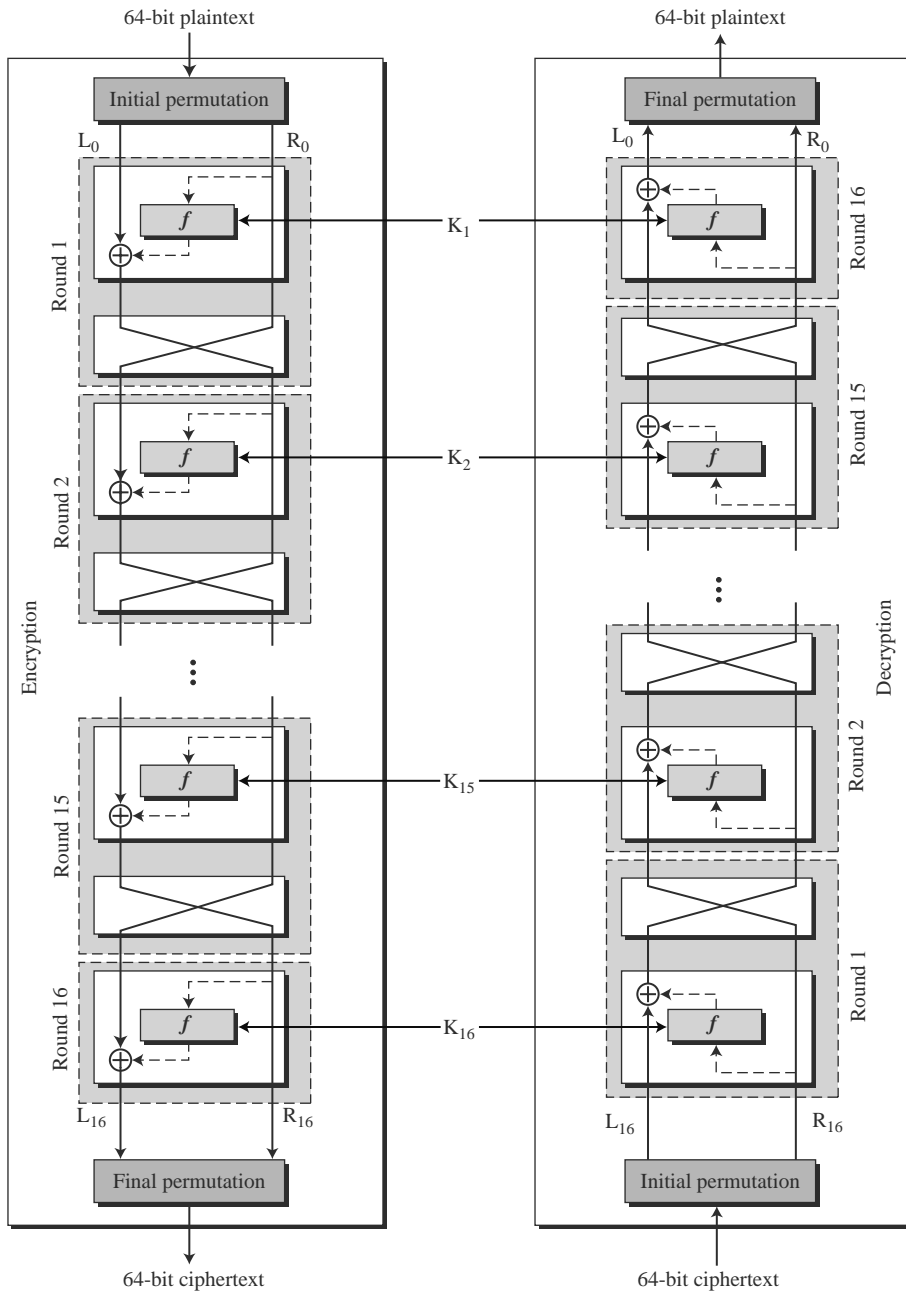


Fig. 6.9 DES cipher and reverse cipher for the first approach

Algorithm

Algorithm 6.1 gives the pseudocode for the cipher and four corresponding routines in the first approach. The codes for the rest of the routines are left as exercises.

Algorithm 6.1 **Pseudocode for DES cipher**

```

Cipher (plainBlock[64], RoundKeys[16, 48], cipherBlock[64])
{
    permute (64, 64, plainBlock, inBlock, InitialPermutationTable)
    split (64, 32, inBlock, leftBlock, rightBlock)
    for (round = 1 to 16)
    {
        mixer (leftBlock, rightBlock, RoundKeys[round])
        if (round!=16) swapper (leftBlock, rightBlock)
    }
    combine (32, 64, leftBlock, rightBlock, outBlock)
    permute (64, 64, outBlock, cipherBlock, FinalPermutationTable)
}

mixer (leftBlock[48], rightBlock[48], RoundKey[48])
{
    copy (32, rightBlock, T1)
    function (T1, RoundKey, T2)
    exclusiveOr (32, leftBlock, T2, T3)
    copy (32, T3, rightBlock)
}

swapper (leftBlock[32], rightBlock[32])
{
    copy (32, leftBlock, T)
    copy (32, rightBlock, leftBlock)
    copy (32, T, rightBlock)
}

function (inBlock[32], RoundKey[48], outBlock[32])
{
    permute (32, 48, inBlock, T1, ExpansionPermutationTable)
    exclusiveOr (48, T1, RoundKey, T2)
    substitute (T2, T3, SubstituteTables)
    permute (32, 32, T3, outBlock, StraightPermutationTable)
}

substitute (inBlock[32], outBlock[48], SubstitutionTables[8, 4, 16])
{
    for (i = 1 to 8)
    {
        row  $\leftarrow 2 \times \text{inBlock}[i \times 6 + 1] + \text{inBlock}[i \times 6 + 6]$ 
        col  $\leftarrow 8 \times \text{inBlock}[i \times 6 + 2] + 4 \times \text{inBlock}[i \times 6 + 3] +$ 
             $2 \times \text{inBlock}[i \times 6 + 4] + \text{inBlock}[i \times 6 + 5]$ 

        value = SubstitutionTables [i][row][col]

        outBlock[[i  $\times$  4 + 1]  $\leftarrow$  value / 8; value  $\leftarrow$  value mod 8
        outBlock[[i  $\times$  4 + 2]  $\leftarrow$  value / 4; value  $\leftarrow$  value mod 4
        outBlock[[i  $\times$  4 + 3]  $\leftarrow$  value / 2; value  $\leftarrow$  value mod 2
        outBlock[[i  $\times$  4 + 4]  $\leftarrow$  value
    }
}

```


Alternative Approach In the first approach, round 16 is different from other rounds; there is no swapper in this round. This is needed to make the last mixer in the cipher and the first mixer in the reverse cipher aligned. We can make all 16 rounds the same by including one swapper to the 16th round and add an extra swapper after that (two swappers cancel the effect of each other). We leave the design for this approach as an exercise.

Key Generation The **round-key generator** creates sixteen 48-bit keys out of a 56-bit cipher key. However, the cipher key is normally given as a 64-bit key in which 8 extra bits are the parity bits, which are dropped before the actual key-generation process, as shown in Fig. 6.10.

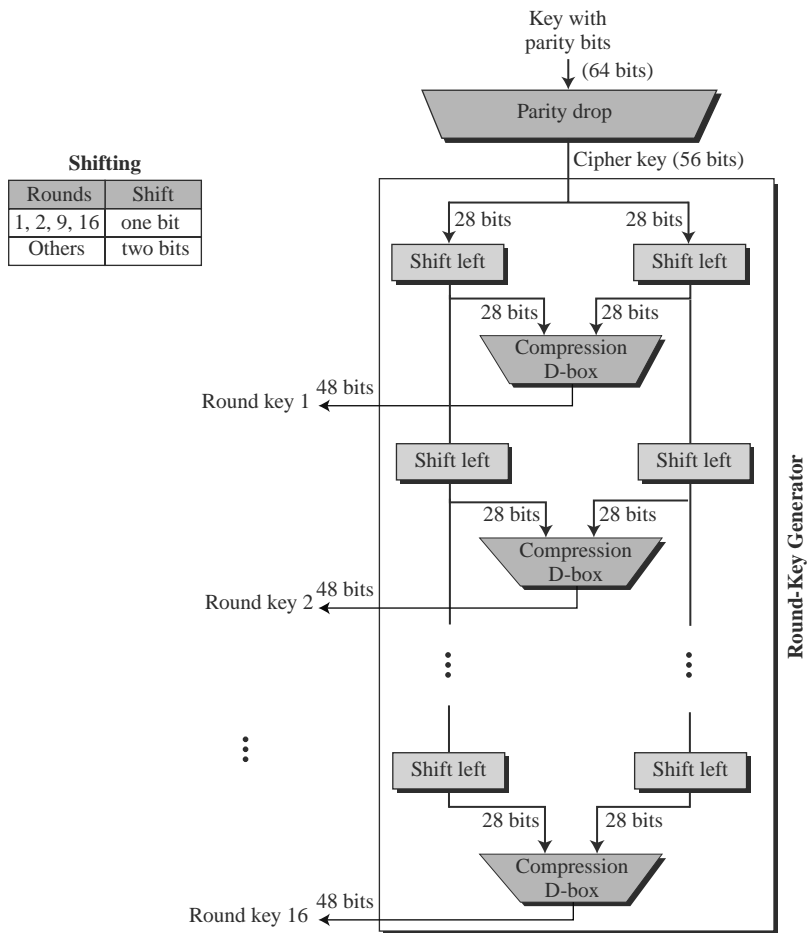


Fig. 6.10 Key generation

Parity Drop The preprocess before key expansion is a compression transposition step that we call **parity bit drop**. It drops the parity bits (bits 8, 16, 24, 32, ..., 64) from the 64-bit key and permutes the rest of the bits according to Table 6.12. The remaining 56-bit value is the actual cipher key which is used to generate round keys. The parity drop step (a compression D-box) is shown in Table 6.12.

Table 6.12 *Parity-bit drop table*

57	49	41	33	25	17	09	01
58	50	42	34	26	18	10	02
59	51	43	35	27	19	11	03
60	52	44	36	63	55	47	39
31	23	15	07	62	54	46	38
30	22	14	06	61	53	45	37
29	21	13	05	28	20	12	04

Shift Left After the straight permutation, the key is divided into two 28-bit parts. Each part is shifted left (circular shift) one or two bits. In rounds 1, 2, 9, and 16, shifting is one bit; in the other rounds, it is two bits. The two parts are then combined to form a 56-bit part. Table 6.13 shows the number of shifts for each round.

Table 6.13 *Number of bit shifts*

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bit shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Compression D-box The compression D-box changes the 58 bits to 48 bits, which are used as a key for a round. The compression step is shown in Table 6.14.

Table 6.14 *Key-compression table*

14	17	11	24	01	05	03	28
15	06	21	10	23	19	12	04
26	08	16	07	27	20	13	02
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

Algorithm Let us write a simple algorithm to create round keys from the key with parity bits. Algorithm 6.2 uses several routines from Algorithm 6.1. The new one is the shiftLeft routine, for which the code is given.

Algorithm 6.2 Algorithm for round-keys generation

```

Key_Generator (keyWithParities[64], RoundKeys[16, 48], ShiftTable[16])
{
    permute (64, 56, keyWithParities, cipherKey, ParityDropTable)
    split (56, 28, cipherKey, leftKey, rightKey)
    for (round = 1 to 16)
    {
        shiftLeft (leftKey, ShiftTable[round])
        shiftLeft (rightKey, ShiftTable[round])
    }
}

```

Algorithm 6.2 (Contd.)

```

    combine (28, 56, leftKey, rightKey, preRoundKey)
    permute (56, 48, preRoundKey, RoundKeys[round], KeyCompressionTable)
  }
}
shiftLeft (block[28], numOfShifts)
{
  for (i = 1 to numOfShifts)
  {
    T ← block[1]
    for (j = 2 to 28)
    {
      block [j-1] ← block [j]
    }
    block[28] ← T
  }
}

```

6.2.4 Examples

Before analyzing DES, let us look at some examples to see the how encryption and decryption change the value of bits in each round.

Example 6.5 We choose a random plaintext block and a random key, and determine what the ciphertext block would be (all in hexadecimal):

Plaintext: 123456ABCD132536

Key: AAB09182736CCDD

CipherText: C0B7A8D05F3A829C

Let us show the result of each round and the text created before and after the rounds. Table 6.15 first shows the result of steps before starting the round.

Table 6.15 Trace of data for Example 6.5

Plaintext: 123456ABCD132536			
After initial permutation: 14A7D67818CA18AD			
After splitting: $L_0=14A7D678$ $R_0=18CA18AD$			
Round	Left	Right	Round Key
Round 1	18CA18AD	5A78E394	194CD072DE8C
Round 2	5A78E394	4A1210F6	4568581ABCCE
Round 3	4A1210F6	B8089591	06EDA4ACF5B5
Round 4	B8089591	236779C2	DA2D032B6EE3
Round 5	236779C2	A15A4B87	69A629FEC913
Round 6	A15A4B87	2E8F9C65	C1948E87475E
Round 7	2E8F9C65	A9FC20A3	708AD2DDB3C0
Round 8	A9FC20A3	308BEE97	34F822F0C66D
Round 9	308BEE97	10AF9D37	84BB4473DCCC

Table 6.15 (Contd.)

Round 10	10AF9D37	6CA6CB20	02765708B5BF
Round 11	6CA6CB20	FF3C485F	6D5560AF7CA5
Round 12	FF3C485F	22A5963B	C2C1E96A4BF3
Round 13	22A5963B	387CCDAA	99C31397C91F
Round 14	387CCDAA	BD2DD2AB	251B8BC717D0
Round 15	BD2DD2AB	CF26B472	3330C5D9A36D
Round 16	19BA9212	CF26B472	181C5D75C66D
After combination: 19BA9212CF26B472			
Ciphertext: C0B7A8D05F3A829C		(after final permutation)	

The plaintext goes through the initial permutation to create completely different 64 bits (16 hexadecimal digit). After this step, the text is split into two halves, which we call L_0 and R_0 . The table shows the result of 16 rounds that involve mixing and swapping (except for the last round). The results of the last rounds (L_{16} and R_{16}) are combined. Finally the text goes through final permutation to create the ciphertext.

Some points are worth mentioning here. First, the right section out of each round is the same as the left section out of the next round. The reason is that the right section goes through the mixer without change, but the swapper moves it to the left section. For example, R_1 passes through the mixer of the second round without change, but then it becomes L_2 because of the swapper. The interesting point is that we do not have a swapper at the last round. That is why R_{15} becomes R_{16} instead of becoming L_{16} .

Example 6.6 Let us see how Bob, at the destination, can decipher the ciphertext received from Alice using the same key. We have shown only a few rounds to save space. Table 6.16 shows some interesting points. First, the round keys should be used in the reverse order. Compare Table 6.15 and Table 6.16. The round key for round 1 is the same as the round key for round 16. The values of L_0 and R_0 during decryption are the same as the values of L_{16} and R_{16} during encryption. This is the same with other rounds. This proves not only that the cipher and the reverse cipher are inverses of each other in the whole, but also that each round in the cipher has a corresponding reverse round in the reverse cipher. The result proves that the initial and final permutation steps are also inverses of each other.

Table 6.16 Trace of data for Example 6.6

Ciphertext: C0B7A8D05F3A829C			
After initial permutation: 19BA9212CF26B472			
After splitting: $L_0=19BA9212$ $R_0=CF26B472$			
Round	Left	Right	Round Key
Round 1	CF26B472	BD2DD2AB	181C5D75C66D
Round 2	BD2DD2AB	387CCDAA	3330C5D9A36D
...
Round 15	5A78E394	18CA18AD	4568581ABCCE
Round 16	14A7D678	18CA18AD	194CD072DE8C
After combination: 14A7D67818CA18AD			
Plaintext: 123456ABCD132536		(after final permutation)	

6.3

DES ANALYSIS

Critics have used a strong magnifier to analyze DES. Tests have been done to measure the strength of some desired properties in a block cipher. The elements of DES have gone through scrutinies to see if they have met the established criteria. We discuss some of these in this section.

6.3.1 Properties

Two desired properties of a block cipher are the avalanche effect and the completeness.

Avalanche Effect **Avalanche effect** means a small change in the plaintext (or key) should create a significant change in the ciphertext. DES has been proved to be strong with regard to this property.

Example 6.7 To check the avalanche effect in DES, let us encrypt two plaintext blocks (with the same key) that differ only in one bit and observe the differences in the number of bits in each round.

Plaintext: 0000000000000000	Key: 22234512987ABB23
Ciphertext: 4789FD476E82A5F1	
Plaintext: 0000000000000000 <u>1</u>	Key: 22234512987ABB23
Ciphertext: 0A4ED5C15A63FEA3	

Although the two plaintext blocks differ only in the rightmost bit, the ciphertext blocks differ in 29 bits. This means that changing approximately 1.5 percent of the plaintext creates a change of approximately 45 percent in the ciphertext. Table 6.17 shows the change in each round. It shows that significant changes occur as early as the third round.

Table 6.17 Number of bit differences for Example 6.7

Rounds	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bit differences	1	6	20	29	30	33	32	29	32	39	33	28	30	31	30	29

Completeness Effect **Completeness effect** means that each bit of the ciphertext needs to depend on many bits on the plaintext. The diffusion and confusion produced by D-boxes and S-boxes in DES, show a very strong completeness effect.

6.3.2 Design Criteria

The design of DES was revealed by IBM in 1994. Many tests on DES have proved that it satisfies some of the required criteria as claimed. We briefly discuss some of these design issues.

S-Boxes We have discussed the general design criteria for S-boxes in Chapter 5; we only discuss the criteria selected for DES here. The design provides confusion and diffusion of bits from each round to the next. According to this revelation and some research, we can mention several properties of S-boxes.

1. The entries of each row are permutations of values between 0 and 15.
2. S-boxes are nonlinear. In other words, the output is not an affine transformation of the input. See Chapter 5 for discussion on the linearity of S-boxes.
3. If we change a single bit in the input, two or more bits will be changed in the output.

4. If two inputs to an S-box differ only in two middle bits (bits 3 and 4), the output must differ in at least two bits. In other words, $S(x)$ and $S(x \oplus 001100)$ must differ in at least two bits where x is the input and $S(x)$ is the output.
5. If two inputs to an S-box differ in the first two bits (bits 1 and 2) and are the same in the last two bits (5 and 6), the two outputs must be different. In other words, we need to have the following relation $S(x) \neq S(x \oplus 11bc00)$, in which b and c are arbitrary bits.
6. There are only 32 6-bit input-word pairs $(x_i \text{ and } x_j)$, in which $x_i \oplus x_j \neq (000000)_2$. These 32 input pairs create 32 4-bit output-word pairs. If we create the difference between the 32 output pairs, $d = y_i \oplus y_j$, no more than 8 of these d 's should be the same.
7. A criterion similar to # 6 is applied to three S-boxes.
8. In any S-box, if a single input bit is held constant (0 or 1) and the other bits are changed randomly, the differences between the number of 0s and 1s are minimized.

D-Boxes

Between two rows of S-boxes (in two subsequent rounds), there are one straight D-box (32 to 32) and one expansion D-box (32 to 48). These two D-boxes together provide diffusion of bits. We have discussed the general design principle of D-boxes in Chapter 5. Here we discuss only the ones applied to the D-boxes used inside the DES function. The following criteria were implemented in the design of D-boxes to achieve this goal:

1. Each S-box input comes from the output of a different S-box (in the previous round).
2. No input to a given S-box comes from the output from the same box (in the previous round).
3. The four outputs from each S-box go to six different S-boxes (in the next round).
4. No two output bits from an S-box go to the same S-box (in the next round).
5. If we number the eight S-boxes, S_1, S_2, \dots, S_8 ,
 - a. An output of S_{j-2} goes to one of the first two bits of S_j (in the next round).
 - b. An output bit from S_{j-1} goes to one of the last two bits of S_j (in the next round).
 - c. An output of S_{j+1} goes to one of the two middle bits of S_j (in the next round).
6. For each S-box, the two output bits go to the first or last two bits of an S-box in the next round. The other two output bits go to the middle bits of an S-box in the next round.
7. If an output bit from S_j goes to one of the middle bits in S_k (in the next round), then an output bit from S_k cannot go to the middle bit of S_j . If we let $j = k$, this implies that none of the middle bits of an S-box can go to one of the middle bits of the same S-box in the next round.

Number of Rounds DES uses sixteen rounds of Feistel ciphers. It has been proved that after eight rounds, each ciphertext is a function of every plaintext bit and every key bit; the ciphertext is thoroughly a random function of plaintext and ciphertext. Therefore, it looks like eight rounds should be enough. However, experiments have found that DES versions with less than sixteen rounds are even more vulnerable to known-plaintext attacks than brute-force attack, which justifies the use of sixteen rounds by the designers of DES.

6.3.3 DES Weaknesses

During the last few years critics have found some weaknesses in DES.

Weaknesses in Cipher Design

We will briefly mention some weaknesses that have been found in the design of the cipher.

S-boxes At least three weaknesses are mentioned in the literature for S-boxes.

1. In S-box 4, the last three output bits can be derived in the same way as the first output bit by complementing some of the input bits.
2. Two specifically chosen inputs to an S-box array can create the same output.
3. It is possible to obtain the same output in a single round by changing bits in only three neighboring S-boxes.

D-boxes One mystery and one weakness were found in the design of D-boxes:

1. It is not clear why the designers of DES used the initial and final permutations; these have no security benefits.
2. In the expansion permutation (inside the function), the first and fourth bits of every 4-bit series are repeated.

Weakness in the Cipher Key

Several weaknesses have been found in the cipher key.

Key Size Critics believe that the most serious weakness of DES is in its key size (56 bits). To do a brute-force attack on a given ciphertext block, the adversary needs to check 2^{56} keys.

- a. With available technology, it is possible to check one million keys per second. This means that we need more than two thousand years to do brute-force attacks on DES using only a computer with one processor.
- b. If we can make a computer with one million chips (parallel processing), then we can test the whole key domain in approximately 20 hours. When DES was introduced, the cost of such a computer was over several million dollars, but the cost has dropped rapidly. A special computer was built in 1998 that found the key in 112 hours.
- c. Computer networks can simulate parallel processing. In 1977 a team of researchers used 3500 computers attached to the Internet to find a key challenged by RSA Laboratories in 120 days. The key domain was divided among all of these computers, and each computer was responsible to check the part of the domain.
- d. If 3500 networked computers can find the key in 120 days, a secret society with 42,000 members can find the key in 10 days.

The above discussion shows that DES with a cipher key of 56 bits is not safe enough to be used comfortably. We will see later in the chapter that one solution is to use triple DES (3DES) with two keys (112 bits) or triple DES with three keys (168 bits).

Weak Keys Four out of 2^{56} possible keys are called **weak keys**. A weak key is the one that, after parity drop operation (using Table 6.12), consists either of all 0s, all 1s, or half 0s and half 1s. These keys are shown in Table 6.18.

Table 6.18 Weak keys

Keys before parities drop (64 bits)	Actual key (56 bits)
0101 0101 0101 0101	0000000 0000000
1F1F 1F1F 0E0E 0E0E	0000000 FFFFFFFF
E0E0 E0E0 F1F1 F1F1	FFFFFFF 0000000
FEFE FEFE FEFE FEFE	FFFFFFF FFFFFFFF

The round keys created from any of these weak keys are the same and have the same pattern as the cipher key. For example, the sixteen round keys created from the first key is all made of 0s; the one from the second is made of half 0s and half 1s. The reason is that the key-generation algorithm first divides the cipher key into two halves. Shifting or permutation of a block does not change the block if it is made of all 0s or all 1s.

What is the disadvantage of using a weak key? If we encrypt a block with a weak key and subsequently encrypt the result with the same weak key, we get the original block. The process creates the same original block if we decrypt the block twice. In other words, each weak key is the inverse of itself $E_k(E_k(P)) = P$, as shown in Fig. 6.11.

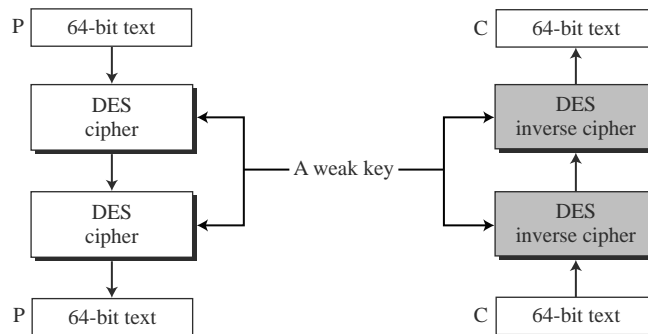


Fig. 6.11 Double encryption and decryption with a weak key

Weak keys should be avoided because the adversary can easily try them on the intercepted ciphertext. If after two decryptions the result is the same, the adversary has found the key.

Example 6.8 Let us try the first weak key in Table 6.18 to encrypt a block two times. After two encryptions with the same key the original plaintext block is created. Note that we have used the encryption algorithm two times, not one encryption followed by another decryption.

Key: 0x0101010101010101

Plaintext: 0x1234567887654321

Ciphertext: 0x814FE938589154F7

Key: 0x0101010101010101

Plaintext: 0x814FE938589154F7

Ciphertext: 0x1234567887654321

Semi-weak Keys There are six key pairs that are called **semi-weak keys**. These six pairs are shown in Table 6.19 (64-bit format before dropping the parity bits).

Table 6.19 Semi-weak keys

First key in the pair	Second key in the pair
01FE 01FE 01FE 01FE	FE01 FE01 FE01 FE01
1FE0 1FE0 0EF1 0EF1	E01F E01F F10E F10E
01E0 01E1 01F1 01F1	E001 E001 F101 F101
1FFE 1FFE 0EFE 0EFE	FE1F FE1F FE0E FE0E
011F 011F 010E 010E	1F01 1F01 0E01 0E01
E0FE E0FE F1FE F1FE	FEE0 FEE0 FEF1 FEF1

A semi-weak key creates only two different round keys and each of them is repeated eight times. In addition, the round keys created from each pair are the same with different orders. To show the idea, we have created the round keys from the first pairs as shown below:

Round key 1	9153E54319BD	6EAC1ABCE642
Round key 2	6EAC1ABCE642	9153E54319BD
Round key 3	6EAC1ABCE642	9153E54319BD
Round key 4	6EAC1ABCE642	9153E54319BD
Round key 5	6EAC1ABCE642	9153E54319BD
Round key 6	6EAC1ABCE642	9153E54319BD
Round key 7	6EAC1ABCE642	9153E54319BD
Round key 8	6EAC1ABCE642	9153E54319BD
Round key 9	9153E54319BD	6EAC1ABCE642
Round key 10	9153E54319BD	6EAC1ABCE642
Round key 11	9153E54319BD	6EAC1ABCE642
Round key 12	9153E54319BD	6EAC1ABCE642
Round key 13	9153E54319BD	6EAC1ABCE642
Round key 14	9153E54319BD	6EAC1ABCE642
Round key 15	9153E54319BD	6EAC1ABCE642
Round key 16	6EAC1ABCE642	9153E54319BD

As the list shows, there are eight equal round keys in each semi-weak key. In addition, round key 1 in the first set is the same as round key 16 in the second; round key 2 in the first is the same as round key 15 in the second; and so on. This means that the keys are inverses of each other $E_{k_2}(E_{k_1}(P)) = P$, as shown in Fig. 6.12.

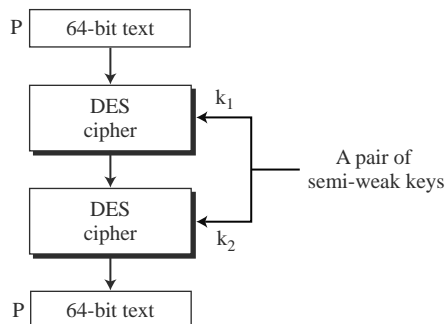


Fig. 6.12 A pair of semi-weak keys in encryption and decryption

Possible Weak Keys There are also 48 keys that are called **possible weak keys**. A possible weak key is a key that creates only four distinct round keys; in other words, the sixteen round keys are divided into four groups and each group is made of four equal round keys.

Example 6.9 What is the probability of randomly selecting a weak, a semi-weak, or a possible weak key?

Solution DES has a key domain of 2^{56} . The total number of the above keys are 64 ($4 + 12 + 48$). The probability of choosing one of these keys is 8.8×10^{-16} , almost impossible.

Key Complement In the key domain (2^{56}), definitely half of the keys are *complement* of the other half. A **key complement** can be made by inverting (changing 0 to 1 or 1 to 0) each bit in the key. Does a key complement simplify the job of the cryptanalysis? It happens that it does. Eve can use only half of the possible keys (2^{55}) to perform brute-force attack. This is because

$$C = E(K, P) \rightarrow \bar{C} = E(\bar{K}, \bar{P})$$

In other words, if we encrypt the complement of plaintext with the complement of the key, we get the complement of the ciphertext. Eve does not have to test all 2^{56} possible keys, she can test only half of them and then complement the result.

Example 6.10 Let us test the claim about the complement keys. We have used an arbitrary key and plaintext to find the corresponding ciphertext. If we have the key complement and the plaintext, we can obtain the complement of the previous ciphertext (Table 6.20).

Table 6.20 Results for Example 6.10

	<i>Original</i>	<i>Complement</i>
Key	1234123412341234	EDCBEDCBEDCBEDCB
Plaintext	12345678ABCDEF12	EDCBA987543210ED
Ciphertext	E112BE1DEFC7A367	1EED41E210385C98

Key Clustering Key clustering refers to the situation in which two or more different keys can create the same ciphertext from the same plaintext. Obviously, each pair of the semi-weak keys is a key cluster. However, no more clusters have been found for the DES. Future research may reveal some more.

6.4 SECURITY OF DES

DES, as the first important block cipher, has gone through much scrutiny. Among the attempted attacks, three are of interest: brute-force, differential cryptanalysis, and linear cryptanalysis.

6.4.1 Brute-Force Attack

We have discussed the weakness of short cipher key in DES. Combining this weakness with the key complement weakness, it is clear that DES can be broken using 2^{55} encryptions. However, today most applications use either 3DES with two keys (key size of 112) or 3DES with three keys (key size of 168). These two multiple-DES versions make DES resistant to brute-force attacks.

6.4.2 Differential Cryptanalysis

We discussed the technique of differential cryptanalysis on modern block ciphers in Chapter 5. DES is not immune to that kind of attack. However, it has been revealed that the designers of DES already knew about this type of attack and designed S-boxes and chose 16 as the number of rounds to make DES specifically resistant to this type of attack. Today, it has been shown that DES can be broken using differential cryptanalysis if we have 2^{47} chosen plaintexts or 2^{55} known plaintexts. Although this looks more efficient than a brute-force attack, finding 2^{47} chosen plaintexts or 2^{55} known plaintexts is impractical. Therefore, we can say that DES is resistant to differential cryptanalysis. It has also been shown that increasing the number of rounds to 20 require more than 2^{64} chosen plaintexts for this attack, which is impossible because the possible number of plaintext blocks in DES is only 2^{64} .

We show an example of DES differential cryptanalysis in Appendix N.

6.4.3 Linear Cryptanalysis

We discussed the technique of linear cryptanalysis on modern block ciphers in Chapter 5. Linear cryptanalysis is newer than differential cryptanalysis. DES is more vulnerable to linear cryptanalysis than to differential cryptanalysis, probably because this type of attack was not known to the designers of DES. S-boxes are not very resistant to linear cryptanalysis. It has been shown that DES can be broken using 2^{43} pairs of known plaintexts. However, from the practical point of view, finding so many pairs is very unlikely.

We show an example of DES linear cryptanalysis in Appendix N.

6.5 MULTIPLE DES—CONVENTIONAL ENCRYPTION ALGORITHMS

If a block cipher has a key size, which is small in context to the present day computation power, then a natural way out may be to perform multiple encryptions by the block cipher. As an example, consider the DES algorithm which has a key size of 56 bits, which is short in context to the modern computation capability. The threat is that such a key value can be evaluated by brute force key search. Hence two DES applications give what is known as 2-DES.

6.5.1 2-DES and Meet in the Middle Attack

Consider a message m , which is to be encrypted. The corresponding block cipher for one application of the DES applications is represented by E_k , where k is the corresponding DES key. The output of 2-DES is $c = E_{k_2}(E_{k_1}(m))$. To decrypt similarly, $m = D_{k_1}(D_{k_2}(c))$. This cipher, 2-DES should offer additional security, equivalent to both k_1 and k_2 . The cipher 2-DES obtained by the repeated application of DES is called, $2 - DES = DES \times DES$. This is called a product cipher obtained by the composition of two ciphers. Such an idea can similarly be extended to multiple ciphers.

It may be noted that such a product on the DES cipher is expected to provide additional security, because DES does not form a group under the composition operation. That is the composition (application) of two ciphers with two different keys cannot be obtained by a single application of DES with a key. Thus 2-DES is expected to provide security equivalent to $56 \times 2 = 112$ bits. However it can be shown that such a cipher can be attacked by an attack method which is called Meet-in-the-Middle attack.

6.5.2 Meet-in-the-Middle (MIM) Attack and 3-DES

Consider the cipher 2-DES as defined above. The plaintext and the ciphertext of the cipher is $P = \{0, 1\}^m$. The key space of DES is $K = \{0, 1\}^n$, the key size of the product cipher is expected to be $K_1 \times K_2$, where the key is represented as the ordered pair (k_1, k_2) , where k_1 belongs to K_1 and k_2 belongs to K_2 .

The attacker obtains l pairs of plaintexts and ciphertexts: $(p_1, c_1), \dots, (p_l, c_l)$. The key is say (K_1, K_2) but unknown to the attacker (obviously, else why will he/she be an attacker).

It is easy to prove that for all $1 \leq i \leq l$, $DES_{K_1}(p_i) = DES_{K_2}^{-1}(c_i)$. There are in total 2^{2n} keys. The probability of a key satisfying this equation for a particular value of i is 2^{-m} , as that is the block size of the cipher. Since all the i values of the plaintext, ciphertext pairs are independent, the probability of a key satisfying the above equation for all the l values of i , is 2^{-ml} .

Thus the reduced key space which satisfies the above test is expected to be $2^{2n} \cdot 2^{-ml} = 2^{2n-ml}$.

Suppose $l \geq 2n/m$, hence the number of keys passing the above test is ≤ 1 . Thus if for a key (K_1, K_2) , for all $1 \leq i \leq l$, $DES_{K_1}(p_i) = DES_{K_2}^{-1}(c_i)$ is satisfied, there is a high probability that the key is the correct key.

The attacker maintains two lists L_1 and L_2 as follows:

L_1 contains 2^n rows, where each row stores one round DES encryptions of the l plaintexts, p_1, \dots, p_l . L_2 contains also 2^n rows where each row stores one round DES decryptions of the l ciphertexts, c_1, \dots, c_l . The lists are sorted in lexicographical order with respect to the plaintexts and ciphertexts. The lists look like as shown in the Fig. 6.13.

L ₁ : Plain Texts	Key	L ₂ : Cipher Texts	Key
$DES_{K_1^1}(p_1)DES_{K_1^1}(p_2)...DES_{K_1^1}(p_l)$	K_1^1	$DES_{K_1^1}^{-1}(c_1)DES_{K_1^1}^{-1}(c_2)...DES_{K_1^1}^{-1}(c_l)$	K_2^1
$DES_{K_1^{2^n}}(p_1)DES_{K_1^{2^n}}(p_2)...DES_{K_1^{2^n}}(p_l)$	$K_1^{2^n}$	$DES_{K_1^{2^n}}^{-1}(c_1)DES_{K_1^{2^n}}^{-1}(c_2)...DES_{K_1^{2^n}}^{-1}(c_l)$	$K_2^{2^n}$

(a) (b)

Fig. 6.13

The attacker now searches the lists L_1 and L_2 and looks for a row i in L_1 which matches with a row j in L_2 . Then by the above discussion, if $l \geq 2n/m$ there is a high probability that the key is (K_1^i, K_2^j) . What is the complexity of the attack? Each table has 2^n rows. Each row has l blocks of size m bits each plus an additional n bits for the key. Hence each row of the table has $ml+n$ bits. Thus the memory required by the attacker per table is $2^n(ml+n)$, and for the two tables it is equal to $2^{n+1}(ml+n)$. The time complexity of the attack is proportional to the number of encryptions or decryptions required. This works out to $2 \cdot l \cdot 2^n = l \cdot 2^{n+l}$.

This is an example of known plaintext attack, because the plaintext is known but not chosen.

Thus we see that for typical values of DES, where $n = m = 56$, the security provided by DES against a meet-in-the-middle attack is that of 57 bits, as opposed to the expected security of 112 bits. Also it may be noted that the attack works with a high probability of success if $l \geq 2$, which means that only two plaintexts needs to be known for the attack.

Since double DES or 2-DES has a problem of this meet-in-the-middle attack, Triple-DES or 3-DES was developed. The expected security of 3-DES is 112 bits (why?).

There are in general two flavors of 3-DES. There are at least two flavors of implementation of 3-DES. The first implementation uses three keys, namely K_1, K_2, K_3 . The ciphertext of m is thus obtained by $C = DES_{K_1}[DES_{K_2}(DES_{K_3}(m))]$. The second way to implement 3-DES is using two keys, thus $C = DES_{K_1}[DES_{K_2}^{-1}(DES_{K_1}(m))]$. Thus if the keys K_1 and K_2 are the same then we obtain a single DES. This backward compatibility of the two key version of 3-DES is the reason why the middle layer is a decryption. It has otherwise no security implications.

6.6 EXAMPLES OF BLOCK CIPHERS INFLUENCED BY DES

6.6.1 The CAST Block Cipher

The CAST Block Cipher is an improvement of the DES block cipher, invented in Canada by Carlisle Adams and Stafford Tavares. The name of the cipher seems to be after the initials of the inventors. The CAST algorithm has 64 bit block size and has a key of size 64 bits.

CAST is based on the Feistel structure to implement the substitution permutation network. The authors state that they use the Feistel structure, as it is well studied and free of basic structural weaknesses.

S-Boxes of CAST CAST uses S-Boxes of dimension $m \times n$ ($m < n$). The typical dimension of the S-Boxes of CAST is 8×32 . The principle behind the construction is as follows: choose n distinct binary bent functions of length 2^m , such that the linear combinations of these functions sum to highly non-linear, Boolean functions. Bent function are Boolean functions with even input variables having the highest possible non-linearity. The resultant functions also satisfy Strict Avalanche Criteria (SAC). SAC states that S-Box output bit j should change with probability $\frac{1}{2}$ when any single input bit is changed, for all i, j . Note that the probability is computed over the set of all pairs of input vectors which differ only in bit i . Half of the bent functions have a weight of $(2^{m-1} + 2^{(m/2)-1})$ and the other have a weight of $(2^{m-1} - 2^{(m/2)-1})$.

Encryption Function The plaintext block is divided into a left half and a right half. The algorithm has 8 rounds. Each round is essentially a Feistel structure. In each round the right half is combined with the round key using a function f and then XOR-ed with the left half. The new left half after the round is the same as the right half before the round. After 8 iterations of the rounds, the left and the right half are concatenated to form the ciphertext.

The Round Function f The round function in CAST can be realized as follows. The 32 bit input can be combined with 32 bits of the round key through a function, denoted by “a” (refer Fig. 6.14).

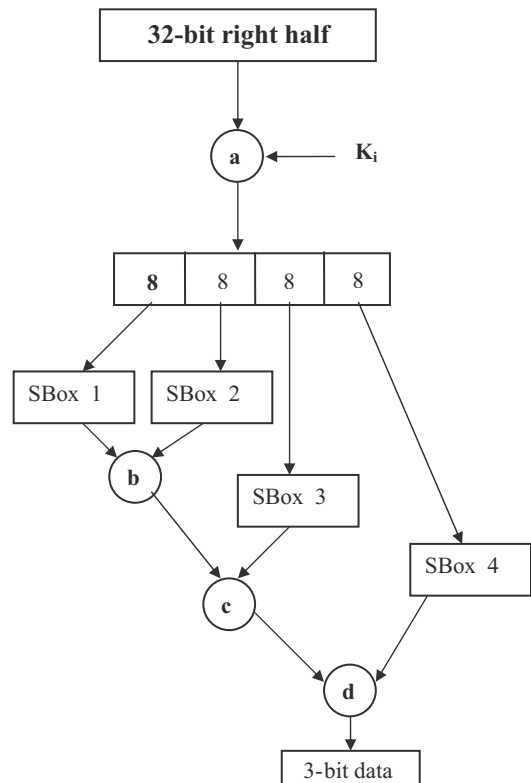


Fig. 6.14

The 32-bit data half is combined using operation “ a ” and the 32-bit result is split into 8 bit pieces. Each piece is input into a 8×32 S-Box. The output of S-Box 1 and 2 are combined using the operation “ b ”; the 32 bit output is combined with the output of S-Box 3, the output is combined in turn with the output of S-Box 4. The combining functions are denoted in the figure by “ c ” and “ d ”. A simple way would be where all the combining functions are XOR functions, however more complex operations may also be used.

Key Scheduling of CAST The key scheduling in CAST has three main components:

1. A key transformation step which converts the primary key (input key) to an intermediate key.
2. A relatively simple bit-selection algorithm mapping the primary key and the intermediate key to a form, referred as partial key bits.
3. A set of key-schedule S-Boxes which are used to create subkeys from the partial key bits.

Let, the input key be denoted by $KEY = k_1k_2k_3k_4k_5k_6k_7k_8$, where k_i is the i^{th} byte of the primary key. The key transformation step generates the intermediate key, $KEY' = k'_1k'_2k'_3k'_4k'_5k'_6k'_7k'_8$ as follows:

$$k'_1k'_2k'_3k'_4 = k_1k_2k_3k_4 \oplus S_1[k_5] \oplus S_2[k_7]$$

$$k'_5k'_6k'_7k'_8 = k_5k_6k_7k_8 \oplus S_1[k'_2] \oplus S_2[k'_4]$$

Here, S_1 and S_2 are key-schedule S-Boxes of dimension 8×32 .

Subsequently, there is a bit-selection step which operates as shown below:

$$K'_1 = k_1k_2$$

$$K'_2 = k_3k_4$$

$$K'_3 = k_5k_6$$

$$K'_4 = k_7k_8$$

$$K'_5 = k'_4k'_3$$

$$K'_6 = k'_2k'_1$$

$$K'_7 = k'_8k'_7$$

$$K'_8 = k'_6k'_5$$

The partial key bits are used to obtain the subkeys, K_i . The subkeys are 32 bits, and are obtained as follows:

$$K_i = S_1(K'_{i,1}) \oplus S_2(K'_{i,2})$$

Here, $K'_{i,j}$ is the j^{th} byte of K'_i . Thus the 8 round subkeys are obtained.

The CAST block cipher can also be implemented with 128 bits, and is referred to as CAST-128. The essential structure of the cipher is still the same as discussed above.

6.6.2 Blowfish

Blowfish is a 64-bit block cipher invented by Bruce Schneier. Blowfish was designed for fast ciphering on 32-bit microprocessors. Blowfish is also compact and has a variable key length which can be increased to 448 bits.

Blowfish is suitable for applications where the key does not change frequently like communication links or file encryptors. However for applications like packet switching or as an one-way hash function, it is unsuitable. Blowfish is not ideal for smart cards, which requires even more compact ciphers. Blowfish is faster than DES when implemented on 32-bit microprocessors. Next we discuss on the round structure of Blowfish.

Round Structure The algorithm is based on the Feistel structure and has two important parts: the round structure and the key expansion function.

There are 16 rounds, and each round are made of simple transformations which are iterated. Each round consists of a key-dependent permutation, and a key and data-dependent substitution. All the operations are additions and XORs on 32 bit words, and lookups in 4 32-bit S-Boxes. Blowfish has a P-array, P_0, P_1, \dots, P_{18} each of which are 32 bit subkeys. There are 4 S-Boxes, each of which maps an 8-bit input to 32-bits. The round structure of Blowfish is illustrated in Fig. 6.15.

The round function is also explained underneath with a pseudo-code.

Divide x into two 32-bit halves: x_L, x_R

For $i = 1$ to 16:

$$x_L = x_L \oplus P_i$$

$$x_R = F[x_L] \oplus x_R$$

Swap x_L and x_R

(undo the last swap)

$$x_R = x_R \oplus P_{17}$$

$$x_L = x_L \oplus P_{18}$$

Ciphertext = Concatenation of x_L and x_R

The function F is central to the security of the block cipher and is defined as below:

Divide x_L into four 8-bit parts: a, b, c, d

$$F[x_L] = ((S_1[a] + S_2[b] \bmod 2^{32}) \oplus S_3[c]) + S_4[d] \bmod 2^{32}$$

Key Scheduling Algorithm The subkeys are computed using the following method:

1. The P-array and then the four S-Boxes are initialized with a fixed string. The string is the hexadecimal digits of π .
2. P_1 is XOR-ed with 32 bits of the key, P_2 is XOR-ed with the next 32 bits of the key, and so on for all the bits of the key. If needed the key bits are cycled to ensure that all the P-array elements are XOR-ed.

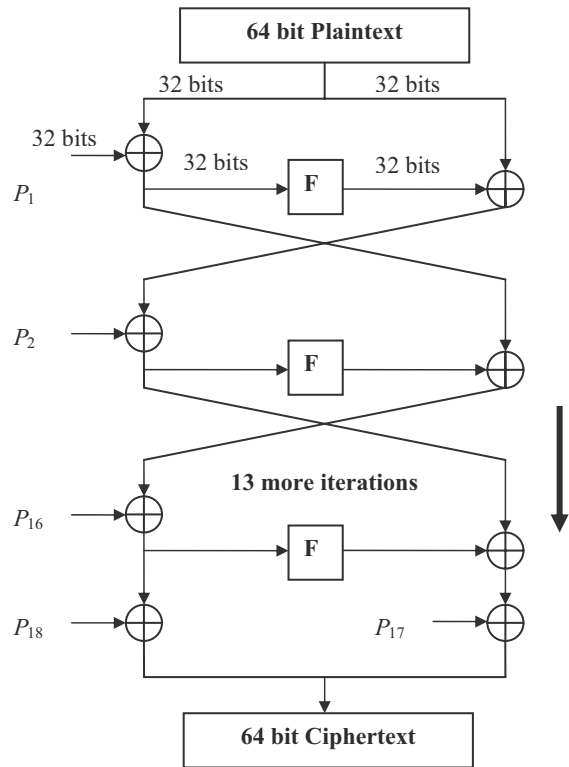


Fig. 6.15

3. An all-zero string is encrypted with the Blowfish algorithm, with the subkeys P_1 to P_{18} obtained so far in steps 1 and 2.
4. P_1 and P_2 are replaced by the 64 bit output of step 3.
5. The output of step 3 is now encrypted with the updated subkeys to replace P_3 and P_4 with the ciphertext of step 4.
6. This process is continued to replace all the P -arrays and the S -Boxes in order.

This complex key-scheduling implies that for faster operations the subkeys should be precomputed and stored in the cache for faster access.

Security analysis by Serge Vaudenay shows that for a Blowfish algorithm implemented with known S -Boxes (note that in the original cipher the S -Boxes are generated during the encryption process) and with r -rounds, a differential attack can recover the P -array with 2^{8r+1} chosen plaintexts.

6.6.3 IDEA

IDEA is another block cipher. It operates on 64 bit data blocks and the key is 128 bit long. It was invented by Xuejia Lai and James Massey, and named IDEA (International Data Encryption Algorithm) in 1990, after modifying and improving the initial proposal of the cipher based on the seminal work on Differential cryptanalysis by Biham and Shamir.

The design principle behind IDEA is the “mixing of arithmetical operations from different algebraic groups”. These arithmetical operations are easily implemented both in hardware and software.

The underlying operations are XOR, addition modulo 2^{16} , multiplication modulo $2^{10}+1$.

The cipher obtains the much needed non-linearity from the later two arithmetical operations and does not use an explicit S -Box.

Round Transformation of IDEA The 64-bit data is divided into four 16 bit blocks: X_1, X_2, X_3, X_4 . These four blocks are processed through eight rounds and transformed by the above arithmetical operations among each other and with six 16 bit subkeys. In each round the sequence of operations is as follows:

1. Multiply X_1 and the first subkey.
2. Add X_2 and the second subkey.
3. Add X_3 and the third subkey.
4. Multiply X_4 and the fourth subkey.
5. XOR the results of step 1 and 3.
6. XOR the results of step 2 and 4.
7. Multiply the results of steps 5 with the fifth subkey.
8. Add the results of steps 6 and 7.
9. Multiply the results of steps 8 with the sixth subkey.
10. Add the results of steps 7 and 9.
11. XOR the results of steps 1 and 9.
12. XOR the results of steps 3 and 9.
13. XOR the results of steps 2 and 10.
14. XOR the results of steps 4 and 10.

The outputs of steps 11, 12, 13 and 14 are stored in four words of 16 bits each, namely Y_1, Y_2, Y_3 and Y_4 . The blocks Y_2 and Y_3 are swapped, and the resultant four blocks are the output of a round of IDEA. It may be noted that the last round of IDEA does not have the swap step.

Instead the last round has the following additional transformations:

1. Multiply Y_1 and the first subkey.

2. Add Y_2 and the second subkey.
3. Add Y_3 and the third subkey.
4. Multiply Y_4 and the fourth subkey.

Finally, the ciphertext is the concatenation of the blocks Y_1 , Y_2 , Y_3 and Y_4 .

Key Scheduling of IDEA IDEA has a very simple key scheduling. It takes the 128 bit key and divides it into eight 16 bit blocks. The first six blocks are used for the first round, while the remaining two are to be used for the second round. Then the entire 128 bit key is given a rotation for 25 steps to the left and again divided into eight blocks. The first four blocks are used as the remaining subkeys for the second round, while the last four blocks are to be used for the third round. The key is then again given a left shift by 25 bits, and the other subkeys are obtained. The process is continued till the end of the algorithm.

For decryption, the subkeys are reversed and are either the multiplicative or additive inverse of the encryption subkeys. The all zero subkey is considered to represent $2^{16}-1$ for the modular multiplication operation, mod $2^{16}+1$. Thus the multiplicative inverse of 0 is itself, as -1 multiplied with -1 gives 1, the multiplicative identity in the group. Computing these keys may have its overhead, but it is a one time operation, at the beginning of the decryption process.

IDEA has resisted several cryptanalytic efforts. The designers gave argument to justify that only 4 rounds of the cipher makes it immune to differential cryptanalysis.

Joan Daemen, Rene Govaerts and Joos Vandewalle showed that the cipher had certain keys which can be easily discovered in a chosen plaintext attack.

They used the fact that the use of multiplicative subkeys with the value of 1 or -1 gives rise to linear factors in the round function. A linear factor is a linear equation in the key, input and output bits that hold for all possible input bits. The linear factors can be revealed by expressing the modulo 2 sum of LSBs of the output subblocks of an IDEA round in terms of inputs and key bits.

From the round structure of IDEA, the XOR of the LSBs of the first and second output subblock of a round are represented by y_1 and y_2 .

$$y_1 \oplus y_2 = (X_1 Z_1)_0 \oplus 1 \oplus x_3 \oplus z_3$$

If $Z_1 = (-)1 = 0 \dots 01$ (i.e if the 15 MSB bits of the Z_1 are 0), we have the following linear equation:

$$y_1 \oplus y_2 = x_1 \oplus x_3 \oplus z_1 \oplus z_3 \oplus 1$$

If the key bits are considered as constants, this linear factor can be interpreted as the propagation of knowledge from $x_1 \oplus x_3$ to $y_1 \oplus y_2$. This is indicated by $(1,0,1,0) \rightarrow (1,1,0,0)$.

Similar factors and their corresponding conditions on subkey blocks can be found for all 15 combinations of LSB output bits and are listed in the following table:

Table 6.21 Linear Factors in the round function with conditions on the subkeys

Linear Factor	Z_1	Z_4	Z_5	Z_6
$(0,0,0,1) \rightarrow (0,0,1,0)$	–	$(-)1$	–	$(-)1$
$(0,0,1,0) \rightarrow (1,0,1,1)$	–	–	$(-)1$	$(-)1$
$(0,0,1,1) \rightarrow (1,0,0,1)$	–	$(-)1$	$(-)1$	–
$(0,1,0,0) \rightarrow (0,0,0,1)$	–	–	–	$(-)1$
$(0,1,0,1) \rightarrow (0,0,1,1)$	–	$(-)1$	–	–

$(0,1,1,0) \rightarrow (1,0,1,0)$	–	–	(-1)	–
$(0,1,1,1) \rightarrow (1,0,0,0)$	–	(-1)	(-1)	(-1)
$(1,0,0,0) \rightarrow (0,1,1,1)$	(-1)	–	(-1)	(-1)
$(1,0,0,1) \rightarrow (0,1,0,1)$	(-1)	(-1)	(-1)	–
$(1,0,1,0) \rightarrow (1,1,0,0)$	(-1)	–	–	–
$(1,0,1,1) \rightarrow (1,1,1,0)$	(-1)	(-1)	–	(-1)
$(1,1,0,0) \rightarrow (0,1,1,0)$	(-1)	–	(-1)	–
$(1,1,0,1) \rightarrow (0,1,0,0)$	(-1)	(-1)	(-1)	(-1)
$(1,1,1,0) \rightarrow (1,1,0,1)$	(-1)	–	–	(-1)
$(1,1,1,1) \rightarrow (1,1,1,1)$	(-1)	(-1)	–	–

The linear factors in the rounds can be combined to obtain multiple round linear factors, by combining linear factors such that the intermediate terms cancel out. For every round they impose conditions on subkeys that can be converted into conditions on global keys, using the following table (which follows from the key scheduling algorithm of IDEA):

Table 6.22 *Derivation of encryption subkeys from the global key of size 128 bits*

r	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6
1	0–15	16–31	32–47	48–63	64–79	80–95
2	96–111	112–127	25–40	41–56	57–72	73–88
3	89–104	105–120	121–8	9–24	50–65	66–81
4	82–97	98–113	114–1	2–17	18–33	34–49
5	75–90	91–106	107–122	123–10	11–26	27–42
6	43–58	59–74	100–115	116–3	4–19	20–35
7	36–51	52–67	68–83	84–99	125–12	13–28
8	29–44	45–60	61–76	77–92	93–108	109–124
9	22–37	38–53	54–69	70–85	–	–

A possible combination for a multiple round linear factor for IDEA is shown in the underlying table. The conditions on the global key bits are also mentioned. The global key bits whose indices are there in the table should be zero. Since key bits with indices 26–28, 72–74 or 111–127 do not appear, there are 2^{23} global keys that can have this linear factor. This is called a class of weak keys as they can be detected by checking the satisfaction of linear factors by some plaintext-ciphertext combinations.

Table 6.23 *Conditions on key bits for linear factor $(1,0,1,0) \rightarrow (0,1,1,0)$*

Round	Input Term	Z_1	Z_5
1	$(1,0,1,0)$	0–14	–
2	$(1,1,0,0)$	96–110	57–71
3	$(0,1,1,0)$	–	50–64
4	$(1,0,1,0)$	82–96	–

5	(1,1,0,0)	75–89	11–25
6	(0,1,1,0)	–	4–18
7	(1,0,1,0)	36–50	–
8	(1,1,0,0)	29–44	93–107
9	(0,1,1,1)	–	–

6.7 RECOMMENDED READING

The following books and websites provide more details about subjects discussed in this chapter. The items enclosed in brackets [...] refer to the reference list at the end of the book.

Books

[Sta06], [Sti06], [Rhe03], [Sal03], [Mao04], and [TW06] discuss DES.

WebSites

The following websites give more information about topics discussed in this chapter.

<http://www.itl.nist.gov/fipspubs/fip46-2.htm>
www.nist.gov/director/prog-ofc/report01-2.pdf
www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.ps
islab.oregonstate.edu/koc/ece575/notes/dc1.pdf
homes.esat.kuleuven.be/~abiryuko/Cryptan/matsui_des
<http://nsfsecurity.pr.erau.edu/crypto/lincrypt.html>

Key Terms

avalanche effect	National Security Agency (NSA)
completeness effect	parity bit drop
Data Encryption Standard (DES)	possible weak keys
double DES (2DES)	round-key generator
Federal Information Processing (FIPS)	Standard semi-weak keys
key complement	triple DES (3DES)
meet-in-the-middle attack	triple DES with three keys
National Institute of Standards and Technology (NIST)	triple DES with two keys
	weak keys

Summary

- ★ The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST) as FIPS 46 in the *Federal Register*.
- ★ At the encryption site, DES takes a 64-bit plaintext and creates a 64-bit ciphertext. At the decryption site, DES takes a 64-bit ciphertext and creates a 64-bit block of plaintext. The same 56-bit cipher key is used for both encryption and decryption.
- ★ The encryption process is made of two permutations (P-boxes), which we call initial and final permutations,

and sixteen Feistel rounds. Each round of DES is a Feistel cipher with two elements (mixer and swapper). Each of these elements is invertible.

- ★ The heart of DES is the DES function. The DES function applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output. This function is made up of four operations: an expansion permutation, a whitener (that adds key), a group of S-boxes, and a straight permutation.
- ★ The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. However, the cipher key is normally presented as a 64-bit key in which 8 extra bits are the parity bits, which are dropped before the actual key-generation process.
- ★ DES has shown a good performance with respect to avalanche and completeness effects. Areas of weaknesses in DES include cipher design (S-boxes and P-boxes) and cipher key (length, weak keys, semi-weak keys, possible weak keys, and key complements).
- ★ Since DES is not a group, one solution to improve the security of DES is to use multiple DES (double and triple DES). Double DES is vulnerable to meet-in-the-middle attack, so triple DES with two keys or three keys is common in applications.
- ★ The design of S-boxes and number of rounds makes DES almost immune from the differential cryptanalysis. However, DES is vulnerable to linear cryptanalysis if the adversary can collect enough known plaintexts.

Practice Set

Review Questions

- 6.1 What is the block size in DES? What is the cipher key size in DES? What is the round-key size in DES?
- 6.2 What is the number of rounds in DES?
- 6.3 How many mixers and swappers are used in the first approach of making encryption and decryption inverses of each other? How many are used in the second approach?
- 6.4 How many permutations are used in a DES cipher algorithm? How many permutations are used in the round-key generator?
- 6.5 How many exclusive-or operations are used in the DES cipher?
- 6.6 Why does the DES function need an expansion permutation?
- 6.7 Why does the round-key generator need a parity drop permutation?
- 6.8 What is the difference between a weak key, a semi-weak key, and a possible weak key?
- 6.9 What is double DES? What kind of attack on double DES makes it useless?
- 6.10 What is triple DES? What is triple DES with two keys? What is triple DES with three keys?

Exercises

- 6.11 Answer the following questions about S-boxes in DES:
 - a. Show the result of passing 110111 through S-box 3.
 - b. Show the result of passing 001100 through S-box 4.
 - c. Show the result of passing 000000 through S-box 7.
 - d. Show the result of passing 111111 through S-box 2.

- 6.12** Draw the table to show the result of passing 000000 through all 8 S-boxes. Do you see a pattern in the outputs?
- 6.13** Draw the table to show the result of passing 111111 through all 8 S-boxes. Do you see a pattern in the outputs?
- 6.14** Check the third criterion for S-box 3 using the following pairs of inputs.
- a. 000000 and 000001
 - b. 111111 and 111011
- 6.15** Check the fourth design criterion for S-box 2 using the following pairs of inputs.
- a. 001100 and 110000
 - b. 110011 and 001111
- 6.16** Check the fifth design criterion for S-box 4 using the following pairs of inputs.
- a. 001100 and 110000
 - b. 110011 and 001111
- 6.17** Create 32 6-bit input pairs to check the sixth design criterion for S-box 5.
- 6.18** Show how the eight design criteria for S-box 7 are fulfilled.
- 6.19** Prove the first design criterion for P-boxes by checking the input to S-box 2 in round 2.
- 6.20** Prove the second design criterion for P-boxes by checking inputs to S-box 3 in round 4.
- 6.21** Prove the third design criterion for P-boxes by checking the output of S-box 4 in round 3.
- 6.22** Prove the fourth design criterion for P-boxes by checking the output of S-box 6 in round 12.
- 6.23** Prove the fifth design criteria for P-boxes by checking the relationship between S-boxes 3, 4, and 5 in rounds 10 and 11.
- 6.24** Prove the sixth design criteria for P-boxes by checking the destination of an arbitrary S-box.
- 6.25** Prove the seventh design criterion for P-boxes by checking the relationship between S-box 5 in round 4 and S-box 7 in round 5.
- 6.26** Redraw Fig. 6.9 using the alternate approach.
- 6.27** Prove that the reverse cipher in Fig. 6.9 is in fact the inverse of the cipher for a three-round DES. Start with a plaintext at the beginning of the cipher and prove that you can get the same plaintext at the end of the reverse cipher.
- 6.28** Carefully study the key compression permutation of Table 6.14.
- a. Which input ports are missing in the output?
 - b. Do all left 24 output bits come from all left 28 input bits?
 - c. Do all right 24 output bits come from all right 28 input bits?
- 6.29** Show the results of the following hexadecimal data
- 0110 1023 4110 1023
- after passing it through the initial permutation box.
- 6.30** Show the results of the following hexadecimal data
- AAAA BBBB CCCC DDDD
- after passing it through the final permutation box.

- 6.31 If the key with parity bit (64 bits) is 0123 ABCD 2562 1456, find the first round key.
- 6.32 Using a plaintext block of all 0s and a 56-bit key of all 0s, prove the key-complement weakness assuming that DES is made only of one round.
- 6.33 Can you devise a meet-in-the-middle attack for a triple DES?
- 6.34 Write pseudocode for the *permute* routine used in Algorithm 6.1

permute (n, m, inBlock[n], outBlock[m], permutationTable[m])

- 6.35 Write pseudocode for the *split* routine used in Algorithm 6.1

split (n, m, inBlock[n], leftBlock[m], rightBlock[m])

- 6.36 Write pseudocode for the *combine* routine used in Algorithm 6.1

combine (n, m, leftBlock[n], rightBlock[n], outBlock[m])

- 6.37 Write pseudocode for the *exclusiveOr* routine used in Algorithm 6.1

exclusiveOr (n, firstInBlock[n], secondInBlock[n], outBlock[n])

- 6.38 Change Algorithm 6.1 to represent the alternative approach.
- 6.39 Augment Algorithm 6.1 to be used for both encryption and decryption.

AES Example - Input (128 bit key and message)

Key in English: **Thats my Kung Fu** (16 ASCII characters, 1 byte each)

Translation into Hex:

T	h	a	t	s		m	y		K	u	n	g		F	u
54	68	61	74	73	20	6D	79	20	4B	75	6E	67	20	46	75

Key in Hex (128 bits): **54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75**

Plaintext in English: **Two One Nine Two** (16 ASCII characters, 1 byte each)

Translation into Hex:

T	w	o		O	n	e		N	i	n	e		T	w	o
54	77	6F	20	4F	6E	65	20	4E	69	6E	65	20	54	77	6F

Plaintext in Hex (128 bits): **54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F**

AES Example - The first Roundkey

- Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- $w[0] = (54, 68, 61, 74)$, $w[1] = (73, 20, 6D, 79)$, $w[2] = (20, 4B, 75, 6E)$, $w[3] = (67, 20, 46, 75)$
- $g(w[3])$:
 - circular byte left shift of $w[3]$: $(20, 46, 75, 67)$
 - Byte Substitution (S-Box): $(B7, 5A, 9D, 85)$
 - Adding round constant $(01, 00, 00, 00)$ gives: $g(w[3]) = (B6, 5A, 9D, 85)$
- $w[4] = w[0] \oplus g(w[3]) = (E2, 32, FC, F1)$:

0101 0100	0110 1000	0110 0001	0111 0100
1011 0110	0101 1010	1001 1101	1000 0101
1110 0010	0011 0010	1111 1100	1111 0001
E2	32	FC	F1

- $w[5] = w[4] \oplus w[1] = (91, 12, 91, 88)$, $w[6] = w[5] \oplus w[2] = (B1, 59, E4, E6)$,
 $w[7] = w[6] \oplus w[3] = (D6, 79, A2, 93)$
- first roundkey: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93

AES Example - All RoundKeys

- Round 0: 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- Round 1: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93
- Round 2: 56 08 20 07 C7 1A B1 8F 76 43 55 69 A0 3A F7 FA
- Round 3: D2 60 0D E7 15 7A BC 68 63 39 E9 01 C3 03 1E FB
- Round 4: A1 12 02 C9 B4 68 BE A1 D7 51 57 A0 14 52 49 5B
- Round 5: B1 29 3B 33 05 41 85 92 D2 10 D2 32 C6 42 9B 69
- Round 6: BD 3D C2 B7 B8 7C 47 15 6A 6C 95 27 AC 2E 0E 4E
- Round 7: CC 96 ED 16 74 EA AA 03 1E 86 3F 24 B2 A8 31 6A
- Round 8: 8E 51 EF 21 FA BB 45 22 E4 3D 7A 06 56 95 4B 6C
- Round 9: BF E2 BF 90 45 59 FA B2 A1 64 80 B4 F7 F1 CB D8
- Round 10: 28 FD DE F8 6D A4 24 4A CC C0 A4 FE 3B 31 6F 26

AES Example - Add Roundkey, Round 0

- State Matrix and Roundkey No.0 Matrix:

$$\begin{pmatrix} 54 & 4F & 4E & 20 \\ 77 & 6E & 69 & 54 \\ 6F & 65 & 6E & 77 \\ 20 & 20 & 65 & 6F \end{pmatrix} \quad \begin{pmatrix} 54 & 73 & 20 & 67 \\ 68 & 20 & 4B & 20 \\ 61 & 6D & 75 & 46 \\ 74 & 79 & 6E & 75 \end{pmatrix}$$

- XOR the corresponding entries, e.g., $69 \oplus 4B = 22$

$$\begin{array}{r} 0110 \ 1001 \\ 0100 \ 1011 \\ \hline 0010 \ 0010 \end{array}$$

- the new State Matrix is

$$\begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix}$$

AES Example - Round 1, Substitution Bytes

- current State Matrix is

$$\begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix}$$

- substitute each entry (byte) of current state matrix by corresponding entry in AES S-Box
- for instance: byte 6E is substituted by entry of S-Box in row 6 and column E, i.e., by 9F
- this leads to new State Matrix

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix}$$

- this non-linear layer is for resistance to differential and linear cryptanalysis attacks

AES Example - Round 1, Shift Row

- the current State Matrix is

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix}$$

- four rows are shifted cyclically to the left by offsets of 0,1,2, and 3
- the new State Matrix is

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix}$$

- this linear mixing step causes diffusion of the bits over multiple rounds

AES Example - Round 1, Mix Column

- Mix Column multiplies fixed matrix against current State Matrix:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix} = \begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix}$$

- entry BA is result of $(02 \bullet 63) \oplus (03 \bullet 2F) \oplus (01 \bullet AF) \oplus (01 \bullet A2)$:
 - $02 \bullet 63 = 00000010 \bullet 01100011 = 11000110$
 - $03 \bullet 2F = (02 \bullet 2F) \oplus 2F = (00000010 \bullet 00101111) \oplus 00101111 = 01110001$
 - $01 \bullet AF = AF = 10101111$ and $01 \bullet A2 = A2 = 10100010$
 - hence

$$\begin{array}{r} 11000110 \\ 01110001 \\ 10101111 \\ 10100010 \\ \hline 10111010 \end{array}$$

AES Example - Add Roundkey, Round 1

- State Matrix and Roundkey No.1 Matrix:

$$\begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix} \quad \begin{pmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{pmatrix}$$

- XOR yields new State Matrix

$$\begin{pmatrix} 58 & 15 & 59 & CD \\ 47 & B6 & D4 & 39 \\ 08 & 1C & E2 & DF \\ 8B & BA & E8 & CE \end{pmatrix}$$

- AES output after Round 1: 58 47 08 8B 15 B6 1C BA 59 D4 E2 E8 CD 39 DF CE

AES Example - Round 2

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 6A & 59 & CB & BD \\ A0 & 4E & 48 & 12 \\ 30 & 9C & 98 & 9E \\ 3D & F4 & 9B & 8B \end{pmatrix} \qquad \begin{pmatrix} 6A & 59 & CB & BD \\ 4E & 48 & 12 & A0 \\ 98 & 9E & 30 & 9B \\ 8B & 3D & F4 & 9B \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 15 & C9 & 7F & 9D \\ CE & 4D & 4B & C2 \\ 89 & 71 & BE & 88 \\ 65 & 47 & 97 & CD \end{pmatrix} \qquad \begin{pmatrix} 43 & 0E & 09 & 3D \\ C6 & 57 & 08 & F8 \\ A9 & C0 & EB & 7F \\ 62 & C8 & FE & 37 \end{pmatrix}$$

AES Example - Round 3

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 1A & AB & 01 & 27 \\ B4 & 5B & 30 & 41 \\ D3 & BA & E9 & D2 \\ AA & E8 & BB & 9A \end{pmatrix} \qquad \begin{pmatrix} 1A & AB & 01 & 27 \\ 5B & 30 & 41 & B4 \\ E9 & D2 & D3 & BA \\ A9 & AA & E8 & BB \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} AA & 65 & FA & 88 \\ 16 & 0C & 05 & 3A \\ 3D & C1 & DE & 2A \\ B3 & 4B & 5A & 0A \end{pmatrix} \qquad \begin{pmatrix} 78 & 70 & 99 & 4B \\ 76 & 76 & 3C & 39 \\ 30 & 7D & 37 & 34 \\ 54 & 23 & 5B & F1 \end{pmatrix}$$

AES Example - Round 4

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} BC & 51 & EE & B3 \\ 38 & 38 & EB & 12 \\ 04 & FF & 9A & 18 \\ 20 & 26 & 39 & A1 \end{pmatrix} \qquad \begin{pmatrix} BC & 51 & EE & B3 \\ 38 & EB & 12 & 38 \\ 9A & 18 & 04 & FF \\ A1 & 20 & 26 & 39 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 10 & BC & D3 & F3 \\ D8 & 94 & E0 & E0 \\ 53 & EA & 9E & 25 \\ 24 & 40 & 73 & 7B \end{pmatrix} \qquad \begin{pmatrix} B1 & 08 & 04 & E7 \\ CA & FC & B1 & B2 \\ 51 & 54 & C9 & 6C \\ ED & E1 & D3 & 20 \end{pmatrix}$$

AES Example - Round 5

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} C8 & 30 & F2 & 94 \\ 74 & B0 & C8 & 37 \\ D1 & 20 & DD & 50 \\ 55 & F8 & 66 & B7 \end{pmatrix} \qquad \begin{pmatrix} C8 & 30 & F2 & 94 \\ B0 & C8 & 37 & 74 \\ DD & 50 & D1 & 20 \\ B7 & 55 & F8 & 66 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 2A & 26 & 8F & E9 \\ 78 & 1E & 0C & 7A \\ 1B & A7 & 6F & 0A \\ 5B & 62 & 00 & 3F \end{pmatrix} \qquad \begin{pmatrix} 9B & 23 & 5D & 2F \\ 51 & 5F & 1C & 38 \\ 20 & 22 & BD & 91 \\ 68 & F0 & 32 & 56 \end{pmatrix}$$

AES Example - Round 6

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 14 & 26 & 4C & 15 \\ D1 & CF & 9C & 07 \\ B7 & 93 & 7A & 81 \\ 45 & 8C & 23 & B1 \end{pmatrix} \qquad \begin{pmatrix} 14 & 26 & 4C & 15 \\ CF & 9C & 07 & D1 \\ 7A & 81 & B7 & 93 \\ B1 & 45 & 8C & 23 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} A9 & 37 & AA & F2 \\ AE & D8 & 0C & 21 \\ E7 & 6C & B1 & 9C \\ F0 & FD & 67 & 3B \end{pmatrix} \qquad \begin{pmatrix} 14 & 8F & C0 & 5E \\ 93 & A4 & 60 & 0F \\ 25 & 2B & 24 & 92 \\ 77 & E8 & 40 & 75 \end{pmatrix}$$

AES Example - Round 7

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} FA & 73 & BA & 58 \\ DC & 49 & D0 & 76 \\ 3F & F1 & 36 & 4F \\ F5 & 9B & 09 & 9D \end{pmatrix} \qquad \begin{pmatrix} FA & 73 & BA & 58 \\ 49 & D0 & 76 & DC \\ 36 & 4F & 3F & F1 \\ 9D & F5 & 9B & 09 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 9F & 37 & 51 & 37 \\ AF & EC & 8C & FA \\ 63 & 39 & 04 & 66 \\ 4B & FB & B1 & D7 \end{pmatrix} \qquad \begin{pmatrix} 53 & 43 & 4F & 85 \\ 39 & 06 & 0A & 52 \\ 8E & 93 & 3B & 57 \\ 5D & F8 & 95 & BD \end{pmatrix}$$

AES Example - Round 8

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} ED & 1A & 84 & 97 \\ 12 & 6F & 67 & 00 \\ 19 & DC & E2 & 5B \\ 4C & 41 & 2A & 7A \end{pmatrix} \qquad \begin{pmatrix} ED & 1A & 84 & 97 \\ 6F & 67 & 00 & 12 \\ E2 & 5B & 19 & DC \\ 7A & 4C & 41 & 2A \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} E8 & 8A & 4B & F5 \\ 74 & 75 & EE & E6 \\ D3 & 1F & 75 & 58 \\ 55 & 8A & 0C & 38 \end{pmatrix} \qquad \begin{pmatrix} 66 & 70 & AF & A3 \\ 25 & CE & D3 & 73 \\ 3C & 5A & 0F & 13 \\ 74 & A8 & 0A & 54 \end{pmatrix}$$

AES Example - Round 9

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 33 & 51 & 79 & 0A \\ 3F & 8B & 66 & 8F \\ EB & BE & 76 & 7D \\ 92 & C2 & 67 & 20 \end{pmatrix}$$

$$\begin{pmatrix} 33 & 51 & 79 & 0A \\ 8B & 66 & 8F & 3F \\ 76 & 7D & EB & BE \\ 20 & 92 & C2 & 67 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} B6 & E7 & 51 & 8C \\ 84 & 88 & 98 & CA \\ 34 & 60 & 66 & FB \\ E8 & D7 & 70 & 51 \end{pmatrix}$$

$$\begin{pmatrix} 09 & A2 & F0 & 7B \\ 66 & D1 & FC & 3B \\ 8B & 9A & E6 & 30 \\ 78 & 65 & C4 & 89 \end{pmatrix}$$

AES Example - Round 10

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 01 & 3A & 8C & 21 \\ 33 & 3E & B0 & E2 \\ 3D & B8 & 8E & 04 \\ BC & 4D & 1C & A7 \end{pmatrix} \quad \begin{pmatrix} 01 & 3A & 8C & 21 \\ 3E & B0 & E2 & 33 \\ 8E & 04 & 3D & B8 \\ A7 & BC & 4D & 1C \end{pmatrix}$$

- after Roundkey (Attention: no Mix columns in last round):

$$\begin{pmatrix} 29 & 57 & 40 & 1A \\ C3 & 14 & 22 & 02 \\ 50 & 20 & 99 & D7 \\ 5F & F6 & B3 & 3A \end{pmatrix}$$

- ciphertext: 29 C3 50 5F 57 14 20 F6 40 22 99 B3 1A 02 D7 3A

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL