Made By Rohan Phutke
Dated 16 September 2024

Book

Cryptographic algorithms and protocols can be grouped into four main areas:
■ Symmetric encryption: Used to conceal the contents of blocks or streams of data of any size, including messages, files, encryption keys, and passwords.
■ Asymmetric encryption: Used to conceal small blocks of data, such as encryption keys and hash function values, which are used in digital signatures.
■ Data integrity algorithms: Used to protect blocks of data, such as messages, from alteration.
■ Authentication protocols: These are schemes based on the use of cryptographic algorithms designed to authenticate the identity of entit

Computer Security: The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)

■ Confidentiality: This term covers two related concepts:
Data confidentiality: Assures that private or confidential information is not made available or disclosed to unauthorized individuals.
Privacy: Assures that individuals control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed.

■ Integrity: This term covers two related concepts:
Data integrity: Assures that information (both stored and in transmitted packets) and programs are changed only in a specified and authorized manner.
System integrity: Assures that a system performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system.

■Availability: Assures that systems work promptly and service is not denied to authorized users.

These three concepts form what is often referred to as the **CIA triad**. The three concepts embody the fundamental security objectives for both data and for information and computing services.

Although the use of the CIA triad to define security objectives is well established, some in the security field feel that additional concepts are needed to present a complete picture

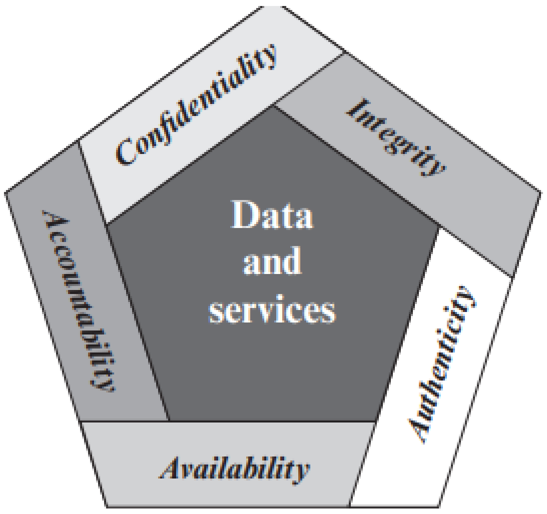


Figure 1.1 Essential Network and Computer Security Requirements

■ **Authenticity**: The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source.
■ **Accountability**: The security goal that generates the requirement for actions of an entity to be traced uniquely to that entity. This supports nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and afteraction recovery and legal action. Because truly secure systems are not yet an achievable goal, we must be able to trace a security breach to a responsible party. Systems must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes.

Computer and network security is essentially a battle of wits between a perpetrator who tries to find holes and the designer or administrator who tries to close them. The great advantage that the attacker has is that he or she need only find a single weakness, while the designer must find and eliminate all weaknesses to achieve perfect security

---

The **OSI security architecture** focuses on security attacks, mechanisms, and services. These can be defined briefly as
■ Security attack: Any action that compromises the security of information owned by an organization.
■ Security mechanism: A process (or a device incorporating such a process) that is designed to

detect, prevent, or recover from a security attack.
■ Security service: A processing or communication service that enhances the security of the data processing systems and the information transfers of an organization. The services are intended to counter security attacks, and they make use of one or more security mechanisms to provide the service

---

Some **Number Theory** (Essential)

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.

Subsequently, we will need some simple properties of divisibility for integers, which are as follows:
■ If a | 1, then a = +-1.
■ If a | b and b | a, then a = +-b.
■ Any b ≠ 0 divides 0.
■ If a | b and b | c, then a | c:

■ If b | g and b | h, then b |(mg + nh) for arbitrary integers m and n. To see this last point, note that
■ If b | g, then g is of the form g = b *g1 for some integer g1.*
■ *If b | h, then h is of the form h = b* h1 for some integer h1.
So mg + nh = mbg1 + nbh1 = b * (mg1 + nh1) and therefore b divides mg + nh

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 0 <= r < n; q = floor(a/n); (2.1)

where floor(x) is the largest integer less than or equal to x
. Equation (2.1) is referred to as the division algorithm.

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 … a and (q + 1)n 7 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.

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. (coprime)

gcd(a, b) = max[k, such that k | a and k | b]
gcd(a, 0) = | a |.
for coprimes gcd(a,b) = 1;

Euclid Algo for finding GCD

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 >= b > 0.
2. Dividing a by b and applying the division algorithm, we can state: a = q1b + r1
0 <= r1 < b (2.2)
3. First consider the case in which r1 = 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 r1 ≠ 0. For this case, we can state that d | r1. This is due to the basic properties of divisibility: the relations d | a and d | b together imply that d | (a - q1b), which is the same as d | r1.

Before proceeding with the Euclidian algorithm, we need to answer the question: What is the gcd(b, r1)? We know that d | b and d | r1. Now take any arbitrary integer c that divides both b and r1. Therefore, c | (q1b + r1) = a. Because c divides both a and b, we must have c <= d, which is the greatest common divisor of a and b. Therefore d = gcd(b, r1)

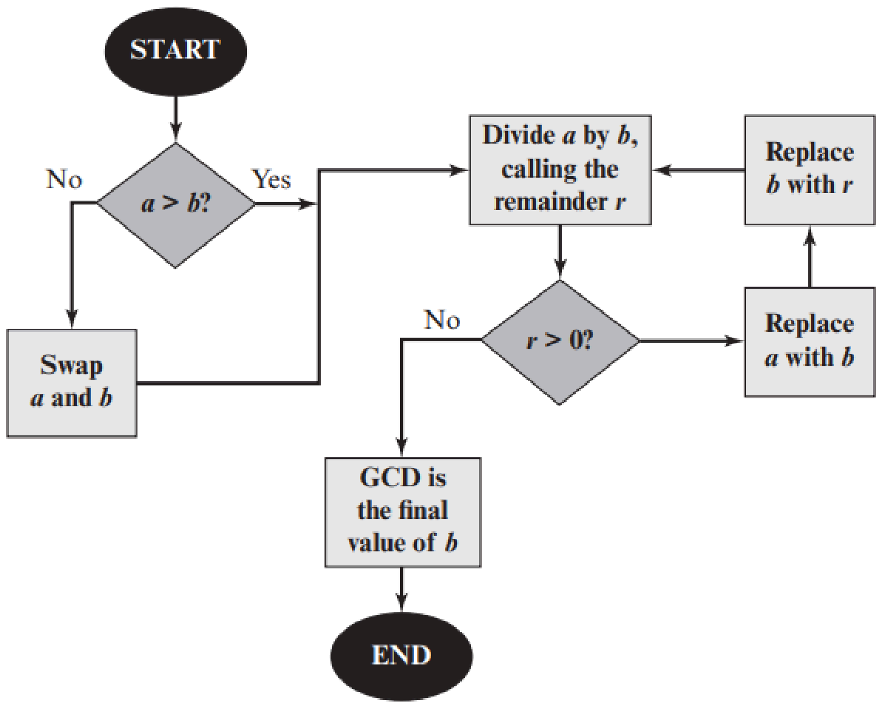


Figure 2.2 Euclidean Algorithm

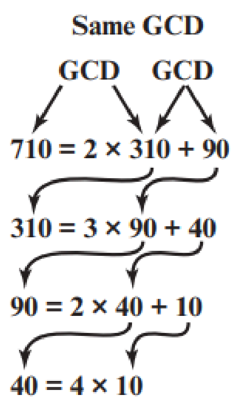


Figure 2.3 Euclidean Algorithm Example: gcd(710, 310)

Same as recursion algorithm for GCD we learned basically called as Euclidean Algorithm for calculating GCD.

The **Euclidean algorithm** is an efficient method for computing the **greatest common divisor (GCD)** of two integers. Understanding its **worst-case time complexity** is essential for evaluating its performance, especially with large inputs.

Worst-Case Time Complexity

- **Time Complexity:** (O(\log \min(a, b))) steps

Explanation

Number of Steps:

- **Worst-Case Scenario:** The algorithm takes the maximum number of steps when the input numbers are consecutive **Fibonacci numbers**. This is because Fibonacci numbers grow exponentially, ensuring that each step only reduces the problem size minimally.
- **Example:**
  - Consider two consecutive Fibonacci numbers, say (F*{n}) and (F*{n-1}).

    - Each step of the algorithm will reduce the problem to (\text{GCD}(F*{n-1}, F*{n-2})), and so on, until it reaches (\text{GCD}(F*{2}, F*{1})).

Logarithmic Relationship:

- The number of steps required is proportional to the number of digits (in any fixed base) of the smaller input number.
- Since Fibonacci numbers grow exponentially, the number of steps (k) relates to the size of the input (b) as:
  [
  k \approx \log_{\phi} b
  ]
  where (\phi) (the golden ratio) is approximately 1.618.
- Simplifying, this gives a **time complexity of (O(\log b))**, where (b) is the smaller of the two input numbers.

Bit Complexity:

- If considering the number of **bit operations** (where (n) is the number of bits in the smaller number), the Euclidean algorithm operates in **(O(n^2))** time in the worst case. However, more advanced versions of the algorithm can achieve better performance.

Summary

- **Iterations:** The Euclidean algorithm completes in a number of steps proportional to the logarithm of the smaller input number, specifically (O(\log \min(a, b))) steps.
- **Bit Operations:** When considering bit-level operations, the worst-case time complexity is (O(n^2)), where (n) is the number of bits in the smaller number.

Practical Implications

- **Efficiency:** The logarithmic time complexity ensures that the Euclidean algorithm remains efficient even for very large integers.
- **Applications:** Beyond computing GCDs, the algorithm is foundational in areas like cryptography, number theory, and algorithms that require modular inverses.

Example

Let's illustrate the worst-case scenario with consecutive Fibonacci numbers:

- **Inputs:** (a = 34) and (b = 21) (where both are Fibonacci numbers)

  Steps:
  1. (\text{GCD}(34, 21)) → (\text{GCD}(21, 13))
  2. (\text{GCD}(21, 13)) → (\text{GCD}(13, 8))
  3. (\text{GCD}(13, 8)) → (\text{GCD}(8, 5))
  4. (\text{GCD}(8, 5)) → (\text{GCD}(5, 3))
  5. (\text{GCD}(5, 3)) → (\text{GCD}(3, 2))
  6. (\text{GCD}(3, 2)) → (\text{GCD}(2, 1))
  7. (\text{GCD}(2, 1)) → (\text{GCD}(1, 0)) → **GCD is 1**
- **Total Steps:** 7, which aligns with (\log_{\phi} 21 \approx 7)

Conclusion

The Euclidean algorithm is highly efficient for computing the GCD, with a worst-case time complexity that grows logarithmically relative to the size of the smaller input number. This makes it suitable for applications requiring rapid GCD computations, even with large integers.

Two integers a and b are said to be congruent modulo n, if (a mod n) = (b mod n). This is written as a === b * ((mod n)^2).

Properties of Congruences Congruences have the following properties:

1. a === b (mod n) if n | (a - b).
2. a = b (mod n) implies b = a (mod n).
3. a = b (mod n) and b = c (mod n) imply a === c (mod n).

Properties of Modular Arithmetic

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

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

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 (mod n). To be more precise, each integer in Zn represents a residue class. We can label the residue classes (mod n) as [0], [1], [2], c , [n - 1], where [r] = {a: a is an integer, a === r (mod n)}

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

The residue classes (mod 4) are

$$[0] = \{\ldots, -16, -12, -8, -4, 0, 4, 8, 12, 16, \ldots\}$$
$$[1] = \{\ldots, -15, -11, -7, -3, 1, 5, 9, 13, 17, \ldots\}$$
$$[2] = \{\ldots, -14, -10, -6, -2, 2, 6, 10, 14, 18, \ldots\}$$
$$[3] = \{\ldots, -13, -9, -5, -1, 3, 7, 11, 15, 19, \ldots\}$$

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:

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

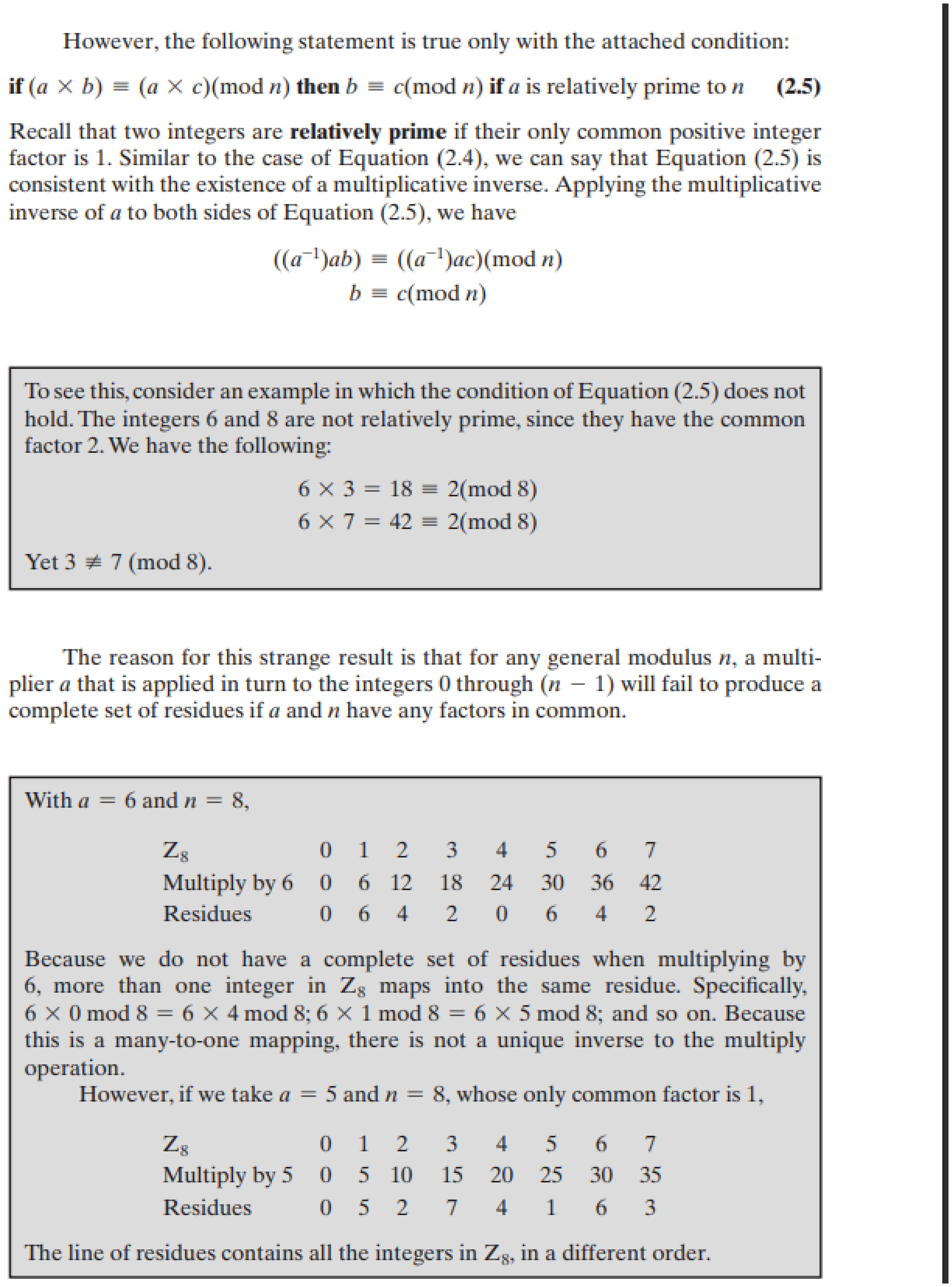
$$\boxed{(5 + 23) \equiv (5 + 7) \pmod{8};\ 23 \equiv 7 \pmod{8}}$$

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

$$((-a) + a + b) \equiv ((-a) + a + c) \pmod{n}$$
$$b \equiv c \pmod{n}$$

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 $w + z \equiv 0 \bmod n$



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

if $(a \times b) \equiv (a \times c)(\text{mod } n)$ **then** $b \equiv c(\text{mod } n)$ **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

$$((a^{-1})ab) \equiv ((a^{-1})ac)(\text{mod } n)$$
$$b \equiv c(\text{mod } n)$$

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:

$$6 \times 3 = 18 \equiv 2(\text{mod } 8)$$
$$6 \times 7 = 42 \equiv 2(\text{mod } 8)$$

Yet $3 \not\equiv 7$ (mod 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, $6 \times 0 \text{ mod } 8 = 6 \times 4 \text{ mod } 8$; $6 \times 1 \text{ mod } 8 = 6 \times 5 \text{ mod } 8$; 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.

VERY IMP

In general, an integer has a multiplicative inverse in Zn 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 Z8; but 2, 4, and 6 do not.

Fermat’s Theorem

Fermat’s theorem states the following: If p is prime and a is a positive integer not divisible by p, then a^p-1 === 1 (mod p)

An alternative form of Fermat’s theorem is also useful: If p is prime and a is a positive integer, then a^p === a(mod p)

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

Euler’s Theorem

Euler’s theorem states that for every a and n that are relatively prime:
a^ phi(n) === 1(mod n)
where phi(n) is the eulers totient function.

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

$$a^{\phi(n)+1} \equiv a(\text{mod } n) \qquad \textbf{(2.13)}$$

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.

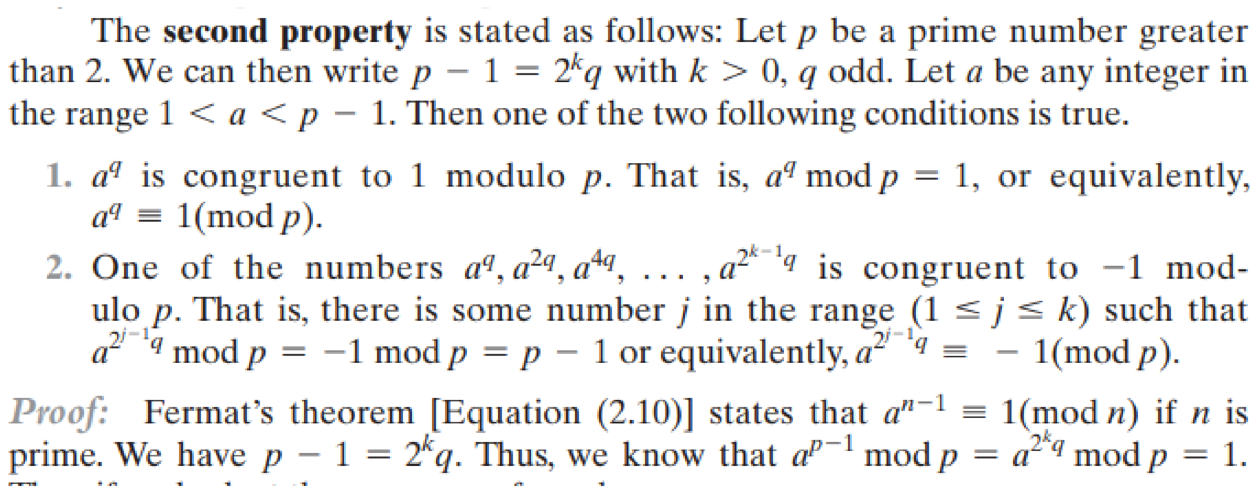
For many cryptographic algorithms, it is necessary to select one or more very large prime numbers at random. Thus, we are faced with the task of determining whether a given large number is prime. There is no simple yet efficient means of accomplishing this task. In this section, we present one attractive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

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 >= 3 can be expressed as n - 1 = (2^k)*q with k > 0, q 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 mod p = 1 if and only if either (a mod p = 1) or (a mod p) = (-1 mod p) = (p - 1). By the rules of modular arithmetic (a mod p) (a mod p) = a^2 mod p. Thus, if either a mod p = 1 or a mod p = -1, then a^2 mod p = 1. Conversely, if a^2 mod p = 1, then (a mod p) 2 = 1, which is true only for a mod p = 1 or a mod p = -1.

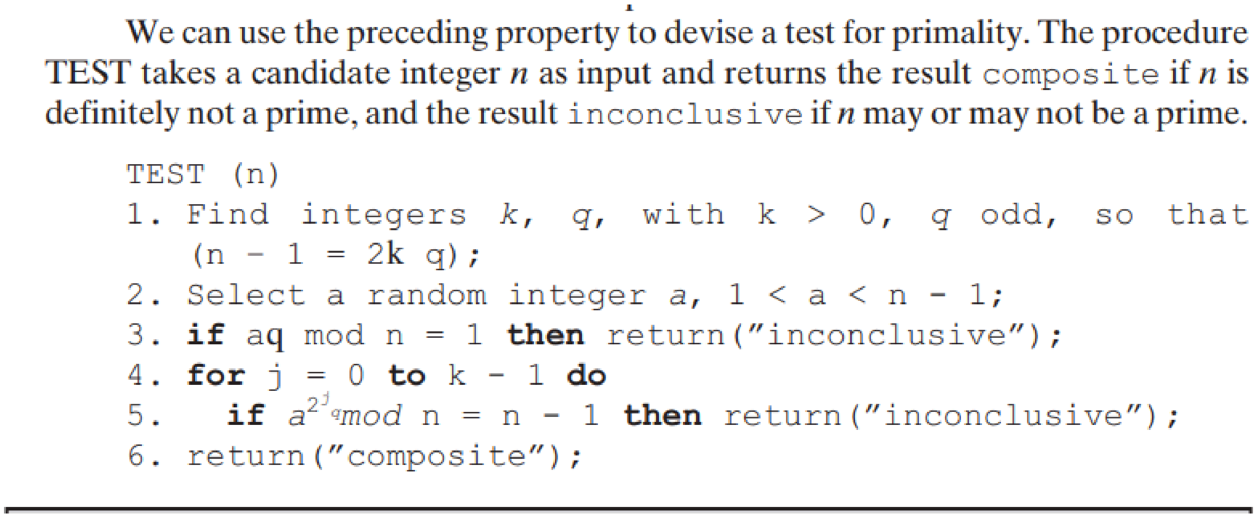


The **second property** is stated as follows: Let p be a prime number greater than 2. We can then write $p - 1 = 2^k q$ with $k > 0$, q odd. Let a be any integer in the range $1 < a < p - 1$. Then one of the two following conditions is true.

1. a^q is congruent to 1 modulo p. That is, $a^q \bmod p = 1$, or equivalently, $a^q \equiv 1 (\bmod p)$.
2. One of the numbers $a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}q}$ 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, $a^{2^{j-1}q} \equiv -1 (\bmod p)$.

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

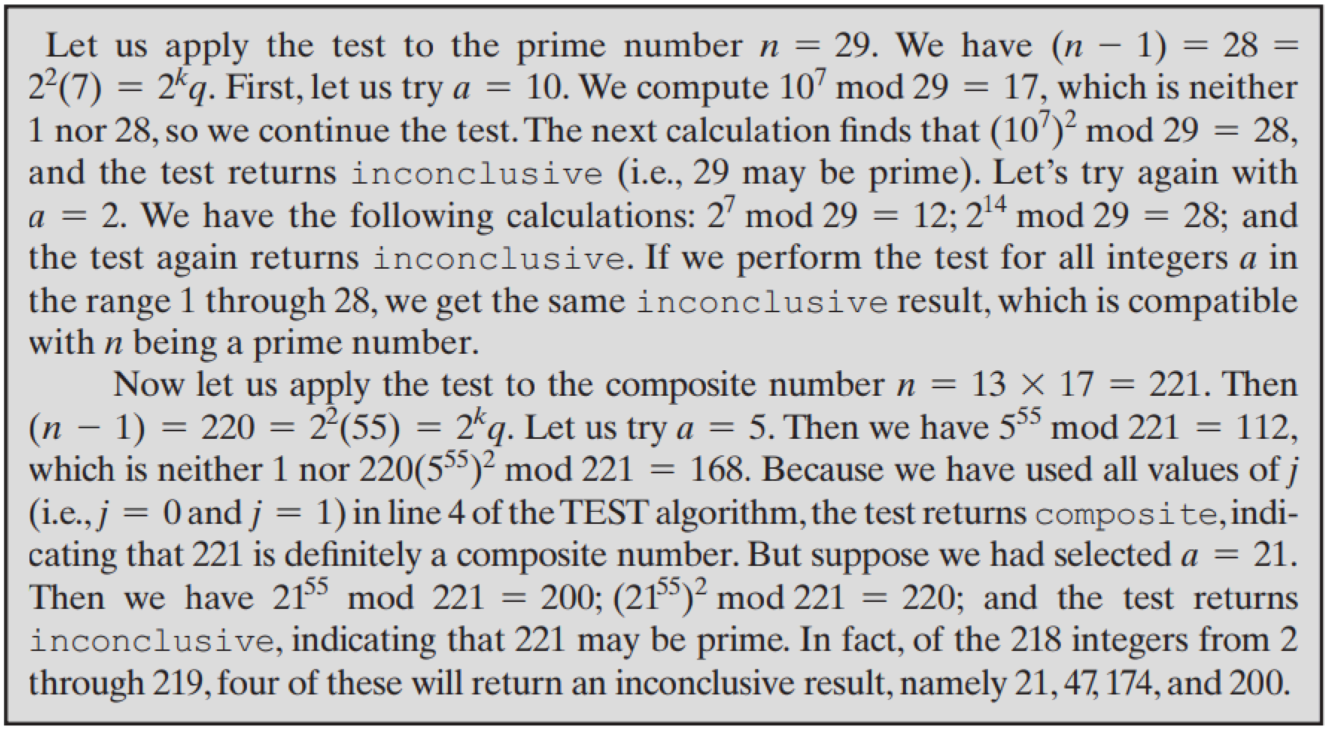
These considerations lead to the conclusion that, if n is prime, then either the first element in the list of residues, or remainders, (aq , a2q , c , a2k - 1 q , a2k q) modulo n equals 1; or some element in the list equals (n - 1);
otherwise n is composite (i.e., not a prime). On the other hand, if the condition is met, that does not necessarily mean that n is prime. For example, if n = 2047 = 23 *89, then n - 1 = 2* 1023. We compute 2^1023 mod 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 = 2k q);
2. Select a random integer a, 1 < a < n - 1;
3. if aq mod n = 1 then return("inconclusive");
4. for j = 0 to k - 1 do
5.   if $a^{2^j q}$ mod 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 = 22 (7) = 2k q. First, let us try a = 10. We compute 107 mod 29 = 17, which is neither 1 nor 28, so we continue the test. The next calculation finds that (107) 2 mod 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: 27 mod 29 = 12; 214 mod 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 * 17 = 221. Then (n - 1) = 220 = 22 (55) = 2k q. Let us try a = 5. Then we have 555 mod 221 = 112, which is neither 1 nor 220(555) 2 mod 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, indi□cating that 221 is definitely a composite number. But suppose we had selected a = 21. Then we have 2155 mod 221 = 200; (2155) 2 mod 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.

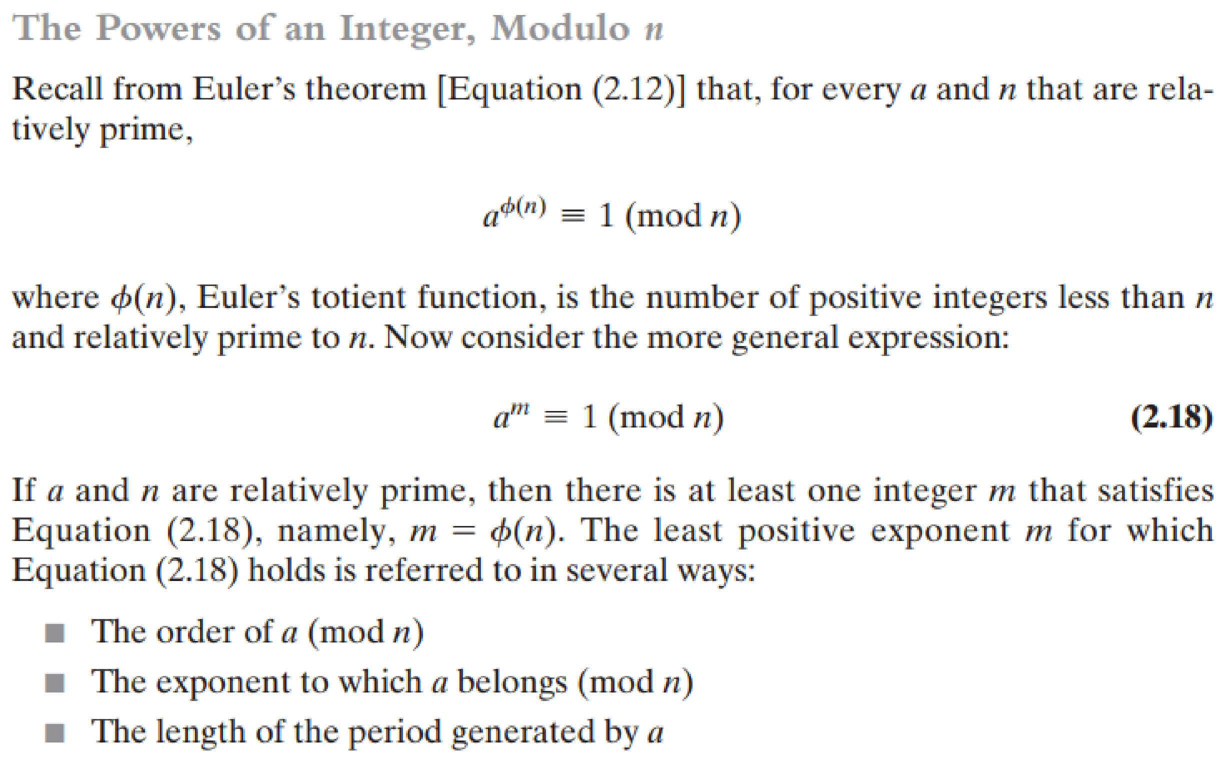

Let us apply the test to the prime number $n = 29$. We have $(n - 1) = 28 = 2^2(7) = 2^k q$. First, let us try $a = 10$. We compute $10^7 \bmod 29 = 17$, which is neither 1 nor 28, so we continue the test. The next calculation finds that $(10^7)^2 \bmod 29 = 28$, and the test returns `inconclusive` (i.e., 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.

look for a number a we will try that for k times,

Discrete Logarithms

Discrete logarithms are fundamental to a number of public-key algorithms, including Diffie–Hellman key exchange and the digital signature algorithm (DSA).



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,

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

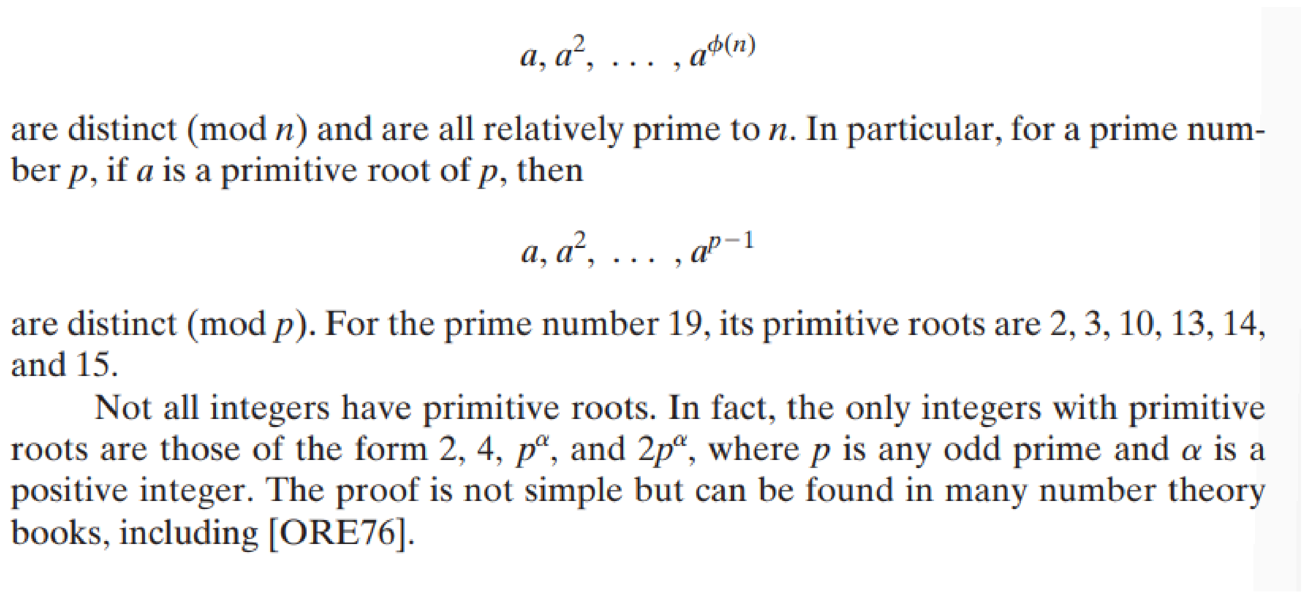
where $\phi(n)$, Euler's totient function, is the number of positive integers less than n and relatively prime to n. Now consider the more general expression:

$$a^{m} \equiv 1 \pmod{n} \qquad \textbf{(2.18)}$$

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

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



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

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

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

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].

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

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

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

Note the following:

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

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

***VERY IMP

Calculation of Discrete Logarithms

Consider the equation

$$y = g^x \text{ mod } p$$

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.

THE DATA ENCRYPTION STANDARD

Until the introduction of the Advanced Encryption Standard (AES) in 2001, the Data Encryption Standard (DES) was the most widely used encryption scheme. The algorithm itself is referred to as the Data Encryption Algorithm (DEA). For DEA, data are encrypted in 64-bit blocks using a 56-bit key. The algorithm transforms 64-bit input in a series of steps into a 64-bit output. The same steps, with the same key, are used to reverse the encryption. Over the years, DES became the dominant symmetric encryption algorithm, especially in financial applications.

DES Encryption

The overall scheme for DES encryption is illustrated in Figure 4.5. As with any encryption scheme, there are two inputs to the encryption function: the plaintext to be encrypted and the key. In this case, the plaintext must be 64 bits in length and the key is 56 bits in length.

Looking at the left-hand side of the figure, we can see that the processing of the plaintext proceeds in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that rearranges the bits to produce the *permuted input*.

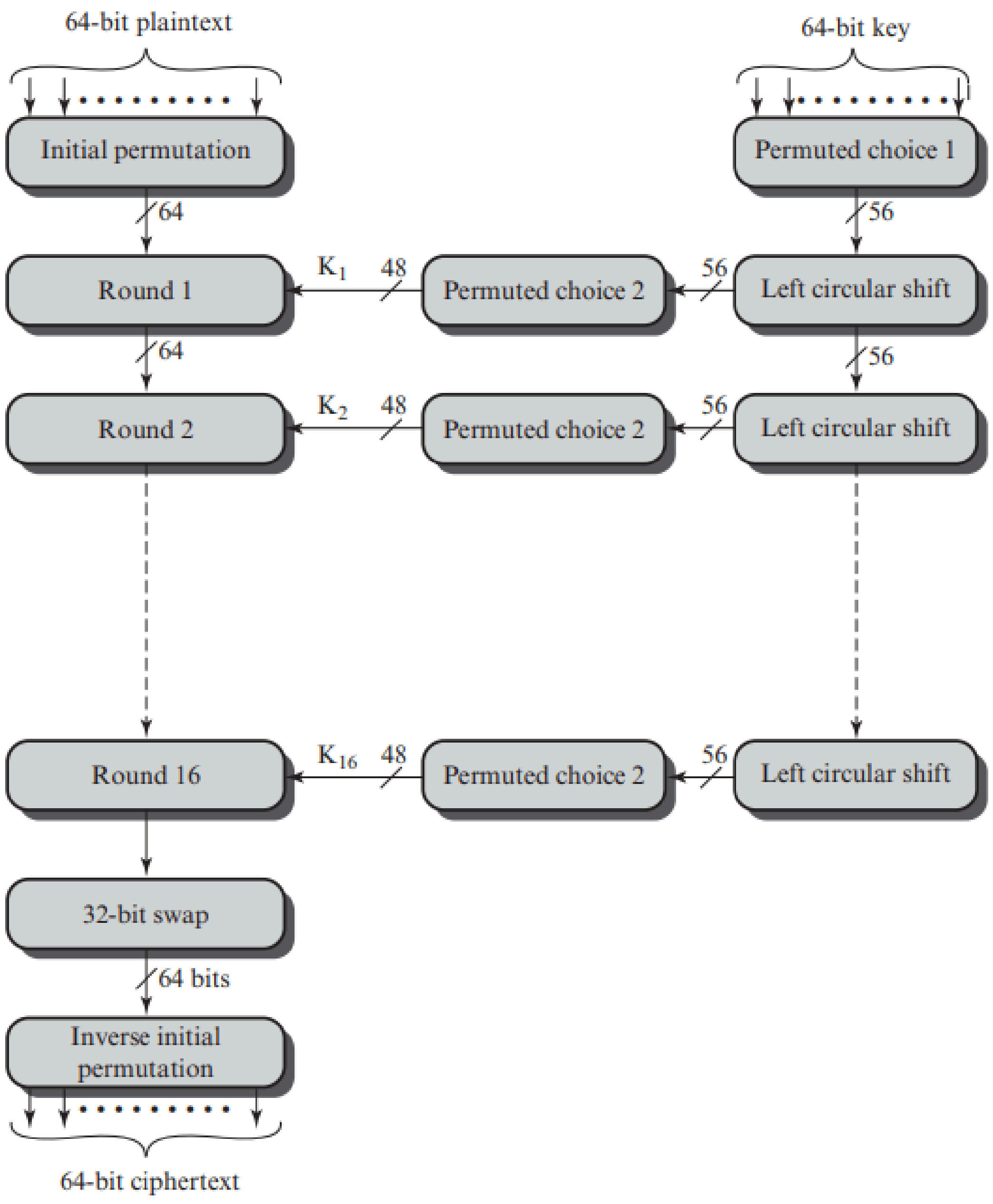


Figure 4.5 General Depiction of DES Encryption Algorithm

Actually, the function expects a 64-bit key as input. However, only 56 of these bits are ever used; the other 8 bits can be used as parity bits or simply set arbitrarily.

This is followed by a phase consisting of sixteen rounds of the same function, which involves both permutation and substitution functions. The output of the last (six□teenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation [IP-1] that is the inverse of the initial permutation function, to produce the 64-bit ciphertext. With the excep□tion of the initial and final permutations, DES has the exact structure of a Feistel cipher, as shown in Figure 4.3. The right-hand portion of Figure 4.5 shows the way in which the 56-bit key is used. Initially, the key is passed through a permutation function. Then, for each of the sixteen rounds, a subkey (Ki) is produced by the combination of a left circular shift and a permutation. The permutation function is the same for each round, but a different subkey is produced because of the repeated shifts of the key bits.

DES Decryption
As with any Feistel cipher, decryption uses the same algorithm as encryption, except that the application of the subkeys is reversed. Additionally, the initial and final permutations are reversed.

Since its adoption as a federal standard, there have been lingering concerns about the level of security provided by DES. These concerns, by and large, fall into two areas: key size and the nature of the algorithm

With a key length of 56 bits, there are 256 possible keys, which is approximately 7.2 * 10^16 keys. Thus, on the face of it, a brute-force attack appears impractical. Assuming that, on average, half the key space has to be searched, a single machine performing one DES encryption per microsecond would take more than a thousand years to break the cipher. However, the assumption of one encryption per microsecond is overly conservative. As far back as 1977, Diffie and Hellman postulated that the technology existed to build a parallel machine with 1 million encryption devices, each of which could perform one encryption per microsecond [DIFF77]. This would bring the average search time down to about 10 hours. The authors estimated that the cost would be about $20 million in 1977 dollars.

With current technology, it is not even necessary to use special, purpose-built hardware. Rather, the speed of commercial, off-the-shelf processors threaten the security of DES. A recent paper from Seagate Technology [SEAG08] suggests that a rate of 1 billion (109) key combinations per second is reasonable for today’s mul□ticore computers. Recent offerings confirm this. Both Intel and AMD now offer hardware-based instructions to accelerate the use of AES. Tests run on a contem□porary multicore Intel machine resulted in an encryption rate of about half a bil□lion encryptions per second [BASU12]. Another recent analysis suggests that with contemporary supercomputer technology, a rate of 1013 encryptions per second is reasonable [AROR12]

Key sizes of 128 bits or greater are effectively unbreakable using simply a brute-force approach. Even if we managed to speed up the attacking system by a factor of 1 trillion (1012), it would still take over 100,000 years to break a code using a 128-bit key. Fortunately, there are a number of alternatives to DES, the most important of which are AES and triple DES

Public Key CryptoSystems

Before proceeding, we should mention several common misconceptions concerning public-key encryption. One such misconception is that public-key encryption is more secure from cryptanalysis than is symmetric encryption. In fact, the security of any encryption scheme depends on the length of the key and the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis

A second misconception is that public-key encryption is a general-purpose technique that has made symmetric encryption obsolete. On the contrary, because of the computational overhead of current public-key encryption schemes, there seems no foreseeable likelihood that symmetric encryption will be abandoned. As one of the inventors of public-key encryption has put it [DIFF88], “the restriction of public-key cryptography to key management and signature applications is almost universally accepted.

Asymmetric Keys

Two related keys, a public key and a private key, that are used to perform complementary operations, such as encryption and decryption or signature generation and signature verification.

Public Key Certificate

A digital document issued and digitally signed by the private key of a Certification Authority that binds the name of a subscriber to a public key. The certificate indicates that the subscriber identified in the certificate has sole control and access to the corresponding private key.

Public Key (Asymmetric) Cryptographic Algorithm

A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys have the property that deriving the private key from the public key is computationally infeasible.

Public Key Infrastructure (PKI)

A set of policies, processes, server platforms, software and workstations used for the purpose of administering certificates and public-private key pairs, including the ability to issue, maintain, and revoke public key certificates.

Table 9.2 Conventional and Public-Key Encryption

Conventional Encryption	Public-Key Encryption
Needed to Work:	*Needed to Work:*
1. The same algorithm with the same key is used for encryption and decryption.	1. One algorithm is used for encryption and a related algorithm for decryption with a pair of keys, one for encryption and one for decryption.
2. The sender and receiver must share the algorithm and the key.	2. The sender and receiver must each have one of the matched pair of keys (not the same one).
Needed for Security:	*Needed for Security:*
1. The key must be kept secret.	1. One of the two keys must be kept secret.
2. It must be impossible or at least impractical to decipher a message if the key is kept secret.	2. It must be impossible or at least impractical to decipher a message if one of the keys is kept secret.
3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Requirements for Public-Key Cryptography

The cryptosystem illustrated in Figures 9.2 through 9.4 depends on a cryptographic algorithm based on two related keys. Diffie and Hellman postulated this system without demonstrating that such algorithms exist. However, they did lay out the conditions that such algorithms must fulfill [DIFF76b].

1. It is computationally easy for a party B to generate a key pair (public key PU_b, private key PR_b).
2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext:

$$C = \mathrm{E}(PU_b, M)$$

3. It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message:

$$M = \mathrm{D}(PR_b, C) = \mathrm{D}[PR_b, \mathrm{E}(PU_b, M)]$$

4. It is computationally infeasible for an adversary, knowing the public key, PU_b, to determine the private key, PR_b.

Table 9.3 Applications for Public-Key Cryptosystems

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie–Hellman	No	No	Yes
DSS	No	Yes	No

5. It is computationally infeasible for an adversary, knowing the public key, PU_b, and a ciphertext, C, to recover the original message, M.

We can add a sixth requirement that, although useful, is not necessary for all public-key applications:

6. The two keys can be applied in either order:

$$M = \mathrm{D}[PU_b, \mathrm{E}(PR_b, M)] = \mathrm{D}[PR_b, \mathrm{E}(PU_b, M)]$$

These are formidable requirements, as evidenced by the fact that only a few algorithms (RSA, elliptic curve cryptography, Diffie–Hellman, DSS) have received widespread acceptance in the several decades since the concept of public-key cryptography was proposed.

Before elaborating on why the requirements are so formidable, let us first recast them. The requirements boil down to the need for a trap-door one-way function. A **one-way function**[3] is one that maps a domain into a range such that every function value has a unique inverse, with the condition that the calculation of the function is easy, whereas the calculation of the inverse is infeasible:

$$\begin{aligned} Y &= \mathrm{f}(X) && \text{easy} \\ X &= \mathrm{f}^{-1}(Y) && \text{infeasible} \end{aligned}$$

Definition of a trap-door one-way function
which is easy to calculate in one direction and infeasible to calculate in the other direction unless certain additional information is known. With the additional information the inverse can be calculated in polynomial time. We can summarize as follows: A trapdoor one-way function is a family of invertible functions fk, such that

$$\begin{aligned} Y &= \mathrm{f}_k(X) && \text{easy, if } k \text{ and } X \text{ are known} \\ X &= \mathrm{f}_k^{-1}(Y) && \text{easy, if } k \text{ and } Y \text{ are known} \\ X &= \mathrm{f}_k^{-1}(Y) && \text{infeasible, if } Y \text{ is known but } k \text{ is not known} \end{aligned}$$

One of the first successful responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78].5 The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose approach to public-key encryption.

We are now ready to state the RSA scheme. The ingredients are the following:

p, q, two prime numbers	(private, chosen)
$n = pq$	(public, calculated)
e, with $\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$	(public, chosen)
$d \equiv e^{-1} \pmod{\phi(n)}$	(private, calculated)

The private key consists of $\{d, n\}$ and the public key consists of $\{e, n\}$. Suppose that user A has published its public key and that user B wishes to send the message M to A. Then B calculates $C = M^e \bmod n$ and transmits C. On receipt of this ciphertext, user A decrypts by calculating $M = C^d \bmod n$.

For this example, the keys were generated as follows:

1. Select two prime numbers, $p = 17$ and $q = 11$.
2. Calculate $n = pq = 17 \times 11 = 187$.
3. Calculate $\phi(n) = (p - 1)(q - 1) = 16 \times 10 = 160$.
4. Select e such that e is relatively prime to $\phi(n) = 160$ and less than $\phi(n)$; we choose $e = 7$.
5. Determine d such that $de \equiv 1 \pmod{160}$ and $d < 160$. The correct value is $d = 23$, because $23 \times 7 = 161 = (1 \times 160) + 1$; d can be calculated using the extended Euclid's algorithm (Chapter 2).

The resulting keys are public key $PU = \{7, 187\}$ and private key $PR = \{23, 187\}$. The example shows the use of these keys for a plaintext input of $M = 88$. For encryption, we need to calculate $C = 88^7 \bmod 187$. Exploiting the properties of modular arithmetic, we can do this as follows.

$$\begin{aligned}
88^7 \bmod 187 &= [(88^4 \bmod 187) \times (88^2 \bmod 187) \\
&\quad \times (88^1 \bmod 187)] \bmod 187 \\
88^1 \bmod 187 &= 88 \\
88^2 \bmod 187 &= 7744 \bmod 187 = 77 \\
88^4 \bmod 187 &= 59{,}969{,}536 \bmod 187 = 132 \\
88^7 \bmod 187 &= (88 \times 77 \times 132) \bmod 187 = 894{,}432 \bmod 187 = 11
\end{aligned}$$

For decryption, we calculate $M = 11^{23} \bmod 187$:

$$\begin{aligned}
11^{23} \bmod 187 &= [(11^1 \bmod 187) \times (11^2 \bmod 187) \times (11^4 \bmod 187) \\
&\quad \times (11^8 \bmod 187) \times (11^8 \bmod 187)] \bmod 187 \\
11^1 \bmod 187 &= 11 \\
11^2 \bmod 187 &= 121 \\
11^4 \bmod 187 &= 14{,}641 \bmod 187 = 55 \\
11^8 \bmod 187 &= 214{,}358{,}881 \bmod 187 = 33 \\
11^{23} \bmod 187 &= (11 \times 121 \times 55 \times 33 \times 33) \bmod 187 \\
&= 79{,}720{,}245 \bmod 187 = 88
\end{aligned}$$

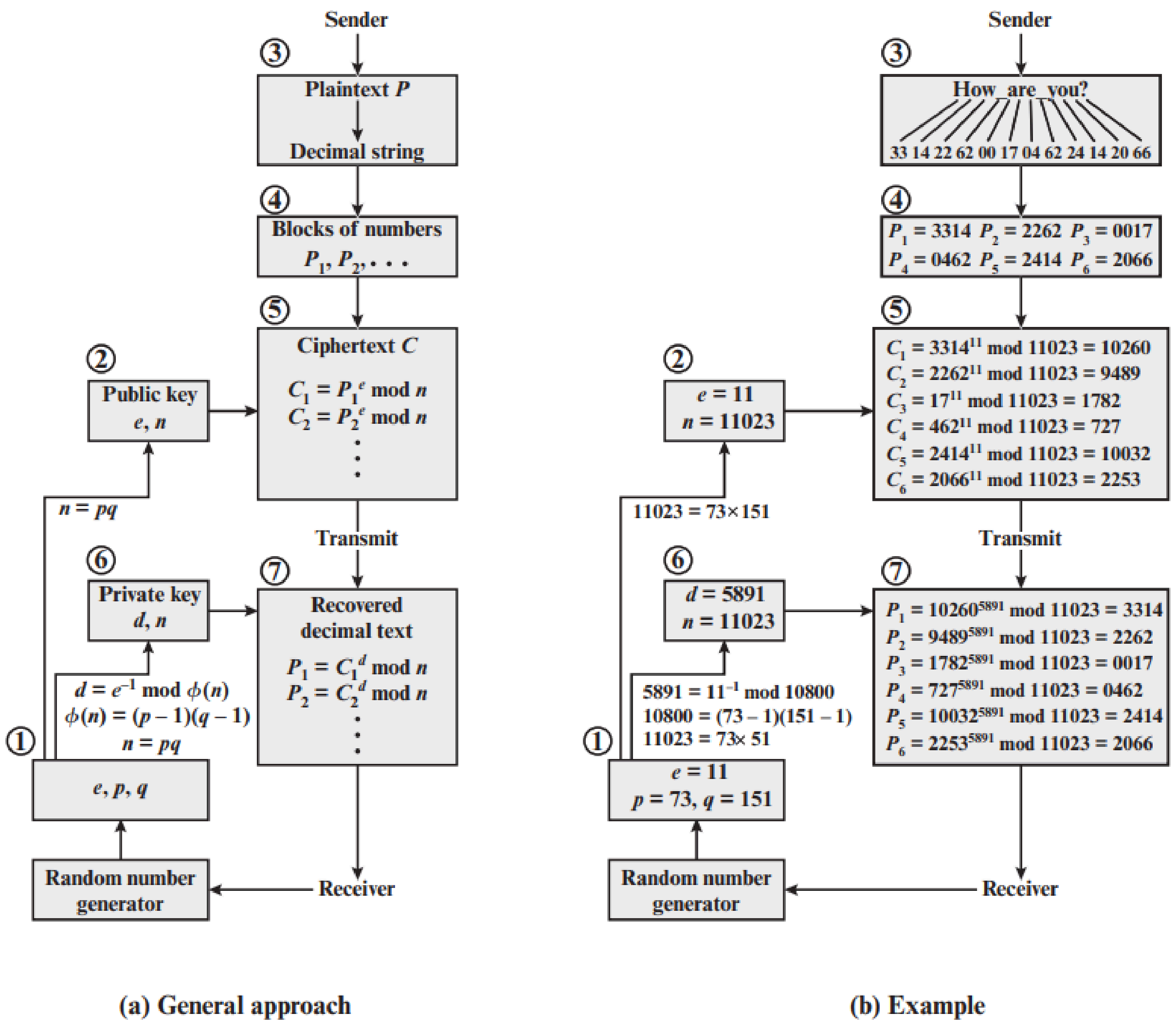


Figure 9.7 RSA Processing of Multiple Blocks

The Security of RSA

Five possible approaches to attacking the RSA algorithm are

- **Brute force:** This involves trying all possible private keys.
- **Mathematical attacks:** There are several approaches, all equivalent in effort to factoring the product of two primes.
- **Timing attacks:** These depend on the running time of the decryption algorithm.
- **Hardware fault-based attack:** This involves inducing hardware faults in the processor that is generating digital signatures.
- **Chosen ciphertext attacks:** This type of attack exploits properties of the RSA algorithm.

The Factoring Problem We can identify three approaches to attacking RSA mathematically.

1. Factor n into its two prime factors. This enables calculation of $\phi(n) = (p - 1) \times (q - 1)$, which in turn enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
2. Determine $\phi(n)$ directly, without first determining p and q. Again, this enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
3. Determine d directly, without first determining $\phi(n)$.

In addition to specifying the size of n, a number of other constraints have been suggested by researchers. To avoid values of n that may be factored more easily, the algorithm's inventors suggest the following constraints on p and q.

1. p and q should differ in length by only a few digits. Thus, for a 1024-bit key (309 decimal digits), both p and q should be on the order of magnitude of 10^{75} to 10^{100}.
2. Both $(p - 1)$ and $(q - 1)$ should contain a large prime factor.
3. $\gcd(p - 1, q - 1)$ should be small.

DIFFIE–HELLMAN KEY EXCHANGE

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography [DIFF76b] and is generally referred to as Diffie–Hellman key exchange. A number of commercial products employ this key exchange technique. The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent symmetric encryption of messages. The algorithm itself is limited to the exchange of secret values.

The Diffie–Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms.

For any integer b and a primitive root a of prime number p, we can find a unique exponent i such that

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

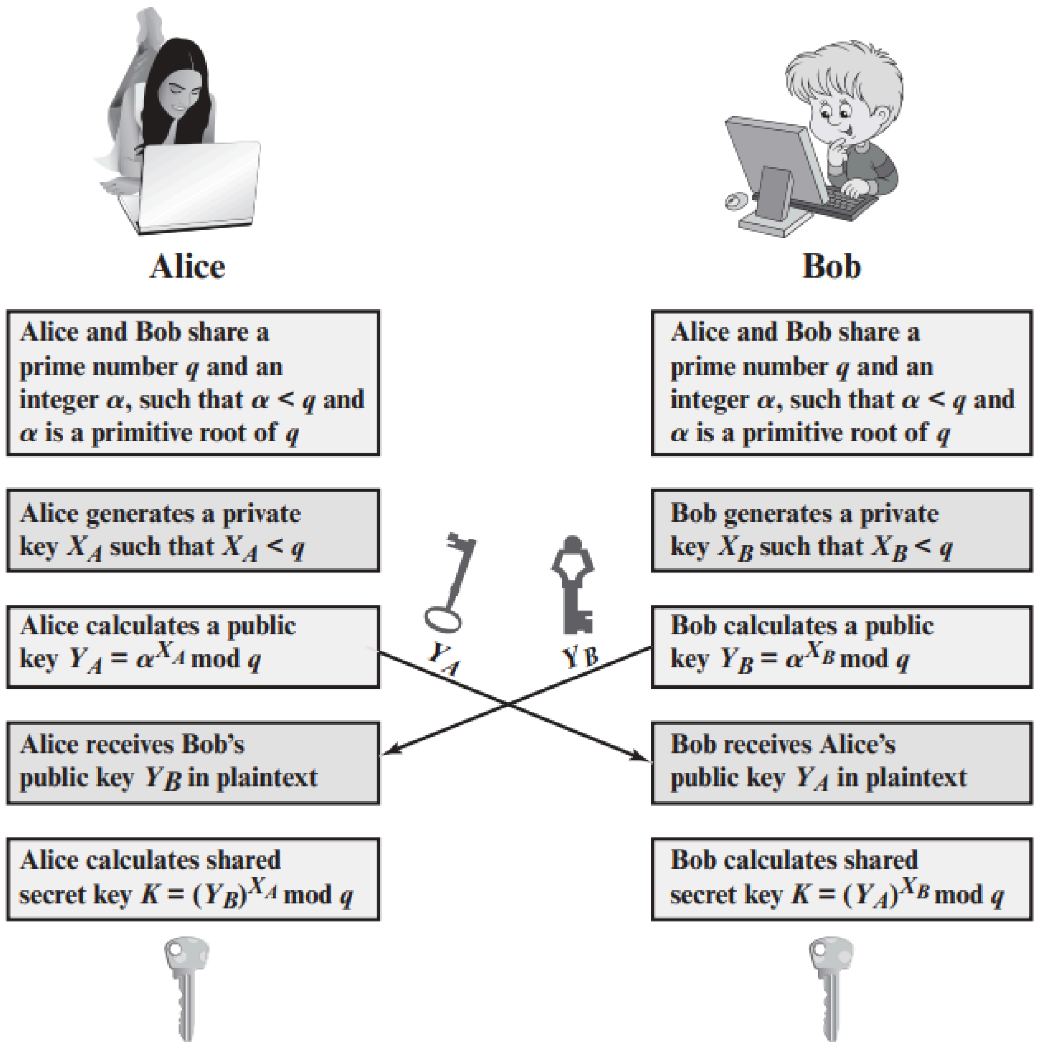
[1]Williamson of Britain's CESG published the identical scheme a few months earlier in a classified document [WILL76] and claims to have discovered it several years prior to that; see [ELLI99] for a discussion.

The exponent i is referred to as the **discrete logarithm** of b for the base a, mod p. We express this value as $\text{dlog}_{a,p}(b)$. See Chapter 2 for an extended discussion of discrete logarithms.

The Algorithm

Figure 10.1 summarizes the Diffie–Hellman key exchange algorithm. For this scheme, there are two publicly known numbers: a prime number q and an integer α that is a primitive root of q. Suppose the users A and B wish to create a shared key.

User A selects a random integer $X_A < q$ and computes $Y_A = \alpha^{X_A} \bmod q$. Similarly, user B independently selects a random integer $X_B < q$ and computes $Y_B = \alpha^{X_B} \bmod q$. Each side keeps the X value private and makes the Y value available publicly to the other side. Thus, X_A is A's private key and Y_A is A's corresponding public key, and similarly for B. User A computes the key as $K = (Y_B)^{X_A} \bmod q$ and user B computes the key as $K = (Y_A)^{X_B} \bmod q$. These two calculations produce identical results:



Alice
Bob
Alice and Bob share a prime number q and an integer α, such that $\alpha < q$ and α is a primitive root of q
Alice and Bob share a prime number q and an integer α, such that $\alpha < q$ and α is a primitive root of q
Alice generates a private key X_A such that $X_A < q$
Bob generates a private key X_B such that $X_B < q$
Alice calculates a public key $Y_A = \alpha^{X_A} \bmod q$
Bob calculates a public key $Y_B = \alpha^{X_B} \bmod q$
Y_A
Y_B
Alice receives Bob's public key Y_B in plaintext
Bob receives Alice's public key Y_A in plaintext
Alice calculates shared secret key $K = (Y_B)^{X_A} \bmod q$
Bob calculates shared secret key $K = (Y_A)^{X_B} \bmod q$

$$
\begin{aligned}
K &= (Y_B)^{X_A} \bmod q \\
&= (\alpha^{X_B} \bmod q)^{X_A} \bmod q \\
&= (\alpha^{X_B})^{X_A} \bmod q \\
&= \alpha^{X_B X_A} \bmod q \\
&= (\alpha^{X_A})^{X_B} \bmod q \\
&= (\alpha^{X_A} \bmod q)^{X_B} \bmod q \\
&= (Y_A)^{X_B} \bmod q
\end{aligned}
$$

By the rules of modular arthimetic.

The result is that the two sides have exchanged a secret value. Typically, this secret value is used as shared symmetric secret key. Now consider an adversary who can observe the key exchange and wishes to determine the secret key K. Because X_A and X_B are private, an adversary only has the following ingredients to work with: q, α, Y_A, and Y_B. Thus, the adversary is forced to take a discrete logarithm to determine the key. For example, to determine the private key of user B, an adversary must compute

$$X_B = \text{dlog}_{\alpha,q}(Y_B)$$

The adversary can then calculate the key K in the same manner as user B calculates it. That is, the adversary can calculate K as

$$K = (Y_A)^{X_B} \bmod q$$

The security of the Diffie–Hellman key exchange lies in the fact that, while it is relatively easy to calculate exponentials modulo a prime, it is very difficult to calculate discrete logarithms. For large primes, the latter task is considered infeasible.

Man-in-the-Middle Attack

The protocol depicted in Figure 10.1 is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is the adversary. The attack proceeds as follows (Figure 10.2).

1. Darth prepares for the attack by generating two random private keys X_{D1} and X_{D2} and then computing the corresponding public keys Y_{D1} and Y_{D2}.
2. Alice transmits Y_A to Bob.
3. Darth intercepts Y_A and transmits Y_{D1} to Bob. Darth also calculates $K2 = (Y_A)^{X_{D2}} \bmod q$.
4. Bob receives Y_{D1} and calculates $K1 = (Y_{D1})^{X_B} \bmod q$.
5. Bob transmits Y_B to Alice.
6. Darth intercepts Y_B and transmits Y_{D2} to Alice. Darth calculates $K1 = (Y_B)^{X_{D1}} \bmod q$.
7. Alice receives Y_{D2} and calculates $K2 = (Y_{D2})^{X_A} \bmod q$.

At this point, Bob and Alice think that they share a secret key, but instead Bob and Darth share secret key $K1$ and Alice and Darth share secret key $K2$. All future communication between Bob and Alice is compromised in the following way.

1. Alice sends an encrypted message M: $\text{E}(K2, M)$.
2. Darth intercepts the encrypted message and decrypts it to recover M.
3. Darth sends Bob $E(K1, M)$ or $\text{E}(K1, M')$, where M' is any message. In the first case, Darth simply wants to eavesdrop on the communication without altering it. In the second case, Darth wants to modify the message going to Bob.

The key exchange protocol is vulnerable to such an attack because it does not authenticate the participants. This vulnerability can be overcome with the use of digital signatures and public-key certificates;

Abhi tak idhr tak hi
baaki chize baadme padhte hain
We are left with Hash Chapter(35 p), MAC Chapter(35 p) and Digital Signature Chapter(20 Pages)
and PKI (2 pages). in book baaki ig book ka khatm then.

Hash Function

A hash function H accepts a variable-length block of data M as input and produces a fixed-size hash value h = H(M). A “good” hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random. In general terms, the principal object of a hash function is data integrity. A change to any bit or bits in M results, with high probability, in a change to the hash value

When a hash function is used to provide message authentication, the hash function value is often referred to as a message digest.

More commonly, message authentication is achieved using a message authentication code (MAC), also known as a keyed hash function. Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC, which is associ□ated with the protected message. If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the associated MAC value. An attacker who alters the message will be unable to alter the associated MAC value without knowledge of the secret key. Note that the verifying party also knows who the sending party is because no one else knows the secret key.

Digital Signatures

Another important application, which is similar to the message authentication application, is the digital signature. The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, the hash value of a message is encrypted with a user’s private key. Anyone who knows the user’s public key can verify the integrity of the message that is associated with the digital signature. In this case, an attacker who wishes to alter the message would need to know the user’s private key.

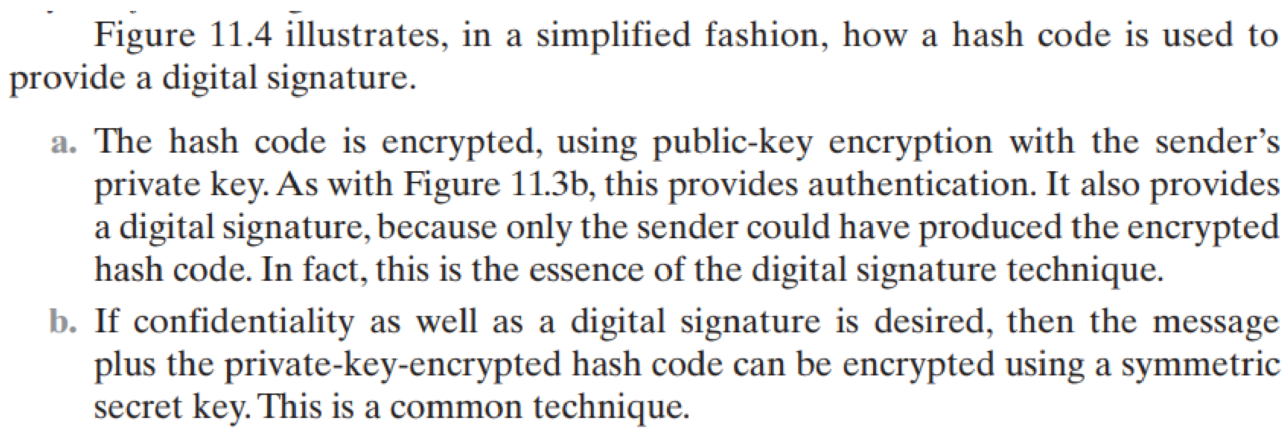


Figure 11.4 illustrates, in a simplified fashion, how a hash code is used to provide a digital signature.

a. The hash code is encrypted, using public-key encryption with the sender's private key. As with Figure 11.3b, this provides authentication. It also provides a digital signature, because only the sender could have produced the encrypted hash code. In fact, this is the essence of the digital signature technique.
b. If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.

A cryptographic hash function can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG). A common application for a hash-based PRF is for the generation of symmetric keys.

REQUIREMENTS AND SECURITY

Before proceeding, we need to define two terms. For a hash value $h = H(x)$, we say that x is the **preimage** of h. That is, x is a data block whose hash value, using the function H, is h. Because H is a many-to-one mapping, for any given hash value h, there will in general be multiple preimages. A **collision** occurs if we have $x \neq y$ and $H(x) = H(y)$. Because we are using hash functions for data integrity, collisions are clearly undesirable.

Let us consider how many preimages are there for a given hash value, which is a measure of the number of potential collisions for a given hash value. Suppose the length of the hash code is n bits, and the function H takes as input messages or data blocks of length b bits with $b > n$. Then, the total number of possible messages is 2^b and the total number of possible hash values is 2^n. On average, each hash value corresponds to 2^{b-n} preimages. If H tends to uniformly distribute hash values then, in fact, each hash value will have close to 2^{b-n} preimages. If we now allow inputs of arbitrary length, not just a fixed length of some number of bits, then the number of preimages per hash value is arbitrarily large. However, the security risks in the use of a hash function are not as severe as they might appear from this analysis. To understand better the security implications of cryptographic hash functions, we need precisely define their security requirements.

Table 11.1 Requirements for a Cryptographic Hash Function H

Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	$H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that $H(y) = h$.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) with $x \neq y$, such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

A function that is collision resistant is also second preimage resistant, but the reverse is not necessarily true. A function can be collision resistant but not preimage resistant and vice versa. A function can be preimage resistant but not second preimage resistant and vice versa.

350 CHAPTER 11 / CRYPTOGRAPHIC HASH FUNCTIONS

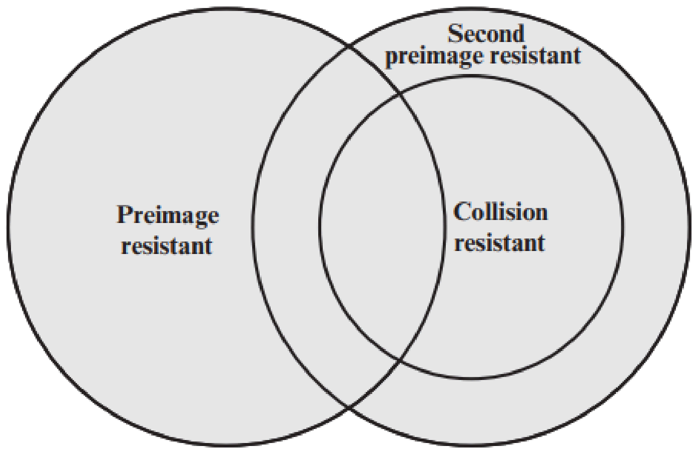


Figure 11.6 Relationship Among Hash Function Properties

Table 11.2 Hash Function Resistance Properties Required for Various Data Integrity Applications

	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + digital signature	yes	yes	yes*
Intrusion detection and virus detection		yes	
Hash + symmetric encryption			
One-way password file	yes		
MAC	yes	yes	yes*

*Resistance required if attacker is able to mount a chosen message attack

meaning. Figure 11.7 provides an example.

To summarize, for a hash code of length m, the level of effort required, as we have seen, is proportional to the following.

Preimage resistant	2^m
Second preimage resistant	2^m
Collision resistant	$2^{m/2}$

Keep in mind that for any hash function there must exist collisions, because we are mapping a message of length at least equal to twice the block size b (because we must append a length field) into a hash code of length n, where b Ú n. What is required is that it is computationally infeasible to find collisions

Digital Signatures Chapter

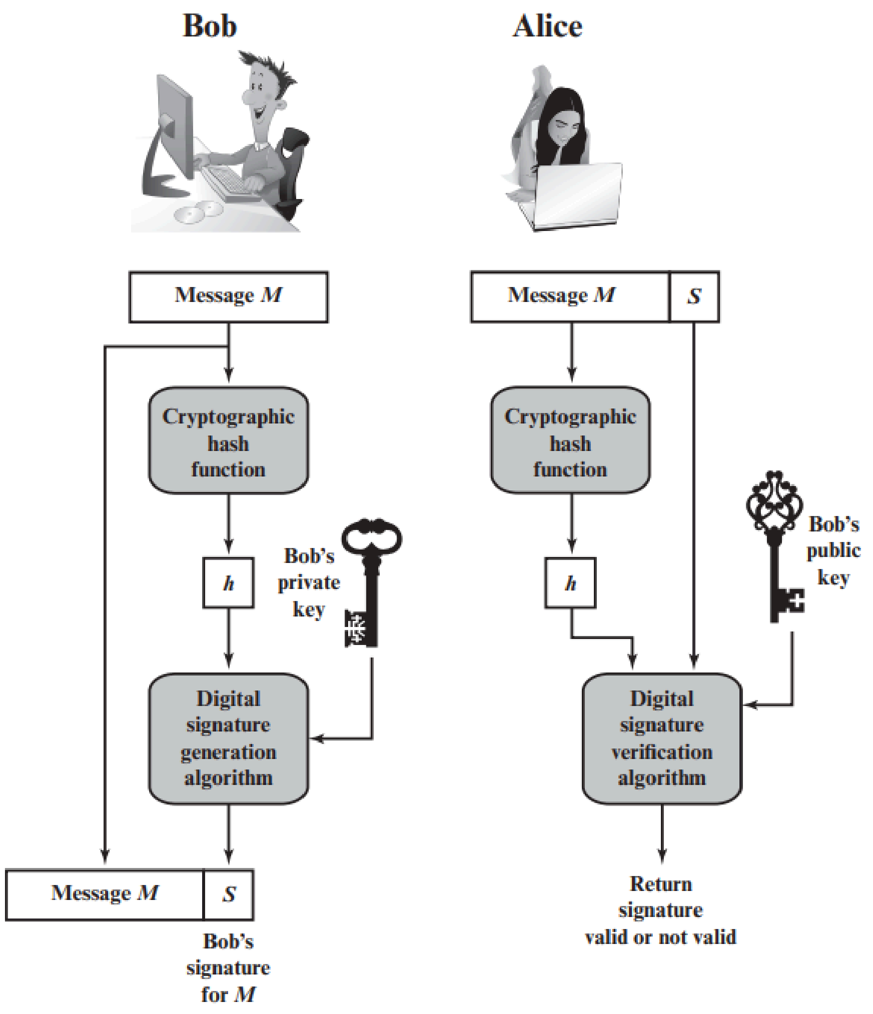

Bob
Alice
Message M
Message M
S
Cryptographic hash function
Cryptographic hash function
h
Bob's private key
h
Bob's public key
Digital signature generation algorithm
Digital signature verification algorithm
Message M
S
Bob's signature for M
Return signature valid or not valid
(a) Bob signs a message
(b) Alice verifies the signature

Figure 13.1 Simplified Depiction of Essential Elements of Digital Signature Process

Message authentication protects two parties who exchange messages from any third party. However, it does not protect the two parties against each other. Several forms of dispute between the two parties are possible

In situations where there is not complete trust between sender and receiver, something more than authentication is needed. The most attractive solution to this problem is the digital signature. The digital signature must have the following properties:
■ It must verify the author and the date and time of the signature.
■ It must authenticate the contents at the time of the signature.
■ It must be verifiable by third parties, to resolve disputes.

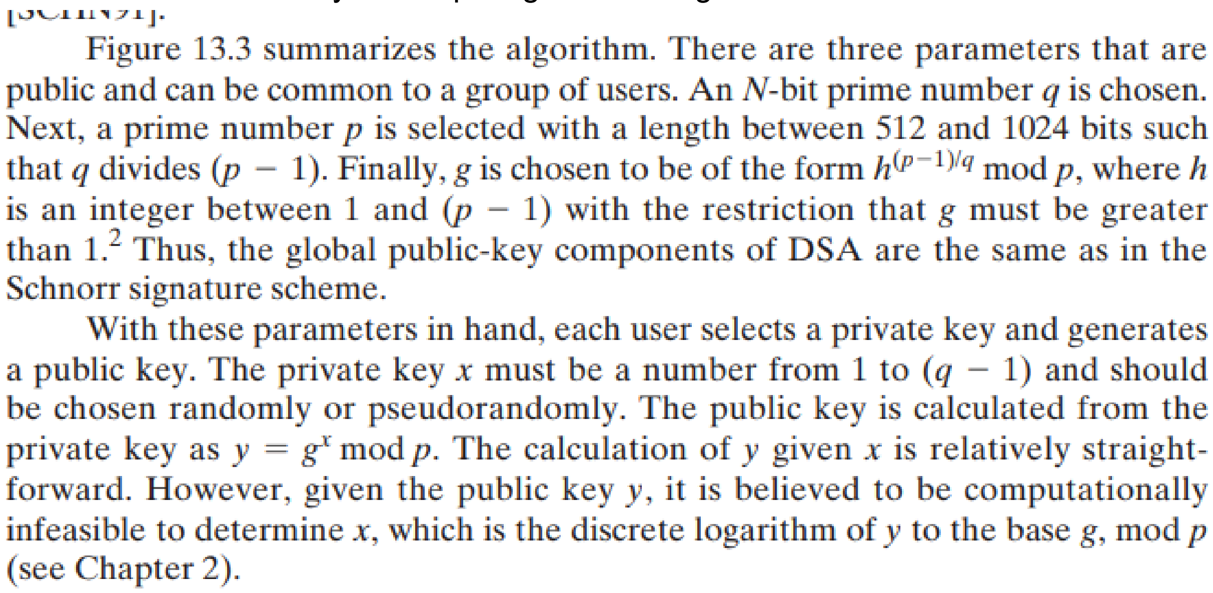
Digital Signature Requirements
On the basis of the properties and attacks just discussed, we can formulate the following requirements for a digital signature.
■ The signature must be a bit pattern that depends on the message being signed.
■ The signature must use some information only known to the sender to prevent both forgery

and denial.
■ It must be relatively easy to produce the digital signature. It must be relatively easy to recognize and verify the digital signature.
■ It must be computationally infeasible to forge a digital signature, either by constructing a new message for an existing digital signature or by constructing a fraudulent digital signature for a given message. ■ It must be practical to retain a copy of the digital signature in storage.

Direct Digital Signature
The term direct digital signature refers to a digital signature scheme that involves only the communicating parties (source, destination). It is assumed that the destina□tion knows the public key of the source.

The Digital Signature Algorithm

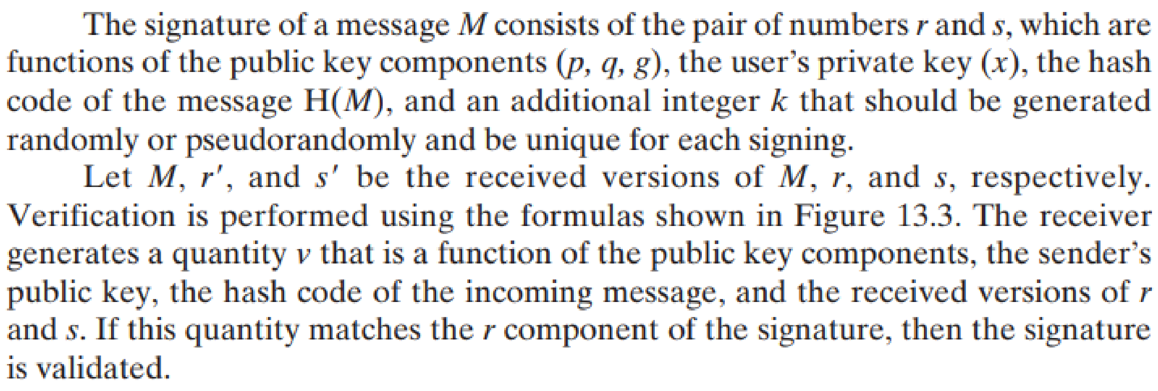
DSA is based on the difficulty of computing discrete logarithms



[SCHN91].

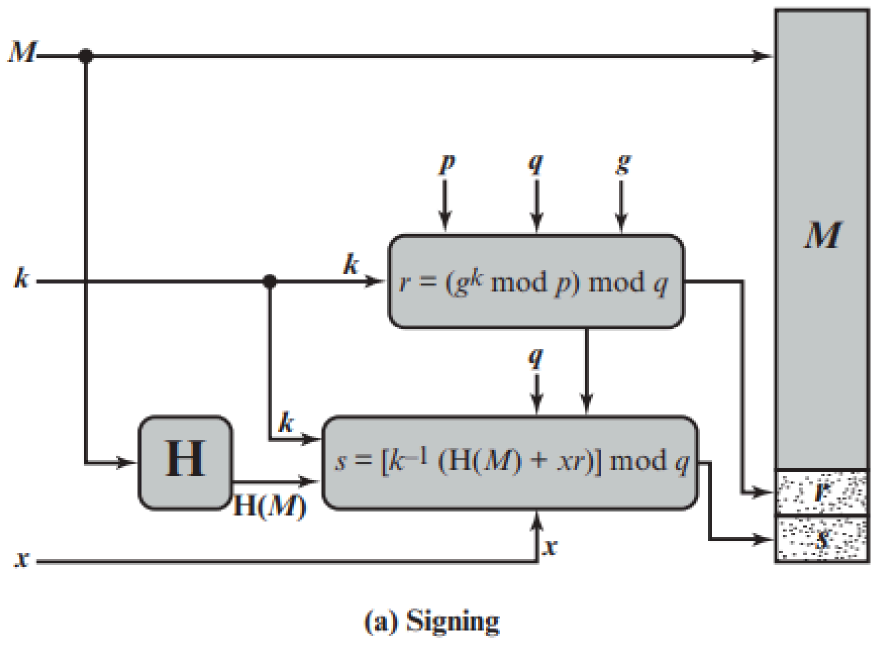
Figure 13.3 summarizes the algorithm. There are three parameters that are public and can be common to a group of users. An N-bit prime number q is chosen. Next, a prime number p is selected with a length between 512 and 1024 bits such that q divides $(p - 1)$. Finally, g is chosen to be of the form $h^{(p-1)/q} \bmod p$, where h is an integer between 1 and $(p - 1)$ with the restriction that g must be greater than 1.[2] Thus, the global public-key components of DSA are the same as in the Schnorr signature scheme.

With these parameters in hand, each user selects a private key and generates a public key. The private key x must be a number from 1 to $(q - 1)$ and should be chosen randomly or pseudorandomly. The public key is calculated from the private key as $y = g^x \bmod p$. The calculation of y given x is relatively straightforward. However, given the public key y, it is believed to be computationally infeasible to determine x, which is the discrete logarithm of y to the base g, mod p (see Chapter 2).

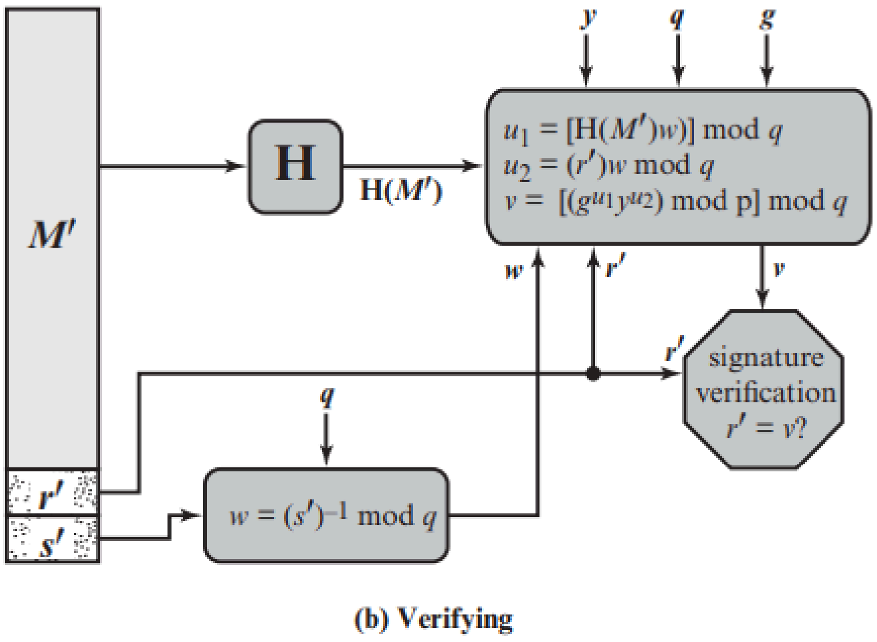


The signature of a message M consists of the pair of numbers r and s, which are functions of the public key components (p, q, g), the user's private key (x), the hash code of the message $H(M)$, and an additional integer k that should be generated randomly or pseudorandomly and be unique for each signing.

Let M, r', and s' be the received versions of M, r, and s, respectively. Verification is performed using the formulas shown in Figure 13.3. The receiver generates a quantity v that is a function of the public key components, the sender's public key, the hash code of the incoming message, and the received versions of r and s. If this quantity matches the r component of the signature, then the signature is validated.

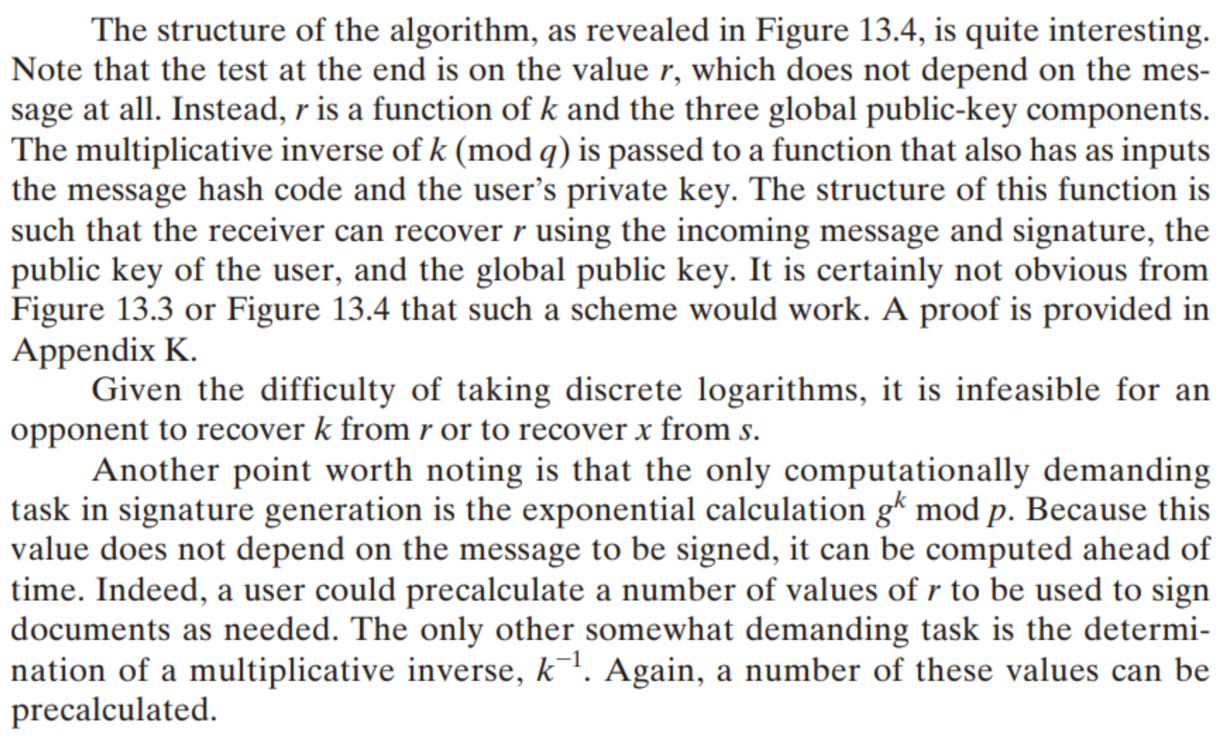


M
p
q
g
M
k
k
$r = (g^k \bmod p) \bmod q$
q
k
H
$s = [k^{-1}(H(M) + xr)] \bmod q$
H(M)
r
s
x
x
(a) Signing



y
q
g
M′
H
H(M′)
$u_1 = [H(M')w)] \bmod q$
$u_2 = (r')w \bmod q$
$v = [(g^{u_1}y^{u_2}) \bmod p] \bmod q$
w
r′
v
r′
signature verification $r' = v$?
q
r′
s′
$w = (s')^{-1} \bmod q$
(b) Verifying

Figure 13.4 DSA Signing and Verifying


The structure of the algorithm, as revealed in Figure 13.4, is quite interesting. Note that the test at the end is on the value r, which does not depend on the message at all. Instead, r is a function of k and the three global public-key components. The multiplicative inverse of $k \pmod q$ is passed to a function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover r using the incoming message and signature, the public key of the user, and the global public key. It is certainly not obvious from Figure 13.3 or Figure 13.4 that such a scheme would work. A proof is provided in Appendix K.

Given the difficulty of taking discrete logarithms, it is infeasible for an opponent to recover k from r or to recover x from s.

Another point worth noting is that the only computationally demanding task in signature generation is the exponential calculation $g^k \bmod p$. Because this value does not depend on the message to be signed, it can be computed ahead of time. Indeed, a user could precalculate a number of values of r to be used to sign documents as needed. The only other somewhat demanding task is the determination of a multiplicative inverse, k^{-1}. Again, a number of these values can be precalculated.

PKI Public Key Infrastructure

RFC 4949 (Internet Security Glossary) defines public-key infrastructure (PKI) as the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates based on asymmetric cryptography. The principal objective for developing a PKI is to enable secure, convenient, and efficient acquisition of public keys.

model. These elements are

- **End entity:** A generic term used to denote end users, devices (e.g., servers, routers), or any other entity that can be identified in the subject field of a public-key certificate. End entities typically consume and/or support PKI-related services.
- **Certification authority (CA):** The issuer of certificates and (usually) certificate revocation lists (CRLs). It may also support a variety of administrative functions, although these are often delegated to one or more Registration Authorities.
- **Registration authority (RA):** An optional component that can assume a number of administrative functions from the CA. The RA is often associated with the end entity registration process but can assist in a number of other areas as well.
- **CRL issuer:** An optional component that a CA can delegate to publish CRLs.
- **Repository:** A generic term used to denote any method for storing certificates and CRLs so that they can be retrieved by end entities.

PKIX Management Functions

PKIX identifies a number of management functions that potentially need to be supported by management protocols. These are indicated in Figure 14.17 and include the following:

- **Registration:** This is the process whereby a user first makes itself known to a CA (directly or through an RA), prior to that CA issuing a certificate or certificates for that user. Registration begins the process of enrolling in a PKI. Registration usually involves some offline or online procedure for mutual authentication. Typically, the end entity is issued one or more shared secret keys used for subsequent authentication.
- **Initialization:** Before a client system can operate securely, it is necessary to install key materials that have the appropriate relationship with keys stored elsewhere in the infrastructure. For example, the client needs to be securely initialized with the public key and other assured information of the trusted CA(s), to be used in validating certificate paths.
- **Certification:** This is the process in which a CA issues a certificate for a user's public key, returns that certificate to the user's client system, and/or posts that certificate in a repository.
- **Key pair recovery:** Key pairs can be used to support digital signature creation and verification, encryption and decryption, or both. When a key pair is used for

encryption/decryption, it is important to provide a mechanism to recover the necessary decryption keys when normal access to the keying material is no longer possible, otherwise it will not be possible to recover the encrypted data. Loss of access to the decryption key can result from forgotten passwords/PINs, corrupted disk drives, damage to hardware tokens, and so on. Key pair recovery allows end entities to restore their encryption/decryption key pair from an authorized key backup facility (typically, the CA that issued the end entity's certificate).

- **Key pair update:** All key pairs need to be updated regularly (i.e., replaced with a new key pair) and new certificates issued. Update is required when the certificate lifetime expires and as a result of certificate revocation.
- **Revocation request:** An authorized person advises a CA of an abnormal situation requiring certificate revocation. Reasons for revocation include private-key compromise, change in affiliation, and name change.
- **Cross certification:** Two CAs exchange information used in establishing a cross-certificate. A cross-certificate is a certificate issued by one CA to another CA that contains a CA signature key used for issuing certificates.

End for Midsem