**SUBJECT TITLE AND CODE : CRYPTOGRAPHY AND NETWORK SECURITY(BCS703)**

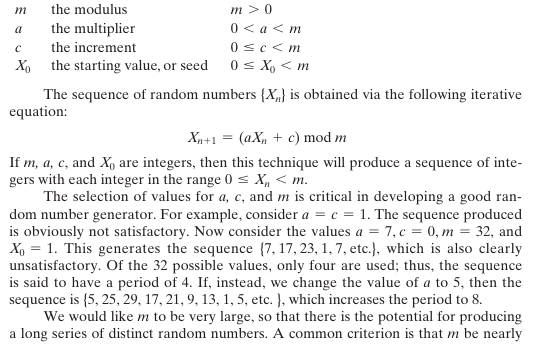
**MODULE 2**

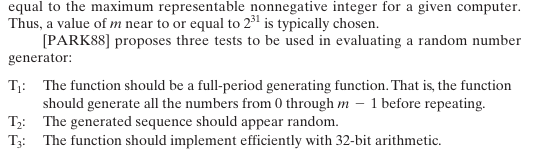
**PSEUDORANDOM NUMBER GENERATORS**

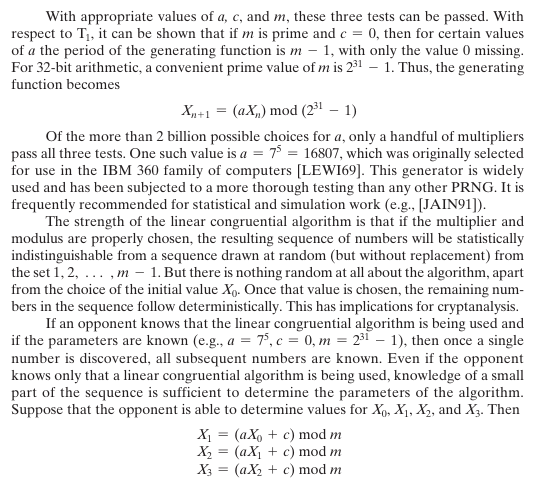
In this section, we look at two types of algorithms for PRNGs.

**Linear Congruential Generators**

A widely used technique for pseudorandom number generation is an algorithm first proposed by Lehmer [LEHM51], which is known as the linear congruential method. The algorithm is parameterized with four numbers, as follows:







These equations can be solved for a, c, and m.

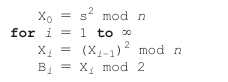
Thus, although it is nice to be able to use a good PRNG, it is desirable to make the actual sequence used nonreproducible, so that knowledge of part of the sequence on the part of an opponent is insufficient to determine future elements of the sequence. This goal can be achieved in a number of ways. For example, [BRIG79] suggests using an internal system clock to modify the random number stream. One way to use the clock would be to restart the sequence after every N numbers using the current clock value (mod m) as the new seed. Another way would be simply to add the current clock value to each random number (mod m).

**Blum Blum Shub Generator**

A popular approach to generating secure pseudorandom numbers is known as the Blum Blum Shub (BBS) generator (see Figure 8.3), named for its developers [BLUM86]. It has perhaps the strongest public proof of its cryptographic strength of any purpose-built algorithm. The procedure is as follows. First, choose two large prime numbers, p and q, that both have a remainder of 3 when divided by 4. That is,

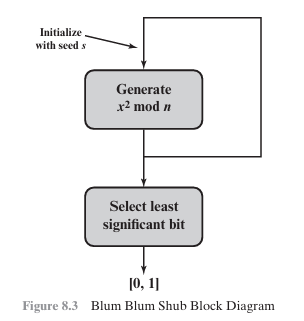
p = q = 3(mod 4)

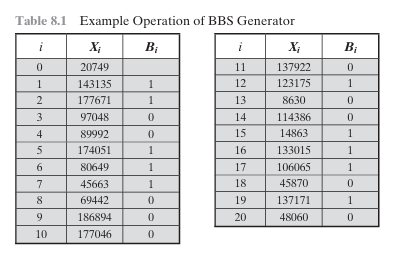
This notation, explained more fully in Chapter 4, simply means that (p mod 4) = (q mod 4) = 3. For example, the prime numbers 7 and 11 satisfy 7 K 11 K 3(mod 4). Let n = p \* q. Next, choose a random number s, such that s is relatively prime to n; this is equivalent to saying that neither p nor q is a factor of s. Then the BBS genera tor produces a sequence of bits Bi according to the following algorithm:



Thus, the least significant bit is taken at each iteration. Table 8.1 shows an example of BBS operation. Here, n = 192649 = 383 \* 503, and the seed s = 101355.

The BBS is referred to as a **cryptographically secure pseudorandom bit generator (CSPRBG).** A CSPRBG is defined as one that passes the next-bit test, which, in turn, is defined as follows [MENE97]: A pseudorandom bit generator is said to pass the next-bit test if there is not a polynomial-time algorithm1 that, on input of the first k bits of an output sequence, can predict the (k + 1)st bit with probability significantly greater than 1/2. In other words, given the first k bits of the





sequence, there is not a practical algorithm that can even allow you to state that the next bit will be 1 (or 0) with probability greater than 1/2. For all practical purposes, the sequence is unpredictable. The security of BBS is based on the difficulty of factoring n. That is, given n, we need to determine its two prime factors p and q.

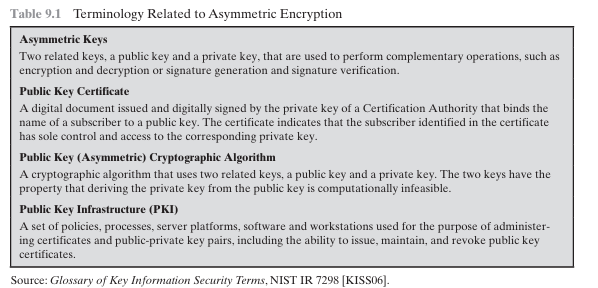
**Public key cryptography and RSA**

The development of public-key, or asymmetric, cryptography is the greatest and per haps the only true revolution in the entire history of cryptography. From its earliest beginnings to modern times, virtually all cryptographic systems have been based on the elementary tools of substitution and permutation. After millennia of working with algorithms that could be calculated by hand, a major advance in symmetric cryptography occurred with the development of the rotor encryption/decryption machine. The electromechanical rotor enabled the development of fiendishly complex cipher systems. With the availability of computers, even more complex systems were devised, the most prominent of which was the Lucifer effort at IBM that culminated in the Data Encryption Standard (DES). But both rotor machines and DES, although representing significant advances, still relied on the bread-and-butter tools of substitution and permutation.

Public-key cryptography provides a radical departure from all that has gone be fore. For one thing, public-key algorithms are based on mathematical functions rather than on substitution and permutation. More important, public-key cryptography is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication, as we shall see.

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 tech nique 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.”



Finally, there is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centres for symmetric encryption. In fact, some form of protocol is needed, generally involving a central agent, and the procedures involved are not simpler nor any more efficient than those required for symmetric encryption (e.g., see analysis in [NEED78]).

This chapter and the next provide an overview of public-key cryptography. First, we look at its conceptual framework. Interestingly, the concept for this technique was developed and published before it was shown to be practical to adopt it. Next, we ex amine the RSA algorithm, which is the most important encryption/decryption algorithm that has been shown to be feasible for public-key encryption. Other important public-key cryptographic algorithms are covered in Chapter 10.

Much of the theory of public-key cryptosystems is based on number theory. If one is prepared to accept the results given in this chapter, an understanding of number theory is not strictly necessary. However, to gain a full appreciation of public-key algorithms, some understanding of number theory is required. Chapter 2 provides the necessary background in number theory. Table 9.1 defines some key terms

**PRINCIPLES OF PUBLIC-KEY CRYPTOSYSTEMS**

The concept of public-key cryptography evolved from an attempt to attack two of the most difficult problems associated with symmetric encryption. The first problem is that of key distribution, which is examined in some detail in Chapter 14.

As Chapter 14 discusses, key distribution under symmetric encryption requires either (1) that two communicants already share a key, which somehow has been dis tributed to them; or (2) the use of a key distribution centre. Whitfield Diffie, one of the discoverers of public-key encryption (along with Martin Hellman, both at Stanford University at the time), reasoned that this second requirement negated the very essence of cryptography: the ability to maintain total secrecy over your own communication. As Diffie put it [DIFF88], “what good would it do after all to develop impenetrable cryptosystems, if their users were forced to share their keys with a KDC that could be compromised by either burglary or subpoena?”

\

The second problem that Diffie pondered, and one that was apparently un related to the first, was that of digital signatures. If the use of cryptography was to become widespread, not just in military situations but for commercial and private purposes, then electronic messages and documents would need the equivalent of signatures used in paper documents. That is, could a method be devised that would stipulate, to the satisfaction of all parties, that a digital message had been sent by a particular person? This is a somewhat broader requirement than that of authentication, and its characteristics and ramifications are explored in Chapter 13.

Diffie and Hellman achieved an astounding breakthrough in 1976 [DIFF76 a, b] by coming up with a method that addressed both problems and was radically different from all previous approaches to cryptography, going back over four millennia.

In the next subsection, we look at the overall framework for public-key cryptography. Then we examine the requirements for the encryption/decryption algorithm that is at the heart of the scheme.

**Public-Key Cryptosystems**

Asymmetric algorithms rely on one key for encryption and a different but related key for decryption. These algorithms have the following important characteristic.

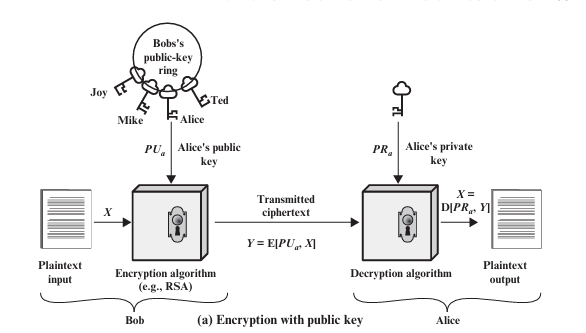
■ It is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key. In addition, some algorithms, such as RSA, also exhibit the following characteristic.

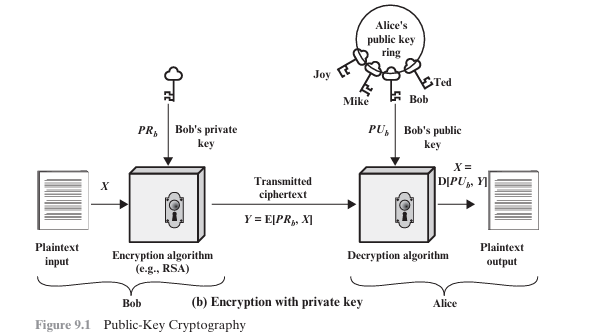
■ Either of the two related keys can be used for encryption, with the other used for decryption.

**A public-key encryption scheme** has six ingredients (Figure 9.1a; compare with Figure 3.1).

■ **Plaintext:** This is the readable message or data that is fed into the algorithm as input.

■ **Encryption algorithm:** The encryption algorithm performs various transformations on the plaintext.





**■ Public and private keys**: This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.

■ **Ciphertext:** This is the encrypted message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.

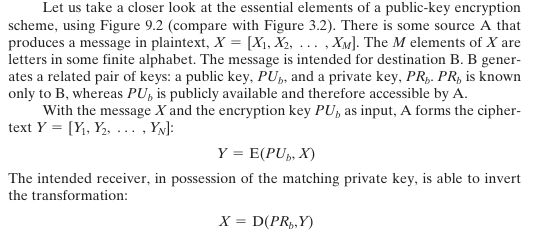
■ **Decryption algorithm:** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

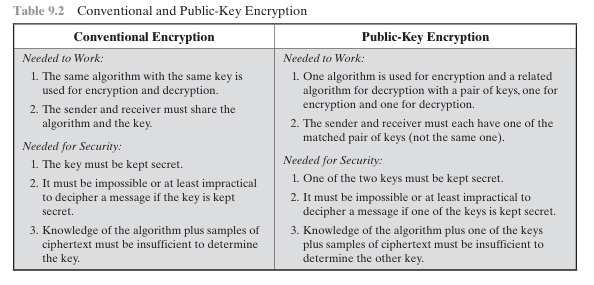
The essential steps are the following.

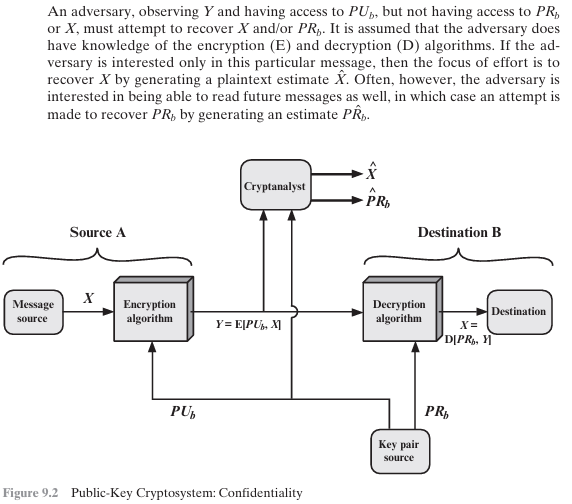
1. Each user generates a pair of keys to be used for the encryption and decryption of messages
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure 9.1a suggests, each user maintains a collection of public keys obtained from others.
3. If Bob wishes to send a confidential message to Alice, Bob encrypts the message using Alice’s public key.
4. When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows Alice’s private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed. As long as a user’s private key remains protected and secret, incoming communication is secure. At any time, a system can change its private key and publish the companion public key to replace its old public key.

Table 9.2 summarizes some of the important aspects of symmetric and public key encryption. To discriminate between the two, we refer to the key used in sym metric encryption as a secret key. The two keys used for asymmetric encryption are referred to as the public key and the private key.2 Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with symmetric encryption.





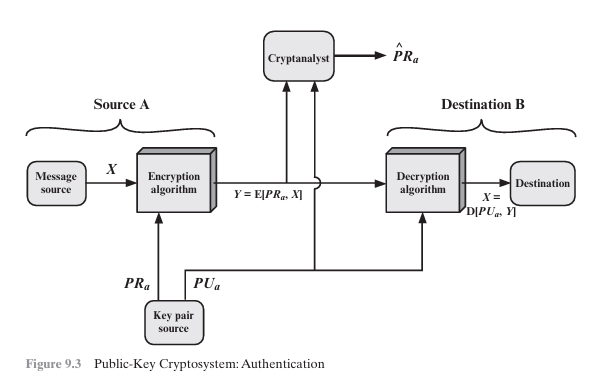


We mentioned earlier that either of the two related keys can be used for encryption, with the other being used for decryption. This enables a rather different cryptographic scheme to be implemented. Whereas the scheme illustrated in Figure 9.2 provides confidentiality, Figures 9.1b and 9.3 show the use of public-key encryption to provide authentication:



In this case, A prepares a message to B and encrypts it using A’s private key before transmitting it. B can decrypt the message using A’s public key. Because the message was encrypted using A’s private key, only A could have prepared the message. Therefore, the entire encrypted message serves as a **digital signature.** In addition, it is impossible to alter the message without access to A’s private key, so the message is authenticated both in terms of source and in terms of data integrity.

In the preceding scheme, the entire message is encrypted, which, although validating both author and contents, requires a great deal of storage. Each document must be kept in plaintext to be used for practical purposes. A copy also must be stored in ciphertext so that the origin and contents can be verified in case of a dispute. A more efficient way of achieving the same results is to encrypt a small block of bits that is a function of the document. Such a block, called an authenticator, must have the property that it is infeasible to change the document without changing the authenticator. If the authenticator is encrypted with the sender’s private key, it serves as a signature that verifies origin, content, and sequencing. Chapter 13 examines this technique in detail.



It is important to emphasize that the encryption process depicted in Figures 9.1b and 9.3 does not provide confidentiality. That is, the message being sent is safe from alteration but not from eavesdropping. This is obvious in the case of a signature based on a portion of the message, because the rest of the message is transmitted in the clear. Even in the case of complete encryption, as shown in Figure 9.3, there is no protection of confidentiality because any observer can decrypt the message by using the sender’s public key.

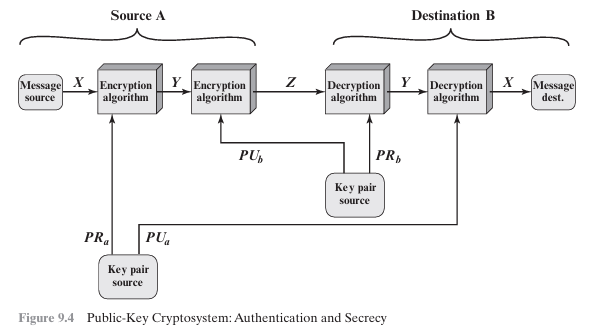
It is, however, possible to provide both the authentication function and confidentiality by a double use of the public-key scheme (Figure 9.4):



In this case, we begin as before by encrypting a message, using the sender’s private key. This provides the digital signature. Next, we encrypt again, using the receiver’s public key. The final ciphertext can be decrypted only by the intended receiver, who alone has the matching private key. Thus, confidentiality is provided. The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.

**Applications for Public-Key Cryptosystems**

Before proceeding, we need to clarify one aspect of public-key cryptosystems that is otherwise likely to lead to confusion. Public-key systems are characterized by the use of a cryptographic algorithm with two keys, one held private and one available publicly. Depending on the application, the sender uses either the sender’s private key or the receiver’s public key, or both, to perform some type of cryptographic



function. In broad terms, we can classify the use of public-key cryptosystems into three categories

■ **Encryption/decryption:** The sender encrypts a message with the recipient’s public key, and the recipient decrypts the message with the recipient’s private key.

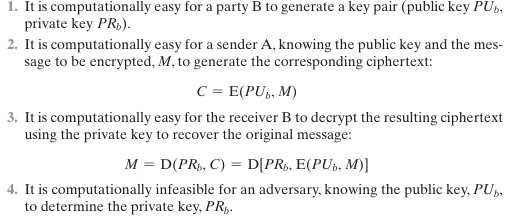
■ **Digital signature:** The sender “signs” a message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.

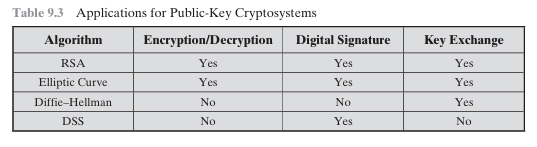
■ **Key exchange:** Two sides cooperate to exchange a session key, which is a secret key for symmetric encryption generated for use for a particular transaction (or session) and valid for a short period of time. Several different approaches are possible, involving the private key(s) of one or both parties; this is discussed in Chapter 10.

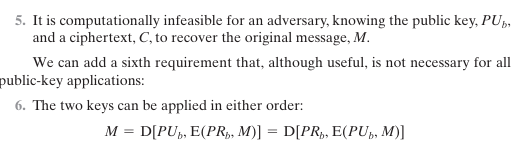
Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. Table 9.3 indicates the applications supported by the algorithms discussed in this book.

**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 fulfil [DIFF76b].







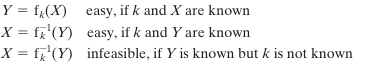
These are formidable requirements, as evidenced by the fact that only a few algorithms (RSA, elliptic curve cryptography, Diffie–Hellman, DSS) have received widespread acceptance in the several decades since the concept of public-key cryptography was proposed. Before elaborating on why the requirements are so formidable, let us first re cast them. The requirements boil down to the need for a trap-door one-way function. **A one-way function3** 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:

Y = f(X) easy

X = f-1(Y) infeasible

Generally, easy is defined to mean a problem that can be solved in polynomial time as a function of input length. Thus, if the length of the input is n bits, then the time to compute the function is proportional to n a, where a is a fixed constant. Such algorithms are said to belong to the class P. The term infeasible is a much fuzzier concept. In general, we can say a problem is infeasible if the effort to solve it grows faster than polynomial time as a function of input size. For example, if the length of the input is n bits and the time to compute the function is proportional to 2n, the problem is considered infeasible. Unfortunately, it is difficult to determine if a particular algorithm exhibits this complexity. Furthermore, traditional notions of computational complexity focus on the worst-case or average-case complexity of an algorithm. These measures are inadequate for cryptography, which requires that it be infeasible to invert a function for virtually all inputs, not for the worst case or even average case. A brief introduction to some of these concepts is provided in Appendix W.

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



Thus, the development of a practical public-key scheme depends on discovery of a suitable trap-door one-way function.

**Public-Key Cryptanalysis**

As with symmetric encryption, a public-key encryption scheme is vulnerable to a brute-force attack. The countermeasure is the same: Use large keys. However, there is a tradeoff to be considered. Public-key systems depend on the use of some sort of invertible mathematical function. The complexity of calculating these functions may not scale linearly with the number of bits in the key but grow more rapidly than that. Thus, the key size must be large enough to make brute-force attack impractical but small enough for practical encryption and decryption. In practice, the key sizes that have been proposed do make brute-force attack impractical but result in encryption/decryption speeds that are too slow for general-purpose use. Instead, as was mentioned earlier, public-key encryption is currently confined to key management and signature applications.

Another form of attack is to find some way to compute the private key given the public key. To date, it has not been mathematically proven that this form of attack is infeasible for a particular public-key algorithm. Thus, any given algorithm, including the widely used RSA algorithm, is suspect. The history of cryptanalysis shows that a problem that seems insoluble from one perspective can be found to have a solution if looked at in an entirely different way.

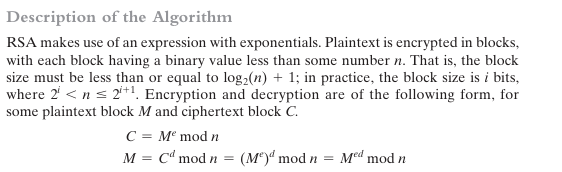
Finally, there is a form of attack that is peculiar to public-key systems. This is, in essence, a probable-message attack. Suppose, for example, that a message were to be sent that consisted solely of a 56-bit DES key. An adversary could encrypt all possible 56-bit DES keys using the public key and could discover the encrypted key by matching the transmitted ciphertext. Thus, no matter how large the key size of the public-key scheme, the attack is reduced to a brute-force attack on a 56-bit key. This attack can be thwarted by appending some random bits to such simple messages.

**THE RSA ALGORITHM**

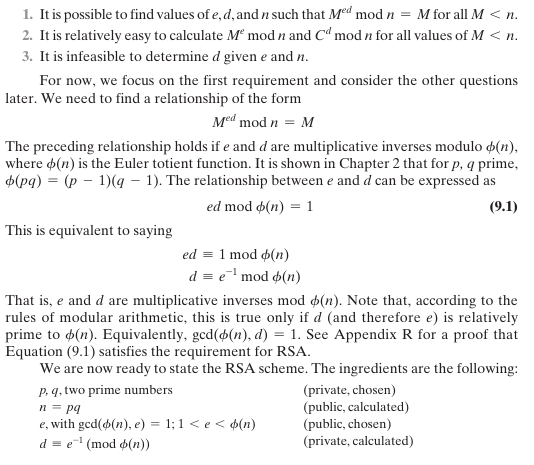
The pioneering paper by Diffie and Hellman [DIFF76b] introduced a new approach to cryptography and, in effect, challenged cryptologists to come up with a crypto graphic algorithm that met the requirements for public-key systems. A number of algorithms have been proposed for public-key cryptography. Some of these, though initially promising, turned out to be breakable.

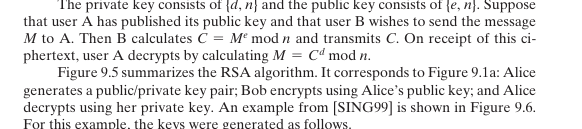
One of the first successful responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78].5 The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose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

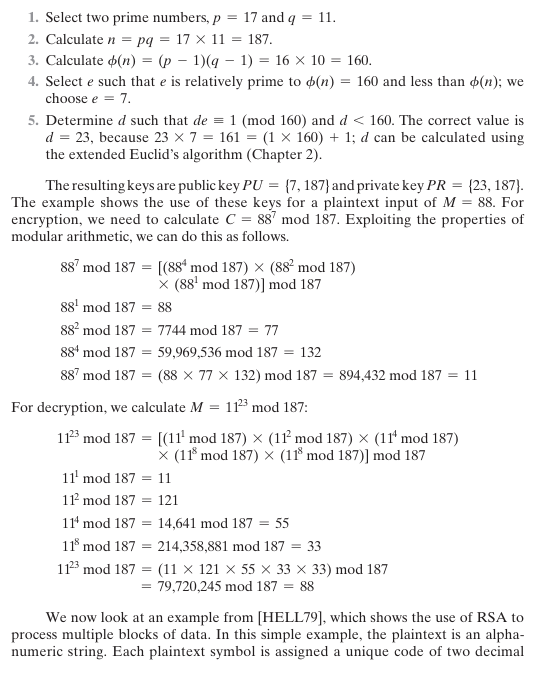
The RSA scheme is a cipher in which the plaintext and ciphertext are integers between 0 and n- 1 for some n. A typical size for n is 1024 bits, or 309 decimal digits. That is, n is less than 21024. We examine RSA in this section in some detail, beginning with an explanation of the algorithm. Then we examine some of the computational and cryptanalytical implications of RSA.

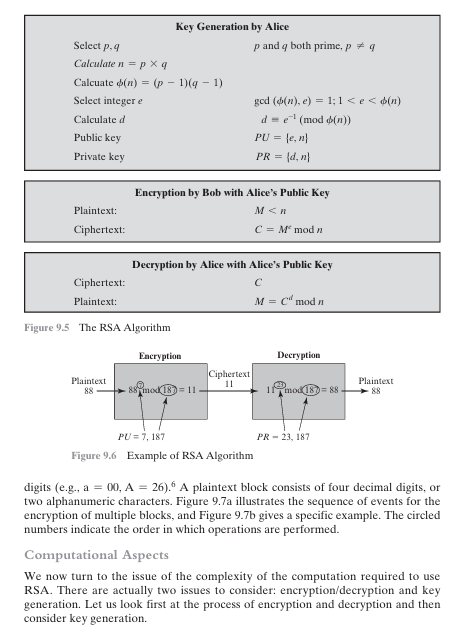


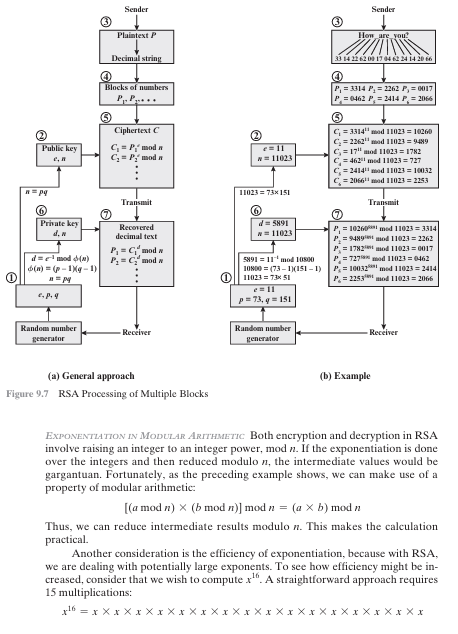
Both sender and receiver must know the value of n. The sender knows the value of e, and only the receiver knows the value of d. Thus, this is a public key encryption algorithm with a public key of PU = {e, n} and a private key of PR = {d, n}. For this algorithm to be satisfactory for public-key encryption, the following requirements must be met.

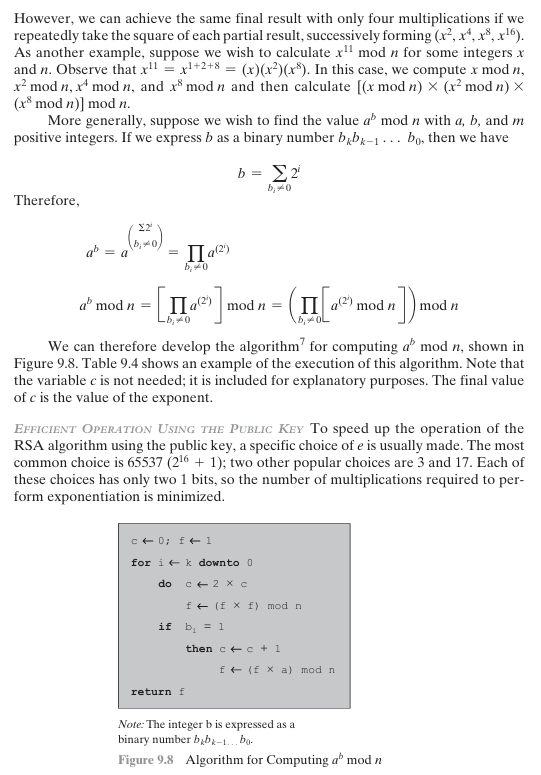


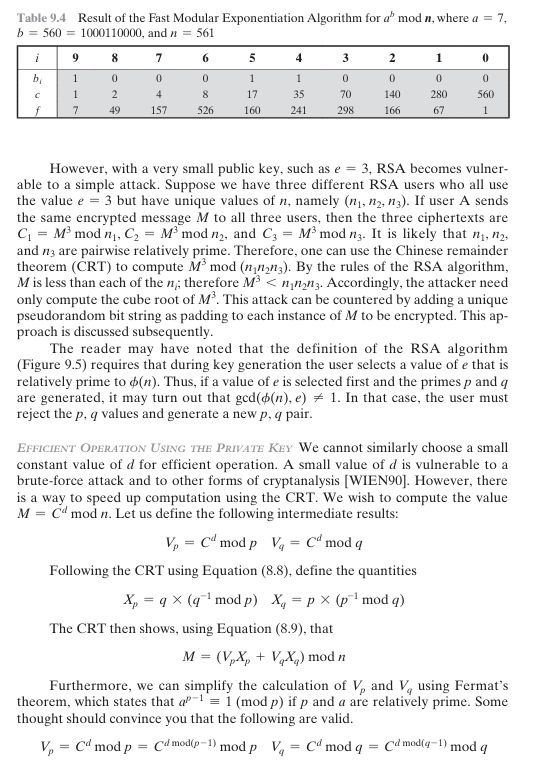












The quantities d mod (p- 1) and d mod (q- 1) can be precalculated. The end result is that the calculation is approximately four times as fast as evaluating M = Cd mod n directly [BONE02].

**KEY GENERATION**

Before the application of the public-key cryptosystem, each participant must generate a pair of keys. This involves the following tasks.

■ Determining two prime numbers, p and q.

■ Selecting either e or d and calculating the other.

First, consider the selection of p and q. Because the value of n = pq will be known to any potential adversary, in order to prevent the discovery of p and q by exhaustive methods, these primes must be chosen from a sufficiently large set (i.e., p and q must be large numbers). On the other hand, the method used for finding large primes must be reasonably efficient.

At present, there are no useful techniques that yield arbitrarily large primes, so some other means of tackling the problem is needed. The procedure that is generally used is to pick at random an odd number of the desired order of magnitude and test whether that number is prime. If not, pick successive random numbers until one is found that tests prime.

A variety of tests for primality have been developed (e.g., see [KNUT98] for a description of a number of such tests). Almost invariably, the tests are probabilistic. That is, the test will merely determine that a given integer is probably prime. Despite this lack of certainty, these tests can be run in such a way as to make the probability as close to 1.0 as desired. As an example, one of the more efficient and popular algorithms, the Miller–Rabin algorithm, is described in Chapter 2. With this algorithm and most such algorithms, the procedure for testing whether a given integer n is prime is to perform some calculation that involves n and a randomly chosen integer a. If n “fails” the test, then n is not prime. If n “passes” the test, then n may be prime or nonprime. If n passes many such tests with many different randomly chosen values for a, then we can have high confidence that n is, in fact, prime.

In summary, the procedure for picking a prime number is as follows.

1. Pick an odd integer n at random (e.g., using a pseudorandom number generator).
2. Pick an integer a 6 n at random.
3. Perform the probabilistic primality test, such as Miller–Rabin, with a as a parameter. If n fails the test, reject the value n and go to step 1.
4. If n has passed a sufficient number of tests, accept n; otherwise, go to step 2.

This is a somewhat tedious procedure. However, remember that this process is per formed relatively infrequently: only when a new pair (PU, PR) is needed.

It is worth noting how many numbers are likely to be rejected before a prime number is found. 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. Actually, because all even integers can be immediately rejected, the correct figure is ln(N)/2. For example, if a prime on the order of magnitude of 2200 were sought, then about ln(2200)/2 = 70 trials would be needed to find a prime.

Having determined prime numbers p and q, the process of key generation is completed by selecting a value of e and calculating d or, alternatively, selecting a value of d and calculating e. Assuming the former, then we need to select an e such that gcd(f(n), e) = 1 and then calculate d K e-1 (mod f(n)). Fortunately, there is a single algorithm that will, at the same time, calculate the greatest common divisor of two integers and, if the gcd is 1, determine the inverse of one of the integers modulo the other. The algorithm, referred to as the extended Euclid’s algorithm, is explained in Chapter 2. Thus, the procedure is to generate a series of random numbers, testing each against f(n) until a number relatively prime to f(n) is found. Again, we can ask the question: How many random numbers must we test to find a usable number, that is, a number relatively prime to f(n)? It can be shown easily that the probability that two random numbers are relatively prime is about 0.6; thus, very few tests would be needed to find a suitable integer (see Problem 2.18).

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 defense against the brute-force approach is the same for RSA as for other cryptosystems, namely, to use a large key space. Thus, the larger the number of bits in d, the better. However, because the calculations involved, both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run.

In this subsection, we provide an overview of mathematical and timing attacks.

**THE FACTORING PROBLEM** We can identify three approaches to attacking RSA mathematically.

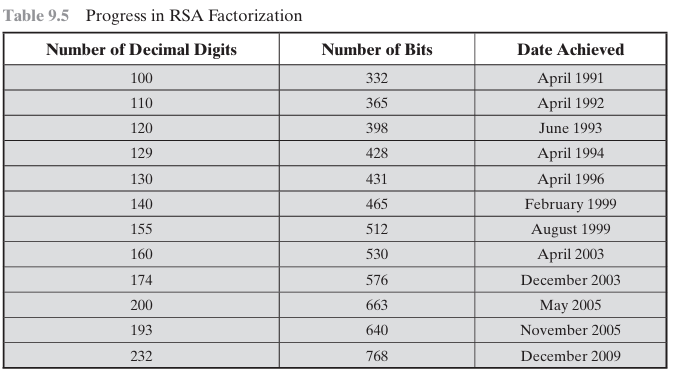
1. Factor n into its two prime factors. This enables calculation of f(n) = (p- 1) \* (q- 1), which in turn enables determination of d K e-1 (mod f(n)).
2. Determine f(n) directly, without first determining p and q. Again, this enables determination of d K e-1 (mod f(n)).
3. Determine d directly, without first determining f(n).

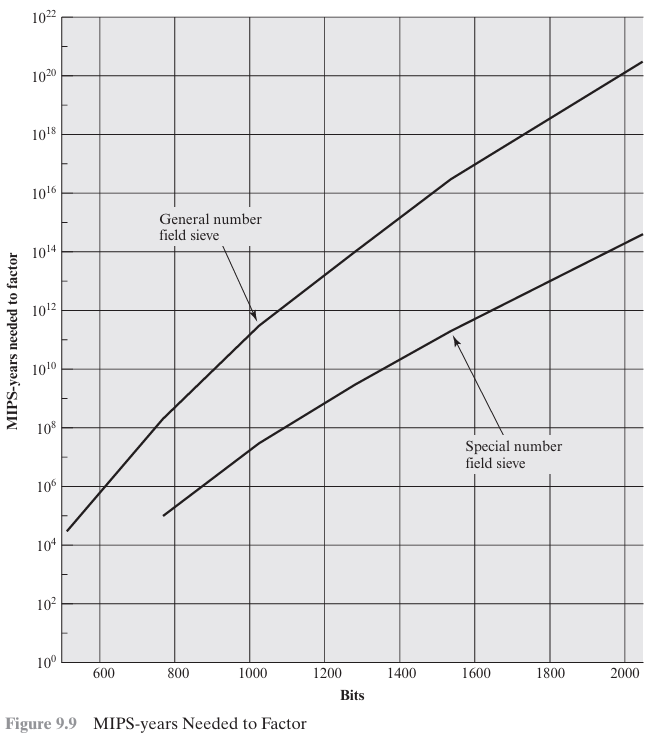
Most discussions of the cryptanalysis of RSA have focused on the task of factoring n into its two prime factors. Determining f(n) given n is equivalent to factoring n [RIBE96]. With presently known algorithms, determining d given e and n appears to be at least as time-consuming as the factoring problem [KALI95]. Hence, we can use factoring performance as a benchmark against which to evaluate the security of RSA.

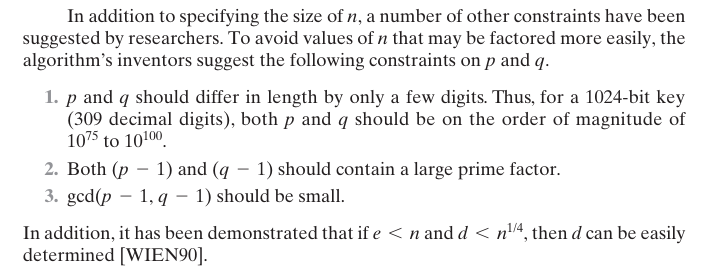
For a large n with large prime factors, factoring is a hard problem, but it is not as hard as it used to be. A striking illustration of this is the following. In 1977, the three inventors of RSA dared Scientific American readers to decode a cipher they printed in Martin Gardner’s “Mathematical Games” column [GARD77]. They of fered a $100 reward for the return of a plaintext sentence, an event they predicted might not occur for some 40 quadrillion years. In April of 1994, a group working over the Internet claimed the prize after only eight months of work [LEUT94]. This challenge used a public key size (length of n) of 129 decimal digits, or around 428 bits. In the meantime, just as they had done for DES, RSA Laboratories had issued challenges for the RSA cipher with key sizes of 100, 110, 120, and so on, digits. The latest challenge to be met is the RSA-768 challenge with a key length of 232 decimal digits, or 768 bits. Table 9.5 shows the results.

A striking fact about the progress reflected in Table 9.5 concerns the method used. Until the mid-1990s, factoring attacks were made using an approach known as the quadratic sieve. The attack on RSA-130 used a newer algorithm, the gen realized number field sieve (GNFS), and was able to factor a larger number than RSA-129 at only 20% of the computing effort.

The threat to larger key sizes is twofold: the continuing increase in computing power and the continuing refinement of factoring algorithms. We have seen that the move to a different algorithm resulted in a tremendous speedup. We can expect further refinements in the GNFS, and the use of an even better algorithm is also a possibility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve. Figure 9.9 compares the performance of the two algorithms. It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better [ODLY95]. Thus, we need to be careful in choosing a key size for RSA. The team that produced the 768-bit factorization [KLEI10] observed that factoring a 1024-bit RSA modulus would be about a thousand times harder than factoring a 768-bit modulus, and a 768-bit RSA modulus is several thousands times harder to factor than a 512-bit one. Based on the amount of time between the 512-bit and 768-bit factorization successes, the team felt it to be reasonable to expect that the 1024-bit RSA moduli could be factored well within the next decade by a similar academic effort. Thus, they recommended phasing out usage of 1024-bit RSA within the next few years (from 2010).







**TIMING ATTACKS** If one needed yet another lesson about how difficult it is to assess the security of a cryptographic algorithm, the appearance of timing attacks provides a stunning one. Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer takes to decipher messages [KOCH96, KALI96b]. Timing attacks are applicable not just to RSA, but to other public-key cryptography systems. This attack is alarming for two reasons: It comes from a completely unexpected direction, and it is a ciphertext only attack.

A **timing attack** is somewhat analogous to a burglar guessing the combi nation of a safe by observing how long it takes for someone to turn the dial from number to number. We can explain the attack using the modular exponentiation algorithm of Figure 9.8, but the attack can be adapted to work with any implementation that does not run in fixed time. In this algorithm, modular exponentiation is accomplished bit by bit, with one modular multiplication per formed at each iteration and an additional modular multiplication performed for each 1 bit.

As Kocher points out in his paper, the attack is simplest to understand in an extreme case. Suppose the target system uses a modular multiplication function that is very fast in almost all cases but in a few cases takes much more time than an entire average modular exponentiation. The attack proceeds bit-by-bit starting with the leftmost bit, bk. Suppose that the first j bits are known (to obtain the entire exponent, start with j = 0 and repeat the attack until the entire exponent is known). For a given ciphertext, the attacker can complete the first j iterations of the for loop. The operation of the subsequent step depends on the unknown exponent bit. If the bit is set, d d (d \* a) mod n will be executed. For a few values of a and d, the modular multiplication will be extremely slow, and the attacker knows which these are. Therefore, if the observed time to execute the decryption algorithm is always slow when this particular iteration is slow with a 1 bit, then this bit is assumed to be 1. If a number of observed execution times for the entire algorithm are fast, then this bit is assumed to be 0.

In practice, modular exponentiation implementations do not have such extreme timing variations, in which the execution time of a single iteration can exceed the mean execution time of the entire algorithm. Nevertheless, there is enough variation to make this attack practical. For details, see [KOCH96]

Although the timing attack is a serious threat, there are simple countermeasures that can be used, including the following.

■ **Constant exponentiation time:** Ensure that all exponentiations take the same amount of time before returning a result. This is a simple fix but does degrade performance.

■ **Random delay:** Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack. Kocher points out that if defenders don’t add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.

■ **Blinding:** Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what cipher text bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.

RSA Data Security incorporates a blinding feature into some of its products. The private-key operation M = Cd mod n is implemented as follows.

1. Generate a secret random number r between 0 and n- 1.
2. Compute C′ = C(re) mod n, where e is the public exponent.
3. Compute M′ = (C′)d mod n with the ordinary RSA implementation.
4. Compute M = M′r-1 mod n. In this equation, r-1 is the multiplicative inverse of r mod n; see Chapter 2 for a discussion of this concept. It can be demon started that this is the correct result by observing that red mod n = r mod n.

RSA Data Security reports a 2 to 10% performance penalty for blinding.

**FAULT-BASED ATTACK** Still another unorthodox approach to attacking RSA is re ported in [PELL10]. The approach is an attack on a processor that is generating RSA digital signatures. The attack induces faults in the signature computation by reducing the power to the processor. The faults cause the software to produce in valid signatures, which can then be analysed by the attacker to recover the private key. The authors show how such an analysis can be done and then demonstrate it by extracting a 1024-bit private RSA key in approximately 100 hours, using a commercially available microprocessor.

The attack algorithm involves inducing single-bit errors and observing the results. The details are provided in [PELL10], which also references other proposed hardware fault-based attacks against RSA. This attack, while worthy of consideration, does not appear to be a serious threat to RSA.

It requires that the attacker have physical access to the target ma chine and that the attacker is able to directly control the input power to the processor. Controlling the input power would for most hardware require more than simply controlling the AC power, but would also involve the power supply control hardware on the chip

**CHOSEN CIPHERTEXT ATTACK AND OPTIMAL ASYMMETRIC ENCRYPTION PADDING**

The basic RSA algorithm is vulnerable to a chosen ciphertext attack (CCA). CCA is defined as an attack in which the adversary chooses a number of ciphertexts and is then given the corresponding plaintexts, decrypted with the target’s private key. Thus, the adversary could select a plaintext, encrypt it with the target’s public key, and then be able to get the plaintext back by having it decrypted with the private key. Clearly, this provides the adversary with no new information. Instead, the ad versary exploits properties of RSA and selects blocks of data that, when processed using the target’s private key, yield information needed for cryptanalysis.

A simple example of a CCA against RSA takes advantage of the following property of RSA:

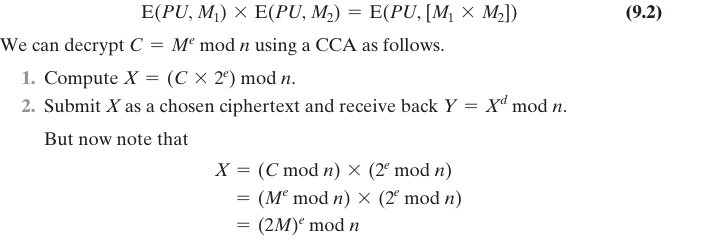
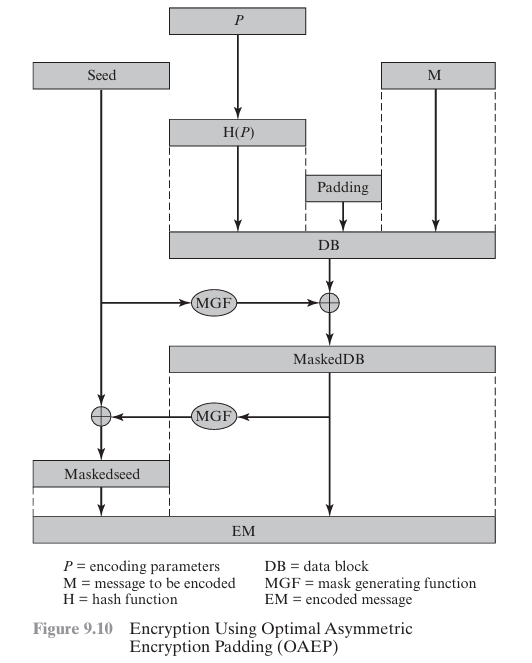
 Therefore, Y = (2M) mod n. From this, we can deduce M. To overcome this simple attack, practical RSA-based cryptosystems randomly pad the plaintext prior to encryption. This randomizes the ciphertext so that Equation (9.2) no longer holds. However, more sophisticated CCAs are possible, and a simple padding with a random value has been shown to be insufficient to provide the desired security. To counter such attacks, RSA Security Inc., a leading RSA vendor and former holder of the RSA patent, recommends modifying the plaintext using a procedure known as optimal asymmetric encryption padding (OAEP). A full discussion of the threats and OAEP are beyond our scope; see [POIN02] for an introduction and [BELL94] for a thorough analysis. Here, we simply summarize the OAEP procedure.

Figure 9.10 depicts OAEP encryption. As a first step, the message M to been crypted is padded. A set of optional parameters, P, is passed through a hash function, H.8 The output is then padded with zeros to get the desired length in the overall data block (DB). Next, a random seed is generated and passed through another hash function, called the mask generating function (MGF). The resulting hash value is bit by-bit XORed with DB to produce a maskedDB. The maskedDB is in turn passed through the MGF to form a hash that is XORed with the seed to produce the masked seed. The concatenation of the maskedSeed and the maskedDB forms the encoded message EM. Note that the EM includes the padded message, masked by the seed, and the seed, masked by the maskedDB. The EM is then encrypted using RSA.



**Diffie-Hellman key exchange**

This chapter begins with a description of one of the earliest and simplest PKCS: Diffie–Hellman key exchange. The chapter then looks at another important scheme, the Elgamal PKCS. Next, we look at the increasingly important PKCS known as elliptic curve cryptography. Finally, the use of public-key algorithms for pseudorandom num ber generation is examined.

**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. Briefly, we can define the discrete logarithm in the following way. Recall from Chapter 2 that a primitive root of a prime number p is one whose powers modulo p generate all the integers from 1 to p- 1. That is, if



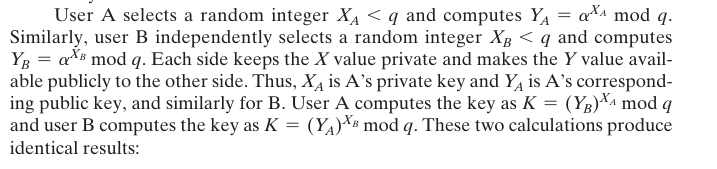
are distinct and consist of the integers from 1 through p- 1 in some permutation. For any integer b and a primitive root a of prime number p, we can find a unique exponent i such that

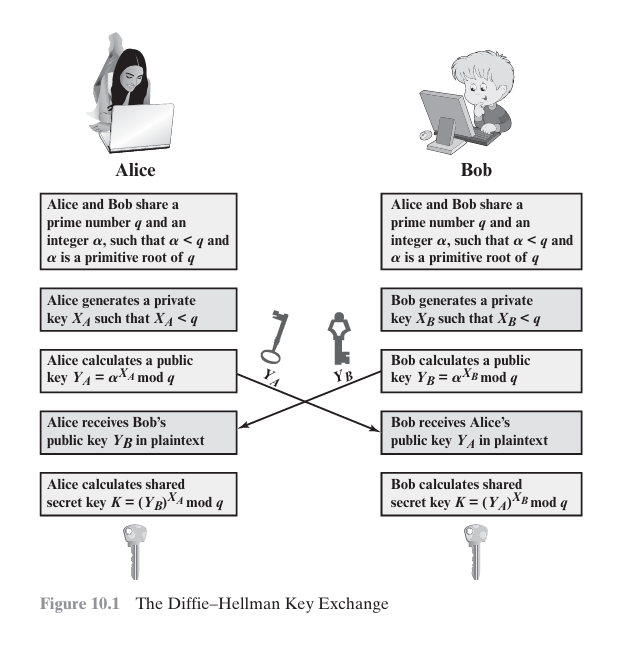


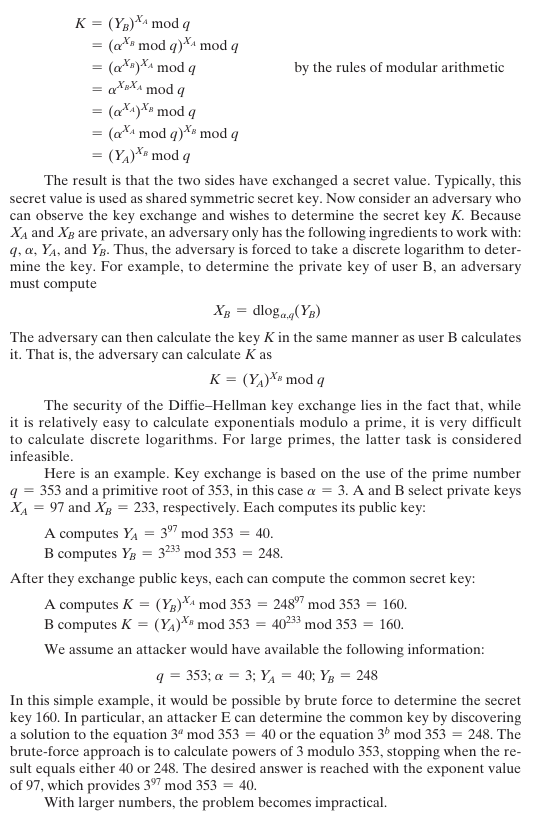
The exponent i is referred to as the discrete logarithm of b for the base a, mod p. We express this value as dloga,p(b). See Chapter 2 for an extended discussion of discrete logarithms.

**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 a that is a primitive root of q. Suppose the users A and B wish to create a shared key.



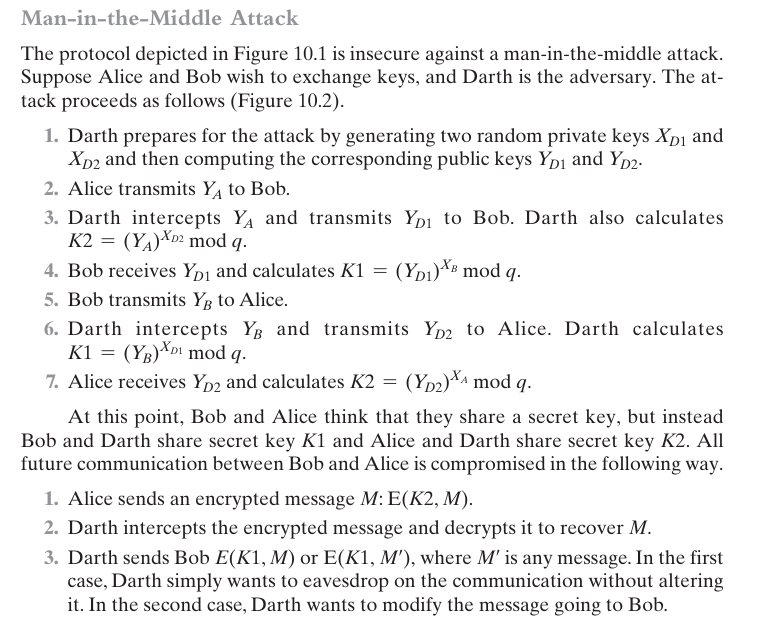


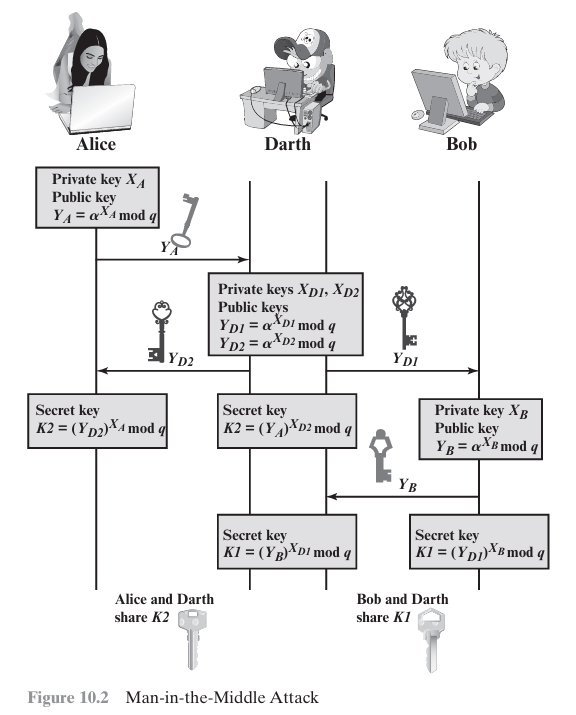


**Key Exchange Protocols**

Figure 10.1 shows a simple protocol that makes use of the Diffie–Hellman calculation. Suppose that user A wishes to set up a connection with user B and use a secret key to encrypt messages on that connection. User A can generate a one-time private key XA, calculate YA, and send that to user B. User B responds by generating a private value XB, calculating YB, and sending YB to user A. Both users can now calculate the key. The necessary public values q and a would need to be known ahead of time. Alternatively, user A could pick values for q and a and include those in the first message.

As an example of another use of the Diffie–Hellman algorithm, suppose that a group of users (e.g., all users on a LAN) each generate a long-lasting private value Xi (for user i) and calculate a public value Yi. These public values, together with global public values for q and a, are stored in some central directory. At any time, user j can access user i’s public value, calculate a secret key, and use that to send an encrypted message to user A. If the central directory is trusted, then this form of communication provides both confidentiality and a degree of authentication. Because only i and j can determine the key, no other user can read the message (confidential ity). Recipient i knows that only user j could have created a message using this key (authentication). However, the technique does not protect against replay attacks.





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; these topics are explored in Chapters 13 and 14

**ELLIPTIC CURVE CRYPTOGRAPHY**

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple addition is the counterpart of modular exponentiation. To form a cryptographic system using elliptic curves, we need to find a “hard problem” corresponding to factoring the product of two primes or taking the discrete logarithm.

Consider the equation Q = kP where Q, P ∈ EP(a, b) and k 6 p. It is relatively easy to calculate Q given k and P, but it is hard to determine k given Q and P. This is called the discrete logarithm problem for elliptic curves.

We give an example taken from the Certicom Web site (www.certicom. com). Consider the group E23(9,17). This is the group defined by the equation y2 mod 23 = (x3 + 9x + 17) mod 23. What is the discrete logarithm k of Q = (4, 5) to the base P = (16, 5)? The brute-force method is to compute multiples of P until Q is found. Thus,

P = (16,5); 2P = (20, 20); 3P = (14, 14); 4P = (19, 20); 5P = (13, 10);

6P = (7, 3); 7P = (8, 7); 8P = (12, 17); 9P = (4, 5).

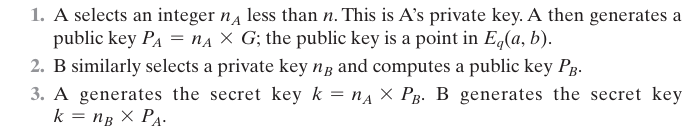
Because 9P = (4, 5) = Q, the discrete logarithm Q = (4, 5) to the base P = (16, 5) is k = 9. In a real application, k would be so large as to make the brute force approach infeasible.

In the remainder of this section, we show two approaches to ECC that give the flavour of this technique.

**Analog of Diffie–Hellman Key Exchange**

Key exchange using elliptic curves can be done in the following manner. First pick a large integer q, which is either a prime number p or an integer of the form 2m, and elliptic curve parameters a and b for Equation (10.5) or Equation (10.7). This defines the elliptic group of points Eq(a, b). Next, pick a base point G = (x1, y1) in Ep(a, b) whose order is a very large value n. The order n of a point G on an elliptic curve is the smallest positive integer n such that nG = 0 and G are parameters of the cryptosystem known to all participants.

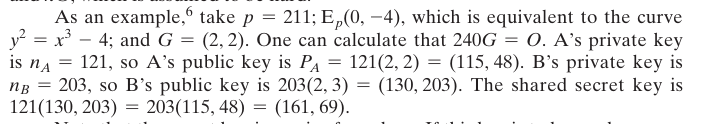
A key exchange between users A and B can be accomplished as follows (Figure 10.7)



The two calculations in step 3 produce the same result because

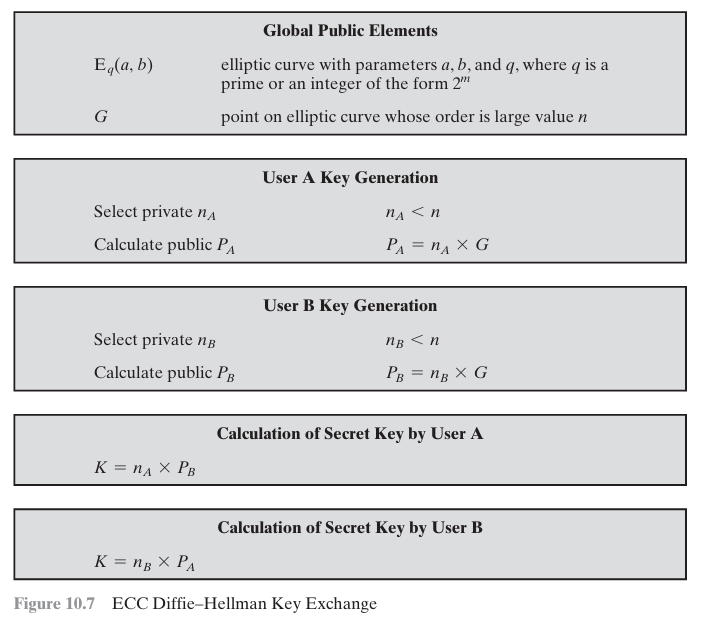


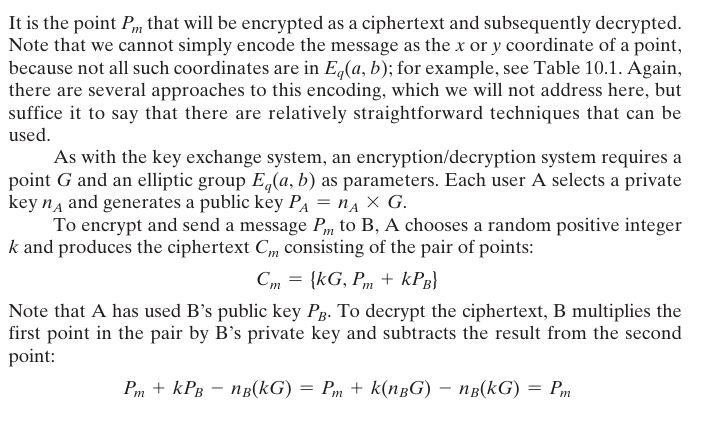
To break this scheme, an attacker would need to be able to compute k given G and kG, which is assumed to be hard.

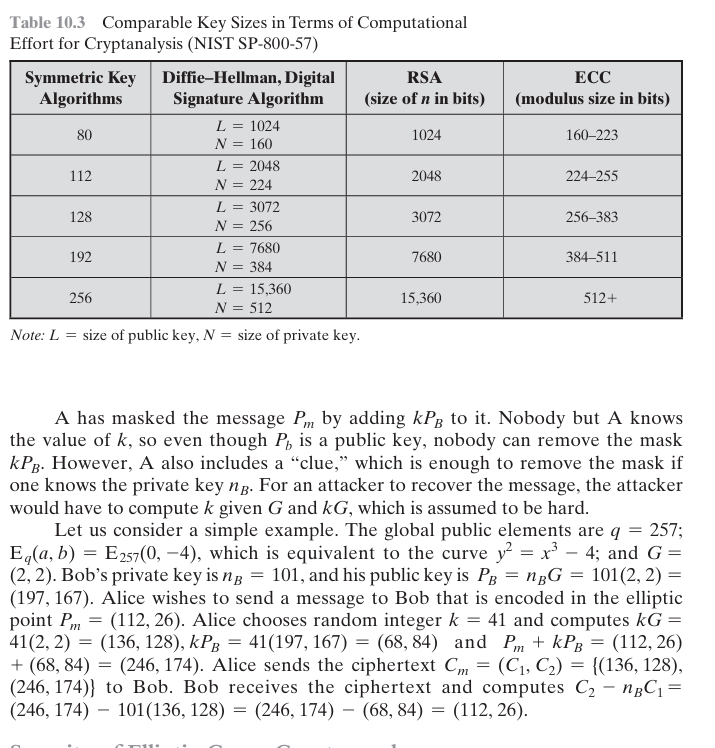


**Elliptic Curve Encryption/Decryption**

Several approaches to encryption/decryption using elliptic curves have been ana lyzed in the literature. In this subsection, we look at perhaps the simplest. The first task in this system is to encode the plaintext message m to be sent as an (x, y) point Pm.







**Security of Elliptic Curve Cryptography**

The security of ECC depends on how difficult it is to determine k given kP and P. This is referred to as the elliptic curve logarithm problem. The fastest known technique for taking the elliptic curve logarithm is known as the Pollard rho method. Table 10.3, from NIST SP 800-57 (Recommendation for Key Management—Part 1: General, September 2015), compares various algorithms by showing comparable key sizes in terms of computational effort for cryptanalysis. As can be seen, a considerably smaller key size can be used for ECC compared to RSA.

Based on this analysis, SP 800-57 recommends that at least through 2030, acceptable key lengths are from 3072 to 14,360 bits for RSA and 256 to 512 bits for ECC. Similarly, the European Union Agency for Network and Information Security (ENISA) recommends in their 2014 report (Algorithms, Key Size and Parameters report—2014, November 2014) minimum key lengths for future system of 3072 bits and 256 bits for RSA and ECC, respectively.

Analysis indicates that for equal key lengths, the computational effort re quired for ECC and RSA is comparable [JURI97]. Thus, there is a computational advantage to using ECC with a shorter key length than a comparably secure RSA.