Chapter Seven
Key Management; Other Public-Key Cryptosystems

7.1. Key Management

In Chapter 7, we examined the problem of the distribution of secret keys. One of the major roles of public-key encryption has been to address the problem of key distribution. There are actually two distinct aspects to the use of public-key cryptography in this regard:

- The distribution of public keys
- The use of public-key encryption to distribute secret keys

We examine each of these areas in turn.

7.1.1 Distribution of Public Keys

Several techniques have been proposed for the distribution of public keys. Virtually all these proposals can be grouped into the following general schemes:

- Public announcement
- Publicly available directory
- Public-key authority
- Public-key certificates
-

1. Public Announcement of Public Keys

On the face of it, the point of public-key encryption is that the public key is public. Thus, if there is some broadly accepted public-key algorithm, such as RSA, any participant can send his or her public key to any other participant or broadcast the key to the community at large (Figure 7.1).

Figure 7.1. Uncontrolled Public-Key Distribution

Although this approach is convenient, it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be user A and send a public key to another participant or broadcast such a public key. Until such time as user A discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for A and can use the forged keys for authentication.

2. Publicly Available Directory

A greater degree of security can be achieved by maintaining a publicly available dynamic directory of public keys. Maintenance and distribution of the public directory would have to be the responsibility of some trusted entity or organization (Figure 7.2). Such a scheme would include the following elements:

1. The authority maintains a directory with a {name, public key} entry for each participant.
2. Each participant registers a public key with the directory authority. Registration would have to be in person or by some form of secure authenticated communication.
3. A participant may replace the existing key with a new one at any time, either because of the desire to replace a public key that has already been used for a large amount of data, or because the corresponding private key has been compromised in some way.
4. Participants could also access the directory electronically. For this purpose, secure, authenticated communication from the authority to the participant is mandatory.

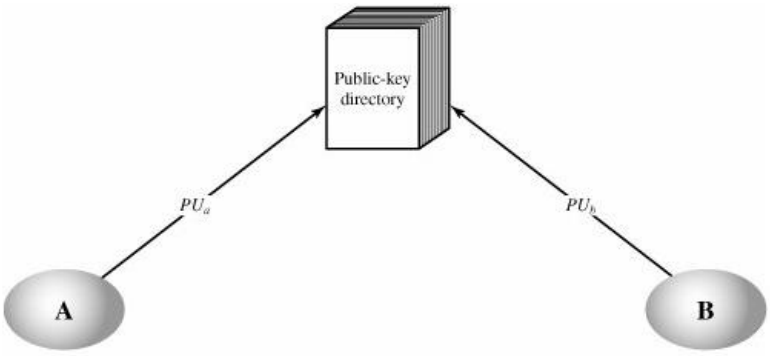


Figure 7.2. Public-Key Publication

This scheme is clearly more secure than individual public announcements but still has vulnerabilities. If an adversary succeeds in obtaining or computing the private key of the directory authority, the adversary could authoritatively pass out counterfeit public keys and subsequently impersonate any participant and eavesdrop on messages sent to any participant. Another way to achieve the same end is for the adversary to tamper with the records kept by the authority.

3. Public-Key Authority

Stronger security for public-key distribution can be achieved by providing tighter control over the distribution of public keys from the directory. A typical scenario is illustrated in Figure 7.3. As before, the scenario assumes that a central authority maintains a dynamic directory of public keys of all participants. In addition, each participant reliably knows a public key for the authority, with only the authority knowing the corresponding private key. The following steps (matched by number to Figure 7.3) occur:

1. A sends a timestamped message to the public-key authority containing a request for the current public key of B.

2. The authority responds with a message that is encrypted using the authority's private key, PR_{auth} Thus, A is able to decrypt the message using the authority's public key. Therefore, A is assured that the message originated with the authority. The message includes the following:

- B's public key, PU_b which A can use to encrypt messages destined for B
- The original request, to enable A to match this response with the corresponding earlier request and to verify that the original request was not altered before reception by the authority
- The original timestamp, so A can determine that this is not an old message from the authority containing a key other than B's current public key

3. A stores B's public key and also uses it to encrypt a message to B containing an identifier of A (ID_A) and a nonce (N_1), which is used to identify this transaction uniquely.

4, B retrieves A's public key from the authority in the same manner as A

5. retrieved B's public key.

At this point, public keys have been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

6. B sends a message to A encrypted with PU_a and containing A's nonce (N_1) as well as a new nonce generated by B (N_2) Because only B could have decrypted message (3), the presence of N_1 in message (6) assures A that the correspondent is B.

7. A returns N_2, encrypted using B's public key, to assure B that its correspondent is A.

]

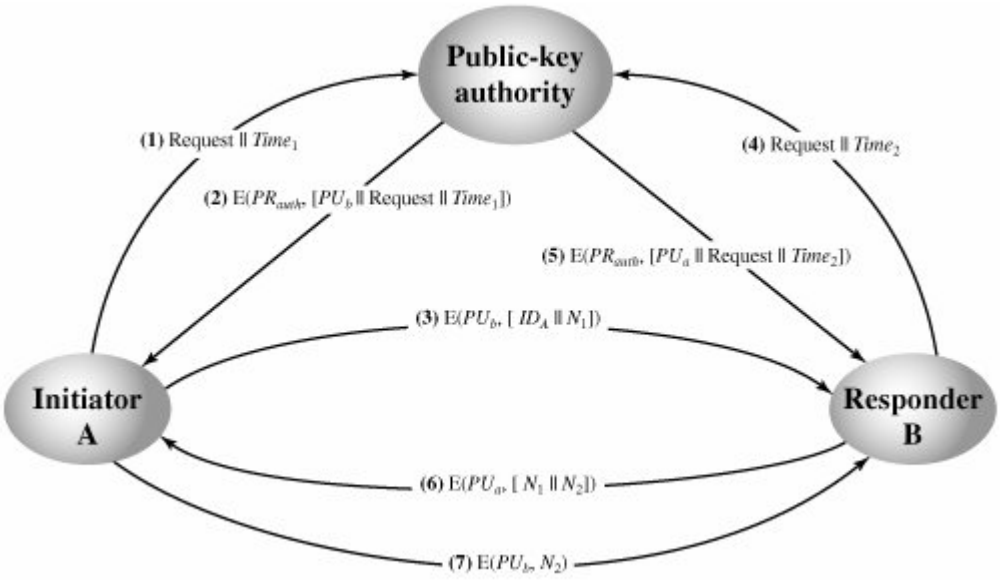


Figure 7.3. Public-Key Distribution Scenario

Thus, a total of seven messages are required. However, the initial four messages need be used only infrequently because both A and B can save the other's public key for future use, a technique known as caching. Periodically, a user should request fresh copies of the public keys of its correspondents to ensure currency.

4. Public-Key Certificates

The scenario of Figure 7.3 is attractive, yet it has some drawbacks. The public-key authority could be somewhat of a bottleneck in the system, for a user must appeal to the authority for a public key for every other user that it wishes to contact. As before, the directory of names and public keys maintained by the authority is vulnerable to tampering.

An alternative approach, first suggested by Kohnfelder, is to use certificates that can be used by participants to exchange keys without contacting a public-key authority, in a way that is as reliable as if the keys were obtained directly from a public-key authority. In essence, a certificate consists of a public key plus an identifier of the key owner, with the whole block signed by a trusted third party. Typically, the third party is a certificate authority, such as a government agency or a financial institution, that is trusted by the user community. A user can present his or her public key to the authority in a secure manner, and obtain a certificate. The user can then publish the certificate. Anyone needed this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signature. A participant can also convey its key information to another by transmitting its certificate. Other

participants can verify that the certificate was created by the authority. We can place the following requirements on this scheme:

1. Any participant can read a certificate to determine the name and public key of the certificate's owner.
2. Any participant can verify that the certificate originated from the certificate authority and is not counterfeit.
3. Only the certificate authority can create and update certificates.
4. Any participant can verify the currency of the certificate.

A certificate scheme is illustrated in Figure 7.4. Each participant applies to the certificate authority, supplying a public key and requesting a certificate.

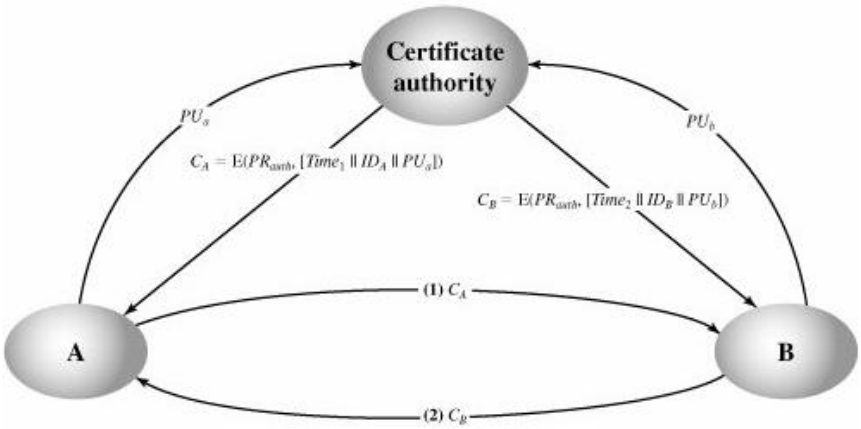


Figure 7.4. Exchange of Public-Key Certificates

Application must be in person or by some form of secure authenticated communication. For participant A, the authority provides a certificate of the form

$C_A = E(PR_{auth}, [T||ID_A||PU_a])$

where PR_{auth} is the private key used by the authority and T is a timestamp. A may then pass this certificate on to any other participant, who reads and verifies the certificate as follows:

$D(PU_{auth}, C_A) = D(PU_{auth}, E(PR_{auth}, [T||ID_A||PU_a])) = (T||ID_A||PU_a)$

The recipient uses the authority's public key, PU_{auth} to decrypt the certificate. Because the certificate is readable only using the authority's public key, this verifies that the certificate came from the certificate authority. The elements ID_A and PU_a provide the recipient with the name and public key of the certificate's holder. The timestamp T validates the currency of the certificate. The timestamp counters the following scenario. A's private key is learned by an adversary. A generates a new private/public key pair and applies to the certificate authority for a new certificate. Meanwhile, the adversary replays the old certificate to B. If B

then encrypts messages using the compromised old public key, the adversary can read those messages.

7.1.2 Distribution of Secret Keys Using Public-Key Cryptography

Once public keys have been distributed or have become accessible, secure communication that thwarts eavesdropping, tampering, or both is possible. However, few users will wish to make exclusive use of public-key encryption for communication because of the relatively slow data rates that can be achieved. Accordingly, public-key encryption provides for the distribution of secret keys to be used for conventional encryption.

1. Simple Secret Key Distribution

An extremely simple scheme was put forward by Merkle, as illustrated in Figure 7.5. If A wishes to communicate with B, the following procedure is employed:

1. A generates a public/private key pair $\{PU_a, PR_a\}$ and transmits a message to B consisting of PU_a and an identifier of A, ID_A.
2. B generates a secret key, K_s, and transmits it to A, encrypted with A's public key.
3. A computes $D(PR_a, E(PU_a, K_s))$ to recover the secret key. Because only A can decrypt the message, only A and B will know the identity of K_s.
4. A discards PU_a and PR_a and B discards PU_a.

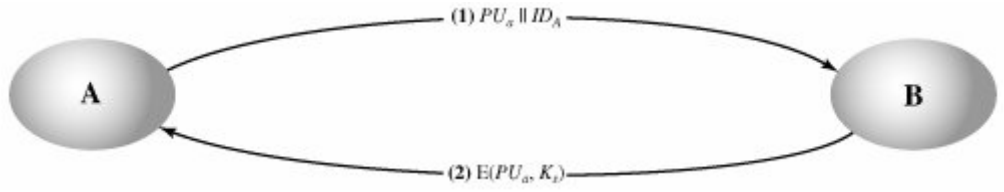


Figure 7.5. Simple Use of Public-Key Encryption to Establish a Session Key

A and B can now securely communicate using conventional encryption and the session key K_s. At the completion of the exchange, both A and B discard K_s. Despite its simplicity, this is an attractive protocol. No keys exist before the start of the communication and none exist after the completion of communication. Thus, the risk of compromise of the keys is minimal. At the same time, the communication is secure from eavesdropping.

The protocol depicted in Figure 7.5 is insecure against an adversary who can intercept messages and then either relay the intercepted message or substitute another message.

Such an attack is known as a **man-in-the-middle attack**. In this case, if an adversary, E, has control of the intervening communication channel, then E can compromise the communication in the following fashion without being detected:

1. A generates a public/private key pair $\{PU_a, PR_a\}$ and transmits a message intended for B consisting of PU_a and an identifier of A, ID_A.
2. E intercepts the message, creates its own public/private key pair $\{PU_e, PR_e\}$ and transmits $PU_e||ID_A$ to B.
3. B generates a secret key, K_s, and transmits $E(PU_e, K_s)$.
4. E intercepts the message, and learns K_s by computing $D(PR_e, E(PU_e, K_s))$.
5. E transmits $E(PU_a, K_s)$ to A.

The result is that both A and B know K_s and are unaware that K_s has also been revealed to E. A and B can now exchange messages using K_s E no longer actively interferes with the communications channel but simply eavesdrops. Knowing K_s E can decrypt all messages, and both A and B are unaware of the problem. Thus, this simple protocol is only useful in an environment where the only threat is eavesdropping.

2. Secret Key Distribution with Confidentiality and Authentication

Figure 7.6, based on an approach suggested, provides protection against both active and passive attacks. We begin at a point when it is assumed that A and B have exchanged public keys by one of the schemes described earlier in this section. Then the following steps occur:

1. A uses B's public key to encrypt a message to B containing an identifier of A (ID_A) and a nonce (N_1), which is used to identify this transaction uniquely.

2. B sends a message to A encrypted with PU_a and containing A's nonce (N_1) as well as a new nonce generated by B (N_2) Because only B could have decrypted message (1), the presence of N_1 in message (2) assures A that the correspondent is B.

[Page 297]

3. A returns N_2 encrypted using B's public key, to assure B that its correspondent is A.

4. A selects a secret key K_s and sends $M = E(PU_b, E(PR_a, K_s))$ to B. Encryption of this message with B's public key ensures that only B can read it; encryption with A's private key ensures that only A could have sent it.

5. B computes $D(PU_a, D(PR_b, M))$ to recover the secret key.

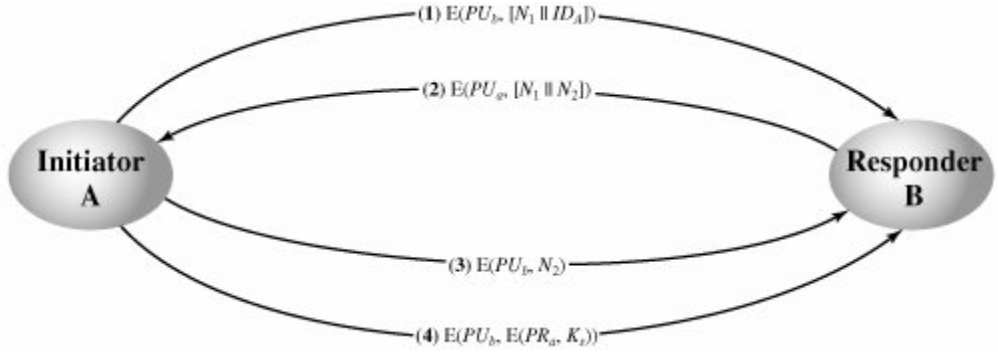


Figure 10.6. Public-Key Distribution of Secret Keys

Notice that the first three steps of this scheme are the same as the last three steps of Figure 7.3. The result is that this scheme ensures both confidentiality and authentication in the exchange of a secret key.

7.2. Diffie-Hellman Key Exchange

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography and is generally referred to as Diffie-Hellman key exchange. A number of commercial products employ this key exchange technique.

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. First, we define a primitive root of a prime number p as one whose powers modulo p generate all the integers from 1 to p 1. That is, if a is a primitive root of the prime number p, then the numbers

a mod p, a^2 mod p,..., a^{p-1} mod p

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

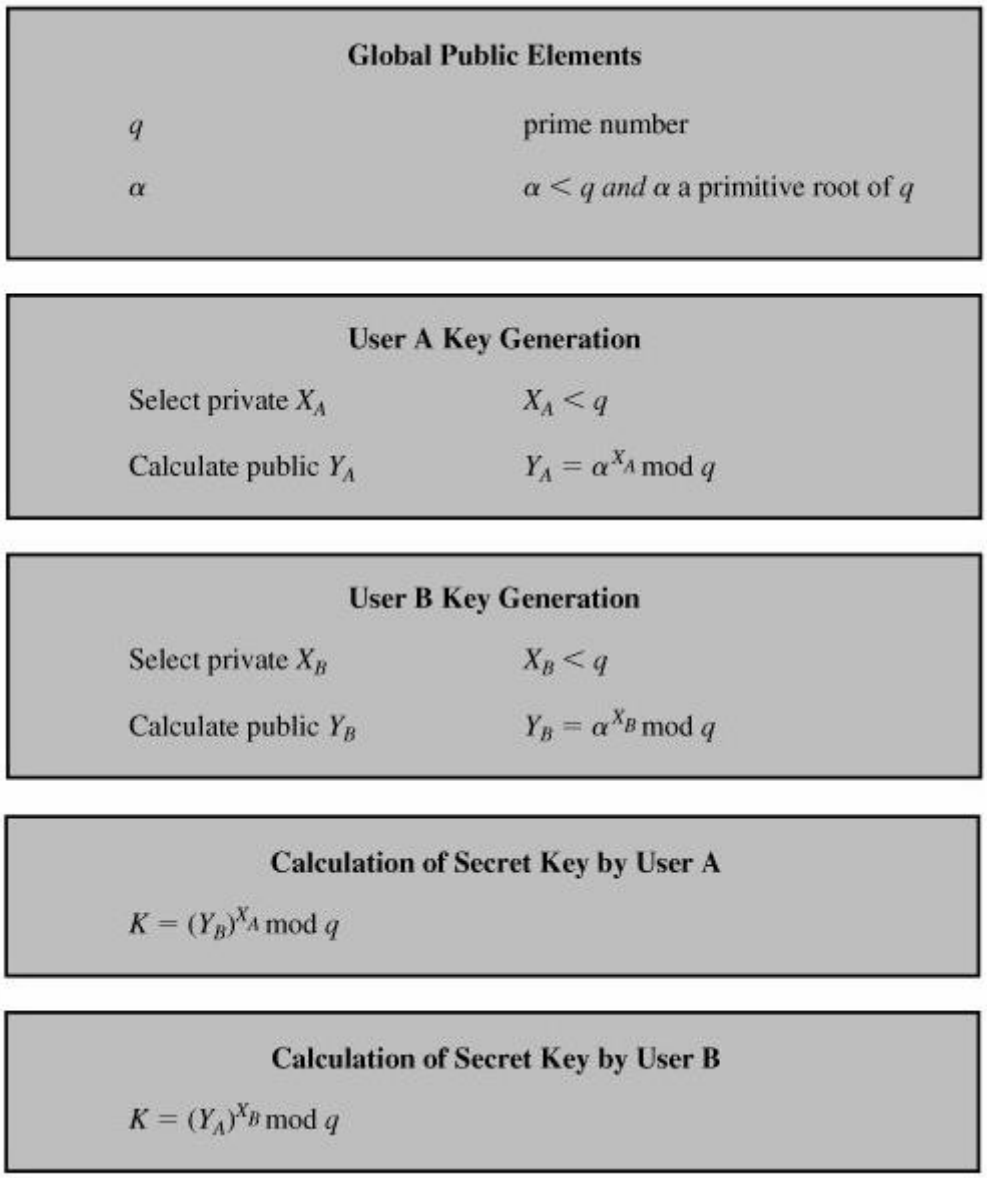
$b \equiv a^i \pmod{p}$ where $0 \leq i \leq (p - 1)$

The exponent i is referred to as the discrete logarithm of b for the base a, mod p.

The Algorithm

Figure 7.7 summarizes the Diffie-Hellman key exchange algorithm. For this scheme, there are two publicly known numbers: a prime number q and an integer that is a primitive root of q.

Figure 10.7. The Diffie-Hellman Key Exchange Algorithm



The result is that the two sides have exchanged a secret value. Furthermore, because X_A and X_B are private.

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.

Here is an example. Key exchange is based on the use of the prime number q = 353 and a primitive root of 353, in this case □ = 3. A and B select secret keys $X_A = 97$ and $X_B = 233$, respectively. Each computes its public key:

A computes $Y_A = 3^{97} \bmod 353 = 40$.

B computes $Y_B = 3^{233} \bmod 353 = 248$.

After they exchange public keys, each can compute the common secret key:

A computes K $= (Y_B)^{X_A} \bmod 353 = 248^{97} \bmod 353 = 160.$

B computes K $= (Y_A)^{X_B} \bmod 353 = 40^{233} \bmod 353 = 160.$

Key Exchange Protocols

Figure 7.8 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 X_A, calculate Y_A, and send that to user B. User B responds by generating a private value X_B calculating Y_B, and sending Y_B to user A. Both users can now calculate the key. The necessary public values q and □ would need to be known ahead of time. Alternatively, user A could pick values for q and □ and include those in the first message.

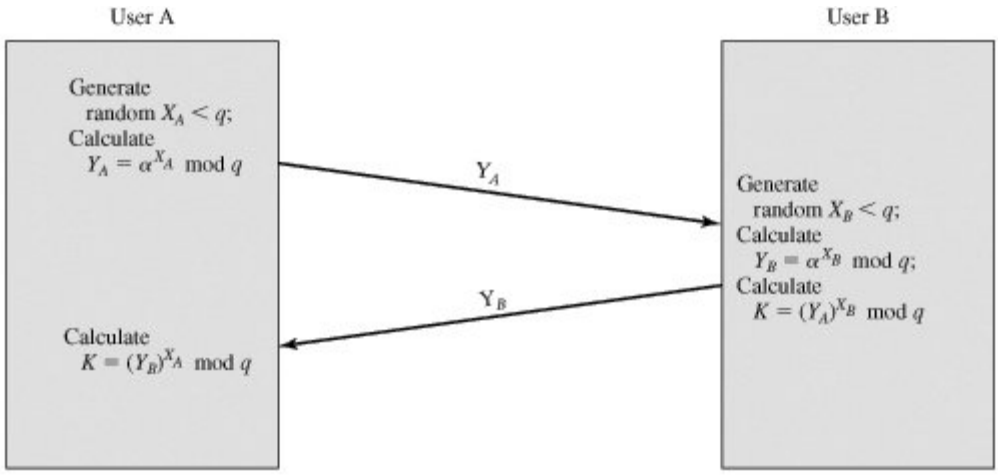


Figure 7.8. Diffie-Hellman Key Exchange

Man-in-the-Middle Attack

The protocol depicted in Figure 7.8 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:

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 $K1 = (Y_{D1})^{X_E} \bmod q$.
5. Bob transmits X_A to Alice.
6. Darth intercepts X_A 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: E(K2, M).
2. Darth intercepts the encrypted message and decrypts it, to recover M.
3. Darth sends Bob E(K1, M) or 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.

7.3. Elliptic Curve Arithmetic

Most of the products and standards that use public-key cryptography for encryption and digital signatures use RSA. As we have seen, the key length for secure RSA use has increased over recent years, and this has put a heavier processing load on applications using RSA. This burden has ramifications, especially for electronic commerce sites that conduct large numbers of secure transactions. Recently, a competing system has begun to challenge RSA: elliptic curve cryptography (ECC). Already, ECC is showing up in standardization efforts, including the IEEE P1363 Standard for Public-Key Cryptography.

The principal attraction of ECC, compared to RSA, is that it appears to offer equal security for a far smaller key size, thereby reducing processing overhead. On the other hand, although the theory of ECC has been around for some time, it is only recently that products have begun to appear and that there has been sustained cryptanalytic interest in probing for weaknesses. Accordingly, the confidence level in ECC is not yet as high as that in RSA.

ECC is fundamentally more difficult to explain than either RSA or Diffie-Hellman, and a full mathematical description is beyond the scope of this book. This section and the next give some background on elliptic curves and ECC. We begin with a brief review of the concept of abelian group.

An elliptic curve is defined by an equation in two variables, with coefficients. For cryptography, the variables and coefficients are

restricted to elements in a finite field, which results in the definition of a finite abelian group. Before looking at this, we first look at elliptic curves in which the variables and coefficients are real numbers. This case is perhaps easier to visualize.

For ECC, we are concerned with a restricted form of elliptic curve that is defined over a finite field

***Definition**: An elliptic curve E over the finite field F_p is given through an equation of the form:*

$y^2 = x^3 + a\,x + b\,, \quad a,b \in F_p \quad , \qquad$ *where* $4a^3 + 27b^2 \neq 0 \bmod p$

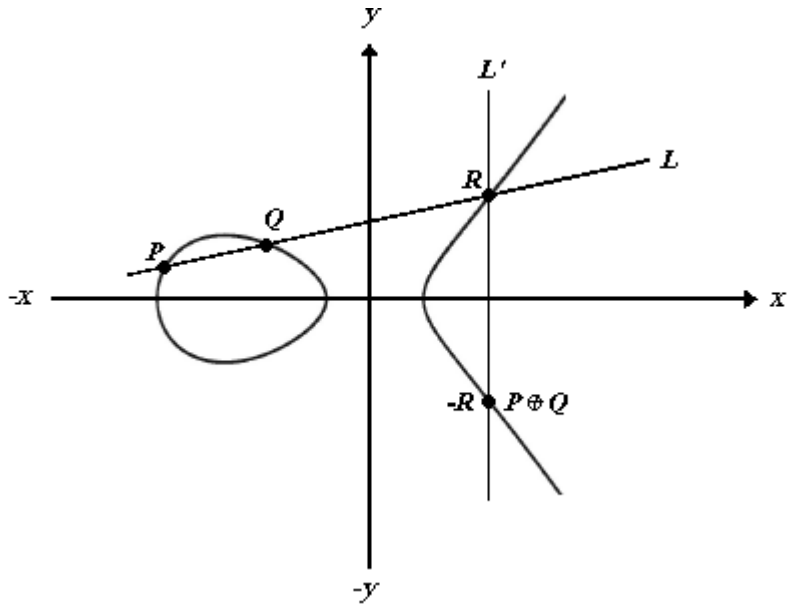


Figure 7.9. An Elliptic Curve

Example: An elliptic curve E over the finite field F_p is given through the equation

Let p = 23 and consider the elliptic curve $E(F_{23})$: $y^2 = x^3 + x + 1$ defined over F_{23}. In the notation of equation, we have a =1 and b =1. Note that $\Delta = 4a^3+27b^2 = 4 + 4 = 8 \neq 0$, *so E is indeed an elliptic curve.*

The points in $E(F_{23})$ and O are the following:

Table 7.1. Points on the Elliptic Curve $E_{23}(1,1)$

(0, 1)	(6, 4)	(12, 19)
(0, 22)	(6, 19)	(13, 7)
(1, 7)	(7, 11)	(13, 16)
(1, 16)	(7, 12)	(17, 3)
(3, 10)	(9, 7)	(17, 20)

Table 7.1. Points on the Elliptic Curve $E_{23}(1,1)$		
(0, 1)	(6, 4)	(12, 19)
(3, 13)	(9, 16)	(18, 3)
(4, 0)	(11, 3)	(18, 20)
(5, 4)	(11, 20)	(19, 5)
(5, 19)	(12, 4)	(19, 18)

Addition formulas of elliptic curve over F_p

Let $P=(x_1, y_1)$, $Q=(x_2, y_2)$, $R=(x_3, y_3)$, $P,Q,R \in E(F_p)$

if $Q \neq -P$ *then* $R=P+Q$.

where

if $Q \neq P$ *then* *(Point addition)*

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda (x_1 - x_3) - y_1$$

where

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$

if $Q = P$ *then* *(Point doubling)*

$$x_3 = \lambda^2 - 2x_1$$
$$y_3 = \lambda (x_1 - x_3) - y_1$$

where

$$\lambda = \frac{3x_1^2 + a}{2y_1},$$

Example:: Addition formulas of elliptic curve over F_p

Consider the elliptic curve defined in the example(2.1).

1.Point addition: *Let P = (3, 10) and Q = (9, 7). Then P + Q = (x_3, y_3) is computed as follows:*

$$\lambda = \frac{7 - 10}{9 - 3} = \frac{-3}{6} = \frac{-1}{2} = 11 \pmod{23}$$

$x_3 = 11^2 - 3 - 9 = 6 - 3 - 9 = -6 = 17 \pmod{23}$,
and
$y_3 = 11(3 - (17)) - 10 = 11(-14) - 10 = -164 = 20 \pmod{23}$.
Hence P + Q = (17, 20).

2.Point doubling: *Let P = (3,10). Then 2P = P + P = (x_3, y_3) is computed as follows:*

$$\lambda = \frac{3(3^2) + 1}{2(10)} = \frac{5}{20} = \frac{1}{4} = 6 \pmod{23}$$

$x_3 = 6^2 - 2(3) = 30 = 7 \pmod{23}$,
$y_3 = 6\,(3 - 7) - 10 = -24 - 10 = -11 = 12 \pmod{23}$.
Hence $2P = (7,12)$.

7.4. 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 $E_p(a, b)$ and k < p. It is relatively easy to calculate Q given k and P, but it is relatively hard to determine k given Q and P. This is called the discrete logarithm problem for elliptic curves.
Consider the group $E_{23}(9, 17)$. This is the group defined by the equation $y^2 \bmod 23 = (x^3 + 9x + 17) \bmod 23$. 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 flavor of this technique.

7.4.1. Elliptic Curves Discrete Logarithm Problem

One of the most interesting open problem in cryptography is the realization of a trapdoor on the discrete logarithm, in which to solve the DLP is hard only if published parameters are used, while it is easy by using a secret key (trapdoor key).

Definition (DLP): For some group G, let a, b $\in$ G, recall that in the DLP, is to find an integer x $\in$ Z, such that $a^x = b$.

The DLP can be defined on various finite groups as well as multiplicative group over a finite filed F_q, this idea can be extended to

arbitrary groups and, in particular, to elliptic curve groups [28]. A typical example except the multiplicative group is the discrete logarithm problem on elliptic curve over F_q , and many cryptographic schemes are constructed on the ECDLP.

Definition (ECDLP): For an elliptic curve E, let P, Q $\in$ E, recall that in the ECDLP, is to find an integer k $\in$ Z, such that kP = Q.

Since an elliptic curve *E* make into Abelian group by an additive operation. "The exponential of a point on *E*" actually refer to the repeated addition. Therefore, the *ith* power of $P \in E$ is the *ith* multiple of P (i.e. $Q=P^k$ equivalent to $Q=kP$). The logarithm of Q to the base P would be k (i.e. the inverse of exponentiation). The ECDLP is of interest because its apparent intractability forms the basis for the security of elliptic curve cryptographic schemes.
The ECDLP over F_q is more intractable than the DLP in F_q. It is this feature that makes cryptographic system based on the ECDLP even more secure than that based on the DLP. Since some of the strongest algorithms for solving DLP cannot be adaptive to the ECDLP.

7.4.2. 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 2^m and elliptic curve parameters a and b for Equation. This defines the elliptic group of points $E_q(a, b)$. Next, pick a base point $G = (x_1, y_1)$ in $E_p(a, b)$ 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 $nG = O$. $E_q(a, b)$ and G are parameters of the cryptosystem known to all participants.
A key exchange between users A and B can be accomplished as follows A selects an integer n_A less than n. This is A's private key. A then generates a public key $P_A = n_A \times G$; the public key is a point in Eq(a, b).

1. B similarly selects a private key n_B and computes a public key P_B.
2. A generates the secret key $K = n_A \times P_B$. B generates the secret key $K = n_B \times P_A$.

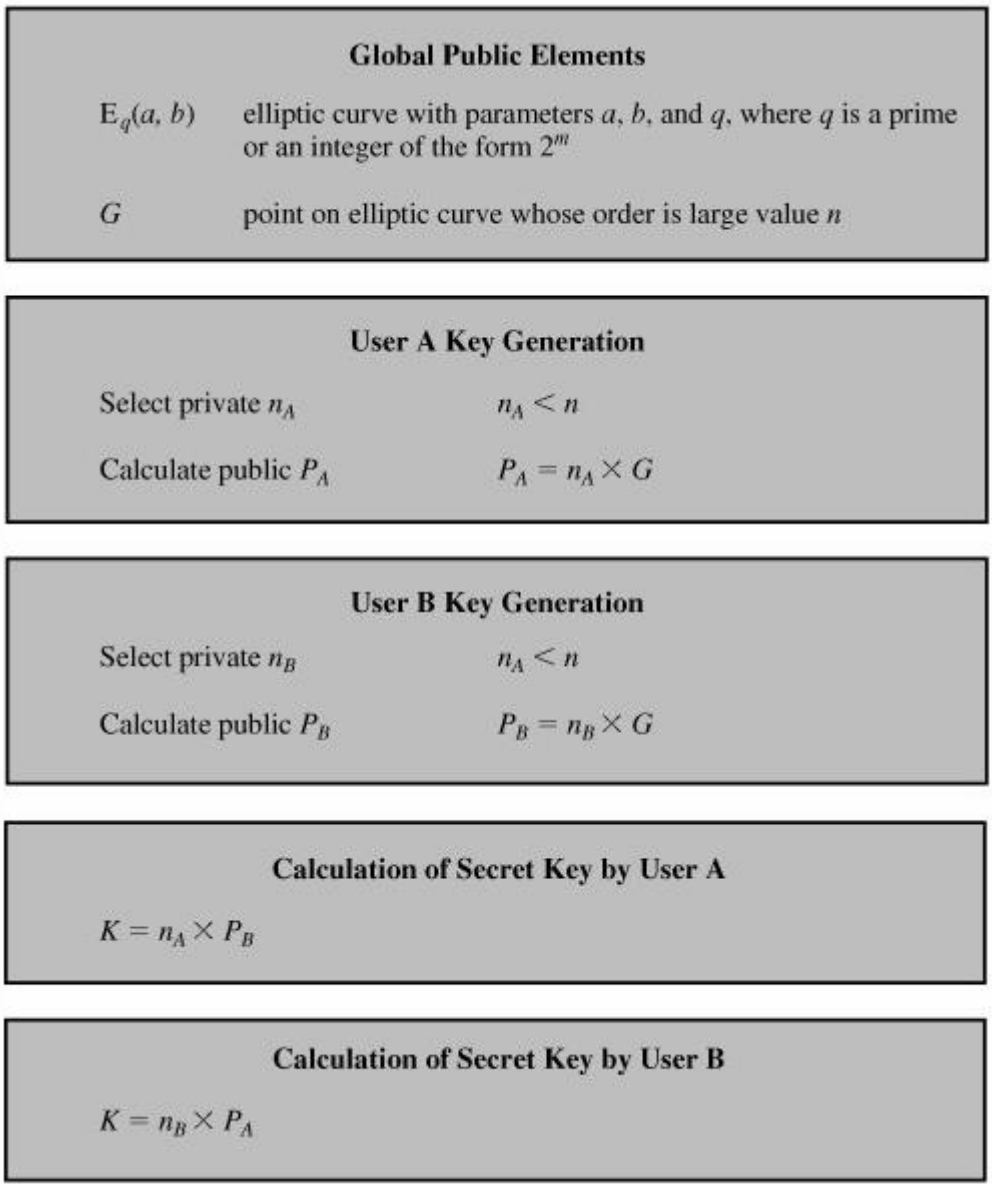


Figure 7.10. ECC Diffie-Hellman Key Exchange

The two calculations in step 3 produce the same result because
n_A x P_B = n_A x (n_B x G) = n_B x (n_A x G) = n_B x P_A
To break this scheme, an attacker would need to be able to compute k given G and kG, which is assumed hard.
As an example, take p = 211; Ep(0, 4), which is equivalent to the curve y^2 = x^3 4; and G = (2, 2). One can calculate that 240G = O. A's private key is n_A = 121, so A's public key is P_A = 121(2, 2) = (115, 48). B's private key is n_B = 203, so B's public key is 203(2, 2) = (130, 203). The shared secret key is 121(130, 203) = 203(115, 48) = (161, 69).
Note that the secret key is a pair of numbers. If this key is to be used as a session key for conventional encryption, then a single number must be

generated. We could simply use the x coordinates or some simple function of the x coordinate.

7.4.3. Elliptic Curve Encryption/Decryption

Several approaches to encryption/decryption using elliptic curves have been analyzed 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 P_m. It is the point P_m that will be encrypted as a ciphertext and subsequently decrypted. Note that we cannot simply encode the message as the x or y coordinate of a point, because not all such coordinates are in Eq(a, b); for example. Again, there are several approaches to this encoding, which we will not address here, but suffice it to say that there are relatively straightforward techniques that can be used.

As with the key exchange system, an encryption/decryption system requires a point G and an elliptic group Eq(a, b) as parameters. Each user A selects a private key n_A and generates a public key $P_A = n_A \times G$.

To encrypt and send a message P_m to B, A chooses a random positive integer k and produces the ciphertext C_m consisting of the pair of points:

$C_m = \{kG, P_m + kP_B\}$

Note that A has used B's public key P_B. To decrypt the ciphertext, B multiplies the first point in the pair by B's secret key and subtracts the result from the second point:

$P_m + kP_B\ n_B(kG) = P_m + k(n_BG)\ n_B(kG) = P_m$

A has masked the message P_m by adding kP_B to it. Nobody but A knows the value of k, so even though P_B is a public key, nobody can remove the mask kP_B. However, A also includes a "clue," which is enough to remove the mask if one knows the private key n_B. For an attacker to recover the message, the attacker would have to compute k given G and kG, which is assumed hard.

As an example of the encryption process, take p = 751; $E_p(1, 188)$, which is equivalent to the curve $y^2 = x^3\ x + 188$; and G = (0, 376). Suppose that A wishes to send a message to B that is encoded in the elliptic point P_m = (562, 201) and that A selects the random number k = 386. B's public key is P_B = (201, 5). We have 386(0, 376) = (676, 558), and (562, 201) + 386(201, 5) = (385, 328). Thus A sends the cipher text {(676, 558), (385, 328)}.

7.4.4. 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 7.3 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. Furthermore, for equal key lengths, the computational effort required for ECC and RSA is comparable. Thus, there is a computational advantage to using ECC with a shorter key length than a comparably secure RSA.

Table 7.3. Comparable Key Sizes in Terms of Computational Effort for Cryptanalysis

Symmetric Scheme (key size in bits)	ECC-Based Scheme (size of n in bits)	RSA/DSA (modulus size in bits)
56	112	512
80	160	1024
112	224	2048
128	256	3072
92	384	7680
256	512	15360

7.5. Digital Signature Standard

The National Institute of Standards and Technology (NIST) have published Federal Information Processing Standard FIPS 186, known as the Digital Signature Standard (DSS). The DSS makes use of the Secure Hash Algorithm (SHA) and presents a new digital signature technique, the Digital Signature Algorithm (DSA). The DSS was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There was a further minor revision in 1996. In 2000, an expanded version of the standard was issued as FIPS 186-2. This latest version also incorporates digital signature algorithms based on RSA and on elliptic curve cryptography. In this section, we discuss the original DSS algorithm.

7.5.1. The DSS Approach

The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique.

Figure 7.11 contrasts the DSS approach for generating digital signatures to that used with RSA. In the RSA approach, the message to be signed is input to a hash function that produces a secure hash code of fixed length.

This hash code is then encrypted using the sender's private key to form the signature. Both the message and the signature are then transmitted. The recipient takes the message and produces a hash code. The recipient also decrypts the signature using the sender's public key. If the calculated hash code matches the decrypted signature, the signature is accepted as valid. Because only the sender knows the private key, only the sender could have produced a valid signature.

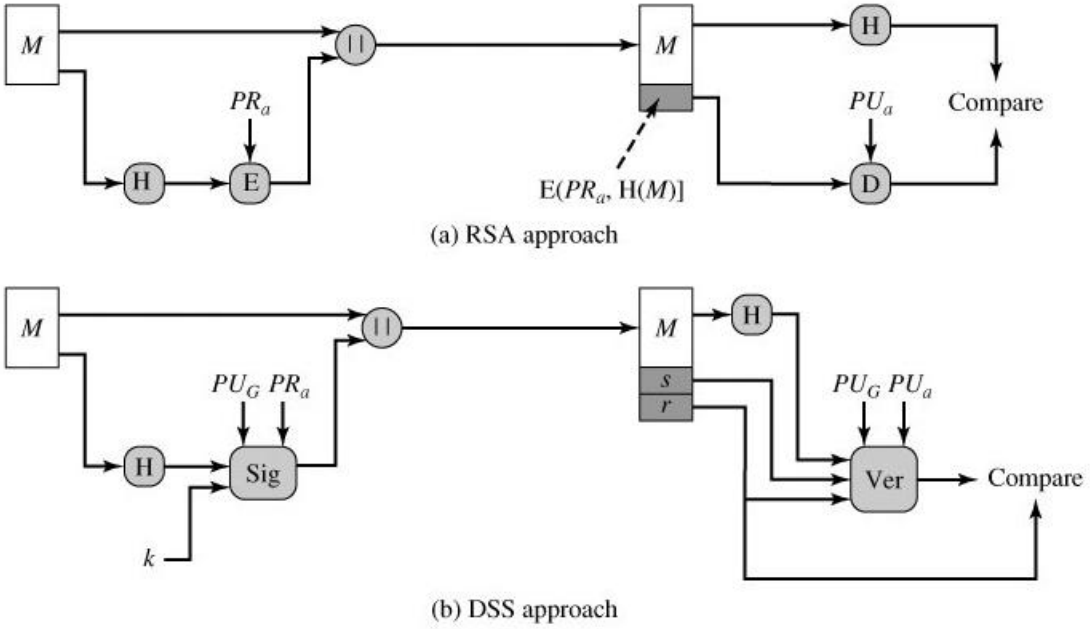


Figure 7.11. Two Approaches to Digital Signatures

The DSS approach also makes use of a hash function. The hash code is provided as input to a signature function along with a random number k generated for this particular signature. The signature function also depends on the sender's private key (PR_a)and a set of parameters known to a group of communicating principals. We can consider this set to constitute a global public key (PU_G). The result is a signature consisting of two components, labeled s and r.

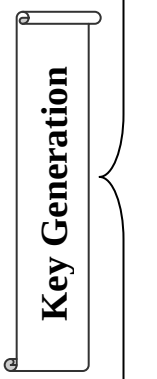
At the receiving end, the hash code of the incoming message is generated. This plus the signature is input to a verification function. The verification function also depends on the global public key as well as the sender's public key (PU_a), which is paired with the sender's private key. The output of the verification function is a value that is equal to the signature component r if the signature is valid. The signature function is such that only the sender, with knowledge of the private key, could have produced the valid signature.

We turn now to the details of the algorithm.

7.5.2. The Digital Signature Algorithm

The DSA is based on the difficulty of computing discrete logarithms and is based on schemes originally presented by ElGamal and Schnorr .

There are three parameters that are public and can be common to a group of users. A 160-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 $h^{(p1)/q}$ mod p where h is an integer between 1 and (p - 1) with the restriction that g must be greater than 1.



Global Public-Key Components

p prime number where $2^{L-1} < p < 2^L$ for $512 \leq L \leq 1024$ and L a multiple of 64; i.e., bit length of between 512 and 1024 bits in increments of 64 bits

q prime divisor of $(p - 1)$, where $2^{159} < q < 2^{160}$; i.e., bit length of 160 bits

g $= h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < (p - 1)$ such that $h^{(p-1)/q} \bmod p > 1$

User's Private Key

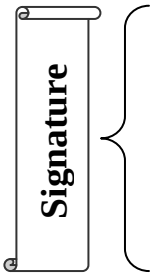
x random or pseudorandom integer with $0 < x < q$

User's Public Key

y $= g^x \bmod p$

User's Per-Message Secret Number

k = random or pseudorandom integer with $0 < k < q$

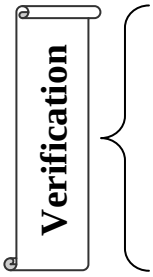


Signing

r $= (g^k \bmod p) \bmod q$

s $= [k^{-1} (H(M) + xr)] \bmod q$

Signature = (r, s)



Verifying

w $= (s')^{-1} \bmod q$

u_1 $= [H(M')w] \bmod q$

u_2 $=(r')w \bmod q$

v $= [(g^{u_1} y^{u_2}) \bmod p] \bmod q$

TEST: $v = r'$

Global Public-Key Components

M = message to be signed

$H(M)$ = hash of M using SHA-1

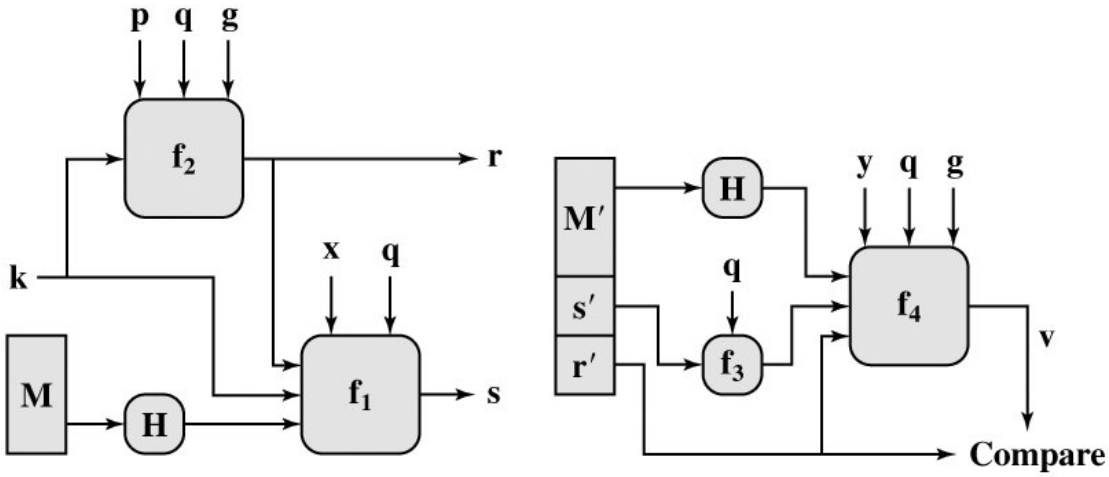
M', r', s' = received versions of M, r, s

With these numbers 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.

To create a signature, a user calculates two quantities, r and s, that 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.

The receiver generates a quantity v that is a function of the public key components, the sender's public key, and the hash code of the incoming message. If this quantity matches the r component of the signature, then the signature is validated.

Figure 7.12 depicts the functions of signing and verifying.



$s = f_1(H(M), k, x, r, q) = (k^{-1} (H(M) + xr)) \bmod q$

$r = f_2(k, p, q, g) = (g^k \bmod p) \bmod q$

(a) Signing

$w = f_3(s', q) = (s')^{-1} \bmod q$

$v = f_4(y, q, g, H(M'), w, r')$

$= ((g^{(H(M')w) \bmod q} y^{r'w \bmod q}) \bmod p) \bmod q$

(b) Verifying

Figure 7.12. DSS Signing and Verifying

The structure of the algorithm, as revealed in Figure 7.12, 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 (mod 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 7.12 that such a scheme would work. A proof is provided at this book's Web site.

7.6. Hash Functions

A hash value h is generated by a function H of the form

$$h = H(M)$$

where M is a variable-length message and H(M) is the fixed-length hash value. The hash value is appended to the message at the source at a time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the hash value. Because the hash function itself is not considered to be secret, some means is required to protect the hash value.

We begin by examining the requirements for a hash function to be used for message authentication. Because hash functions are typically quite complex, it is useful to examine some very simple hash functions to get a feel for the issues involved. We then look at several approaches to hash function design.

Requirements for a Hash Function

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties (adapted from a list in:

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given value h, it is computationally infeasible to find x such that H(x) = h. This is sometimes referred to in the literature as the one-way property.
5. For any given block x, it is computationally infeasible to find $y \neq x$ such that H(y) = H(x). This is sometimes referred to as **weak collision resistance**.
6. It is computationally infeasible to find any pair (x, y) such that H(x) = H(y). This is sometimes referred to as **strong collision resistance**.

Secure Hash Algorithm

The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. The actual standards document is entitled Secure Hash Standard. SHA is based on the hash function MD4 and its design closely models MD4. SHA-1 is also specified in RFC 3174, which essentially duplicates the material in FIPS 180-1, but adds a C code implementation.

SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512 (Table 12.1). These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on the other SHA versions by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using 269 operations, far fewer than the 280 operations previously thought needed to find a collision with an SHA-1 hash [WANG05]. This result should hasten the transition to the other versions of SHA.

Table 12.1. Comparison of SHA Parameters

	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	<264	<264	<2128	<2128
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256

Notes: 1. All sizes are measured in bits.
2. Security refers to the fact that a birthday attack on a message digest of size n produces a collision with a workfactor of approximately 2n/2

In this section, we provide a description of SHA-512. The other versions are quite similar.

SHA-512 Logic

The algorithm takes as input a message with a maximum length of less than 2128 bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks. Figure 12.1 depicts the overall processing of a message to produce a digest.

Figure 12.1. Message Digest Generation Using SHA-512
[View full size image]

This follows the general structure depicted in Figure 11.9. The processing consists of the following steps:

Step 1: Append padding bits. The message is padded so that its length is congruent to 896 modulo 1024 [length 896 (mod 1024)]. Padding is always added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

Step 2: Append length. A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).

The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. In Figure 12.1, the expanded message is represented as the sequence of 1024-bit blocks M1, M2,..., MN, so that the total length of the expanded message is N x 1024 bits.

Step 3: Initialize hash buffer. A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following 64-bit integers (hexadecimal values):

a = 6A09E667F3BCC908

b = BB67AE8584CAA73B

c = 3C6EF372FE94F82B

c = A54FF53A5F1D36F1

e = 510E527FADE682D1

f = 9B05688C2B3E6C1F

g = 1F83D9ABFB41BD6B

h = 5BE0CDI9137E2179

[Page 355]
These values are stored in big-endian format, which is the most significant byte of a word in the low-address (leftmost) byte position. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.
Step 4: Process message in 1024-bit (128-word) blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 12.1. The logic is illustrated in Figure 12.2.
Figure 12.2. SHA-512 Processing of a Single 1024-Bit Block

Each round takes as input the 512-bit buffer value abcdefgh, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, Hi-1. Each round t makes use of a 64-bit value Wt derived from the current 1024-bit block being processed (Mi) These values are derived using a message schedule described subsequently. Each round also makes use of an additive constant Kt

where 0 t 79 indicates one of the 80 rounds. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data.

The output of the eightieth round is added to the input to the first round (Hi-1)to produce Hi. The addition is done independently for each of the

eight words in the buffer with each of the corresponding words in Hi-1 using addition modulo 264.

[Page 356]
Step 5: Output. After all N 1024-bit blocks have been processed, the output from the Nth stage is the 512-bit message digest.

We can summarize the behavior of SHA-512 as follows:

H0 = IV

Hi = SUM64(Hi-1, abcdefghi)

MD = HN

where

IV	= initial value of the abcdefgh buffer, defined in step 3
abcdefghi	= the output of the last round of processing of the ith message block
N	= the number of blocks in the message (including padding and length fields)
SUM64	= Addition modulo 264 performed separately on each word of the pair of inputs
MD	= final message digest value

SHA-512 Round Function
Let us look in more detail at the logic in each of the 80 steps of the processing of one 512-bit block (Figure 12.3). Each round is defined by the following set of equations:

where

t	=step number; 0 t 79
Ch(e, f, g)	= (e AND f) (NOT e AND g) the conditional function: If e then f else g
Maj(a, b, c)	= (a AND b) (a AND c) (b AND c) the function is true only of the majority (two or three) of the arguments are true.

[Page 357]

ROTRn(x)	= circular right shift (rotation) of the 64-bit argument x by n bits
Wt	= a 64-bit word derived from the current 512-bit input block
Kt	= a 64-bit additive constant
+	= addition modulo 264

Figure 12.3. Elementary SHA-512 Operation (single round)

It remains to indicate how the 64-bit word values Wt are derived from the 1024-bit message. Figure 12.4 illustrates the mapping. The first 16 values of Wt are taken directly from the 16 words of the current block. The remaining values are defined as follows:

where

ROTRn(x) = circular right shift (rotation) of the 64-bit argument x by n bits

SHRn(x) = left shift of the 64-bit argument x by n bits with padding by zeros on the right

Figure 12.4. Creation of 80-word Input Sequence for SHA-512 Processing of Single Block
(This item is displayed on page 358 in the print version)
[View full size image]

Thus, in the first 16 steps of processing, the value of Wt is equal to the corresponding word in the message block. For the remaining 64 steps, the value of Wt consists of the circular left shift by one bit of the XOR of four of the preceding values of Wt, with two of those values subjected to shift and rotate operations. This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.