MOD 2

12

MOD 2

10

Chapter 6

Public Key Cryptography and RSA

In 1976, Diffie and Helman proposed the use of a public key/private key pair for session key agreement between two communicating parties. Their scheme is explored further in Chapter 8. In 1977, Rivest, Shamir, and Adelman proposed a scheme using a public key for encrypting messages and a corresponding private key for decryption – this scheme is commonly referred to as RSA.

In this chapter, we first study three RSA operations – key generation, encryption, and decryption. We then examine why RSA works, i.e., why does encryption of a message followed by decryption of its ciphertext *always* end up recovering the original message? We study the time complexity of RSA operations and investigate attacks on it. Finally, we study applications of RSA and introduce the Public Key Cryptography Standard (PKCS).

6.1 RSA OPERATIONS

The first step in the use of RSA is to generate a public key/private key pair. This is usually a one-time operation unless an individual compromises his/her key or needs to obtain a fresh one for security reasons. We then describe the operations of encryption and decryption. For a more complete understanding of this chapter, you may wish to review the mathematical foundations in Sections 3.1 and 3.2 of Chapter 3.

6.1.1 Key Generation

Step 1: Choose two large prime numbers p and q. The product $n = p \times q$ is referred to as the modulus and $\Phi(n)$ is the Euler Totient Function defined in Chapter 3.

Step 2: Choose an encryption key, e, such that $\gcd(e, \Phi(n)) = 1$. The pair of integers, (e, n) is referred to as the public key.

Step 3: Compute the decryption key, $d = e^{-1} \bmod \Phi(n)$. d is also referred to as the private key.

Recall that $\Phi(n)$ is the number of integers between 1 and $n-1$ that are relatively prime to n. Since $n = pq$, the only integers in this range not relatively prime to n are

$$p, 2p, \ldots (q-1)p$$

and

$$q, 2q, \ldots (p-1)q.$$

Hence

$$\Phi(n) = pq - 1 - (p - 1) - (q - 1) = (p-1)(q-1).$$

The number of bits, b, used to represent n is referred to as the key size, b.

So, $$b = \lceil \log_2 n \rceil.$$

Encryption Let m be the message (or plaintext). We use $|m|$ to denote the length of m. In the naive implementation of RSA, a message is split into multiple blocks, each of size b, except possibly for the last block. $|m| \bmod b$, if different from 0, will be the size of the last block. For each block m_i, calculate the corresponding ciphertext c_i as

$$c_i = m_i^e \bmod n \tag{6.1}$$

Decryption Given a block of ciphertext c_i, the corresponding plaintext is

$$m_i = c_i^d \bmod n \tag{6.2}$$

A block of plaintext, m_i, is encrypted as c_i using Eq. (6.1). Will the decryption of c_i using Eq. (6.2) always return the original plaintext? Could two different plaintexts correspond to the same ciphertext? We answer these questions after an example.

Example 6.1

Suppose the RSA prime numbers are $p = 3$ and $q = 11$.

So $$n \equiv pq = 33 \quad \text{and} \quad \Phi(n) = (p-1)(q-1) = 2 \times 10 = 20$$

Choose a public key, $e = 3$. Since gcd(3, 20) = 1, $3^{-1} \bmod 20$ exists and is 7.

So the private key, $d = 7$.

The key size is 6 (minimum number of bits used to represent n), which is also the block size, b. Given a message 00111011, we note that it spans two blocks. The first block is 001110 (or 14 in decimal). The remaining two bits of the message are padded with zeros to the left and comprise the second block, i.e., 000011 (or 3 in decimal).

The ciphertext for Block 1 is

$$\begin{aligned}
14^3 \bmod 33 &= 14 * 14 * 14 && \bmod 33\\
&= 196 * 14 && \bmod 33\\
&\equiv (196 - 33 * 6) * 14 && \bmod 33 \quad \text{// note the reduction}\\
& && \quad\quad\quad\;\; \text{// modulo 33 here}\\
&\equiv -2 * 14 && \bmod 33\\
&= -28 && \bmod 33\\
&\equiv 5 && \bmod 33
\end{aligned}$$

Note that the reduction modulo 33 may be performed at one or more intermediate steps. This helps maintain a bound on the size of the numbers involved.

Decryption is performed below:

$$\begin{aligned}
5^7 \bmod 33 &= 5 * 5 * 5 * 5 * 5 * 5 * 5 && \bmod 33\\
&= 25 * 25 * 25 * 5 && \bmod 33\\
&\equiv (-8) * (-8) * (-8) * 5 && \bmod 33\\
&\equiv (64 - 33 * 2) * (-40) && \bmod 33\\
&= (-2) * (-40) && \bmod 33\\
&= 80 && \bmod 33\\
&\equiv 80 - 33 * 2 && \bmod 33\\
&= 14 && \bmod 33
\end{aligned}$$

The ciphertext for Block 2 is

$$3^3 \bmod 33 \equiv 27$$

Decryption of this ciphertext yields

$$27^7 \bmod 33 \equiv 3.$$

In real applications, the modulus n and decryption key d are both hundreds of digits long. Figure 6.1 shows possible values of p, q, d, and e for a 1024-bit RSA key. The smaller the encryption/decryption keys, the less is the time to perform an encryption/decryption operation. For this reason, the encryption key is often a small integer. $e = 3$ is a popular choice for encryption key. Another popular choice is $2^{16} + 1 = 65{,}537$.

```
p:
11612199208603481528191203480311138551131149330910969186013508178
40489766730995798137305339077858683430364481310884322577520364334
5752164528840671055453193

q:
10073741008106865231276389795238965771248709323399675549509790429
94201294528667174062298925269404436079460104574371305813260841621
6317245870942597743372319

n:
11697828736201497863616515616693701962943165402985819893220131312
90426377816577452140272210039190585143331628702695744509319333385
39865902088800740075367740154897032857980650725590433432639369465
45058613378665041606264003016783561013711658871498490854799867662
2092621965291236839645729546251947370638676364567

Φ(n)  :
11697828736201497863616515616693701962943165402985819893220131312
90426377816577452140272210039190585143331628702695744509319333385
39865902088800740075367718468956816147633891257997157882535047085
59193182314191489276403168325722301350739459267234143591680357837
6233769409007329027586167476841547587369877539056

e:
65537

d:
74195545250228704883472133474941453407330439213164252456074342495
81220329444358382374358793797255633190107747060997102058895898525
14632317321096352206064562204189531657665988954701292225951269855
78916063053895025137178525254989490295182544642017315534578805040
5615961486430055238827198327257800953933056066 57
```

Figure 6.1 *Sample RSA parameters for key size = 1024 bits*

One might wonder whether the use of the same encryption key (such as 3 or 65,537) by so many would compromise the security of RSA. In response, it should be noted that the public key of a person is (e, n) – a pair comprising the encryption key and the modulus. The space of moduli is so large that the probability of two individuals having the same modulus value, n, is infinitesimally small. This, in effect, means that no two persons have the same public key or the same private key.

6.2 WHY DOES RSA WORK?

Let the i-th message block have value = m_i. Let its corresponding ciphertext be c_i. Decrypting the ciphertext using Eq. (6.2) yields

$$\begin{aligned} c_i^d \bmod n &= (m_i^e \bmod n)^d && \bmod n && \text{from Eq. (6.1)} \\ &= m_i^{e \times d} && \bmod n && \text{laws of modulo arithmetic} \\ &= m_i^{1 + k \times \Phi(n)} && \bmod n && \text{for some integer } k \end{aligned}$$

The last step follows from the fact that d was chosen to be the inverse of e modulo $\Phi(n)$. So, $e \times d$ and 1 differ by an integral multiple of $\Phi(n)$.

We next show that $m_i^{1 + k \times \Phi(n)} \bmod p = m_i$

$$\begin{aligned} m_i^{1 + k \times \Phi(n)} \bmod p &= m_i\, m_i^{k \times \Phi(n)} \bmod p \\ &= m_i\, (m_i^{k \times (q-1)})^{(p-1)} \bmod p \end{aligned}$$

(a) Suppose $\gcd(m_i, p) = 1$. Then $\gcd(m_i^{k \times (q-1)}, p) = 1$.

From Fermat's Little Theorem,

$$(m_i^{k \times (q-1)})^{(p-1)} \bmod p = 1$$

So

$$m_i^{1 + k \times \Phi(n)} \bmod p = m_i \bmod p$$

(b) On the other hand, suppose $\gcd(m_i, p) > 1$. This implies that m_i is an integral multiple of p. In that case

$$m_i^{1 + k \times \Phi(n)} \bmod p = m_i \bmod p = 0$$

We conclude that, regardless of the value of the message m_i, (6.3)

$$m_i^{1 + k \times \Phi(n)} \bmod p = m_i \bmod p$$

In a similar manner, it can be shown that regardless of the value of the message m_i, (6.4)

$$m_i^{1 + k \times \Phi(n)} \bmod q = m_i \bmod q$$

From Eq. (6.3), (6.5)

$$m_i^{1 + k \times \Phi(n)} = m_i + c_1 p \qquad \text{for some integer } c_1$$

From Eq. (6.4), (6.6)

$$m_i^{1 + k \times \Phi(n)} = m_i + c_2 q \qquad \text{for some integer } c_2$$

Equating the RHS of Eqs (6.5) and (6.6),

$$m_i + c_1 p = m_i + c_2 q$$

So

$$c_1 p = c_2 q$$

Since p and q are prime, it follows that c_1 should have q as factor.

So

$$c_1 p = c_3 q\, p \qquad \text{for some integer } c_3$$

Substituting in Eq. (6.5), we get

$$m_i^{1 + k \times \Phi(n)} = m_i + c_3 pq = m_i + c_3 n$$

So

$$c_i^d \bmod n = m_i^{1 + k \times \Phi(n)} \bmod n = m_i$$

6.3 PERFORMANCE

6.3.1 Time Complexity

Both encryption and decryption involve repeated multiplications (modulo n) of b-bit numbers. Unoptimized multiplication of two b-bit numbers and reduction modulo n (division), both take $O(b^2)$ time. The encryption key is usually a small integer (relative to n) as described earlier. So encryption involves a small, constant number of modulo n multiplications. Hence, the *time complexity of encryption* is $O(b^2)$.

Decryption, on the other hand, involves raising a b-bit number (in general) to the power of d. A naive implementation of decryption thus involves d multiplications. Since d is of the same order as n, the complexity of a decryption operation is $O(nb^2)$. Given that n is hundreds of digits long, the execution time of this operation is prohibitive but can be reduced by using the "Square and Multiply" technique described below.

6.3.2 Speeding Up RSA

We can speed up the decryption of ciphertext c by computing c, c^2, c^4, c^8, etc., up to a maximum of b terms. Note that each element in this series is the square of the preceding element. Then we multiply elements in this series whose positions correspond to 1's in the binary representation of the decryption key d. Of course, each multiplication is a modulo n multiplication so the intermediate products are never more than b bits wide. This approach, which first computes squares followed by products, is referred to as "*Square and Multiply.*"

Example 6.2

Suppose the decryption key is 57.
Then, naive decryption would involve a total of 56 modulo n multiplications.
However, the "square and multiply" approach involves computing

$$c^2,\ c^4,\ c^8,\ c^{16} \text{ and } c^{32} \qquad \text{each reduced modulo } n.$$

Since the binary representation of 57 is 111001, we selectively multiply c, c^8, c^{16}, *and* c^{32} to obtain the original plaintext (see Fig. 6.2). Thus, we now perform decryption using only *five square operations* and *three multiplications*, a considerable savings compared to 56 *multiplications* without "square and multiply."

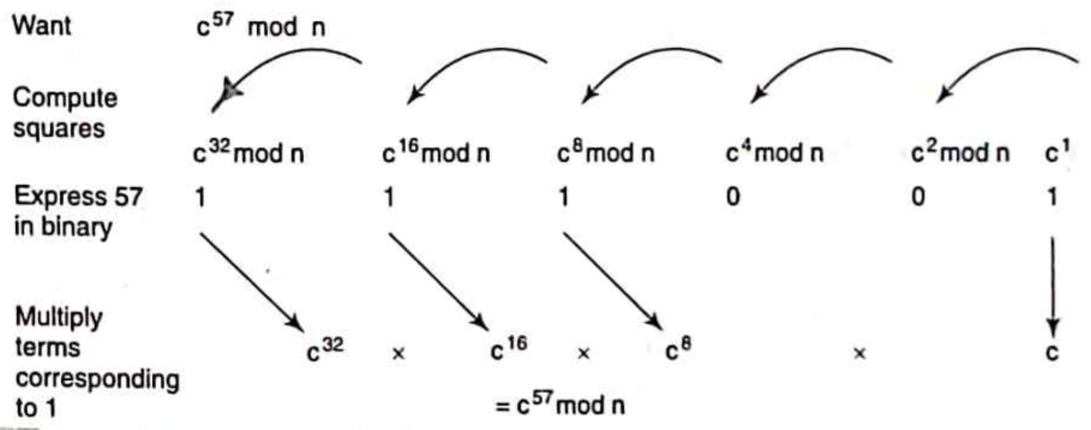


Figure 6.2 *Use of "square and multiply" in RSA*

In general, decryption involves b-1 square operations and at most b-1 multiplications. Also, each square operation and multiplication is followed by a reduction modulo n. Hence, the *time for decryption* is $O(b^3)$ – this is considerably slower than the $O(b^2)$ time necessary for encryption.

The choice of key size represents a trade-off between security and performance. A larger key size provides greater security, but the times for both encryption and decryption increase. The asymptotic complexities tell us that doubling the key size increases the time for encryption by, roughly, a factor of 4, while the time for decryption increases by a factor of 8.

6.3.3 Software Performance

The Java programming language has a number of APIs of relevance to cryptography. These include APIs for key generation and encryption/decryption (both symmetric and asymmetric cryptography), message digests, and digital signatures (Chapter 7). These are contained in the java.security package and its various subpackages.

Java also permits the import of classes created by various third parties that implement cryptographic algorithms. An example of a third party provider is Bouncy Castle. *Bouncy Castle cryptographic APIs* are available for use in both Java and C++ programs.

An example of the use of the Java APIs for key generation, encryption, and decryption is shown in Fig. 6.3.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "BC");
kpg.initialize(1024);
KeyPair kp = kpg.generateKeyPair( );
Cipher c = Cipher.getInstance ("RSA/ECB/PKCS1Padding", "BC");

String plainText = "Hello World!";
c.init(Cipher.ENCRYPT_MODE, kp.getPublic( ));
byte [] encryptedText = c.doFinal(plainText.getBytes( ));

c.init(Cipher.DECRYPT_MODE, kp.getPrivate( ));
byte [] decryptedText = c.doFinal(encryptedText);
String recoveredText = new String(decryptedText);
```

Figure 6.3 *Illustrating Java APIs for RSA encryption/decryption*

Figure 6.4 shows the time to perform encryption and decryption of a 60-byte integer for various key sizes. Execution times are on a Pentium 3.2 GHz machine with 512 MB RAM.

6.4 APPLICATIONS

Providing *message confidentiality* through encryption is an obvious application of public key cryptography.

Suppose A needs to send a confidential message to B. With secret key cryptography, A and B need to share a secret. With public key cryptography, A needs to use B's public key. How does A obtain B's public key? This is done through a digital certificate. We will study digital certificates in Chapter 10. For now, think of B's digital certificate as an authentic electronic document from which one can extract the public key of B.

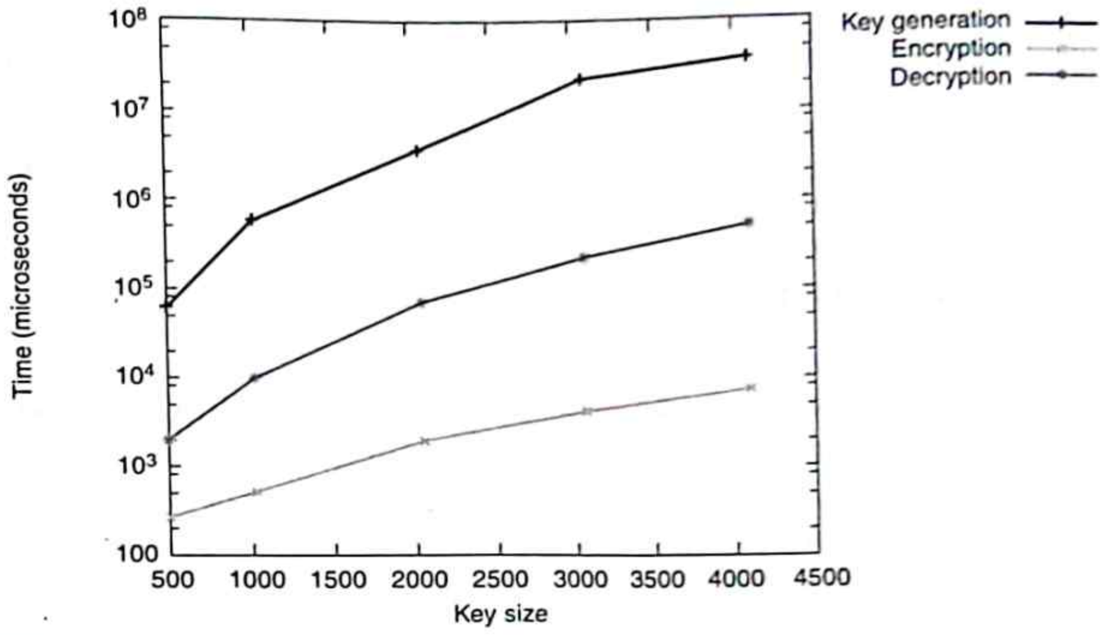


Figure 6.4 *Time for RSA key operations as a function of key size*

It turns out that public key cryptography is far more computationally intensive vis-à-vis secret key cryptography. (Public key cryptography is orders of magnitude more expensive as compared to secret key cryptography). So, is there any reason to use public key encryption at all?

With secret key cryptography, each pair of entities would have to agree upon a secret key (possibly using an offline technique) and then safely store the secret keys – one per entity it wishes to communicate with. With public key cryptography, each entity must only store its private key securely. *In summary, the principal drawback of public key cryptography is speed, while the principal drawback of secret key cryptography is key management.*

To combine the speed of secret key cryptography and the convenience of public key cryptography, a *session key* is often employed. Here's how it works.

The sender

- chooses a fresh random number, *s*, as the secret key. This is referred to as a session key
- encrypts the message with the session key ($E_s(m)$)
- encrypts the session key with the recipient's public key ($E_{B.pu}(s)$)
- sends the encrypted message and the encrypted session key in the same message (see Fig. 6.5).

The receiver

- uses his private key to decrypt the part of the message containing the encrypted session key
- uses the session key to decrypt the message (see Fig. 6.5).

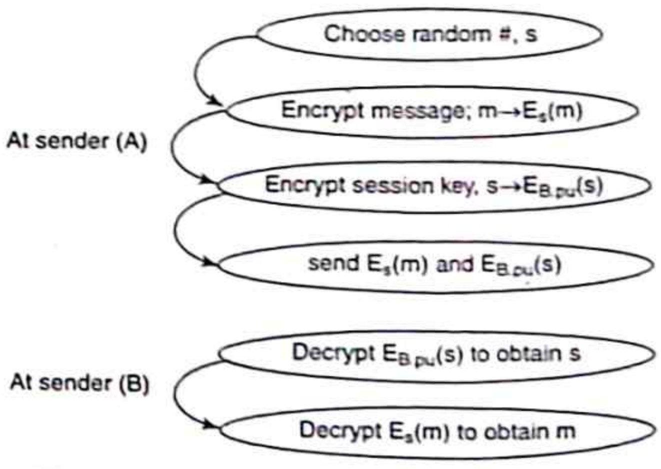

Choose random #, s
Encrypt message; m→E_s(m)
At sender (A)
Encrypt session key, s→$E_{B.pu}$(s)
send E_s(m) and $E_{B.pu}$(s)
Decrypt $E_{B.pu}$(s) to obtain s
At sender (B)
Decrypt E_s(m) to obtain m

Figure 6.5 *Encrypted message with encrypted session key*

The session key is used to encrypt/decrypt the remaining messages in that session. As the name suggests, the session key is valid for the duration of a session and is destroyed thereafter.

There are several other uses of public key cryptography. It is used to generate a *digital signature* that provides *message integrity* and *authentication* together with *non-repudiation*. We study the digital signature in Chapter 7 after introducing the cryptographic hash. Finally, public key cryptography is used in many authentication protocols as illustrated in Chapter 11.

6.5 PRACTICAL ISSUES

6.5.1 Generating Primes

RSA key generation involves choosing two large primes p and q. But how do we know that a number, say p, is prime? We could examine divisibility by all integers less than $\sqrt{p}$. The reason for stopping at $\sqrt{p}$ is that, if p is composite (non-prime), then at least one of its factors must be less than $\sqrt{p}$. We could perform other optimizations such as checking for divisibility by odd integers only, etc. Such naive methods, however, do not easily scale up and are not feasible in primality testing of integers that are hundreds of digits long.

The most widely used test of primality is a probabilistic method called the *Miller-Rabin Test.* This asserts that a number is prime with some probability $1-\varepsilon$. ε can be made arbitrarily small (at the cost of greater computation time). This test uses Fermat's Theorem. It rejects the hypothesis that an integer, p , is prime if, for an arbitrary integer, $i < p$,

$$i^{p-1} \neq 1 \pmod{p}$$

The AKS test was the first deterministic test for primality. Known after its originators, *Manindra Agrawal, Neeraj Kayal, and Nitin Saxena*, its time complexity is $O(\log^{12} p)$. Its claim to fame is that it is the first *deterministic test* that is *polynomial in log p* and that *holds unconditionally* for all candidate integers, not just those with some specific properties. Since then, there have been improvements to AKS that run in $O(\log^6 p)$ time [LENS05].

6.5.2 Side Channel and Other Attacks

There are several ways in which RSA may be attacked; we highlight three of these here. First, factorizing the RSA modulus is an assault on the very foundations of the RSA algorithm. We examine the state of the art in modulus factorization. We then study an attack on the way the RSA algorithm is used. Finally, we study side-channel attacks which exploit timing or power characteristics of RSA implementations.

Modulus Factorization

Factorization of the modulus n, i.e., obtaining its prime factors, p and q, is one way of attacking the RSA algorithm. From p and q, an attacker can obtain $\Phi(n)$ and thence the decryption key, d (using the extended Euclid's algorithm). There could be ways of breaking RSA and recovering the private key through other routes but to date no such breakthroughs have been reported.

What techniques are used to factorize large RSA moduli and to what extent are they successful? Two early algorithms are attributable to Pollard. In the *Pollard rho* algorithm, for example, integers modulo n are randomly selected. Let these numbers be denoted by

$$r_1, r_2, \ldots$$

For each new integer, r_i, selected, $\gcd(r_i - r_j, n)$ is computed for each $j < i$. We stop generating fresh integers when we find that $\gcd(r_i - r_j, n) > 1$ for some j. Note that this will happen when $r_i - r_j$ is a multiple of p or q. Let us focus on finding an $r_i - r_j$ that is a multiple of p. This occurs when $r_i \bmod p = r_j \bmod p$. It can be shown that we need to select, on average, about $O(\sqrt{p})$, random integers before such a "collision" first occurs. (This is a consequence of a well-known phenomenon called the Birthday Paradox studied in Chapter 7.)

The reader will recognize that this approach is very time-consuming and that the number of gcd operations is $O(p)$. In addition, the amount of memory required to save previous values of r_j is prohibitive. Fortunately, using a clever trick, the Pollard rho method uses a loop that involves only two gcd computations per iteration with $O(1)$ storage. The average number of iterations is $O(\sqrt{p})$. (Details of this algorithm are beyond the scope of this text.) This algorithm is a reasonable choice for factorizing RSA moduli that are tens of digits long. But what about real-world moduli that are hundreds of digits long?

It turns out that, thus far, *no polynomial time algorithm* has been devised for factoring an arbitrarily large integer that is itself the product of two very large integers (hundreds of digits long) of comparable size. The best known factorization algorithms together with their running times are

Quadratic sieve	$O\left(e^{(1+o(1))\sqrt{(\ln n)(\ln \ln n)}}\right)$
Elliptic curve	$O\left(e^{(1+o(1))\sqrt{(2 \ln p)(\ln \ln p)}}\right)$
General Number Field Sieve (GNFS)	$O\left(e^{(1.92+o(1))\sqrt{(\ln n)^{1/3}(\ln \ln n)^{2/3}}}\right)$

What is the largest modulus that is known to have been factorized and what horsepower does this task need?

It is customary to express the amount of computation required for this task in units of MIP-years. One *MIP-year* is the amount of processing power made available by one machine running continuously for a year and executing 1 million instructions per second. To put this speed in perspective, today's mid-range desktop (say a 1.5 GHz Pentium-based machine) delivers about 400 million instructions per second. With today's best known factorization algorithms, the horsepower to factorize a 600-bit modulus is about 8000 MIP-years. This translates to a completion time of 20 years on today's mid-range desktop. One option is to employ parallel processing – parallelize the factorization algorithm and then deploy tens or hundreds of high-end machines to obtain results in a few weeks. We conclude this section by summarizing the latest achievements in this area.

Since the 1990s, RSA Security (now a division of EMC Corporation) has been issuing factorization challenges. After one challenge has been met, a new, larger modulus is offered for factorization. The last and most recent challenge was to factorize a *663-bit* (*200 decimal-digit*) number. This task was successfully performed by Jens Franke et al. from the University of Bonn. This feat was achieved in *May 2005* using the GNFS algorithm running for several months on 80 AMD Opteron CPUs.

Thus far, no one has surpassed the above record by factorizing a modulus larger than 663 bits. Nevertheless, it is expected that 768 bit moduli will be factorizable within the next year or two and 1000 bit moduli in the foreseeable future. Consequently, for sensitive financial and military applications, the use of 1024-bit RSA keys is considered insecure today. In the context of those applications, a *key size of 2048 or larger is recommended.*

Small Exponent Attack

There is a well-known attack on RSA that exploits the idiosyncrasies in the way it is used. Consider the scenario in which a person wishes to send the same message, m, to three different parties. Assume further that each party has the same encryption key = 3. We remarked earlier that this choice of encryption key is very common in actual practice. However, the RSA moduli of the three parties would almost certainly be different – let these be n_1, n_2, and n_3, and let $N = n_1 * n_2 * n_3$.

Now suppose an attacker eavesdrops upon the ciphertexts, c_1, c_2, and c_3. These are related to the message, m, by

$$c_1 = m^3 \bmod n_1$$
$$c_2 = m^3 \bmod n_2$$
$$c_3 = m^3 \bmod n_3$$

Since the prime factors of n_1, n_2, and n_3 will almost certainly be distinct, it follows that n_1, n_2, and n_3 are pairwise relatively prime. Hence, knowing the residues, $m^3 \bmod n_1$, $m^3 \bmod n_2$, and $m^3 \bmod n_3$, we can use the Chinese Remainder Theorem (Section 3.4) to reconstruct $m^3 \bmod N$.

Since $m < n_1$, $m < n_2$ and $m < n_3$, $m^3 < N$.
Hence, $m^3 \bmod N = m^3$ and so $m = (m^3 \bmod N)^{1/3}$.

A more obvious attack with an encryption key, $e = 3$, occurs if an attacker knows or guesses that the message $m < N^{1/3}$. In this case, the operation "cube root modulo N" on the ciphertext reduces to the regular algebraic cube root of an integer.

Side Channel Attacks

Traditionally, attacks have been launched based on knowledge of the ciphertext (through eavesdropping on communication channels) and/or partial knowledge of the plaintext (by intelligent guesses). Factorization, on the other hand, is an attack on the mathematical foundations of RSA, while the "Exponent-3" attack targeted the way RSA was used. A very different class of attacks is based on monitoring, for example, the *timing* or *power measurements* of a cryptographic algorithm on a device. These attacks have been shown to be quite successful in leaking sensitive information such as secret/private keys. This is especially the case for *embedded devices* such as *smart cards*.

A smart card is a credit size card used to store details of its owner. It may be used for multiple applications such as a driver's license, a credit/debit card, an identity token, etc. Higher end smart cards have a processor and even a crypto accelerator if used for security applications. The smart card is often used to securely store the owner's private key. Private key operations are performed on the smart card itself. A number of companies manufacture tamper-proof smart cards with a guarantee against being able to ever retrieve the private key from the card. However, a variety of subtle attacks on smart cards, which seek to deduce secrets such as the owner's private key, are possible and are further explored in this section.

It may be relatively easy for an attacker to steal or borrow a smart card. The next task of the attacker is to induce the card to perform cryptographic tasks involving the stored private key. It is generally not possible for the attacker to inspect the contents of registers and the RAM during smart card operations. However, there is inexpensive, off-the shelf equipment available that enables him/her to connect a smart card via probes to equipment that can accurately monitor variables such as timing and power consumption. Because these attacks access such (non-traditional) channels, they are referred to as *side-channel attacks*. Compared to such channels in the case of servers or PCs, side channels from embedded devices are less noisy since cryptographic operations are usually performed with little interference from other operations or processes within the embedded device.

The leakage of key information in a side channel attack is not due to a poor cryptographic algorithm but due to the peculiarities of its implementation. Consider the snippet of code in Fig. 6.6 that performs RSA decryption using the "*Square and Multiply*" technique. The decryption key, d, is assumed to be a k–bit integer with the most significant bit = 1.

```
Given $d$, $n$, and $c$
Want: $c^d$ mod $n$

// $k \equiv \log_2 n$ is the key size
$x = c$
for ($i = k - 2$; $i \geq 0$; $i$--)
    $x = x^2$ mod $n$;              // Square operation
if ($d_i == 1$)
    $x = x \times c$  mod $n$;      // Multiply operation
return ($x$);
```

Figure 6.6 *Implementation of square and multiply with conditional multiply*

In Fig. 6.6, the *square operation* is executed in each iteration. However, the *multiply operation* is skipped if the corresponding bit in the decryption key, d, is a 0. Conditional execution of this type may be exploited by an attacker to deduce, for example, the number and positions of 1's in d. Here are some possible strategies.

The attacker may carefully monitor the power consumed by the smart card over the duration of the decryption. If the power consumption characteristics of the square and multiply operations are dissimilar, then the attacker can identify the iterations during which the multiply operation is skipped. From this, he/she can readily deduce the *position of 1's in the decryption key*, d, as shown in Fig. 6.7. Note that these operations involve modulo n reductions. If the result obtained after performing the multiplication or square operation is less than n, then no reduction is required. Hence, the times for multiplication and squaring are not constants but depend upon the input, c.

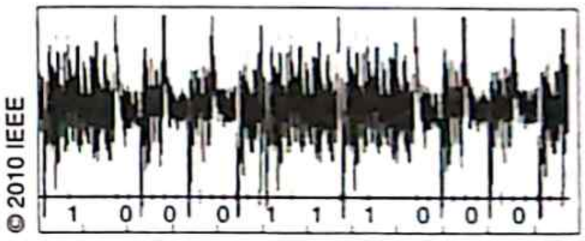


Figure 6.7 *Power profile of decryption operation using implementation of Fig. 6.6**

If the attacker has access to an identical smart card, the attacker may be able to study the times and power consumption for the multiply and square operation. He/she may experiment with different inputs, c, and also with different decryption keys. This study may provide further insights into timing and power requirements.

To thwart the above side channel attacks, the implementation of Fig. 6.8 may be employed.

Now, notice that a multiply operation is always performed in each iteration regardless of the value of the bit in d inspected during that iteration. Thus, the attacker will be unable to launch a successful side channel attack based on timing and power measurements.

Another class of side channel attacks occurs by carefully inducing *transient faults* into the chip in a smart card. How are faults injected into a chip and what effect do they have?

**Source:* Santosh Ghosh, Monjur Alam, Dipanwita Roy Chowdhury, Indranil Sen Gupta, "Effect of Side Channel Attacks on RSA Embedded Devices", *IEEE TENCON*, 2007, Taipei.

```
Given:  $d$, $n$, and $c$
Want:   $c^d \bmod n$

// $k \equiv \log_2 n$ is the key size
$x = c$
for ( $i = k - 2$; $i \geq 0$; $i$-- ) {
    $x = x^2 \bmod n$           // Square operation
    $y = x \times c \bmod n$       // Unconditional
                                // Multiply operation
    if ($d_i == 1$)
      $x = y$
}
return ($x$)
```

Figure 6.8 *Implementation of square and multiply with unconditional multiply*

Around 40 years ago, it was noticed that *radioactive particles* produced by heavy metals such as uranium and thorium caused electronic hardware to malfunction. These metals were present in very tiny quantities in the packaging material around the chip and caused bits in a processor to randomly flip. Since then, sophisticated techniques including those using highly focused *laser beams* have been used to target very specific parts of an embedded processor at specific points during the execution of a given operation.

Other techniques at injecting faults manipulate the *voltage supply* or the *clock* to a smart card. Most smart cards require an external voltage supply and an external clock input. *Glitches* in execution may occur when very high or low clock frequencies are applied or when spikes in the voltage supply are introduced. The effects of such input may cause instructions to be skipped, data to be corrupted, etc. However, the relevant question is "How does the induction of a fault help the attacker deduce the key?"

To answer this question, we turn to Fig. 6.8, which shows the implementation of the Square and Multiply algorithm with the unconditional multiply operation. Now suppose the attacker injects a fault in the system during the multiply step of a certain iteration. Then the result of the multiply operation will be erroneous. If the bit of the decryption key during that iteration is a 0, then the multiply instruction is superfluous and will not affect the final decrypted output. However, if the bit of the decryption key during that iteration is a 1, then the multiply instruction is necessary and an error in it will lead to a faulty decrypted value.

To obtain the decryption key, this experiment would have to be repeated *d* times. Each time a fault is injected in the multiply instruction of a different iteration. The result of the decryption is compared to the correct value (without faults). If the result does not match the correct value, the attacker infers that the bit of the decryption key is a 1.

In Exercise 6.12, a countermeasure against a fault induction attack is studied [JOYE02]. We conclude this section by noting that side channel attacks are, at least in part, a consequence of the need to optimize a design for speed, chip area, power requirements, etc. All these are especially important for embedded applications. As is often the case, security is often sacrificed for lower cost and higher performance. For example, Exercise 6.8 illustrates how RSA performance can be improved using the Chinese Remainder Theorem, while Exercise 6.9 illustrates a serious side channel vulnerability using this optimization.

6.6 PUBLIC KEY CRYPTOGRAPHY STANDARD (PKCS)

A solution to the problems with small encryption keys is to pad the message with non-zero random bits before performing encryption. The number and position of these random bits has been standardized so that the receiver does not misinterpret the random bits for data. Padding is also important if the message contains data that can be guessed. An attacker could guess the plaintext, then encrypt it with the public key, and verify whether its encrypted version coincides with the ciphertext sent. He/she could repeat this sequence of guess–encrypt–verify until a match is found between his/her encrypted value and the ciphertext sniffed by him/her.

The *Public Key Cryptography Standard* (PKCS # 1) specifies, among other things, the format of each block to be encrypted by RSA. As shown in Fig. 6.9, the bytes of the block from left (most significant) should be 00 followed by the byte 02 (in hexadecimal) followed by *at least eight* random non-zero bytes and another 00. The rest of the block is composed of data [Fig. 6.9(a)]. The 00 to the right of the random byte string indicates the start of the data section in a block.

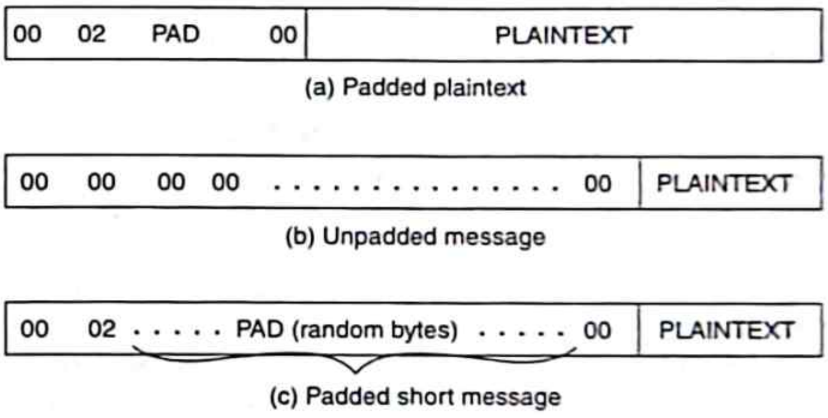

00 02 PAD 00 PLAINTEXT
(a) Padded plaintext
00 00 00 00 . . . . 00 PLAINTEXT
(b) Unpadded message
00 02 . . . . . PAD (random bytes) . . . . . 00 PLAINTEXT
(c) Padded short message

Figure 6.9 *Use of plaintext padding in RSA encryption*

By padding a short message with random bytes in a block of plaintext, the ciphertext will be a function of not just the short message as in Fig. 6.9(b), but also a function of the large sequence of random bytes [Fig. 6.9(c)]. To be successful, an attacker will have to guess not merely the message but also the random bytes. The problems discussed in the last section due to the use of the small exponent such as $e = 3$ are also solved by padding the block of plaintext.

PKCS # 1 is one of a set of 15 standards for Public Key Cryptography developed by RSA Laboratories. The PKCS standards include algorithm-independent syntax for many of the artefacts that we study later in this text – digital signatures, digital envelopes, extended certificates, etc. Details of these standards are found in [RSAL].

SELECTED REFERENCES

One of the earliest documents on RSA is [RIVE78]. An excellent catalogue of attacks on RSA is [BONE99]. There are several references for side channel attacks. One of the earliest is [KOCH99]. A useful survey is [GIRA04]. A recent survey of fault attacks is [BAR06]. A standard scheme for padding in connection with RSA encryption is in [BELL94].

OBJECTIVE-TYPE QUESTIONS

6.1 The relation between the RSA encryption and decryption keys is
(a) $ed \equiv 1 \bmod n$ (b) $ed \equiv 1 \bmod \Phi(n)$
(c) $ed \equiv 0 \bmod n$ (d) $ed \equiv 0 \bmod \Phi(n)$

6.2 Let the RSA modulus, n, be 77. If the encryption key is 7, the decryption key is
(a) 51 (b) 29 (c) 43 (d) 58

6.3 In the above example, if the ciphertext = 5, the corresponding plaintext is
(a) 26 (b) 33 (c) 44 (d) 71

6.4 The time complexities of RSA encryption and decryption (as a function of key size, k) are, respectively,
(a) $O(k^3)$ and $O(k^2)$ (b) $O(k)$ and $O(k^2)$
(c) $O(k^4)$ and $O(k^3)$ (d) $O(k^2)$ and $O(k^3)$

6.5 The principal advantage of public key cryptography over secret key cryptography is
(a) simplified key management (b) lower chip area
(c) improved speed (d) higher security

6.6 What characteristics of an implementation does a side channel attack exploit?
(a) Power consumption (b) Timing
(c) Chip area (d) Algorithmic optimizations

6.7 The main purpose of plaintext padding is to
(a) prevent side channel attacks (b) improve the speed of decryption
(c) prevent plaintext guessing (d) prevent known plaintext attacks

EXERCISES

6.1 The modulus in a toy implementation of RSA is 143.
(a) What is the smallest value of a valid encryption key and the corresponding decryption key?
(b) For the computed encryption key and plaintext = 127, what is the corresponding ciphertext?
(c) For the computed decryption key and ciphertext = 2, what is the corresponding plaintext?

6.2 If the RSA public key is (31, 3599), what is the corresponding private key? (Note that the encryption key is 31 and the modulus is 3599.)

6.3 The following is a valid decryption key for a 1024-bit modulus. Yes or no?

995810327599534649709635311706213974167121442371108705052534808
082717631413914578382358641994087508851470566171200118061886356
927857128023895937686124596803547724139368691984016484827565532
619882877414653050192475986143342001209708918739890065984251840
06657495148764873646150218997013746881994622763397581376

Explain why or why not.

6.4 If an attacker gets to know $\Phi(n)$, will he be able to obtain p and q? If so, how?

6.5 Consider the following modified definition of an RSA decryption key, d'

$$d' = e^{-1} \bmod \Phi' \qquad \text{where}$$

$$\Phi' = \frac{\Phi(n)}{\gcd(p-1, q-1)}$$

Show that RSA decryption works even under the new definition of the decryption key, i.e.,

$$m = c^{d'} \pmod{n}$$

Here, p, q, n, $\Phi(n)$, m, and c are as defined in the text.

6.6 An entity sends the same message, m, to three parties, X, Y, and Z, each encrypted with their respective public keys. Assume that the public key moduli of the three parties are n_x, n_y, and n_z. If the encryption key in each case = 3, we examined how an eavesdropper could recover the original message, m, using the Chinese Remainder Theorem.

Suppose the encryption keys were each = 5 instead, would a similar attack be possible? Explain why or why not.

6.7 The number of 1024-bit primes is roughly 10^x. What is x?

Hint: The number of primes less than m is asymptotically $\dfrac{m}{\ln m}$.

Assume that each human on the planet has a 2048-bit public key/private key pair. What is the probability that no two humans have a common factor in their RSA moduli? Assume that the population of the planet is about 7 billion.

6.8 Consider the following trick to speed up RSA decryption. Compute

$$\begin{aligned}
d_p &= d \bmod (p-1)\\
d_q &= d \bmod (q-1)\\
M_p &= q^{-1} \bmod p\\
M_q &= p^{-1} \bmod q\\
x_p &= c^{d_p} \bmod p\\
x_q &= c^{d_q} \bmod q
\end{aligned}$$

Here, p, q, d, and c are as defined in the text.

(a) Use the Chinese Remainder Theorem to show that the required plaintext is

$$(M_p\, q\, x_p + M_q\, p\, x_q) \bmod n$$

(b) Calculate the savings in time as a percentage of the time taken to do decryption using $m = c^d \bmod n$. Assume that d_p, d_q, M_p, and M_q are pre-computed (note that this can be done since they are not functions of c).

6.9 The previous exercise illustrates how the Chinese Remainder Theorem may be used to speed up private key operations. This is an especially attractive option in resource-constrained devices such as smart cards. However, consider the following side channel attack. A fault is induced during the computation of x_p in the previous exercise so a value x_p' is computed instead. The plaintext is thus computed as

$$(M_p\, q\, x_p' + M_q\, p\, x_q) \bmod n$$

Show that leakage of this erroneous value enables an attacker to factorize n.

6.10 A 60-byte quantity was encrypted/decrypted using the Bouncy Castle Java APIs for RSA. The measured times (ms) for different key sizes are summarized below.

Key Size	*Encryption*	*Decryption*
1024	0.44 (a)	8.60 (b)
2048	1.63 (c)	55.30 (d)
3072	4.02 (e)	173.95 (f)
4096	5.77 (g)	445.15 (h)

Exactly ONE entry in each column (Enc. and Dec.) has been misreported.
Identify the incorrect entries. (Knowledge of the underlying hardware platform, operating system, and compiler are not necessary to answer this question.)
Explain why they are incorrect.

6.11 Suppose you find two integers a and b, $a \neq b$ and $a \neq -b$, such that $a^2 = b^2 \pmod n$ where n is an RSA modulus. How would this help you to factorize n?

6.12 The following are three implementations of the "*Square and Multiply*" technique used in modular exponentiation.

Given: d, n, and m
Want: $m^d \bmod n$
// $k \equiv \log_2 n$ is the key size

Implementation 1

```
x = m
for (i = k − 2; i ≥ 0; i--)
    x = x² mod n
  if (d_i = =1)
    x = x × m mod n
return (x)
```

Implementation 2

```
x = m
    for (i = k − 2; i ≥ 0; i--)
  x = x² mod n
  y = x × m mod n
if (d_i = =1)
  x = y
return (x)
```

Implementation 3

```
x[0] = __ ; x[1] = __
  for (i = k − 1; i ≥ 0; i--)
x[d̄_i] = x[d̄_i] x[d_i]
            mod n
x[d_i] = x[d_i] x[d_i]
            mod n
return (x[0])
```

Fill the blanks in Implementation 3.

For the three implementations above, indicate with a *Y* or *N* in the matrix below whether it is *resistant* to each type of side channel attack.

	Timing	*Power*	*Fault Induction*
Implementation 1			
Implementation 2			
Implementation 3			

6.13 The Bouncy Castle C++ or Java APIs are available at http://www.bouncycastle.org/. Use these APIs to do the following:

(a) Create an RSA public key/private key pair. Display the values of p, q, n, $\Phi(n)$ for different key sizes.

(b) For each key, encrypt a message. Change a single bit in the message and display the new ciphertext. How different is it from the old ciphertext?

(c) Decrypt the ciphertext and verify whether you get the corresponding plaintext.

(d) Time the different operations – key generation, encryption, and decryption. Compare the execution times for different key sizes. What conclusions do you draw?

ANSWERS TO OBJECTIVE-TYPE QUESTIONS

6.1 (b)	6.2 (c)	6.3 (a)	6.4 (d)
6.5 (a)	6.6 (a)(b)	6.7 (c)	

Chapter 7

Cryptographic Hash

7.1 INTRODUCTION

A hash function is a deterministic function that maps an input element from a larger (possibly infinite) set to an output element in a much smaller set. The input element is mapped to a *hash value*. For example, in a district-level database of residents of that district, an individual's record may be mapped to one of 26 hash buckets. Each hash bucket is labelled by a distinct alphabet corresponding to the first alphabet of a person's name. Given a person's name (the input), the output or hash value is simply the first letter of that name (Fig. 7.1).

Hashes are often used to speed up insertion, deletion, and querying of databases. In the example above, two names beginning with the same alphabet map to the same hash bucket and result in a *collision*. A good hash function would tend to map an equal number of inputs to each hash bucket – this is unlikely to be the case with the above hash function.

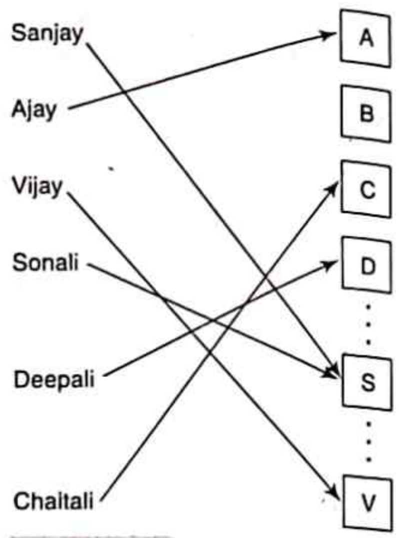


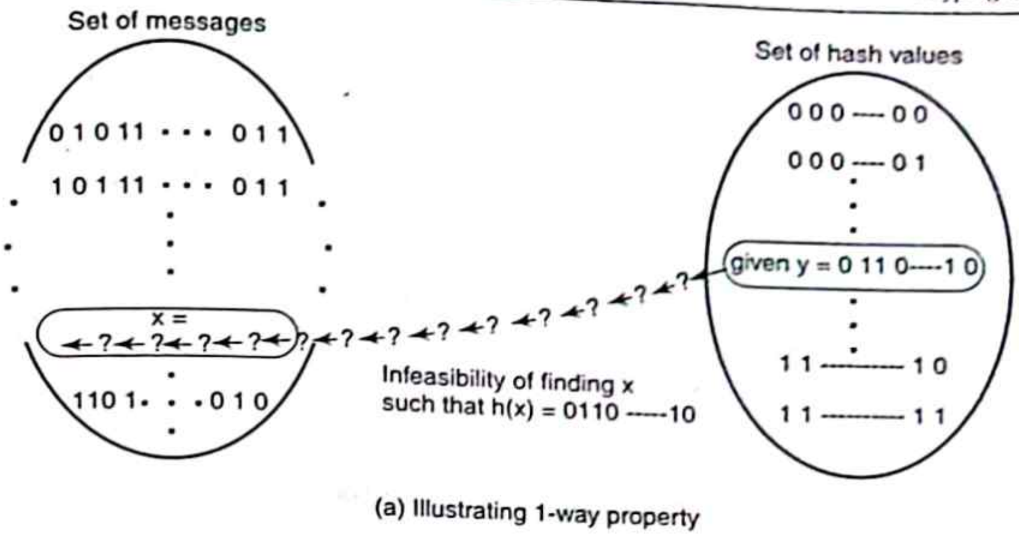
Figure 7.1 *Elementary hash function*

7.2 PROPERTIES

7.2.1 Basics

A cryptographic hash function, $h(x)$, maps a binary string of *arbitrary length* to a *fixed length* binary string. The properties of h (illustrated in Fig. 7.2) are as follows:

- *One-way property*. Given a hash value, y (belonging to the range of the hash function), it is computationally infeasible to find an input x such that $h(x) = y$.
- *Weak collision resistance*. Given an input value x_1, it is computationally infeasible to find another input value x_2 such that $h(x_1) = h(x_2)$.
- *Strong collision resistance*. It is computationally infeasible to find two input values x_1 and x_2 such that $h(x_1) = h(x_2)$.
- *Confusion + diffusion*. If a single bit in the input string is flipped, then each bit of the hash value is flipped with probability roughly equal to 0.5.



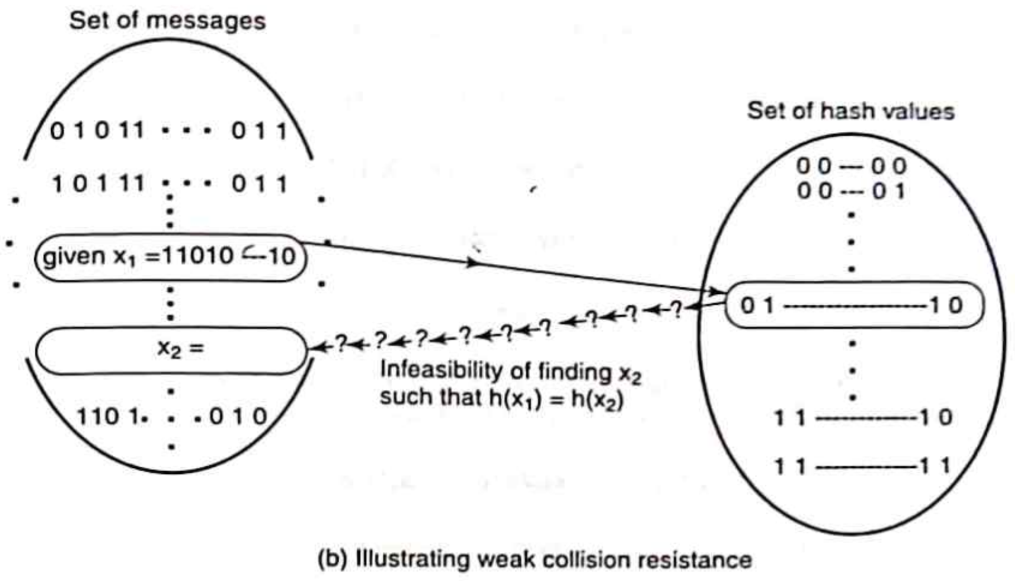


Figure 7.2 *Properties of the cryptographic hash*

Note that there are an infinite number of input strings of arbitrary length, but there are only a finite (though large) number of hash values. For a well-designed hash function, all hash values are equally probable for a random string. Hence, there are an infinite number of inputs that map to the same hash value or digest. However, given an arbitrary hash value, it should be beyond the reach of any supercomputer to find even one of those inputs!

There is a subtle difference between the two collision resistance properties. In the first, the hash designer chooses x_1 and challenges anyone to find an x_2, which maps to the same hash value as that of x_1. This is a more *specific challenge* compared to the one in which the attacker tries to find x_1 and x_2 such that $h(x_1) = h(x_2)$. In the second challenge, the attacker has the *liberty* to choose x_1.

If an attacker can break the first hash collision resistance property, then trivially, he can break the second hash collision resistance property. Indeed, as we will see in the next section, a successful response to the second challenge takes much less time compared to the first challenge. From the *hash designer's perspective*, if an attacker cannot even break the second (and easier) hash collision resistance property, then the cryptographic hash is "strongly collision resistant."

Figure 7.3 shows the hash output or *digest* of a 4 KB message. The 160-bit SHA-1, which is one of the standard cryptographic algorithms, was used. A single bit in the message was flipped and the SHA-1 hash was re-computed. As shown in Fig. 7.3, 84 out of 160 bits of the hash value were flipped. Moreover, the flipped bits are randomly diffused over the entire hash value.

SHA-1 Digest (160 bits) of a 4KB message

```
11101001 11011101 11101101 10001100

10000100 01100001 01001110 10001001

01000101 00000001 10010110 01011010

11110110 00001100 00100101 00100101

01111001 01001010 10001100 01111101
```

Digest after changing second last bit of the same message
(changed bits are shown against grey background)

```
10001010 11100101 10010011 10001000

11101010 10000101 01000111 01010000

01001100 01111110 10101101 11110001

00100000 00011101 10000111 11111110

00010010 00111001 00011001 11010010
```

Figure 7.3 *SHA-1 message digest of a 4KB message*

7.2.2 Attack Complexity

Weak Collision Resistance

How long would it take to find an input, x, that hashes to a given value y?

Assume that the hash value is w bits long. So, the total number of possible hash values is 2^w. Now a brute force attempt to obtain x would be to loop through the following operations

```
do
{
    generate a random string, x'
    compute h(x')
}
while (h(x') != y)
return (x')
```

Assuming that any given string is equally likely to map to any one of the 2^w hash values, it follows that the above loop would have to run, on the average, 2^{w-1} times before finding an x' such that $h(x') = y$.

A similar loop could be used to find a string, x_2, that has the same hash value as a given string x_1. Thus, successful brute-force attacks on both the *one-way function* property and *weak collision resistance* take $O(2^w)$ time.

Strong Collision Resistance

A brute-force attack on strong collision-resistance of a hash function involves looping through the program in Fig. 7.4. Unlike the program that attacks weak collision resistance, this program

```
// S is the set of (input string, hash value) pairs
// encountered so far

notFound = true
while ( notFound )
{
    generate a random string, x'
    search for a pair ( x, y) in S where x = x'
    if ( no such pair exists in S )
    {
        compute y' = h(x')
        search for a pair (x, y) in S where y = y'
        if ( no such pair exists in S )
            insert (x', y') into S
        else
            notFound = false
    }
}
return   ( x and x' )   // these are two strings that have
                        // the same hash value
```

Figure 7.4 Program to attack strong collision resistance

terminates when the hash of a newly chosen random string collides with *any* of the previously computed hash values.

In the Appendix, we show that we need to generate only about $O(\sqrt{n})$ random strings before detecting a repeat in the hash values computed so far. Here, n is the total number of possible hash values. Let w denote the width of a hash value as before. So, $n = 2^w$. We thus find that we need to generate roughly $O(\sqrt{n}) = O(2^{w/2})$ input strings before a hash collision is detected. Note that this involves considerably less work than successfully attacking the weak collision-resistance property which, as described in the previous section, takes $O(2^w)$ time.

The Birthday Analogy

Attacking *strong collision resistance* is analogous to answering the following:

> "What is the minimum number of persons required so that the probability of two or more in that group having the same birthday is greater than ½ ?"

It is known that in a class of only 23 random individuals, there is a greater than 50% chance that the birthdays of at least two persons coincide (a "Birthday Collision"). This statement is referred to as the *Birthday Paradox*.

The random strings generated in Fig. 7.4 are analogous to the random individuals in the Birthday Paradox. The birthday of a randomly chosen individual is analogous to the hash value of a randomly chosen string.

On the other hand, the question posed in the context of *weak collision resistance* is analogous to

> "What is the minimum number of persons required in Tom's class so that the probability of at least one other person sharing Tom's birthday is greater than ½?"

In the latter case, we require Tom's class to have about 365/2 or ~ 183 persons, a much larger number than 23 required by the Birthday paradox.

7.3 CONSTRUCTION

7.3.1 Generic Cryptographic Hash

The input to a cryptographic hash function is often a message or document. To accommodate inputs of arbitrary length, most hash functions (including the commonly used MD-5 and SHA-1) use an *iterative construction* as shown in Fig. 7.5. C is a *compression box*. It accepts two binary strings of lengths b and w and produces an output string of length w. Here, b is the block size and w is the width of the digest.

During the first iteration, the multiplexer at the second input in Fig. 7.5 accepts a pre-defined initialization vector (IV), while the top input is the first block of the message. In subsequent iterations, the "partial hash output" is fed back as the second input to the C-box. The top input is derived from successive blocks of the message. This is repeated until all the blocks of the message have been processed. The above operation is summarized below:

$$\begin{aligned} h_1 &= C\,(IV, m_1) && \text{for first block of message} \\ h_i &= C\,(h_{i-1}, m_i) && \text{for all subsequent blocks of the message} \qquad (7.1) \end{aligned}$$

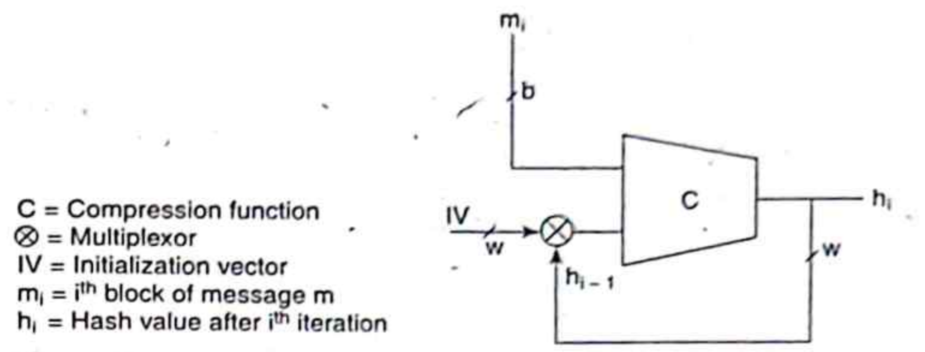

C = Compression function
⊗ = Multiplexor
IV = Initialization vector
m_i = i^th block of message m
h_i = Hash value after i^th iteration
m_i
b
IV
w
C
h_i
w
h_{i-1}

Figure 7.5 *Iterative construction of cryptographic hash*

The above iterative construction of the cryptographic hash function is a simplified version of that proposed by Merkle [MERK89] and Damgard [DAMG89]. It has the property that *if the compression function is collision-resultant, then the resulting hash function is also collision-resultant*. The converse of this result is, however, not necessarily true.

Many of the widely used cryptographic hash functions are based on the above iterative construction; *MD-5* and *SHA-1* are the best known examples. MD-5 is a 128-bit hash, while SHA-1 is a 160-bit hash. (MD stands for Message Digest and SHA stands for Secure Hash Algorithm.) Other examples include *SHA-256* and *SHA-512*, which are 256 and 512 bit hash functions, respectively. We next describe the construction of SHA-1.

7.3.2 Case Study: SHA-1

SHA-1 uses the iterative hash construction of Fig. 7.5. The message is split into *blocks of size 512 bits*. The length of the message, expressed in binary as a 64 bit number, is appended to the message. Between the end of the message and the length field, a pad is inserted so that the length of the (message + pad + 64) is a multiple of 512, the block size. The pad has the form: 1 followed by the required number of 0's.

We now describe how the SHA-1 hash of a message is computed.

Array Initialization

Each block is split into 16 words, each 32 bits wide. These 16 words populate the first 16 positions, $W_1, W_2, \ldots, W_{16}$, of an *array of 80 words*. The remaining 64 words are obtained from

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16} \qquad 16 < i \leq 80 \tag{7.2}$$

This array of words is shown in Fig. 7.6.

Hash Computation

A *160-bit shift register* is used to compute the intermediate hash values (Fig. 7.6). It is initialized to a fixed pre-determined value at the start of the hash computation. We use the notation *S1*, *S2*, *S3*, *S4*, and *S5* to denote the five 32-bit words making up the shift register. The bits of the shift register are then mangled together with each of the words of the array in turn. The mangling is achieved using a combination of the following Boolean operations: – +, ∨, ⊕, ∧, ROTATE.

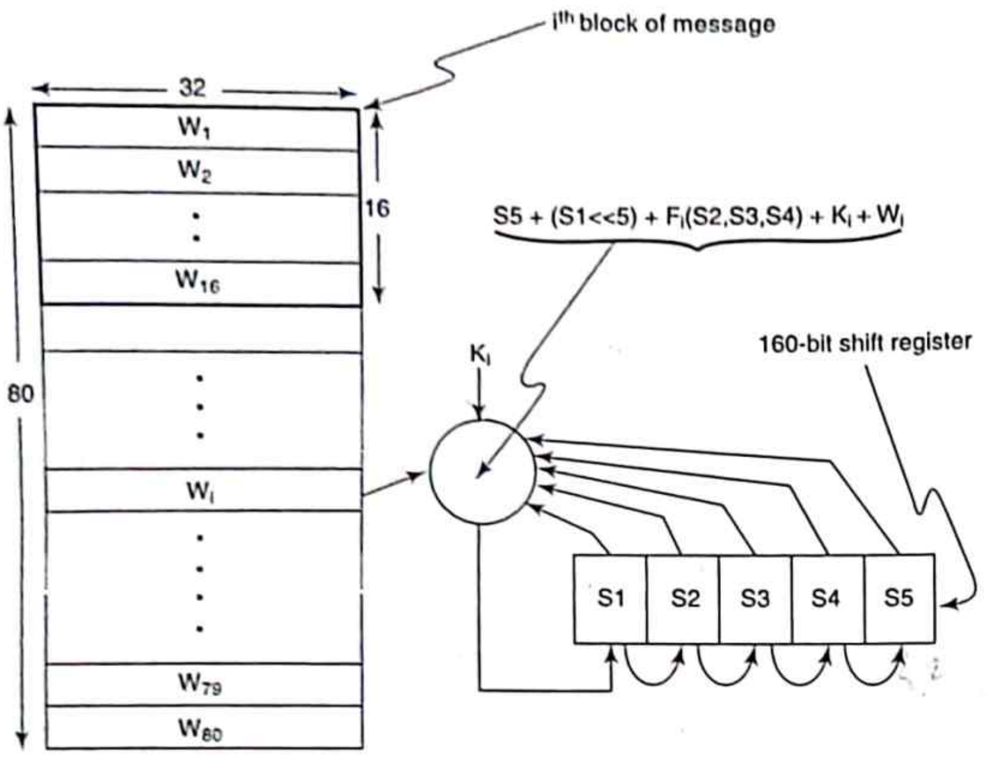


Figure 7.6 *Computation of SHA-1*

The SHA-1 hash of a message is the content of the shift register after all message blocks have been processed using the procedure below.

```
initialize the shift register, S1 S2 S3 S4 S5
for each block of the (message + pad + length field) {
        create the 80-word array [using Eq. (7.2)]
        for $i$ = 1 to 80 {
                temp ← $S5 + (S1 << 5) + F_i(S2, S3, S4) + K_i + W_i$
                S5 ← S4
                S4 ← S3
                S3 ← S2 >> 2
                S2 ← S1
                S1 ← temp
        }
}
```

The notation *S1* << *5* denotes a rotation of *S1* by *5* bit positions to the left. Likewise, *S2* >> 2 indicates a rotation of *S2* by two positions to the right. The initial values in *S1*, *S2*, *S3*, *S4*, and *S5* and the constants K_i, $1 \le i \le 80$ are all predetermined. The function, F_i, is defined below.

$$\begin{aligned}
F_i\,(S2, S3, S4) &= (S2 \wedge S3) \vee (\sim S2 \wedge S4), && 1 \le i \le 20\\
F_i\,(S2, S3, S4) &= S2 \oplus S3 \oplus S4, && 21 \le i \le 40\\
F_i\,(S2, S3, S4) &= (S2 \wedge S3) \vee (S2 \wedge S4) \vee (S3 \wedge S4), && 41 \le i \le 60\\
F_i\,(S2, S3, S4) &= S2 \oplus S3 \oplus S4 && 61 \le i \le 80 \qquad (7.3)
\end{aligned}$$

The cryptographic hash has diverse applications ranging from secure storage of passwords to electronic payments. In conjunction with secret key cryptography or public key cryptography, it is used in numerous security protocols to provide authentication, data integrity, and non-repudiation (Chapter 11). We next introduce some of its most important applications.

7.4 APPLICATIONS AND PERFORMANCE

We first study how the cryptographic hash is used to provide message authentication and message integrity. We then highlight its role in the generation of the digital signature.

7.4.1 Hash-based MAC

In Chapter 5, we introduced the *Message Authentication Code* (MAC) and implemented it using DES in CBC mode – such an implementation is called the CBC-MAC. Recall that a MAC is used as a message integrity check as well as to provide message authentication. It makes use of a common shared secret, k, between two communicating parties. The hash-based MAC that we now introduce is an alternative to the CBC-MAC.

The cryptographic hash applied on a message creates a *digest* or *digital fingerprint* of that message. Suppose that a sender and receiver share a *secret*, k. If the message and secret are concatenated and a hash taken on this string, then the hash value becomes a fingerprint of the *combination* of the message, m and the secret, k.

$$\text{MAC} = h\ (m \,\|\, k) \tag{7.4}$$

The MAC is much more than just a *checksum* on a message. It is computed by the sender, appended to the message, and sent across to the receiver. On receipt of the message + MAC, the receiver performs the computation in Eq. 7.4 using the common secret and the received message. It checks to see whether the MAC computed by it matches the received MAC. A change of even a single bit in the message or MAC will result in a mismatch between the computed MAC and the received MAC. In the event of a *match*, the receiver concludes the following:

(a) The sender of the message is the same entity it shares the secret with – thus the MAC provides *source authentication*.

(b) The message has not been corrupted or tampered with in transit – thus the MAC provides verification of *message integrity*.

It is instructive to ask, "In what way are the properties of the cryptographic hash – the one-way property and collision resistance – relevant to the security provided by the MAC?"

An attacker might obtain one or more message–MAC pairs in an attempt to determine the MAC secret. First, if the hash function is *one-way*, then it is not feasible for an attacker to deduce the input to the hash function that generated the MAC and thus recover the secret. If the hash function is *collision-resistant*, then it is virtually impossible for an attacker to suitably modify a message so that the modified message and the original both map to the same MAC value.

There are other ways of computing the hash MAC besides Eq. (7.4). Another possibility is to use the key itself as the Initialization Vector (IV) instead of concatenating it with the message. In either case, the message length should be also factored in as was done in the construction of the SHA–1. We conclude this subsection by introducing the HMAC.

Bellare, Canetti, and Krawczyk [BELL96] proposed the *HMAC* and showed that their scheme is secure against a number of subtle attacks on the simple hash-based MAC. Figure 7.7 shows how an HMAC is computed given a key and a message.

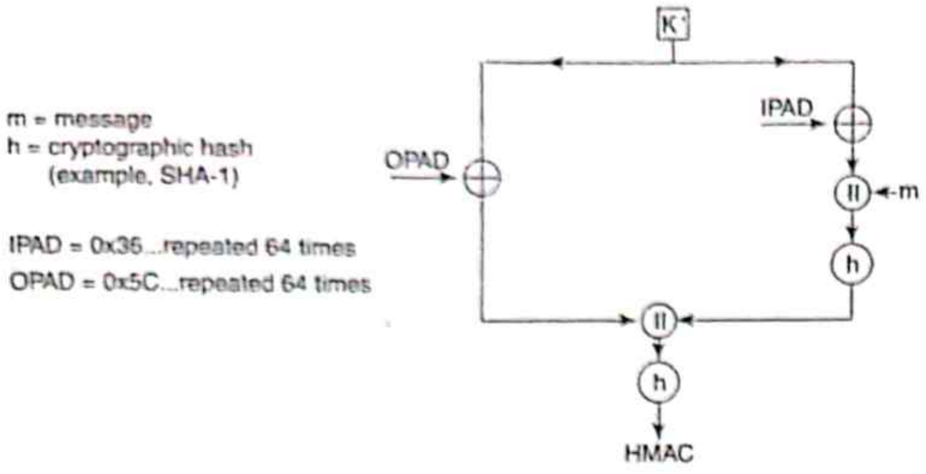


Figure 7.7 Computation of an HMAC

- The key is padded with 0's (if necessary) to form a 64-byte string denoted K' and XORed with a constant (denoted IPAD).
- It is then concatenated with the message and a hash is performed on the result.
- K' is also XORed with another constant (denoted OPAD) after which it is prepended to the output of the first hash.
- A second hash is then computed to yield the HMAC.

As shown in Fig. 7.7, the HMAC performs an extra hash computation but provides greatly enhanced security.

7.4.2 Digital Signatures

The same secret that is used to generate a MAC on a message is the one that is used to verify the MAC. Thus the MAC secret should be known by both parties – the party that generates the MAC and the party that verifies it. A digital signature, on the other hand, uses a secret that only the signer is privy to. An example of such a secret is the signer's *private key*. A crude example of an RSA signature by A on message, *m*, is

$$E_{A.pr}(m) \quad \text{where } A.pr \text{ is A's private key.}$$

The use of the signer's private key is a fundamental aspect of signature generation. Hence, a message sent together with the sender's signature guarantees not just *integrity* and *authentication* but also *non-repudiation*, i.e., the signer of a document cannot later deny having signed it since she alone has knowledge or access to her private key used for signing.

We saw in Chapter 6 that RSA private key operations are very expensive. One way to improve signing efficiency is to perform the private key operation on just the hash of the message rather than on the entire message. The RSA signature is then

$$E_{A.pr}(h(m)) \tag{7.5}$$

We elaborate on the properties of such a signature next.

The RSA signature is *authentic*. Insofar as no two signers have distinct private keys[1], the signatures on the same document by different individuals will be different. Given a signed document, how do

[1] For RSA key sizes (or moduli) equal to 1024 bits or more, the probability of two persons having the same modulus (and hence private key) is infinitesimally small.

we verify its authenticity? The verifier needs to perform only a public key operation on the digital signature (using the signer's public key) and a hash on the message. The verifier concludes that the signature is authentic if the results of these two operations tally, i.e.,

$$E_{A.pu}\left(E_{A.pr}(h(m))\right) \overset{?}{=} h(m) \tag{7.6}$$

Just because a document or message contains A's signature, does it mean that it was actually signed by A? With regular manual signatures, this need not be the case since signatures may be *forged*. In the electronic world, a signature can be forged if one has knowledge of or access to the signer's private key. So long as the latter is secure, a digital signature cannot be forged.

To the extent that one can ascertain that a signature over a given message is unmistakably A's and that a digital signature cannot be forged, it follows that A cannot repudiate his signature over a document. Thus, a digital signature performs not merely message authentication and message integrity checking but also non-repudiation. By contrast, the MAC performs only the first two functions.

We remark on one feature of a digital signature that stands out in comparison with a manual signature. A manual signature depends only on the signer and does not change from document to document. On the other hand, a person's digital signature is also a function of the document that is being signed. This is often mystifying to some who encounters digital signatures for the first time but is easily demystified by inspection of Eq. 7.5.

7.4.3 Performance Estimates

How expensive (computationally) is the cryptographic hash vis-à-vis secret key and public/private key operations? Table 7.1 compares the times to compute the cryptographic hash versus the times to perform encryption/decryption using 128-bit DES and 1024-bit RSA. The input file size in each case is 100 KB. All measurements are reported using the Bouncy Castle Java APIs running on a Pentium-IV 3.2 GHz machine with 1 GB RAM and 2 X 2 MB L2 cache. The OS is a Windows XP Service Pack 2.

Table 7.1 *Computation times for SHA-1, DES, and RSA*

SHA-1	*DES (128-bit)*		*RSA (1024-bit)*	
	Encryption Time (µsec)	Decryption Time (µsec)	Encryption Time (µsec)	Decryption Time (µsec)
1844	3900	4000	520,000	10,020,000

Input in all cases is a 100 KB file.

From Table 7.1, it is clear that DES is more than twice as expensive compared to SHA-1. RSA, however, is about three orders of magnitude more expensive compared to DES and SHA-1. Also, note that RSA decryption is much more expensive compared to RSA encryption. This is consistent with results reported in Chapter 6 and the fact that the time complexity of RSA encryption is $O(k^2)$, while that of RSA decryption is $O(k^3)$, where k is the key size.

Table 7.2 compares the cost of RSA signature generation versus signature verification for two different message sizes and key sizes. Note that signature generation is much more expensive compared to signature verification. This is because RSA signature generation involves a private key operation, while signature verification involves a public key operation.

Table 7.2 also includes the time to perform a SHA-1 digest on the message. Note that the time to compute the hash is a small fraction of the signature generation time for both file sizes (1 and 10 KB) and for both RSA key sizes (1024 and 2048 bits).

Table 7.2 *Computation times for signature generation and verification*

Message Size	Key Size	Signature Generation Time (μsec)	Signature Verification Time (μsec)	Hash Computation Time (μsec)
1 KB	1024	10,550	586	
	2048	70,316	1989	188
10 KB	1024	10,681	835	
	2048	75,386	2172	328

7.5 THE BIRTHDAY ATTACK

The following idea, first proposed by Yuval [YUVA79], illustrates the danger in choosing hash lengths less than 128 bits.

A malicious individual, Malloc, wishes to forge the signature of his victim, Alka, on a fake document, $\mathcal{F}$. $\mathcal{F}$ could, for example, assert that Alka owes Malloc several million rupees. Malloc does the following:

- He creates millions of documents, $\mathcal{F}_1$, $\mathcal{F}_2$, . . . $\mathcal{F}_m$, etc. that are, for all practical purposes, "clones" of $\mathcal{F}$. This is accomplished by, for example, leaving an extra space between two words, etc. If there are 300 words in $\mathcal{F}$, there are 2^{300} ways in which extra spaces may be left between words.
- He computes the hashes, $h(\mathcal{F}_1)$, $h(\mathcal{F}_2)$, . . . $h(\mathcal{F}_m)$ of each of these documents.
- He creates an innocuous document, $\mathcal{D}$ – one that most people would not hesitate to sign. (For example, it could espouse an environmental cause relating to conservation of forests.)
- He creates millions of "clones" of $\mathcal{D}$ in the same way he cloned $\mathcal{F}$ above. Let $\mathcal{D}_1$, $\mathcal{D}_2$, . . . $\mathcal{D}_m$ be the cloned documents of $\mathcal{D}$.
- He computes the hashes, $h(\mathcal{D}_1)$, $h(\mathcal{D}_2)$, . . . $h(\mathcal{D}_m)$ of each of the cloned documents. Using a corollary of the Birthday Paradox, it can be shown that if m is sufficiently large (essentially the square root of $n = 2^w$, where w is the width of the hash value), then with probability approaching ½ there will be *at least one pair of clones*, $\mathcal{F}_i$ and $\mathcal{D}_j$, that *hash to the same value.*
- Malloc asks Alka to sign the document $\mathcal{D}_j$ and Alka obliges.
- Later Malloc accuses Alka of signing the fraudulent document $\mathcal{F}_i$.

Note that the digital signature is obtained by encrypting the hash value of the document using the private key of the signer. Thus, Alka's signature on $\mathcal{D}_j$ is the same as that on $\mathcal{F}_i$. Hence, at a later point in time, Malloc can use Alka's signature on $\mathcal{D}_j$ to claim that she signed the fraudulent document, $\mathcal{F}_i$.

The Birthday Attack can be launched in time $O(n^{1/2})$ where $n = 2^w$. Hence, a 64-bit hash ($w = 64$) is extremely vulnerable. Partly in response to this concern, hashes of length 128 and above are now widely used. Unfortunately, even these have recently been shown to be vulnerable. For example, Wang et al. [WANG05] found hash collisions in the 160-bit SHA-1 in approximately 2^{63} operations rather than 2^{80}. For applications that demand stringent security, one of SHA-224, SHA-256, SHA-384, SHA-512 are recommended. The integer suffixes of these hash algorithms represent the number of bits in the hash value.

APPENDIX

Let w be the number of bits used to represent the output of a cryptographic hash function. Given m random messages (bit strings), what is the probability that at least one pair of messages maps to the same hash value?

If w is the width of the cryptographic hash, there are a total of 2^w possible hash values. Let $n = 2^w$. The probability of at least one hash collision, P_c in these m messages is

$$P_c = 1 - \text{Prob(No collisions)}$$

We use a combinatoric argument to estimate Prob(No collisions). This quantity is simply the ratio of (a) the number of one-to-one functions (these are mappings that avoid collisions) between the set of the m messages and the set of n possible hash values to (b) the number of all possible functions (including many-to-one mappings). The number of one-to-one functions is

$$n\ (n-1)\ (n-2)\ .\ .\ .\ (n-m+1)$$

while the number of all possible functions is n^m.
We thus have

$$P_c = 1 - \frac{n(n-1)(n-2)\cdots(n-m+1)}{n^m}$$
$$= 1 - \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{m-1}{n}\right)$$

$i/n << 1$ for $0 \le i \le m-1$. So, we can approximate $1 - i/n$ by $e^{-i/n}$. This follows from the Taylor's series expansion of e^x ignoring second and higher order terms in x. Thus

$$P_c \sim 1 - e^{-(1/n + 2/n + \ldots (m-1)/n)}$$
$$= 1 - e^{-m\,(m-1)/2n}$$
$$\sim 1 - e^{-m^2/2n}$$

We can express m as a function of the collision probability and n.

$$m \sim \sqrt{-2n \ln (1 - P_c)}$$

We see that for a given P_c, the number of messages to be inspected before a collision is detected is roughly $O(\sqrt{n})$. A brute-force attack on, both, the *one-way property* and on *weak collision resistance* involves generation and hash computation of $O(n)$ messages on the average. On the other hand, we have just demonstrated that a brute-force attack on the *strong collision resistance* property of the cryptographic hash requires generation and hash computation of only $O(\sqrt{n})$ messages on the average.

SELECTED REFERENCES

Good sources for the mathematical treatment of the cryptographic hash are [MENE01] and [STIN05]. [RIVE80] and [FIPS 180-1] introduce the MD-5 and SHA-1 hashes, respectively. The

HMAC was first proposed in [BELL96]. [GILB03] contains a security analysis of SHA-256. [WANG05] was one of the first works to demonstrate collisions with the MD-5 hash.

OBJECTIVE-TYPE QUESTIONS

7.1 Which of the following is/are synonymous with "hash of a message"?
(a) Digital signature over a message (b) Message digest
(c) Message authentication code (d) Message fingerprint

7.2 Which of the following are true statements about the SHA-1 hash?
(a) The block size is 512 bytes
(b) The hash value (output) is 160 bits
(c) The hash value is a function of message length
(d) The number of iterations involved in its implementation = # of blocks in the message

7.3 The MAC provides
(a) message confidentiality (b) non-repudiation
(c) message authentication (d) message integrity

7.4 The digital signature provides
(a) message confidentiality (b) non-repudiation
(c) message authentication (d) message integrity

7.5 To verify a digital signature, the following is/are required
(a) the signer's public key (b) the signer's private key
(c) the hash algorithm used (d) the verifier's public key

7.6 The ratio, $\dfrac{\textit{Time to encrypt a 10KB message with 56-bit DES}}{\textit{Time to compute hash of a 10KB message with SHA-1}}$ is
(a) > 1 (b) $>> 1$
(b) < 1 (d) $<< 1$

7.7 Yuval's Birthday Attack can be prevented by
(a) using a non-cryptographic hash function
(b) increasing the width of the hash value
(c) padding the vulnerable message
(d) using a keyed hash

EXERCISES

7.1 Would the hash function defined below be appropriate for security applications?

$$h(x) = (x \bmod n) \bmod p$$

where n is a 1024-bit prime and p is a 160-bit prime.

7.2 Consider the Cyclic Redundancy Check (CRC) used to protect data in communications in Ethernet LANs, etc.
Here's how the CRC on a message, m, is computed.
A "generator polynomial" of degree r, $g(x)$ with binary coefficients is given.
Let m be expressed as a binary string. Equivalently, it can be expressed as a polynomial, $m(x)$, with binary coefficients. So for example the message 10011010 is expressable as $x^7 + x^4 + x^3 + x$.

The quantity $(m(x) * x^r)$ is divided by $g(x)$. The division is performed as in high school division except that all additions/subtractions of polynomial coefficients are performed modulo 2. The r-bit resulting remainder is the "CRC hash".

Is it appropriate to use the CRC as a cryptographic hash? Explain why or why not.

7.3 In this chapter, we studied two uses of the cryptographic hash – in computing the MAC and the digital signature. Think and elaborate upon any two other applications of the cryptographic hash.

7.4 Assume a scenario in which you are not allowed to use any form of secret or public key cryptography. Can you still perform encryption/decryption using only a cryptographic hash? If so, explain how. Assume that both sides share a common secret.

7.5 If w is the number of bits in a hash value, what is

(a) the time to successfully attack the strong collision-resistance property of the cryptographic hash

(b) the time to successfully attack the weak collision-resistance property of the cryptographic hash

7.6 State one advantage of a CBC-MAC over an H-MAC.
State one advantage of an H-MAC over a CBC-MAC.

7.7 Consider the digital signature created using the signer's private key operation but without the hash function, i.e.,

$$\text{Sign}(m) = E_{A.pr}(m)$$

Demonstrate how a forged signature may be created using this definition of a digital signature.

7.8 Derive an expression for the number of messages that need to be created (variations of the malicious message and the innocuous message) so that Yuval's Birthday attack is successful with a probability 0.75.

7.9 Consider the following construction of a cryptographic hash.
As in the case of SHA-1

- The block size is 512 bits and the hash is computed iteratively, block by block.
- Padding is used so that the width of the (message + pad + length field) is a multiple of 512 bits.
- An array of 80 32-bit words ($W_1, W_2, \ldots W_{80}$) is initialized prior to processing a block. The first 16 words are obtained from the block and the next 64 words are computed from

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16} \qquad 16 < i \le 80$$

- A 160-bit shift register stores the intermediate hash values. It comprises five 32-bit registers, $S1$, $S2$, $S3$, $S4$, $S5$. The shift register is initialized as in SHA-1.

For each block, the following loop is executed:

```
for $i$ = 1 to 80 {

        $temp \leftarrow (S5 \vee S1) \oplus (F_i(S2, S3, S4) \wedge W_i) \oplus K_i$
        $S5 \leftarrow S4$
        $S4 \leftarrow S3$
        $S3 \leftarrow S2$
        $S2 \leftarrow S1$
        $S1 \leftarrow temp$
}
```

The function F_i is as defined in SHA-1.

$F_i\ (S2, S3, S4) = (S2 \wedge S3) \vee (\neg S2 \wedge S4)$, $\quad 1 \le i \le 20$

$F_i\ (S2, S3, S4) = S2 \oplus S3 \oplus S4$, $\quad 21 \le i \le 40$

$F_i\ (S2, S3, S4) = (S2 \wedge S3) \vee (S2 \wedge S4) \vee (S3 \wedge S4)$, $\quad 41 \le i \le 60$

$F_i\ (S2, S3, S4) = S2 \oplus S3 \oplus S4$, $\quad 61 \le i \le 80$

Study the following design carefully and then identify whether it is vulnerable. If so, explain clearly how or why it may be vulnerable.

7.10 Use the Bouncy Castle C++ or Java APIs to compute the SHA-1 hash on a message. Also use the APIs to generate and verify an RSA digital signature on a message.

ANSWERS TO OBJECTIVE-TYPE QUESTIONS

7.1 (b)(d)	7.2 (b)(c)	7.3 (c)(d)	7.4 (b)(c)(d)
7.5 (a)(c)	7.6 (a)	7.7 (b)	

Chapter 8

Discrete Logarithm and its Applications

The security of RSA (Chapter 6) rests, in part, on the infeasibility of factoring integers that are the product of two very large primes. In this chapter, we introduce another computational challenge – the discrete logarithm problem. We then introduce schemes for secure key exchange, encryption, and digital signatures that all depend on the hardness of the discrete logarithm problem.

8.1 INTRODUCTION

Consider the finite, multiplicative group $(Z_p^*, *_p)$, where p is prime. Let g be a generator of the group, i.e., successive powers of g generate all elements of the group. So

$$g^1 \bmod p,\ g^2 \bmod p, \dots\ g^{p-1} \bmod p$$

is a *re-arrangement* of the integers in Z_p^* (see Fig. 8.1 for an example with $p = 29$ and $g = 2$). Let x be an element in $\{0, 1, \dots\ p-2\}$. The function

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

is referred to as *modular exponentiation* with base g and modulus p.
The inverse operation is expressed as

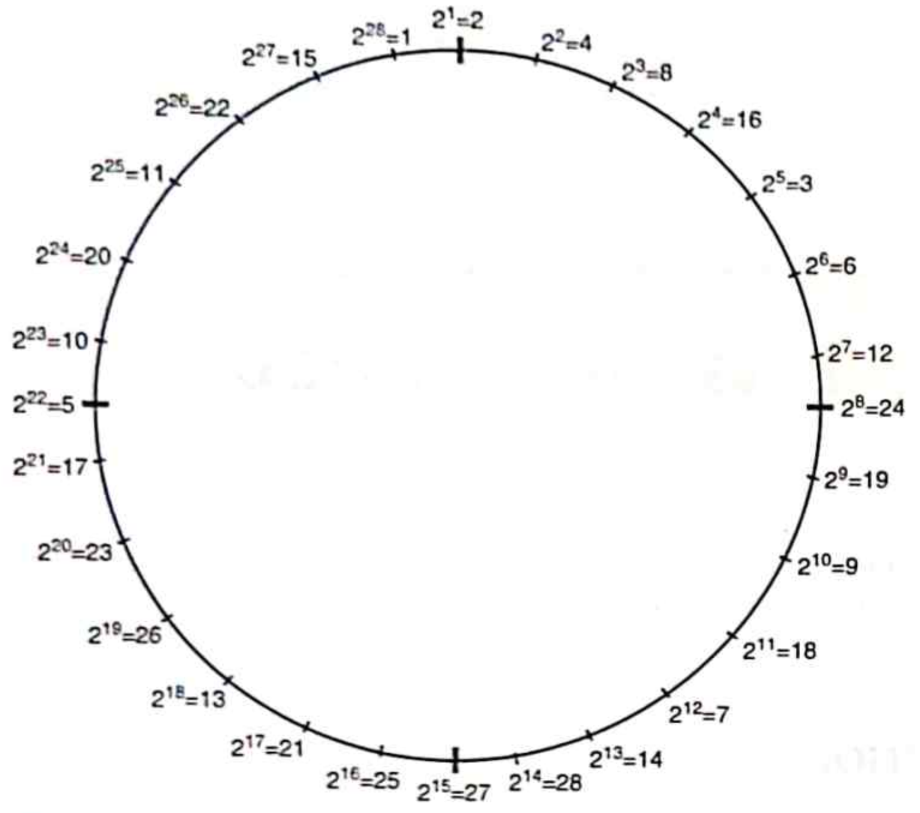
$$x = \log_g y \pmod p$$

and is referred to as the *discrete logarithm*. It involves computing x given the values of p, g, and $y \in Z_p^*$.

For cryptographic applications, the values of p and g are very large (100's of digits long). However, as seen before in Chapter 6, the computation time for performing modular exponentiation can be greatly reduced. Using the *Square and Multiply* strategy, the time to compute $g^x \bmod p$ is reduced from a polynomial in p to a polynomial in $\log p$.

A brute-force algorithm to compute the discrete logarithm is to prepare a table, which pairs each x, $0 \le x \le p-2$, with the corresponding $g^x \bmod p$. We then sort the table on the $g^x \bmod p$ column. Given a value, we use it as an index into the $g^x \bmod p$ column. The desired discrete logarithm is simply the value in the other column of the same row (see Fig. 8.1).

The computation cost of the table approach is proportional to p. For large p (say 2000-bit p), constructing and storing such a large table is infeasible. There have been several attempts to speed up the discrete logarithm problem such as the Pollard-rho algorithm and index calculus methods. However, the time complexity of these methods is $O\sqrt{p}$ which is still prohibitive.



y	$\log_2 y$ (mod 29)
1	28
2	1
3	5
4	2
5	22
6	6
7	12
8	3
9	10
10	23
11	25
12	7
13	18
14	13
15	27
16	4
17	21
18	11
19	9
20	24
21	17
22	26
23	20
24	8
25	16
26	19
27	15
28	14

Figure 8.1 *Discrete logarithm in* $(Z^*_{29}, *_{29})$ *with* $g = 2$

Example 8.1

Let $p = 131$ and $g = 2$. [See Chapter 3, Section 3.3.1 for how to check if an integer is a generator of the group $(Z^*_p, *_p)$].
Then, the discrete logarithm,

$$\log_2 72 \bmod 131 = 17$$

since

$$2^{17} \bmod 131 = 72.$$

It is this *one-way property* of modular exponentiation – easy to compute but hard to invert – that makes it of such great use in cryptography. There are numerous applications of the discrete logarithm ranging from online secret key agreement to encryption and digital signatures. We discuss key agreement in Section 8.2 and encryption/signing in Section 8.3.

8.2 DIFFIE–HELLMAN KEY EXCHANGE

8.2.1 Protocol

Consider two parties, A and B that need to agree upon a shared secret for the duration of their current session. One widely used method is the Diffie–Hellman Key Exchange. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key.

For simplicity, assume that both A and B know the base g and modulus p in advance. They then participate in the following sequence of steps shown in Fig. 8.2:

- A chooses a random integer a, $1 < a < p-1$, computes the "partial key," $g^a \bmod p$ and sends this to B.
- B chooses a random integer b, $1 < b \le p-1$, computes the "partial key," $g^b \bmod p$ and sends this to A.
- On receipt of A's message, B computes $(g^a \bmod p)^b \bmod p = g^{ab} \bmod p$
- On receipt of B's message, A computes $(g^b \bmod p)^a \bmod p = g^{ab} \bmod p$

Now, both A and B share a common secret, $g^{ab} \bmod p$. These steps are clarified in the toy example below.

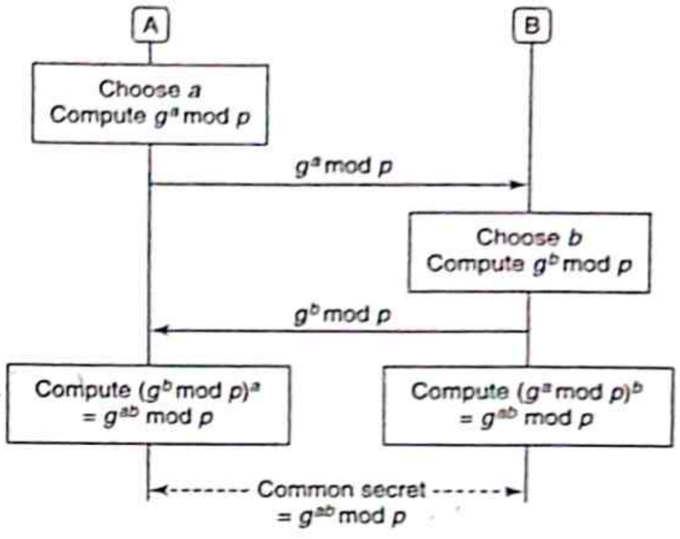


Figure 8.2 *Diffie–Hellman key exchange*

Example 8.2

Let $p = 131$ and $g = 2$.

Let the random number chosen by A be 24. So, her partial key is $2^{24} \bmod 131 \equiv 46$.
Let the random number chosen by B be 17. So, his partial key is $2^{17} \bmod 131 \equiv 72$.
After receiving B's partial key, A computes the shared secret as $72^{24} \bmod 131 \equiv 13$.
After receiving A's partial key, B computes the shared secret as $46^{17} \bmod 131 \equiv 13$.

Note that in this protocol, the partial keys, $g^a \bmod p$ and $g^b \bmod p$ are sent in the clear. Can an eavesdropper with knowledge of the partial keys and the public parameters (p and g) deduce the common secret, $g^{ab} \bmod p$, derived by A and B? This problem is referred to as the *computational Diffie-Hellman Problem.*

One way to solve the Computational Diffie–Hellman Problem is to obtain a and b from g^a *mod* p and g^b *mod* p. But this entails computing the discrete logarithm – a problem that is computationally infeasible in the case where p is sufficiently large. Is there another way of solving the Computational Diffie–Hellman Problem without having to solve the discrete logarithm? To date, there is no known alternative. This leads us to conclude that the security of the Diffie–Hellman protocol is based on the infeasibility of the discrete logarithm problem.

The integers a and g^a *mod* p are, in a sense, A's private key and public key, respectively. As mentioned earlier, the idea of a public key–private key pair originated with this protocol. Later, El Gamal devised schemes for encryption and digital signatures, which take advantage of the infeasibility of computing the discrete logarithm.

8.2.2 Attacks

The Diffie–Hellman protocol, as described in the previous section, can be easily attacked. An attacker, C chooses an integer c and computes $g^c \bmod p$. As shown in Fig. 8.3, C then intercepts

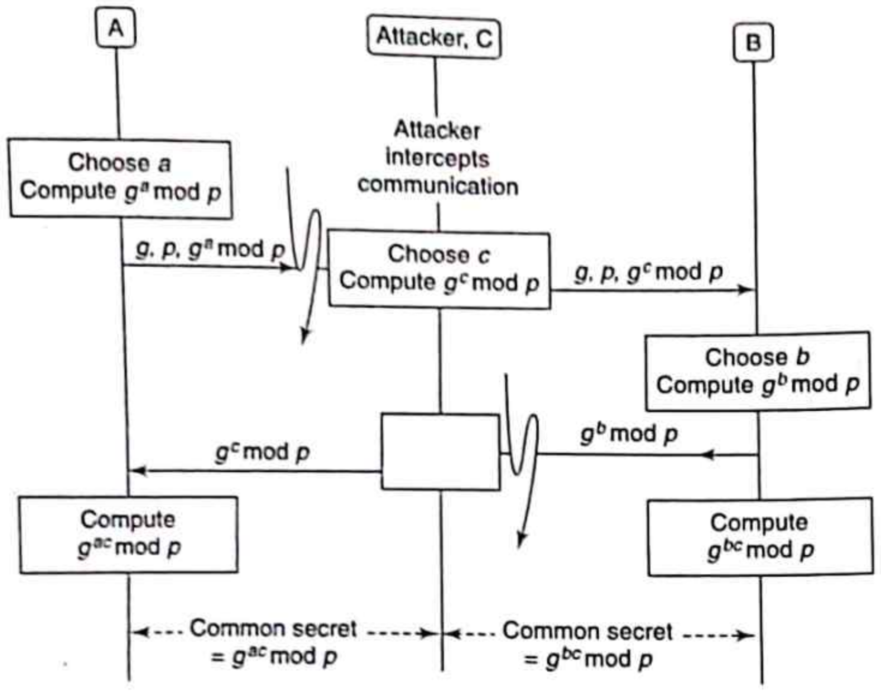


Figure 8.3 *Man-in-the middle attack on Diffie–Hellman key exchange*

A's message to B, substitutes it with g^c mod p, and sends this instead to B. In the same manner, C also intercepts B's message to A sending g^c mod p instead. After the message transfer, B computes

$$(g^c \bmod p)^b \bmod p = g^{bc} \bmod p$$

while A computes

$$(g^c \bmod p)^a \mathit{mod}\ p = g^{ac} \bmod p$$

C also computes the two secrets

$$(g^a \bmod p)^c \bmod p = g^{ac} \bmod p \text{ and}$$
$$(g^b \bmod p)^c \bmod p = g^{bc} \bmod p$$

A and B might think that they have a secure channel for communications by encrypting all messages to each other with their "common secret." Instead, A shares the secret g^{ac} mod p with C and B shares the secret g^{bc} mod p with C.

In reality, every subsequent message encrypted by A and intended for B can be decrypted by C, possibly changed and re-encrypted with the secret shared by B and C. Likewise, every message from B to A can be decrypted and possibly modified by C. This is a classical example of an *active* "*Man in the Middle Attack.*"

This attack is possible because B believed that the partial key (g^c mod p) that it received was from A. Likewise, A believed that the partial key (g^c mod p) that it received was from B. They neglected to *authenticate* the source of the partial keys they had received. This could have been done, for example, by each party sending its partial key together with an RSA signature on it.

8.2.3 Choice of Diffie–Hellman Parameters

We have chosen the modulus p to be prime and the base g as a generator of Z_p^*. Why is this so?

The secrets involved in Diffie–Hellman key exchange include the "private keys," a and b (known to A and B, respectively), and the shared secret, g^{ab} mod p. To minimize the chance of an attacker *guessing* any of these secrets, it is desirable that they be chosen (or computed) from as large a pool of integers as possible.

The number of distinct private keys and the number of distinct public keys are both upper-bounded by the cardinality of Z_p^*. This is $p-1$ for p prime and is less than $p-1$ for non-prime p. Of course, to maximize security, it would be desirable to have as large a p as possible. But, like so much else in cryptography, the choice of size of p is a tradeoff between security and performance. Finally, by choosing g to be a generator of Z_p^*, the space of possible public keys is large (equal to $p-1$).

We next explore subtle attacks that may be possible owing to a poor choice of p. We illustrate the attack with a toy example.

Example 8.3

Let $p = 29$. Figure 8.1 shows the powers of $g = 2$ in the set Z_{29}^*. Now, suppose one of the communicating parties, say A, in the Diffie–Hellman key exchange protocol chooses its private key, $a = 7$. So, the corresponding public key is 2^7 mod 29 = 12.

We next note that the subgroup of Z_{29}^* generated by 12 is

$$\{g^7 \bmod 29,\ g^{14} \bmod 29,\ g^{21} \bmod 29,\ g^{28} \bmod 29, \ldots\}$$

which is the set of four elements,

$$\{12, 28, 17, 1\}$$

We now claim that, *regardless of the value of B's private key*, the *common secret* that they compute will be restricted to an element in the above set. Thus, the common secret is *one of only four elements*, not one of the 28 elements in Z_{29}^*!

To see why, express b as

$$b = 4 * i + j \qquad \text{where} \quad j = b \bmod 4$$

The common secret computed by A and B is

$$\begin{aligned} g^{ab} \bmod 28 &= g^{7(4i+j)} \bmod 29 \\ &= g^{28i+7j} \bmod 29 \\ &= ((g^{28})^i \bmod 29) * (g^{7j} \bmod 29) \end{aligned}$$

Now, g^{28} mod 29 = 1 (by Fermat's Theorem) and g^{7j} mod 29 is an element in {12, 28, 17, 1}. Hence, the common secret is restricted to an element in {12, 28, 17, 1} for ANY choice of B's private key.

In the above example, the common secret computed by A and B is a member of a "small" subgroup and so may be easily guessed. Such an attack is referred to as a "small subgroup attack." An attacker can eavesdrop on the partial secrets exchanged, g^a mod p and g^b mod p and then attempt to compute the elements in the sets

$$\{g^a \bmod p,\ g^{2a} \bmod p,\ g^{3a} \bmod p,\ g^{4a} \bmod p, \ldots\} \text{ and}$$
$$\{g^b \bmod p,\ g^{2b} \bmod p,\ g^{3b} \bmod p,\ g^{4b} \bmod p, \ldots\}$$

If either of these sets is small, the attacker can guess the common secret. He/she can launch a man-in-the-middle attack, which is simpler than the one described in the previous section. In that attack, the attacker decrypted and then re-encrypted every message. This attack, on the other hand, is *passive* (the attacker does not need to change the message). To read the message, he/she needs to decrypt it. Re-encryption is necessary only when the attacker wishes to modify the content of a message.

These attacks are possible if Z_p^* has small subgroups. By Lagrange's theorem, this could occur if $p-1$ has small prime factors.

For any large prime, p, 2 will always be a factor of $p-1$. However, it is possible to find primes such that the rest of the factors of $p-1$ are large, i.e.,

$$p-1 = 2 * f_1 * f_2 \ldots * f_k$$

where $f_1, f_2, \ldots f_k$ are each "large" primes. Such values of p are referred to as *Lim–Lee primes*. Then, by using a Lim–Lee prime for p, we can be sure that the only subgroups of Z_p^* are of size 2 or f_1 or f_2 ... or f_k or larger.

A special case of these primes is when $k = 1$, i.e., $p-1$ has only two prime factors, 2 and a "very large" prime. In this case, the prime, p, is referred to as a *safe prime*.

To prevent small subgroup attacks, Diffie-Hellman key exchange software at both ends should perform tests on the "private keys", a and b, chosen by A and B. (See Exercise 8.4 for the necessary test). If the test fails, a new private key should be chosen.

8.3 OTHER APPLICATIONS

8.3.1 El Gamal Encryption

El Gamal encryption uses a large prime number p and a generator g in $(Z_p^*, *_p)$. An El Gamal private key is an integer, a, $1 < a < p-1$. The corresponding public key is the triplet (p, g, α) where α is the encryption key calculated from

$$\alpha = g^a \bmod p$$

Let (p, g, α) be the public key of A. To encrypt a message $m < p-1$, to be sent to A, B does the following:

- He chooses a random number r, $1 < r < p-1$ such that r is relatively prime to $p-1$.
- He computes

$$C_1 = g^r \bmod p$$

and uses α from A's public key to compute

$$C_2 = (m * \alpha^r) \bmod p$$

- He sends the ciphertext (C_1, C_2) to A.

To decrypt the ciphertext (C_1, C_2), A uses her private key, a and computes

$$(C_1^{-a}) * C_2 \bmod p.$$

To see why decryption does indeed recover the original message, observe that

$$\begin{aligned}(C_1^{-a}) * C_2 \bmod p &= (g^r \bmod p)^{-a} * (m * \alpha^r \bmod p) && \text{using the definitions of } C_1 \text{ and } C_2 \text{ above}\\ &= (g^{-ar} * g^{ar} * m) \bmod p && \text{using the definition of the El Gamal public key}\\ &= m\end{aligned}$$

Unlike with RSA and DES encryption, the ciphertext is now twice the size of the original plaintext.

Example 8.4

Let $p = 131$ and $g = 2$.
Let A's private key, $a = 97$.
So, her public key is $\alpha = 2^{97} \bmod 131 \equiv 14$.
Let the message to be sent be $m = 75$.
Let the sender, B, choose the random number, $r = 33$.
B then computes

$$\begin{aligned} C_1 &= g^r \bmod p \\ &= 2^{33} \bmod 131 \\ &\equiv 103 \end{aligned}$$

and

$$\begin{aligned} C_2 &= (m * \alpha^r) \bmod p \\ &= 75 * 14^{33} \bmod 131 \\ &= 51 \end{aligned}$$

To decrypt the message, A computes

$$\begin{aligned} (C_1^{-a}) * C_2 \bmod p &= 103^{-97} * 51 \bmod 131 \\ &= 75 \end{aligned}$$

Thus, A recovers the original message using her private key.

Note that the strength of this encryption is closely related to the difficulty of solving the discrete logarithm problem for large (several hundred digits in length) values of p. For example, knowing C_1, g, and p, it is computationally infeasible to find r. If not, an attacker would be able to deduce m by computing $\alpha^r \bmod p$ and then computing $(C_2 * (\alpha^r)^{-1} \bmod p)$.

There are also some precautions to be followed in the use of El Gamal encryption. The same random number should not be used again. To see why, suppose that two messages m and m' are encrypted using the same random number, r. The ciphertext corresponding to the first message is

$$(C_1, C_2) = (g^r \bmod p,\ m*\alpha^r \bmod p).$$

The ciphertext corresponding to the second message is

$$(C_1', C_2') = (g^r \bmod p,\ m'*\alpha^r \bmod p).$$

Consider now an eavesdropper who has seen both pairs of ciphertext. If, in addition, the eavesdropper also happens to know the first plaintext, m, then he/she can obtain the value of the second plaintext m' as follows.

$$\begin{aligned} m * C_2' * C_2^{-1} \bmod p &= m * (m' * \alpha^r \bmod p) * (m*\alpha^r \bmod p)^{-1} \bmod p \\ &= m' * m * m^{-1} * \alpha^r * \alpha^{-r} \bmod p \\ &= m' \end{aligned}$$

The above known plaintext attack is not an illustration of the weakness of the El Gamal scheme per se but rather is a consequence of its improper use (in this case using the same random number more than once).

8.3.2 El Gamal Signatures

As before, let a and (p, g, α) be the private and public keys of A. To sign a message m, A does the following:

1. She computes the hash $h(m)$ of the message.
2. She chooses a random number r, $1 < r < p-1$ such that r is relatively prime to $p-1$.

3. She computes

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

4. She computes

$$y = (h(m) - ax)r^{-1} \bmod (p-1)$$

5. The signature is the pair (x, y).

Note that the private key, a, must be used for signature generation as it is in Step 4 above. Signature verification, on the other hand, uses α, a component of the public key triplet. To verify the signature, the following check is performed:

$$\alpha^x * x^y \bmod p \overset{?}{=} g^{h(m)} \bmod p$$

To prove that a valid signature satisfies the above equation, we start with the expression in Step 4.

$$y = (h(m) - ax)r^{-1} \bmod (p-1)$$

So $\quad ry = (h(m) - ax) + k(p-1) \quad$ where k is an integer

Raising both sides to the power of g and reducing modulo p, we get

$$g^{ry} = g^{h(m)}\, g^{-ax} \bmod p \qquad \text{(note that } g^{k(p-1)} = 1 \bmod p \text{ from Fermat's Theorem)}$$

or $\quad g^{ax}\, g^{ry} = g^{h(m)} \bmod p$

So $\quad \alpha^x * x^y = g^{h(m)} \bmod p \qquad$ (since $\alpha = g^a \bmod p$ and $x = g^r \bmod p$)

How easy is it to forge an El Gamal signature? Put differently, can one produce a valid pair (x, y) for a given document (or message) m? By valid, we mean an x and a y that satisfy the equations in Steps 3 and 4 above. One approach to forge a signature would be to proceed to complete Steps 1 through 3 in the signature generation procedure above. Step 4, however, requires knowledge of the private key, a. Without the latter, it is not possible to complete the computation of the digital signature.

It is interesting that a person's El Gamal signature on a given document is not unique. Insofar as he/she uses a different random number to sign the same document, the signature he/she produces each time will be different. Note that, by contrast, a manual signature is unique to a person and does not vary from document to document. The RSA signature depends upon both – the signer and the document but remains the same for a given signer and document. The El Gamal signature is further removed from the idea of the traditional manual signature in that there are *multiple valid signatures* for a *given signer* and a *given document.*

8.3.3 Related Signature Schemes

There are a number of attacks on the discrete logarithm using index calculus methods, the Pollard-rho algorithm, etc. [STIN02]. To render these attacks infeasible, p should be about a 1000-bit integer. The El Gamal signature is comprised of two integers of this size. So, an El Gamal signature occupies 2000 bits. A scheme by Schnorr greatly helps reduce the size of the signature to less than 400 bits with no loss of security. We discuss this scheme next.

Choose a large prime, p (about 1000 bits) so that the following holds:

q is a prime about 160 bits wide that divides $p-1$.

Let g be a q^{th} root of 1 mod p. So, $g^q = 1 \bmod p$.
Let a be an integer, $1 \le a \le q-1$. (As before a is the private key).

Let $\alpha = g^a \bmod p$. [(The public key is $(p, q, g, g^a \bmod p)$]
Let r be a random number, $1 \le r \le q-1$.

Then the *Schnorr signature* is the pair, (x, y) where

$$x = h(m \,\|\, g^r \bmod p) \quad \text{and}$$
$$y = (r + ax) \bmod q$$

Signature verification is performed by computing the value of x using the signer's public key and checking whether it equals the value of x received as shown below.

$$x \overset{?}{=} h(m \,\|\, g^y \alpha^{-x} \bmod p)$$

We note that

$$\begin{aligned} g^y \alpha^{-x} \bmod p &= g^y (g^a)^{-x} && \bmod p \\ &= g^{y-ax} && \bmod p \\ &= g^{r+kq} && \bmod p, \text{ where } k \text{ is an integer.} \\ & && \text{This follows from the definition of } y. \\ &= g^r && \bmod p \\ & && \text{since } g \text{ is the } q\text{-th root of } 1 \end{aligned}$$

The signer computed the values of x and y as a function of the message, a nonce and her private key. The verifier computes x as a function of the corresponding public key. If the computed value of x matches the received value, the verifier can be sure that the signer has correctly used the genuine private key (corresponding to her public key). Thus, her signature must be authentic.

Closely related to the Schnorr signature is the *Digital Signature Algorithm* (DSA), which is used in the *Digital Signature Standard* (*DSS*). This algorithm is further explored in the exercises at the end of this chapter.

We have, thus far, considered the discrete logarithm problem in Z_p^* or in prime subgroups of Z_p^*. Are these the only groups in which the discrete logarithm problem is infeasible? It turns out that there are other groups that may be considered. These include the multiplicative group of or a binary field $GF(2^n)$. This field is discussed in Chapter 3. In addition, the group of points on certain elliptic curves are excellent examples of where the discrete logarithm is infeasible. We defer a discussion of this group to Chapter 9 where *elliptic curve cryptography* is introduced.

SELECTED REFERENCES

The first application of the discrete logarithm problem to key agreement appeared in [DIFF76]. El Gamal's application to encryption and signatures is in [ELGA85]. The Schnorr signature first appeared in [SCHN91]. The Digital Signature Standard (DSS) is described in [FIPS186]. An excellent study of the strength of the discrete log problem and attempted attacks on it is [ODLY99].

OBJECTIVE-TYPE QUESTIONS

8.1 The discrete logarithm of 28 to the base 2 modulo 53 is
(a) 29 (b) 16 (c) 47 (d) 42

8.2 A and B perform Diffie–Hellman key exchange using $p = 53$ and $g = 2$. If A chooses her secret $= 10$ and B chooses his secret $= 33$, then the common secret they agree upon is
(a) 44 (b) 12 (c) 27 (d) 6

8.3 The computational Diffie–Hellman problem is
(a) Given p, g, $g^a \bmod p$, compute a
(b) Given p, g, a, compute $g^a \bmod p$
(c) Given p, g, $g^b \bmod p$, and $g^{ab} \bmod p$, compute $g^a \bmod p$
(d) Given p, g, $g^a \bmod p$, and $g^b \bmod p$, compute $g^{ab} \bmod p$

8.4 The original Diffie–Hellman key exchange protocol is vulnerable to a man-in-the-middle attack because
(a) key exchange takes place in the clear (unencrypted)
(b) the prime numbers chosen are not safe
(c) the computational Diffie–Hellman problem is not infeasible
(d) the two communicating parties do not authenticate themselves to each other

8.5 The small sub-group attack can be prevented by
(a) authenticating each message in the Diffie–Hellman protocol
(b) use of a safe prime
(c) use of an appropriate generator
(d) encryption of each key exchange message

8.6 The number of distinct El Gamal ciphertexts that could be generated by the same person for a given block of plaintext is
(a) 1 (b) 2 (c) infinite (d) none of these

8.7 Compared to the El Gamal signature, which of the following is/are true of the Schnorr signature?
(a) Signature generation is faster (b) It is less secure
(c) It is more space-efficient (d) It does not require generation of a random number.

EXERCISES

8.1 Write a program to compute the following discrete logarithms.
– $\log_3 326 \bmod 881$
– $\log_7 200 \bmod 911$

8.2 Generalize the Diffie–Hellman key exchange protocol to more than two parties.

8.3 Two parties use the Diffie–Hellman key exchange protocol with $p = 23$ and $g = 5$. If the common secret that both sides compute $= 21$, then what are the possible values of the initial secrets chosen by each of them?

8.4 In Diffie–Hellman key exchange, each party, A and B, chooses a secret (a and b, respectively). Show that the "small subgroup attack" will be thwarted if a and b satisfy:

$$gcd\ (a, p-1) = 1 \quad \text{and}$$
$$gcd\ (b, p-1) = 1$$

8.5 A block of plaintext has been encrypted using El Gamal encryption. Assume that $p = 977$, $g = 3$ and the recipient's public key $= 477$. What is the plaintext corresponding to the ciphertext, $C_1 = 108$ and $C_2 = 562$?

8.6 In this chapter, we examined why using the *same* random number to perform El Gamal encryptions on multiple messages could be exploited by an attacker to recover plaintext. In the case of El Gamal signatures, is there a danger in using the same random number to sign multiple messages? Explain.

8.7 Compute the El Gamal signature on a message, m, where $h(m) = 3008$. Assume that $p = 4793$, $g = 5$, and that the signer's private key $= 2157$.
Also, compute both sides of the signature verification equation.

8.8 The digital signature algorithm (DSA) is defined below:
- The values of p, q, and g are chosen as in the Schnorr signature presented in this chapter.
- a, where $1 \le a \le q - 1$ and $\alpha = g^a$ *mod* p are, respectively, the private and public keys of the signer, A.

To sign a message m, A performs the following steps:
- She chooses a random #, r relatively prime to p–1.
- She computes $x = (g^r \bmod p) \bmod q$.
- She computes $y = ((\mathrm{h}(m) + ax)\, r^{-1}) \bmod q$
- The signature is the pair (x, y).

To verify the signature, the verifier performs the following test

$$x = (g^{???}\ \alpha^{???} \bmod p) \bmod q$$

Substitute variables for the ??? above and explain why the equation performs signature verification.

8.9 Given the primes p and q in the Schnorr signature defined in the text, suggest a computationally efficient way of obtaining g. Here, g is a generator of a subgroup of order q.

ANSWERS TO OBJECTIVE-TYPE QUESTIONS

8.1 (b)	8.2 (d)	8.3 (d)	8.4 (d)
8.5 (b)	8.6 (d)	8.7 (c)	