Chapter 8

Cryptography

Contents

8.1 Symmetric Cryptography

Cryptography began primarily as a way for two parties, who are typically called "Alice" and "Bob," to communicate securely even if their messages might be read by an eavesdropper, "Eve." (See Figure 8.1.) It has grown in recent times to encompass much more than this basic scenario. Examples of current applications of cryptography include attesting the identity of the organization operating a web server, digitally signing electronic documents, protecting the confidentiality of online baking and shopping transactions, protecting the confidentiality of the files stored on a hard drive, and protecting the confidentiality of packets sent over a wireless network. Thus, ***cryptography*** deals with many techniques for secure and trustworthy communication and computation.

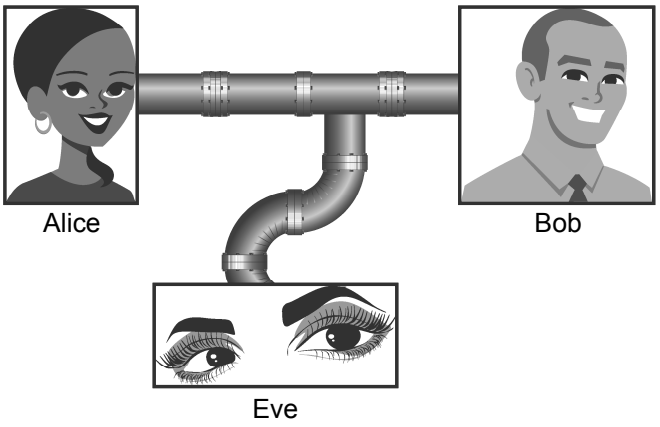


Figure 8.1: The basic scenario for cryptograpy. Alice and Bob encrypt their communications so that the eavesdropper Eve, can't understand the content of their messages.

In ***symmetric cryptography***, which was introduced in Section 1.3.1 and is discussed in more detail in this section, we use the same key for both encryption and decryption. The symmetric encryption algorithm recommended by the U.S. National Institute of Standards and Technology (NIST) is the ***Advanced Encryption Standard***, or ***AES***, which is designed to be a replacement for the legacy ***Data Encryption Standard*** (***DES***) algorithm. Rather than jumping right in to describe the AES cryptosystem, however, let us first describe some classic cryptosystems. Each classic cryptosystem we describe contains an idea that is included in AES; hence, understanding each of these earlier cryptosystems helps us understand AES.

8.1.1 Attacks

Before we describe any cryptosystem in detail, however, let us say a few words about ***cryptosystem attacks***. The science of attacking cryptosystems is known as ***cryptanalysis*** and its practitioners are called ***cryptanalysts***. In performing cryptanalysis, we assume that the cryptanalyst knows the algorithms for encryption and decryption, but that he does not know anything about the keys used. This assumption follows the open design principle (Section 1.1.4). In fact, it is dangerous for us to assume that we gain any degree of security from the fact that the cryptanalyst doesn't know which algorithms we are using. Such ***security by obscurity*** approach is likely to fail, since there are a number of different ways that such information can be leaked. For example, internal company documents could be published or stolen, a programmer who coded an encryption algorithm could be bribed or could voluntarily disclose the algorithm, or the software or hardware that implements an encryption algorithm could be reverse engineered. So we assume the cryptanalyst knows which cryptosystem we are using.

There are four primary types of attacks that a cryptanalyst can attempt to perform on a given cryptosystem.

- ***Ciphertext-only attack***. In this attack, the cryptanalyst has access to the ciphertext of one or more messages, all of which were encrypted using the same key, K. His or her goal is to determine the plaintext for one or more of these ciphertexts or, better yet, to discover K.
- ***Known-plaintext attack***. Inthis attack, the cryptanalyst has access to one or more plaintext-ciphertext pairs, such that each plaintext was encrypted using the same key, K. His or her goal in this case is to determine the key, K.
- ***Chosen-plaintext attack***. In this attack, the cryptanalyst can chose one or more plaintext messages and get the ciphertext that is associated with each one, based on the use of same key, K. In the ***offline chosen-plaintext attack***, the cryptanalyst must choose all the plaintexts in advance, whereas in the ***adaptive chosen-plaintext attack***, the cryptanalyst can choose plaintexts in an iterative fashion, where each plaintext choice can be based on information he gained from previous plaintext encryptions.
- ***Chosen-ciphertext attack***. In this attack, the cryptanalyst can choose one or more ciphertext messages and get the plaintext that is associated with each one, based on the use of same key, K. As with the chosen-plaintext attack, this attack also has both offline and adaptive versions.

We have listed the attacks above in order by the amount of information the cryptanalyst can access when performing them. (See Figure 8.2.)

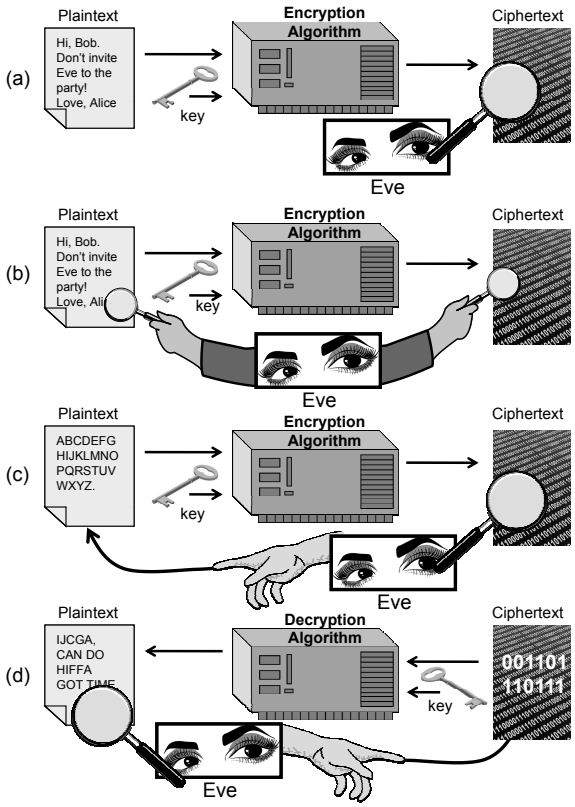


Figure 8.2: Types of attacks: (a) Ciphertext-only attack. (b) Known-plaintext attack. (c) Chosen-plaintext attack. (d) Chosen-ciphertext attack.

One thing that makes these attacks feasible is that it is usually easy to recognize that a message is a valid plaintext. For example, given a certain ciphertext, a cryptanalyst could decrypt it with a given key and get message NGGNPXNGQNJABAVEIVARORNPU, which she can immediately dismiss. But if she gets message ATTACKATDAWNONIRVINEBEACH, then she can be confident she has found the decryption key. This ability is related to the ***unicity distance*** for a cryptosystem, which is the minimum number of characters of ciphertext that are needed so that there is a single intelligible plaintext associated with it. Because of the built-in redundancy that is a part of every natural language (which helps us understand it when it is spoken), the unicity distance, in characters, for most cryptosystems is typically much less than their key lengths, in bits. This concept was previously introduced in Section 1.3.3 in the context of brute-force decryption attacks.

8.1.2 Substitution Ciphers

In the ancient cryptosystem, the ***Caesar cipher***, each Latin letter of a plaintext was substituted by the letter that was three positions away in a cyclic listing of the alphabet, that is, modulo the alphabet size (Section 1.1.1). We can generalize this cipher so that each letter can have an arbitrary substitution, so long as all the substitutions are unique. This approach greatly increases the key space; hence, increasing the security of the cryptosystem. For example, with English plaintexts, there are 26! possible substitution ciphers, that is, there are more than 4.03×10^{26} such ciphers.

An entertaining example of a substitution cipher is shown in the 1983 movie *A Christmas Story*. In this movie, the young character, Ralphie, uses a circular decoder pin representing a substitution cipher to decode a secret message broadcast over the radio. He is a bit disappointed, however, when he discovers that the message is

"BE SURE TO DRINK YOUR OVALTINE,"

which was little more than a commercial.

Simple substitution ciphers like the one Ralphie used, which are based on substituting letters of the alphabet, are easily broken. The main weakness in such ciphers is that they don't hide the underlying frequencies of the different characters of a plaintext. For example, in English text, the letter "E" occurs just over 12% of the time, and the next frequent letter is "T." which occurs less than 10% of the time. So the most frequently occurring character in a ciphertext created from English text with a substitution cipher probably corresponds to the letter "E." In Table 8.1, we give the frequencies of letters that occur in a well-known book, which illustrates the potential weakness of a letter-based substitution cipher to a frequency analysis. A similar table could have easily been constructed for any text or corpus written in any alphabet-based language.

a:	8.05%	b:	1.67%	c:	2.23%	d:	5.10%
e:	12.22%	f:	2.14%	g:	2.30%	h:	6.62%
i:	6.28%	j:	0.19%	k:	0.95%	l:	4.08%
m:	2.33%	n:	6.95%	o:	7.63%	p:	1.66%
q:	0.06%	r:	5.29%	s:	6.02%	t:	9.67%
u:	2.92%	v:	0.82%	w:	2.60%	x:	0.11%
y:	2.04%	z:	0.06%				

Table 8.1: Letter frequencies in the book *The Adventures of Tom Sawyer*, by Mark Twain.

Polygraphic Substitution Ciphers and Substitution Boxes

In a ***polygraphic substitution cipher***, groups of letters are encrypted together. For example, a plaintext could be partitioned into strings of two letters each, that is, divided into ***digrams***, and each digram substituted with a different and unique other digram to create the ciphertext. Since there are $26^2 = 676$ possible English digrams, there are 676! possible keys for such an English digram substitution. The problem with such keys, however, is that they are long—specifying an arbitrary digram substitution key requires that we write down the substitutions for all 676 digrams. Of course, if an alphabet size is smaller than 26, we can write down a digram substitution cipher more compactly. For example, the Hawaiian language uses just 12 letters if we ignore accent marks. Still, even in this case, it would be useful to have a compact way to express digram substitutions.

One way to express a digram substitution that is easy to visualize is to use a two-dimensional table. In such a table, the first letter in a pair would specify a row, the second letter in a pair would specify a column, and each entry would be the unique two-letter substitution to use for this pair. Such a specification is called a ***substitution box*** or ***S-box***.

This visualization approach, of using an S-box to encode a substitution cipher, can be extended to binary words. For example, we could take a b-bit word, x, divide it into two words, y and z, consisting of the first c bits and last d bits, respectively, of x, such that $b = c + d$. Then we could specify the substitution to use for such a word, x, by using an S-box of dimensions $2^c \times 2^d$. We show an example 4×4 S-box for a 4-bit substitution cipher in Figure 8.3. Note that as long as the substitutions specified in an S-box, $\mathcal{S}$, are unique, then there is an inverse S-box, $\mathcal{S}^{-1}$, that can be used to reverse the substitutions specified by $\mathcal{S}$.

In addition to single-letter frequencies, the frequencies of all digram combinations are easy to compute for any alphabet-based written language, given a large enough corpus. Thus, a cryptosystem based only on simple single-character or digram substitution is insecure.

	00	01	10	11
00	0011	0100	1111	0001
01	1010	0110	0101	1011
10	1110	1101	0100	0010
11	0111	0000	1001	1100

(a)

	0	1	2	3
0	3	8	15	1
1	10	6	5	11
2	14	13	4	2
3	7	0	9	12

(b)

Figure 8.3: A 4-bit S-box (a) An S-box in binary. (b) The same S-box in decimal. This particular S-box is used in the Serpent cryptosystem, which was a finalist to become AES, but was not chosen.

8.1.3 One-Time Pads

Substitution can be applied to entire blocks of letters at a time, not just pairs. For example, the ***Vigenère cipher***, first published in 1586, is an example of a polygraphic substitution cipher that applies to blocks of length m, since it amounts to repeatedly using m shift ciphers in parallel. A key in this cryptosystem is a sequence of m shift amounts, $(k_1, k_2, \ldots, k_m)$, modulo the alphabet size (26 for English). Given a block of m characters of plaintext, we encrypt the block by cyclically shifting the first character by k_1, the second by k_2, the third by k_3, and so on. Thus, there are potentially m different substitutions for any given letter in the plaintext (depending on where in the plaintext the letter appears), making this a type of polygraphic substitution cipher. Decryption is done by performing the reverse shifts on each block of m characters in the ciphertext. Unfortunately, as with all substitution ciphers, the Vigenére cipher can be easily broken using statistical techniques, as long as the ciphertext is long enough relative to the value of m.

There is one type of substitution cipher that is absolutely unbreakable, however, which is known as the ***one-time pad***. In the one-time pad, which was invented in 1917 by Joseph Mauborgne and Gilbert Vernam, we apply the same approach as with the Vigenère cipher, in that we use a block of keys, $(k_1, k_2, \ldots, k_m)$, to encrypt a plaintext, M, of length n, but with two critical differences.

1. The length, m, of the block of keys has to be the same as n, the length of the plaintext.

2. Each shift amount, k_i, must be chosen completely at random.

With these two additional rules, there is no statistical analysis that can be applied to a ciphertext. Indeed, since each shift amount is chosen completely at random, each letter of the alphabet is equally likely to appear at any place in the ciphertext. Thus, from the eavesdropper's perspective, every letter of the alphabet is equally likely to have produced any given letter in the ciphertext. That is, this cryptosystem is absolutely unbreakable.

Because of its security, it is widely reported that the hotline connecting Moscow and Washington, D.C., during the Cold War was encrypted using a one-time pad. So long as no one reveals the pads—the sequence of random shifts that were used in one-time pad encryptions—the messages that were sent will be secret forever. But when pads are reused, then the security of the messages is quickly reduced, since it allows for statistical methods to be used to discover parts of the plaintext.

But this requirement of one-time use is hard to achieve, since the pad length has to be as long as the message. If Alice and Bob are encrypting a long conversation using a one-time pad, what happens when one of them runs out of pad? Interestingly, such a situation happened during the Cold War. It is now known that the Soviet Union communicated with its spies using one-time pads, but that these pads were sometimes reused by desperate spies who had used up all the pages of pad in their code books. Anticipating that such reuse would occur, the U.S. government initiated an effort, called the ***Venona Project***, to perform analyses of intercepted traffic between the Soviet Union and its spies. The Venona Project was highly successful, in fact, because a significant amount of pad reuse actually did occur in the field, since the one-time pad is impractical.

Binary One-Time Pads

In spite of its impracticality, some principles of the one-time pad are used in other, more-practical cryptosystems. In particular, there is a binary version of the one-time pad that has an elegant interpretation using the binary exclusive-or (XOR) operation. This operation is used in most modern cryptosystems similarly to how it is used in a binary version of the one-time pad cryptosystem. Recall that the ***exclusive-or*** (***XOR***) operator applied to two bits, a and b, yields 1 if a and b are different, and 0 if a and b are the same. In the binary one-time pad, we view the plaintext message, M, as being a binary string of length n. Likewise, we view the pad, P, to be a completely random binary string of length n. We can then specify how to produce the ciphertext, C, using the formula

$$C = M \oplus P,$$

where we make the common notational use of $\oplus$ here to denote the XOR operator applied bitwise to two equal-length binary strings. Like its letter-based counterpart, the binary one-time pad is absolutely unbreakable, because each bit of the ciphertext is equally likely to be a 0 or 1, independent of the plaintext and the other bits of the ciphertext. In addition, given the pad P it is easy to recover the plaintext from the ciphertext, using the formula

$$M = C \oplus P.$$

Indeed, since XOR is associative, we have

$$C \oplus P = (M \oplus P) \oplus P = M \oplus (P \oplus P) = M \oplus \vec{0} = M.$$

where $\vec{0}$ denotes a vector of all zero bits. Thus, in a binary one-time pad cryptosystem, the pad P is used directly for both encryption and decryption.

8.1.4 Pseudo-Random Number Generators

Randomness is a precious resource, as the historical experience with the one-time pad shows. Ignoring the philosophical argument about whether "true" randomness really exists, and sticking to the practical problem of how to gather unpredictable bits, getting a computer or other digital device to generate random numbers is relatively expensive. Current techniques involve sampling subatomic processes whose unpredictability is derived from quantum mechanics or sampling environmental phenomena, such as user input variations, wind noise, or background radiation coming from outer space. None of these techniques are cheap or fast, from a computer's perspective. Moreover, even with these sources of unpredictability, it is not easy to turn any of these sources into uniformly distributed, unbiased sequences of numbers or bits, such as is needed for the one-time pad.

Randomness is useful, however, for such things as secret keys. So it is helpful if we can expand any sources of randomness we have, to get more useful bits from these sources. We can perform such an expansion of randomness by using a ***pseudo-random number generator*** (***PRNG***), which is a method for generating a sequence of numbers that approximates the properties of a random sequence.

The Linear Congruential Generator

A desirable propery of a random sequence is that the numbers it generates are uniformly distributed. One way to achieve this property is to use a method employed, for instance, by the `java.util.Random` class in Java, which is a ***linear congruential generator***. In this PRNG, we start with a random number, x_0, which is called the ***seed***, and we generate the next number, x_{i+1}, in a sequence, from the previous number, x_i, according to the following formula:

$$x_{i+1} = (ax_i + b) \bmod n.$$

Here, we assume that $a > 0$ and $b \geq 0$ are chosen at random from the range $[0, n-1]$, which is also the range of generated numbers. If a and n are relatively prime, then one can prove that the generated sequence is uniformly distributed. For instance, if n itself is prime, then this PRNG will be uniform, which approximates an important property of a random sequence. For cryptographic purposes, the linear congruential generator produces a sequence of numbers that is insufficient as a random sequence however.

Security Properties for PRNG's

In cryptographic applications, we desire pseudo-random number generators with additional properties that the linear congruential generator does not have. For instance, it should be hard to predict x_{i+1} from previous numbers in the sequence. With the linear congruential generator, it is easy to determine the values of a and b as soon as we have seen three consecutive numbers, and, from that point on, an adversary can predict every number that follows.

Another desired property for a pseudo-random sequence concerns its period. Since a pseudo-random sequence is generated deterministically from a random ***seed***, there will be a point where the sequence starts repeating itself. The number of values that are output by the sequence before it repeats is known as its ***period***. For instance, if a is relatively prime to n, then the period of a linear congruential generator is n.

A More Secure PRNG

There are several PRNGs that are believed to be cryptographically secure. For example, a PRNG more secure than the linear congruential generator is one that takes a secure encryption algorithm, like the Advanced Encryption Standard (AES) algorithm (which operates on fixed-length plaintext blocks) and uses it to encrypt, using a common random key, each number in a deterministic sequence of numbers that starts from a random seed. This sequence could even be a consecutive set of integers, as long as it starts from a random seed. Breaking the predictability of such a sequence amounts to a type of ciphertext-only attack, where the adversary knows that the associated plaintexts are taken from a known sequence. The period of this PRNG is equal to 2^n, where n is the the block size. Thus, such a PRNG is much more secure than the linear congruential generator.

Given a secure PRNG, we can use it for encryption and decryption, by making its seed be a secret key, K, and performing an exclusive-or of the sequence of pseudo-random numbers with the plaintext message, M, to produce a ciphertext, as with the one-time pad. Even so, just as with the one-time pad, we should only perform such an encryption only once for any given key, K, and the length of the plaintext should be much smaller than the period for the PRNG. Otherwise, the security of our scheme would be similar to the weak security that comes from reusing a one-time pad. For this reason, such an encryption scheme is best restricted for use as a ***stream cipher***, where we encrypt a single stream of bits or blocks. Stream ciphers where previously discussed in the context of encryption methods for wireless networks (Section 6.5.2).

8.1.5 The Hill Cipher and Transposition Ciphers

Another classic cryptosystem is the ***Hill cipher***, which was invented in 1929 by Lester S. Hill. The Hill cipher takes an approach based on the use of linear algebra. In the description below, we assume the reader is familiar with the basics of matrix multiplication and inverses.

The Hill cipher takes a block of m letters, each interpreted as a number from 0 to 25, and interprets this block as a vector of length m. Thus, if $m = 3$ and a block is the string "CAT," then we would represent this block as the vector

$$\vec{x} = \begin{bmatrix} 2 \\ 0 \\ 19 \end{bmatrix}.$$

The Hill cipher uses an $m \times m$ random matrix, K, as the key, provided that K is invertible when we perform all arithmetic modulo 26. The ciphertext vector, $\vec{c}$, for $\vec{x}$, is determined by the matrix equation

$$\vec{c} = K \cdot \vec{x},$$

where we use standard matrix multiplication for the operator $(\cdot)$, assuming all arithmetic is modulo 26. Given the inverse, K^{-1}, for K, we can recover the plaintext vector, $\vec{x}$, from $\vec{c}$, using the formula

$$\vec{x} = K^{-1} \cdot \vec{c},$$

since

$$K^{-1} \cdot \vec{c} = K^{-1} \cdot (K \cdot \vec{x}) = (K^{-1} \cdot K) \cdot \vec{x} = \vec{1} \cdot \vec{x} = \vec{x}.$$

This approach allows us to specify an encryption of an entire message, M, mathematically, by interpreting M as a matrix of dimension $m \times N$, where $N = n/m$, and defining the ciphertext, C, as an $m \times N$ matrix defined as

$$C = K \cdot M.$$

Then we can recover the entire message, M, from C as follows:

$$M = K^{-1} \cdot C,$$

where, in both the encryption and decryption, we assume that all arithmetic is done modulo 26.

Although this notation is quite elegant, the Hill cipher is still relatively easy to break given enough plaintext-ciphertext pairs. Nevertheless, its use of interpreting letters as numbers and using linear algebra to perform encryption and decryption is another idea from classic cryptography that finds its way into the AES cryptosystem.

Transposition Ciphers

In a ***transposition cipher***, the letters in a block of length m in a plaintext are shuffled around according to a specific ***permutation*** of length m. Since every permutation, π, also has an inverse permutation, π^{-1}, which undoes all the shuffling that is done by π, it is easy to do encryption and decryption of messages in this cryptosystem if we know π. In particular, the encryption of a plaintext M of length m can be done by the formula

$$C = \pi(M),$$

and decryption can be done by the formula

$$M = \pi^{-1}(C).$$

This formula works independent of whether we are viewing the characters in M as letters or as bits.

Transposition Ciphers as Hill Ciphers

Interestingly, such a transposition cipher is actually a special case of a Hill cipher, because any permutation can be performed using matrix multiplication. For example, if the matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

were used in a Hill cipher, then it would be equivalent to the following permutation:

$$\pi : (1,2,3,4,5) \rightarrow (3,1,2,5,4).$$

Note that when we apply a transpositional cipher to the letters in a plaintext M, we do nothing to hide the statistical distribution of the letters in M. Such lack of hiding can leak information. Moreover, since a transposition cipher is a type of Hill cipher, it is subject to the same weakness as the Hill cipher. In particular, with enough plaintext-ciphertext pairs, we can solve a straightforward linear system to determine all the values in the matrix used for encryption. And once we know the encryption matrix, the entire encryption scheme is broken. Nevertheless, if we use permutations and other matrix operations in a nonlinear encryption scheme like the one we discuss next, then the overall cryptosystem will not have this weakness.

8.1.6 The Advanced Encryption Standard (AES)

In 1997, the U.S. National Institute for Standards and Technology (NIST) put out a public call for a replacement of the symmetric encryption algorithm DES. It narrowed down the list of submissions to five finalists, and ultimately chose an algorithm that was then known only as Rijndael (which is pronounced something like "Rhine doll"), designed by cryptographers Joan Daemen and Vincent Rijmen, as the one to become the new standard, the ***Advanced Encryption Standard*** (***AES***).

AES is a block cipher that operates on 128-bit blocks. It is designed to be used with keys that are 128, 192, or 256 bits long, yielding ciphers known as AES-128, AES-192, and AES-256. A schematic input-output diagram of AES is shown in Figure 8.4. As of early 2010, AES-256 is widely regarded as the best choice for a general-purpose symmetric cryptosystem. It is supported by all mainstream operating systems, including Windows, Mac OS, and Linux.

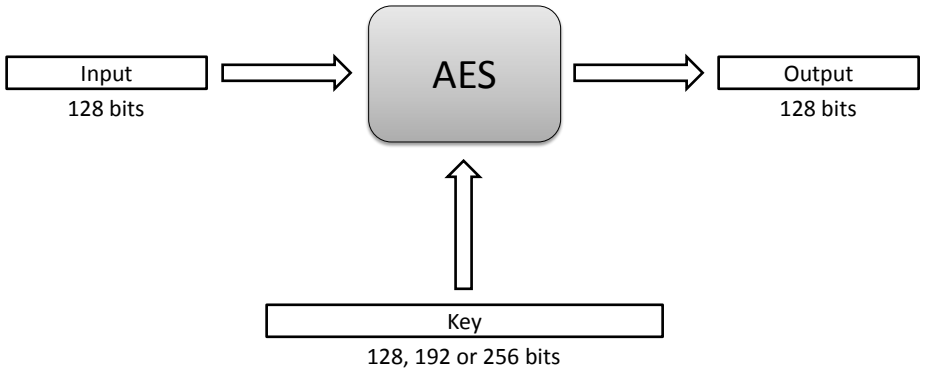


Figure 8.4: Schematic input-output diagram of the AES symmetric block cipher. The block size is always 128 bits. The key length can be 128, 192, or 256 bits.

AES Rounds

The 128-bit version of the AES encryption algorithm proceeds in ten rounds. Each round performs an invertible transformation on a 128-bit array, called ***state***. The initial state X_0 is the XOR of the plaintext P with the key K:

$$X_0 = P \oplus K.$$

Round i ($i = 1, \cdots, 10$) receives state X_{i-1} as input and produces state X_i. The ciphertext C is the output of the final round: $C = X_{10}$. A schematic illustration of the structure of the AES rounds is shown in Figure 8.5.

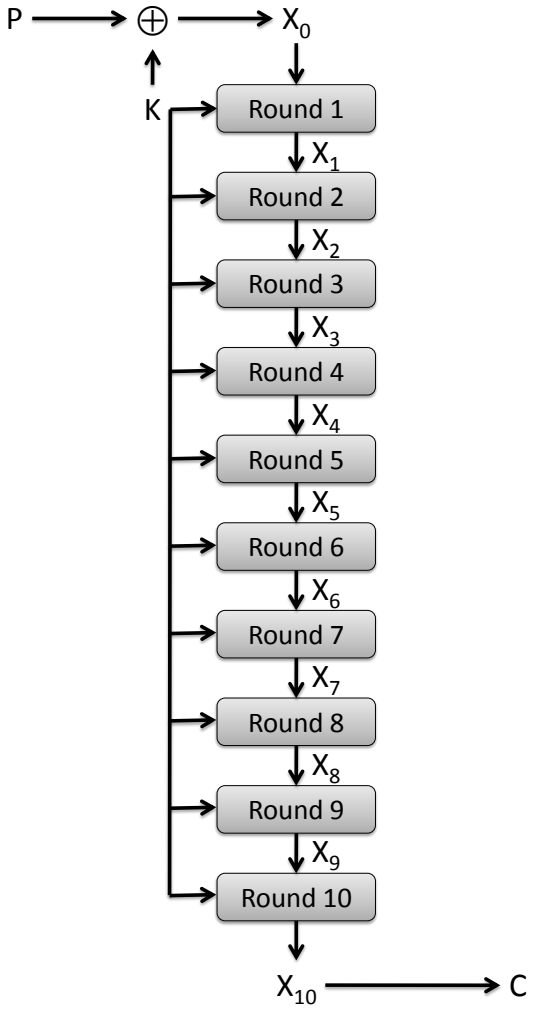


Figure 8.5: The AES rounds.

Each round is built from four basic steps:

1. ***SubBytes step***: an S-box substitution step
2. ***ShiftRows step***: a permutation step
3. ***MixColumns step***: a matrix multiplication (Hill cipher) step
4. ***AddRoundKey step***: an XOR step with a ***round key*** derived from the 128-bit encryption key

These steps are described in detail in Section 8.5.

Implementation of AES

Typical software implementations of AES are optimized for speed of execution and use several ***lookup tables*** to implement the basic steps of each round. A lookup table stores all the possible values of a function into an array that is indexed by the input of the function. It can be shown that the 128-bit version of the AES algorithm can be implemented using exactly eight lookup tables, each mapping an input ***byte*** (an 8-bit word) to an output ***int*** (a 32-bit word). Thus, each of the eight lookup tables stores 256, 32-bit ints. The lookup tables are precomputed and accessed during encryption and decryption.

Using the lookup tables, a round of AES encryption or decryption is implemented by a combination of only three types of operations:

- XOR of two ints: $y = x_1 \oplus x_2$, where x_1, x_2, and y are ints
- Split of an int into 4 bytes: $(y_1, y_2, y_3, y_4) = x$, where y_1, y_2, y_3, and y_4 are bytes and x is an int
- Table lookup of an int indexed by a byte: $y = T[x]$, where y is an int and x is a byte

Attacks on AES

As of early 2010, AES is considered a highly secure symmetric cryptosystem. Indeed, the only known practical attacks on AES are side channel attacks.

Variations of a ***timing attack*** on high-performance software implementations of AES were independently discovered in 2005 by Bernstein and by Osvik, Shamir, and Tromer. Recall that to speed up the running time of AES, the algorithm is implemented using lookup tables. The timing attack is based on the fact that the cache of the processor where the AES algorithm is executed will store portions of the lookup tables used in the implementation of AES. Accessing table entries stored in the cache is much faster that accessing entries in main memory. Thus, the time it takes to execute the algorithm provides information about how the lookup tables are accessed and therefore, the inner workings of the algorithm as well. By timing multiple executions of the algorithm using the same key on a series of known plaintexts of known ciphertexts, the attacker can eventually learn the key.

If the attacker is on the same system where AES is executed, the key can be recovered in less than a second. If the attacker and the AES computation are on different machines, recovering the key takes several hours. To

defend against timing attacks, AES should be implemented in a way that the execution time remains constant, irrespective of the cache architecture.

Other side channel attacks on AES target hardware implementations, such as those on a ***field-programmable gate array*** (FPGA). For example, ***fault attacks*** induce hardware error conditions during the execution of the algorithm and compare the resulting corrupted ciphertext with the correct ciphertext from a regular execution of the algorithm.

8.1.7 Modes of Operation

There are several ways to use a ***block cipher***, such as AES, that operate on fixed-length blocks. The different ways such an encryption algorithm can be used are known as its ***modes of operation***. In this section, we discuss several of the most commonly used modes of operation for block ciphers. The general scenario is that we have a sequence of blocks, B_1, B_2, B_3, and so on, to encrypt, all with the same key, K, using a block cipher algorithm, like AES.

Electronic Codebook (ECB) Mode

The simplest of encryption modes for a block cipher encrypts each block, B_i, independently. That is, this mode, which is known as ***electronic codebook mode*** (***ECB***) mode, involves encrypting the block, B_i, according to the following formula:

$$C_i = E_K(B_i),$$

were E_K denotes the block encryption algorithm using key K. Likewise, decryption is by the following formula:

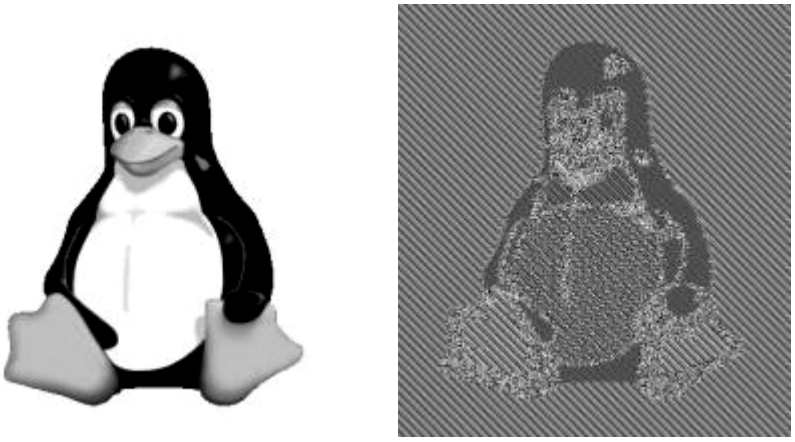
$$B_i = D_K(C_i),$$

where D_K denotes the block decryption algorithm using key K.

This mode has the advantage of simplicity, of course. In addition, it can tolerate the loss of a block, such as might occur if the blocks are being sent as packets over a network. This resilience to block loss comes from the fact that decrypting the ciphertext for a block, B_i, does not depend in any way on the block, B_{i-1}.

The disadvantage of using this mode, however, is that, if our encryption algorithm is completely deterministic, like AES, so that each plaintext has a unique associated ciphertext, then the ECB mode may reveal patterns that might appear in the stream of blocks. In this case, identical blocks will have identical encryptions in ECB mode. For example, in a large image file, blocks of the image that are the same color, and are therefore identical, will

be encrypted in the same way. This disadvantage of ECB mode allows an encryption of a sequence of blocks sometimes to reveal a surprising amount of information, as illustrated in Figure 8.6.



(a) (b)

Figure 8.6: How ECB mode can leave identifiable patterns in a sequence of blocks: (a) An image of Tux the penguin, the Linux mascot. (b) An encryption of the Tux image using ECB mode. (The image in (a) is by Larry Ewing, lewing@isc.tamu.edu, using The Gimp; the image in (b) is by Dr. Juzam. Both are used with permission via attribution.)

Cipher-Block Chaining (CBC) Mode

An encryption mode that avoids the revelation of patterns in a sequence of blocks is the ***cipher-block chaining mode*** (***CBC***). In this mode of operation, the first plaintext block, B_1, is exclusive-ored with an ***initialization vector***, C_0, prior to being encrypted, and each subsequent plaintext block is exclusive-ored with the previous ciphertext block prior to being encrypted. That is, setting C_0 to the initialization vector, then

$$C_i = E_K(B_i \oplus C_{i-1}).$$

Decryption is handled in reverse,

$$B_i = D_K(C_i) \oplus C_{i-1},$$

where we use the same initialization vector, C_0, since exclusive-or is a self-inverting function.

This mode of operation has the advantage that if identical blocks appear at different places in the input sequence, then they are very likely to have

different encryptions in the output sequence. So it is difficult to determine patterns in an encryption that is done using CBC mode, which corrects a disadvantage of ECB mode.

CBC mode does not allow the encryption of the blocks in a sequence to be done independently. That is, the sequence of blocks must be encrypted sequentially, with the encryption of block $i - 1$ completing before the encryption of block i can begin.

Decryption, on the other hand, can proceed in parallel if all the ciphertext blocks are available. This asymmetry is due to the fact that both the encryption and decryption of block i uses the ciphertext block $i - 1$. This block is available during encryption only through a sequential process. But all the encryptions are available for decryption; hence, the decryption can be done in parallel.

In addition, this property implies that the decryption process can tolerate the loss of a ciphertext block. For if block C_i is lost, it implies that decryption of blocks i and $i + 1$ are lost. But decryption of block $i + 2$ can still be done, since it relies only on C_{i+1} and C_{i+2}.

Cipher Feedback (CFB) Mode

The ***cipher feedback mode*** (***CFB***) for block encryption algorithms is similar to that of the CBC mode. Like the CBC, the encryption for block B_i involves the encryption, C_{i-1}, of the previous block. The encryption begins with an initialization vector, C_0. It computes the encryption of the ith block as

$$C_i = E_K(C_{i-1}) \oplus B_i.$$

That is, the ith block is encrypted by first encrypting the previous ciphertext block and then exclusive-oring that with the ith plaintext block. Decryption is done similarly, as follows:

$$B_i = E_K(C_{i-1}) \oplus C_i.$$

That is, decryption of the ith ciphertext block also involves the encryption of the $(i - 1)$st ciphertext block. The decryption algorithm for the block cipher is actually never used in this mode. Depending on the details of the block cipher, this property could allow decryption to proceed faster by using the CFB mode than by using the CBC mode.

Output Feedback (OFB) Mode

In the ***output feedback mode*** (***OFB***), a sequence of blocks is encrypted much as in the one-time pad, but with a sequence of blocks that are generated with the block cipher. The encryption algorithm begins with an initialization vector, V_0. It then generates a sequence of vectors,

$$V_i = E_K(V_{i-1}).$$

Given this sequence of pad vectors, we perform block encryptions as follows:

$$C_i = V_i \oplus B_i.$$

Likewise, we perform block decryptions as follows:

$$B_i = V_i \oplus C_i.$$

Thus, this mode of operation can tolerate block losses, and it can be performed in parallel, both for encryption and decryption, provided the sequence of pad vectors has already been computed.

Counter (CTR) Mode

In ***counter mode*** (***CTR***), every step of encryption and decryption can be done in parallel. This mode is similar to the OFB in that we perform encryption through an exclusive-or with a generated pad. In fact, the method is essentially that mentioned in Section 8.1.4. We start with a random seed, s, and compute the ith offset vector according to the formula

$$V_i = E_K(s + i - 1),$$

so the first pad is an encryption of the seed, the second is an encryption of $s + 1$, the third is an encryption of $s + 2$, and so on. Encryption is performed as in the OFB mode, but with these generated vectors,

$$C_i = V_i \oplus B_i.$$

Likewise, we perform block decryptions as follows:

$$B_i = V_i \oplus C_i.$$

In this case, the generation of the pad vectors, as well as encryptions and decryptions, can all be done in parallel. This mode is also able to recover from dropped blocks.

8.2 Public-Key Cryptography

As we saw to some degree with the AES cryptosystem, a trend in modern cryptography is to view blocks of bits as large numbers represented in binary. Doing this requires that we have a set of tools available for operating on large numbers, many of which we discuss in the next section.

8.2.1 Modular Arithmetic

When we operate on blocks of bits as large numbers, we need to make sure that all our operations result in output values that can be represented using the same number of bits as the input values. The standard way of achieving this is to perform all arithmetic modulo the same number, n. That is, after each operation, be it an addition, multiplication, or other operation, we return the remainder of a division of the result with n. Technically, this means that we are performing arithmetic in Z_n, which is the set of integers

$$Z_n = \{0, 1, 2, \cdots, n-1\}.$$

So algorithms for performing addition, subtraction, and multiplication are basically the same as with standard integers, with this added step of reducing the result to a value in Z_n.

Modulo Operator

Operation x mod n, referred to as x ***modulo*** n, takes an arbitrary integer x and a positive integer n as operands. The result of this operation is a value in Z_n defined using the following rules:

- If $0 \leq x \leq n-1$, that is, $x \in Z_n$, then $x \bmod n = x$. For example, $3 \bmod 13 = 3$ and $0 \bmod 13 = 0$.

- If $x \geq n$, then x mod n is the remainder of the division of x by n. For example, $29 \bmod 13 = 3$ since $29 = 13 \cdot 2 + 3$. Also, $13 \bmod 13 = 0$ and $26 \bmod 13 = 0$ since for any multiple of 13, the remainder of its division by 13 is zero. Note that this rule generalizes the previous rule.

- Finally, if $x < 0$, we add a sufficiently large multiple of n to x, denoted by kn, to get a nonnegative number $y = x + kn$. We have that $x \bmod n = y \bmod n$. Since y is nonnegative, the operation y mod n

can be computed using the previous rules. For example, to compute $-27 \bmod 13$, we can add $3 \cdot 13 = 39$ to -27 to obtain

$$y = -27 + 3 \cdot 13 = -27 + 39 = 12.$$

Thus, we have

$$-27 \bmod 13 = 12 \bmod 13 = 12.$$

In order to find a multiple kn of n greater than x, we can set k as 1 plus the integer division (division without remainder) of $-x$ by n, that is,

$$k = 1 + \left\lfloor \frac{x}{n} \right\rfloor.$$

For example, for $x = -27$ and $n = 13$, we have

$$k = 1 + \lfloor 27/13 \rfloor = 1 + 2 = 3.$$

In general, we have that $x \bmod n$ and $-x \bmod n$ are different.

Several examples of operations modulo 13 are shown below:

$29 \bmod 13 = 3$; $13 \bmod 13 = 0$; $0 \bmod 13 = 0$; $-1 \bmod 13 = 12$.

We can visualize the modulo operator by repeating the sequence of numbers $0, 1, 2, \cdots (n-1)$, as shown in Figure 8.7.

x	...	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	...
$x \bmod 5$	...	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	...

Figure 8.7: Operation $x \bmod 5$.

Modular Inverses

The notion of division in Z_n is not so easy to grasp, however. We can limit ourselves to consider the inverse x^{-1} of a number x in Z_n since we can write a/b as ab^{-1}. We say that y is the ***modular inverse*** of x, modulo n, if the following holds:

$$xy \bmod n = 1.$$

For example, 4 is the inverse of 3 in Z_{11} since

$$4 \cdot 3 \bmod 11 = 12 \bmod 11 = 1.$$

We have that elements 1 and $n-1$ of Z_n always admit an inverse modulo n. Namely, the inverse of 1 is 1 and the inverse of $n-1$ is $n-1$. However, not every other number in Z_n admits a modular inverse, as can be seen from the multiplication table of Figure 8.8.a, which shows the products $xy \bmod 10$ for $x, y \in Z_n$. However, if n is a prime number, then every element but zero in Z_n admits a modular inverse, as shown in Figure 8.8.b.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	**1**	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	**1**	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	**1**	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	**1**

(a)

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	**1**	2	3	4	5	6	7	8	9	10
2	0	2	4	6	8	10	**1**	3	5	7	9
3	0	3	6	9	**1**	4	7	10	2	5	8
4	0	4	8	**1**	5	9	2	6	10	3	7
5	0	5	10	4	9	3	8	2	7	**1**	6
6	0	6	**1**	7	2	8	3	9	4	10	5
7	0	7	3	10	6	2	9	5	**1**	8	4
8	0	8	5	2	10	7	4	**1**	9	6	3
9	0	9	7	5	3	**1**	10	8	6	4	2
10	0	10	9	8	7	6	5	4	3	2	**1**

(b)

Figure 8.8: Modular multiplication tables in Z_n for $n = 10$ and $n = 11$, with highlighted elements that have a modular inverse: (a) xy mod 10. (b) xy mod 11.

Modular Exponentiation

Finally, we consider modular exponentiation, that is, operation

$$x^y \bmod n.$$

Figure 8.9 shows successive modular powers

$$x^1 \bmod n, x^2 \bmod n, \cdots, x^{n-1} \bmod n$$

and illustrates the following patterns:

- If n is not prime, as for $n = 10$ shown in Figure 8.9.a, there are modular powers equal to 1 only for the elements of Z_n that are relatively prime with n. These are exactly the elements x such that the ***greatest common divisor*** (***GCD***) of x and n is equal to 1, as is the case for 1, 3, 7, and 9 for $n = 10$.

- If n is prime, as for $n = 13$ shown in Figure 8.9.b, every nonzero element of Z_n has a power equal to 1. In particular, we always have

$$x^{n-1} \bmod n = 1.$$

We can generalize the patterns above by considering the subset Z_n^* of Z_n consisting of the elements relatively prime with n, that is, the set

$$Z_n^* = \{x \in Z_n \text{ such that } \mathrm{GCD}(x, n) = 1\}.$$

	y								
	1	2	3	4	5	6	7	8	9
1^y	1	1	1	1	1	1	1	1	1
2^y	2	4	8	6	2	4	8	6	2
3^y	3	9	7	1	3	9	7	1	3
4^y	4	6	4	6	4	6	4	6	4
5^y	5	5	5	5	5	5	5	5	5
6^y	6	6	6	6	6	6	6	6	6
7^y	7	9	3	1	7	9	3	1	7
8^y	8	4	2	6	8	4	2	6	8
9^y	9	1	9	1	9	1	9	1	9

(a)

	y											
	1	2	3	4	5	6	7	8	9	10	11	12
1^y	1	1	1	1	1	1	1	1	1	1	1	1
2^y	2	4	8	3	6	12	11	9	5	10	7	1
3^y	3	9	1	3	9	1	3	9	1	3	9	1
4^y	4	3	12	9	10	1	4	3	12	9	10	1
5^y	5	12	8	1	5	12	8	1	5	12	8	1
6^y	6	10	8	9	2	12	7	3	5	4	11	1
7^y	7	10	5	9	11	12	6	3	8	4	2	1
8^y	8	12	5	1	8	12	5	1	8	12	5	1
9^y	9	3	1	9	3	1	9	3	1	9	3	1
10^y	10	9	12	3	4	1	10	9	12	3	4	1
11^y	11	4	5	3	7	12	2	9	8	10	6	1
12^y	12	1	12	1	12	1	12	1	12	1	12	1

(b)

Figure 8.9: Modular exponentiation tables in Z_n for $n = 10$ and $n = 13$, with highlighted powers equal to 1 and elements of Z_n that have some power equal to 1: (a) $x^y \bmod 10$. (b) $x^y \bmod 13$.

For example, for $n = 10$, we have

$$Z_{10}^* = \{1, 3, 7, 9\}.$$

Also, if n is prime, we always have

$$Z_n^* = \{1, 2, \cdots, (n-1)\}.$$

Let $\phi(n)$ be the number of elements of Z_n^*, that is,

$$\phi(n) = |Z_n^*|.$$

Function $\phi(n)$ is called the ***totient*** of n. The following property, known as ***Euler's Theorem***, holds for each element x of Z_n^*:

$$x^{\phi(n)} \bmod n = 1.$$

A consequence of Euler's theorem is that we can reduce the exponent modulo $\phi(n)$:

$$x^y \bmod n = x^{y \bmod \phi(n)} \bmod n.$$

Note given two elements x and y of Z_n^*, their modular product $xy \bmod n$ is also in Z_n^*. Also, for each element x of Z_n^*, the modular inverse of x is $x^{\phi(n)-1}$. Indeed, we have

$$x \cdot x^{\phi(n)-1} \bmod n = x^{\phi(n)} \bmod n = 1.$$

More details on modular arithmetic are given in Section 8.5.2.

8.2.2 The RSA Cryptosystem

Recall that in a public-key cryptosystem, encryption is done with a public key, K_P, associated with the intended recipient, Bob, of the plaintext message, M. The sender, Alice, doesn't have to have a prior relationship with Bob and she doesn't have to have figured out a way to share a secret key with Bob, as she would if she wanted to use a symmetric encryption scheme, like AES, to secretly communicate with Bob. Once the message M has been transformed into a ciphertext, $C = E_{K_P}(M)$, Alice sends C to Bob. Bob is then able to decrypt the ciphertext C using his secret key, K_S, by using the appropriate decryption method, $D_{K_S}(C)$.

In this section, we describe a specific public-key cryptosystem, which is named ***RSA***, after its inventors, Ronal Rivest, Adi Shamir, and Leonard Adleman (see Figure 8.10). In this cryptosystem, we treat plaintext and ciphertext message blocks as large numbers, represented using thousands of bits. Encryption and decryption are done using modular exponentiation and the correctness of these encryption and decryption algorithms is based on Euler's Theorem and other properties of modular arithmetic.

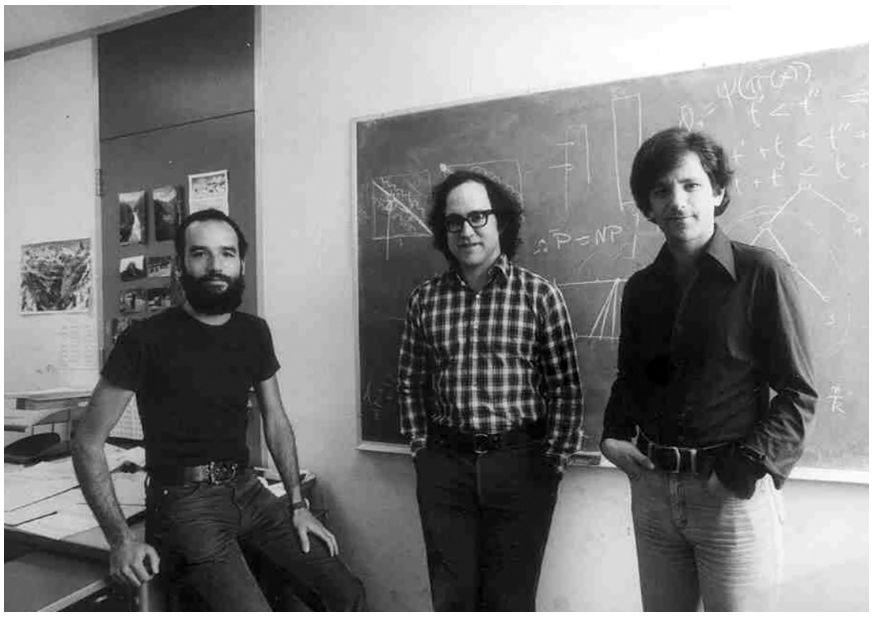


Figure 8.10: The inventors of the RSA cryptosystem, from left to right, Adi Shamir, Ron Rivest, and Len Adleman, who received the Turing Award in 2002 for this achievement. (Image used with permission from Ron Rivest and Len Adleman.)

RSA Encryption and Decryption

The setup for RSA allows a potential message receiver, Bob, to create his public and private keys. It begins with Bob generating two large, random prime numbers, p and q, and setting $n = pq$. He then picks a number, e, that is relatively prime to $\phi(n)$, and he computes $d = e^{-1} \bmod \phi(n)$. From this point on, he can "throw away" the values of p, q, and $\phi(n)$. They are no longer needed. Bob's public key is the pair, (e, n). His private key is d. He needs to keep d a secret, but he should publish (e, n) to any places that might allow others to use it to send Bob encrypted messages.

Given Bob's public key, (e, n), Alice can encrypt a message, M, for him by computing

$$C = M^e \bmod n.$$

Thus, encrypting M requires a single modular exponentiation.

To decrypt the ciphertext, C, Bob performs a modular exponentiation,

$$C^d \bmod n,$$

and sets the result to M. This is, in fact, the plaintext that Alice encrypted, as the following shows for the case when M is relatively prime to n:

$$\begin{aligned} C^d \bmod n &= (M^e)^d \bmod n \\ &= M^{ed} \bmod n \\ &= M^{ed \bmod \phi(n)} \bmod n \\ &= M^1 \bmod n \\ &= M. \end{aligned}$$

When M is not relatively prime to n, it must still be relatively prime to either p or q, since $M < n$. So, in the case that $M = ip$ (with a similar argument for when $M = iq$),

$$M^{\phi(n)} \bmod q = 1,$$

by Euler's Theorem, since $\phi(n) = \phi(p)\phi(q)$. Thus, $M^{k\phi(n)} \bmod q = 1$, where k is defined so that $ed = k\phi(n) + 1$. So $M^{k\phi(n)} = 1 + hq$, for some integer h; hence, multiplying both sides by M, we see that $M^{k\phi(n)+1} = M + Mhq$. But, in this case, $M = ip$, which implies

$$\begin{aligned} M^{k\phi(n)+1} \bmod n &= (M + Mhq) \bmod n \\ &= (M + iphq) \bmod n \\ &= (M + (ih)pq) \bmod n \\ &= (M + (ih)n) \bmod n \\ &= M. \end{aligned}$$

Thus, we have shown the correctness of the RSA decryption method.

The Security of the RSA Cryptosystem

The security of the RSA cryptosystem is based on the difficulty of finding d, given e and n. If we knew $\phi(n) = (p-1)(q-1)$, it would be easy to compute d from e. Thus, Bob needs to keep p and q secret (or even destroy all knowledge of them), since anyone who knows the values of p and q immediately knows the value of $\phi(n)$. Anyone who knows the value of $\phi(n)$ can compute $d = e^{-1} \bmod \phi(n)$, using the extended Euclidian algorithm.

Thus, the security of the RSA cryptosystem is closely tied to factoring n, which would reveal the values of p and q. Fortunately, since this problem has shown itself to be hard to solve, we can continue to rely on the security of the RSA crptosystem, provided we use a large enough modulus. As of 2010, a 2,048-bit modulus is recommended. Side channel attacks have also been demonstrated on RSA, based on measuring the time taken by decryption and/or the power consumption of the CPU performing the operation.

We must take some care in how we use the RSA cryptosystem, however, because of its deterministic nature. For example, suppose we use the RSA algorithm to encrypt two plaintext messages, M_1 and M_2, into the respective ciphertexts, C_1 and C_2, using the same public key. Because RSA is deterministic, we know that, in this case, if $C_1 = C_2$, then $M_1 = M_2$. Unfortunately, this fact could allow a cryptanalyst to infer information from ciphertexts encrypted from supposedly different plaintexts. The cryptosystem we discuss in Section 8.2.3 does not have the same disadvantage.

Efficient Implementation the RSA Cryptosystem

The implementation of the RSA cryptosystem requires efficient algorithms for the following tasks:

- Primality testing, that is, testing if an integer is prime. This algorithm is used in the setup phase to pick the factors p and q of the RSA modulus. Each factor is picked by generating a series of random numbers and stopping as soon as a prime is found.
- Computing the greatest common divisor, which is used in the setup phase to pick the encryption exponent.
- Computing the modular inverse, which is used in the setup phase to compute the decryption exponent given the encryption exponent.
- Modular power, used in the encryption and decryption algorithms. Clearly, first computing the power and then applying the modulo operator is inefficient since the power can be a very large number.

In Section 8.5.2, we present an efficient algorithm for these tasks.

8.2.3 The Elgamal Cryptosystem

The ***Elgamal*** cryptosystem, named after its inventor, Taher Elgamal, is a public-key cryptosystem that uses randomization, so that independent encryptions of the same plaintext are likely to produce different ciphertexts. It is based on viewing input blocks as numbers and applying arithmetic operations on these numbers to perform encryption and decryption. Before we give the details for this cryptosystem, let us discuss some related concepts from number theory.

In the number system Z_p, all arithmetic is done modulo a prime number, p. A number, g in Z_p, is said to be a ***generator*** or ***primitive root*** modulo p if, for each positive integer i in Z_p, there is an integer k such that $i = g^k \bmod p$.

It turns out that there are $\phi(\phi(p)) = \phi(p-1)$ generators for Z_p. So we can test different numbers until we find one that is a generator. To test whether a number, g, is a generator, it is sufficient that we test that

$$g^{(p-1)/p_i} \bmod p \neq 1,$$

for each prime factor, p_i, of $\phi(p) = p - 1$. If a number is not a generator, one of these powers will be equal to 1. Normally, it would be hard to factor $p - 1$, to find all its prime factors. But we can actually make this job easy by choosing candidates for the prime number p in such a way that we know the factoring of $p - 1$. The Elgamal cryptosystem requires such a generator, so let us assume here that we can choose any prime number, p, in a way that facilitates our ability to quickly find a generator, g, for Z_p.

Once we have a generator g, we can efficiently compute $x = g^k \bmod p$, for any value k (see Section 8.5.2 for details). Conversely, given x, g, and p, the problem of determining k such that $x = g^k \bmod p$ is known as the ***discrete logarithm*** problem. Like factoring, the discrete logarithm problem is widely believed to be computationally hard. The security of the Elgamal cryptosystem depends on the difficulty of the discrete logarithm problem.

As a part of the setup, Bob chooses a random large prime number, p, and finds a generator, g, for Z_p. He then picks a random number, x, between 1 and $p - 2$, and computes $y = g^x \bmod p$. The number, x, is Bob's secret key. His public key is the triple (p, g, y).

When Alice wants to encrypt a plaintext message, M, for Bob, she begins by getting his public key, (p, g, y). She then generates a random number, k, between 1 and $p - 2$, and she then uses modular multiplication and exponentiation to compute two numbers:

$$\begin{aligned} a &= g^k \bmod p \\ b &= My^k \bmod p. \end{aligned}$$

The encryption of M is the pair (a, b).

Decryption and Security Properties

Note that an Elgamal encryption is dependent on the choice of the random number, k. Moreover, each time Alice does an Elgamal encryption, she must use a different random number. If she were to reuse the same random number, she would be leaking information much like the one-time pad would leak information if we were to reuse a pad.

Given an Elgamal ciphertext, (a, b), created for Bob, he can decrypt this ciphertext by computing $a^x \bmod p$, computing the inverse of this value modulo p, and multiplying the result by b, modulo p. This sequence of computations gives Bob the following:

$$M = b(a^x)^{-1} \bmod p.$$

The reason this actually decrypts the ciphertext is as follows:

$$\begin{aligned} b(a^x)^{-1} \bmod p &= My^k(g^{kx})^{-1} \bmod p \\ &= M(g^x)^k g^{-kx} \bmod p \\ &= Mg^{xk}g^{-kx} \bmod p \\ &= Mg^{kx}g^{-kx} \bmod p \\ &= M \bmod p \\ &= M. \end{aligned}$$

Note that Bob doesn't need to know the random value, k, to decrypt a message that was encrypted using this value. And Alice didn't need to know Bob's secret key to encrypt the message for him in the first place. Instead, Alice got g^x, as y, from Bob's public key, and Bob got g^k, as a, from Alice's ciphertext. Alice raised y to the power k and Bob raised a to the power x, and in so doing they implicitly computed a type of one-time shared key, g^{xk}, which Alice used for encryption and Bob used for decryption.

The security of this scheme is based on the fact that, without knowing x, it would be very difficult for an eavesdropper to decrypt the ciphertext, (a, b). Since everyone knows $y = g^x \bmod p$, from Bob's public key, the security of this scheme is therefore related to the difficulty of solving the discrete logarithm problem. That is, Elgamal could be broken by an eavesdropper finding the secret key, x, given only y, knowing that y happens to be equal to $g^x \bmod p$. As previously mentioned, the discrete logarithm problem is another one of those problems generally believed to be computationally difficult. Thus, the security of the Elgamal cryptosystem is based on a difficult problem from number theory.

8.2.4 Key Exchange

The use of a symmetric cryptosystem requires that Alice and Bob agree on a secret key before they can send encrypted messages to each other. This agreement can be accomplished, for example, by the one-time use of a private communication channel, such as an in-person meeting in a private room, or mailing in tamper-proof containers. A ***key exchange protocol***, which is also called ***key agreement protocol***, is a cryptographic approach to establishing a shared secret key by communicating solely over an insecure channel, without any previous private communication.

Intuitively, the existence of a key exchange protocol appears unlikely, as the adversary can arbitrarily disrupt the communication between Alice and Bob. Indeed, it can be shown that no key exchange protocol exists if the adversary can actively modify messages sent over the insecure channel. Nevertheless, key exchange can be successfully accomplished if the adversary is limited to only passive eavesdropping on messages.

The classic ***Diffie-Hellman key exchange protocol*** (***DH protocol***), which is named after its inventors, Whitfield Diffie and Martin Hellman, is based on modular exponentiation. The DH protocol assumes that the following two public parameters have been established and are known to all participants (including the attacker): a prime number, p, and a generator (Section 8.2.3), g, for Z_p. The DH protocol consists of the following steps:

1. Alice picks a random positive number x in Z_p and uses it to compute $X = g^x \bmod p$. She sends X to Bob.
2. Bob picks a random positive number y in Z_p and uses it to compute $Y = g^y \bmod p$. He sends Y to Alice.
3. Alice computes the secret key as $K_1 = Y^x \bmod p$.
4. Bob computes the secret key as $K_2 = X^y \bmod p$.

Note that Steps 1–2 can be performed in parallel. Similarly, Steps 3–4 can also be performed in parallel. At the end of the protocol, Alice and Bob have computed the same secret key $K = g^{xy} \bmod p = K_1 = K_2$, since

$$K_1 = Y^x \bmod p = (g^y)^x \bmod p = (g^x)^y \bmod p = X^y \bmod p = K_2.$$

The security of the DH protocol is based on the assumption that it is difficult for the attacker to determine the key K from the public parameters and the eavesdropped values X and Y. Indeed, recovering either x from X or y from Y is equivalent to solving the discrete logarithm problem, which is believed to be computationally hard, as discussed in Section 8.2.3. More generally, no methods are known for efficiently computing $K = g^{xy} \bmod p$ from p, g, $X = g^x \bmod p$ and $Y = g^y \bmod p$, which is called the ***Diffie-Hellman problem***.

Even though it is secure against a passive attacker, the DH protocol is vulnerable to a man-in-the-middle attack if the attacker can intercept and modify the messages exchanged by Alice and Bob. The attack, illustrated in Figure 8.11, results in Alice and Bob unknowingly selecting different keys that are known to the attacker, who can subsequently decrypt all ciphertexts exchanged by Alice and Bob.

The attack works as follows:

1. The attacker picks numbers s and t in Z_p.
2. When Alice sends the value $X = g^x \bmod p$ to Bob, the attacker reads it and replaces it with $T = g^t \bmod p$.
3. When Bob sends the value $Y = g^y \bmod p$ to Alice, the attacker reads it and replaces it with $S = g^s \bmod p$.
4. Alice and the attacker compute key $K_1 = g^{xs} \bmod p$.
5. Bob and the attacker compute key $K^2 = g^{y\,t} \bmod p$.
6. When Alice sends a message to Bob encrypted with the key K_1, the attacker decrypts it, reencrypts it with the key K_2 and sends it to Bob.
7. When Bob sends a message to Alice encrypted with the key K_2, the attacker decrypts it, reencrypts it with the key K_1 and sends it to Alice.

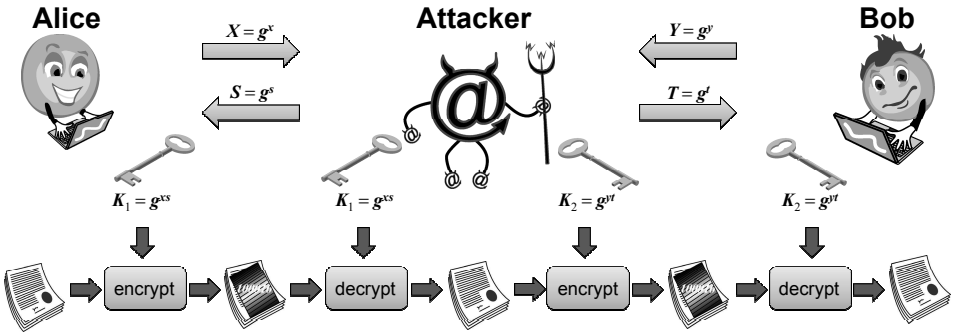


Figure 8.11: The man-in-the-middle attack against the DH protocol. First, by intercepting and modifying the messages of the DH protocol, the attacker establishes a secret key, K_1, with Alice and secret key, K_2, with Bob. Next, using keys K_1 and K_2, the attacker reads and forwards messages between Alice and Bob by decrypting and reencrypting them. Alice and Bob are unaware of the attacker and believe they are communicating securely with each other.

8.3 Cryptographic Hash Functions

As mentioned previously, we often wish to produce a compressed digest of a message. A cryptographic ***hash function*** serves this purpose, while also providing a mapping that is deterministic, ***one-way***, and ***collision-resistant***. Cryptographic hash functions were introduced in Section 1.3.4. In this section, we discuss them in more detail.

8.3.1 Properties and Applications

One of the critical properties of cryptographic hash functions is that they are one-way. That is, given a message, M, it should be easy to compute a hash value, $H(M)$, from that message. However, given only a value, x, it should be difficult to find a message, M, such that $x = H(M)$. Moreover, the hash value should be significantly smaller than a typical message. For example, the commonly used standard hash function SHA-256 produces hash values with 256 bits. This hash function uses several of the techniques employed in symmetric encryption, including substitution, permutation, exclusive-or, and iteration, in a way that provides so much diffusion of the input that changing any bit in the input could potentially impact the value of every bit in the output. Rather than go into these details, however, let us discuss the properties of cryptographic hash functions and how they are used.

Collision Resistance

A hash function, H, is a mapping of input strings to smaller output strings. We say that H has ***weak collision resistance*** if, given any message, M, it is computationally difficult to find another message, $M' \neq M$, such that

$$H(M') = H(M).$$

Hash function H has ***strong collision resistance*** if it is computationally difficult to compute two distinct messages, M_1 and M_2, such that $H(M_1) = H(M_2)$. That is, in weak collision resistance, we are trying to avoid a collision with a specific message, and in strong collision resistance we are trying to avoid collisions in general. It is usually a challenge to prove that real-world cryptographic hash functions have strong collision resistance, so cryptographers typically provide experimental evidence for this property.

The Merkle-Damgård Construction

A common structure for a hash function is to use as a building block a ***cryptographic compression function*** $C(X, Y)$, which is a cryptographic hash function C that takes as input two strings, X and Y, where X has fixed length m and Y has fixed length n, and produces a hash value of length n. Given a message M, we divide M into multiple blocks, M_1, M_2, ..., M_k, each of length m, where the last block is padded in an unambiguous way with additional bits to make it of length m. We start by applying the compression function C to the first block, M_1, and a fixed string v of length n, known as the ***initialization vector***. Denote the resulting hash value with $d_1 = C(M_1, v)$. Next, we apply the compression function to block M_2 and d_1, resulting in hash value $d_2 = C(M_2, d_1)$, and so on. We define the hash value of message H as equal to d_k. This method for constructing a cryptographic hash function from a cryptographic compression function, illustrated in Figure 8.12, is known as the ***Merkle-Damgård construction***, after his inventors Ralph Merkle and Ivan Damgård.

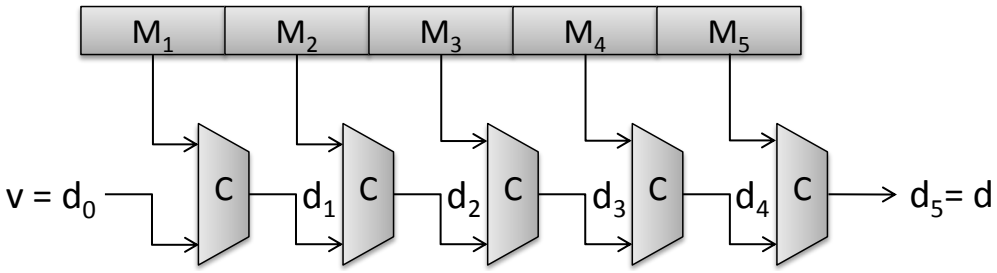


Figure 8.12: The Merkle-Damgård construction.

In the Merkle-Damgård construction, if an attacker finds a collision between two different messages, M_1 and M_2, i.e., $H(M_1) = H(M_2)$, then he can form other arbitrary collisions, Indeed, for any message P, we have

$$H(M_1||P) = H(M_2||P),$$

where the "||" symbol denotes string concatenation. Thus, it is important for a compression function to have strong collision resistance.

Practical Hash Functions for Cryptographic Applications

The hash functions currently recommended for cryptographic applications are the ***SHA-256*** and ***SHA-512*** functions standardized by NIST, where SHA stands for "secure hash algorithm" and the numeric suffix refers to the length of the hash value. These functions follows the Merkle-Damgård

construction. SHA-256 employs a compression function with inputs of $m = 512$ bits and $n = 256$ bits and produces hash values of $n = 256$ bits. These parameters are $m = 1,024$ and $n = 512$ for SHA-512.

The MD5 hash function, where MD refers to "message digest", is still widely used in legacy applications. However, it is considered insecure as several attacks against it have been demonstrated. In particular, it has been shown that given two arbitrary messages, M_1 and M_2, one can efficiently compute suffixes S_1 and S_2 such that $M_1||P1$ and $M_2||P_2$ collide under MD5. For example, using this approach, one can generate different PDF files or executable files with the same MD5 hash, a major vulnerability.

8.3.2 Birthday Attacks

The chief way that cryptographic hash functions are attacked is by compromising their collision resistance. Sometimes this is done by careful cryptanalysis of the algorithms used to perform cryptographic hashing. But it can also be done by using a brute-force technique known as a ***birthday attack***. This attack is based on a nonintuitive statistical phenomenon that states that as soon as there are more than 23 people in a room, there is better than a 50-50 chance that two of the people have the same birthday. And if there are more than 60 people in a room, it is almost certain that two of them share a birthday. The reason for this fact is that if there are 23 people in a room, there are

$$23 \cdot 22/2 = 253$$

possible pairs of people, all of which would have to be different for there to be no two people with the same birthday. When there are 60 people in the room, the number of distinct pairs of people is

$$60 \cdot 59/2 = 1770.$$

Suppose that a cryptographic hash function, H, has a b-bit output. We have that the number of possible hash values is 2^b. We might at first think that an attacker, Eve, needs to generate a number of inputs proportional to 2^b before she finds a collision, but this is not the case.

In the birthday attack, Eve generates a large number of random messages and she computes the cryptographic hash value of each one, hoping to find two messages with the same hash value. By the same type of argument used for the birthday coincidence in a room full of people, if the number of messages generated is sufficiently large, there is a high likelihood that two of the messages will have the same hash value. That is, there is a high likelihood of a collision in the cryptographic hash function

among the candidates tested. All Eve has to do is to sort the set of generated values to find a pair that are equal. Eve does not need to try a number of messages that are proportional to 2^b, but can reduce that to something on the order of $2^{b/2}$. For this reason, we usually think of the security of a cryptographic hash function in terms of half of the size of its output. Thus, a collision-resistant hash function with 256-bit has values has 128-bit security.

Analysis of the Birthday Attack

We now outline the mathematical analysis of the birthday attack. Consider a b-bit hash function and let $m = 2^b$ denote the number of possible hash values. The probability that the i-th message generated by the attacker does not collide with any of the previous $i - 1$ messages is

$$1 - \frac{i-1}{m}.$$

Thus, the ***failure probability*** at round k, that is, the probability the attacker has not found any collisions after generating k message, is

$$F_k = \left(1 - \frac{1}{m}\right) + \left(1 - \frac{2}{m}\right) + \left(1 - \frac{3}{m}\right) + \cdots + \left(1 - \frac{k-1}{m}\right).$$

To find a closed-form expression for F_k, we use the following standard approximation:

$$1 - x \approx e^{-x}$$

Thus, we obtain

$$F_k \approx e^{-\left(\frac{1}{m} + \frac{2}{m} + \frac{3}{m} + \cdots + \frac{k-1}{m}\right)} = e^{-\frac{k(k-1)}{m}}.$$

The attack fails/succeeds with 50% probability when $F_k = \frac{1}{2}$, that is,

$$e^{-\frac{k(k-1)}{m}} = \frac{1}{2}.$$

Solving the above expression for k, we get

$$k \approx 1.17\sqrt{m}.$$

Note that the number of bits of $\sqrt{m}$ is $\frac{b}{2}$, half the number of bits of m. This concludes our justification of the birthday attack.

8.4 Digital Signatures

Digital signatures were introduced in Section 1.3.2. In this section, we recall the definition and main properties of digital signatures and we show how to use the RSA and Elgamal cryptosystems as digital signature schemes.

A ***digital signature*** is a way for an entity to demonstrate the authenticity of a message by binding its identity with that message. The general framework is that Alice, should be able to use her private key with a signature algorithm to produce a digital signature, $S_{\text{Alice}}(M)$, for a message, M. In addition, given Alice's public key, the message, M, and Alice's signature, $S_{\text{Alice}}(M)$, it should be possible for another party, Bob, to verify Alice's signature on M, using just these elements. (See Figure 8.13.)

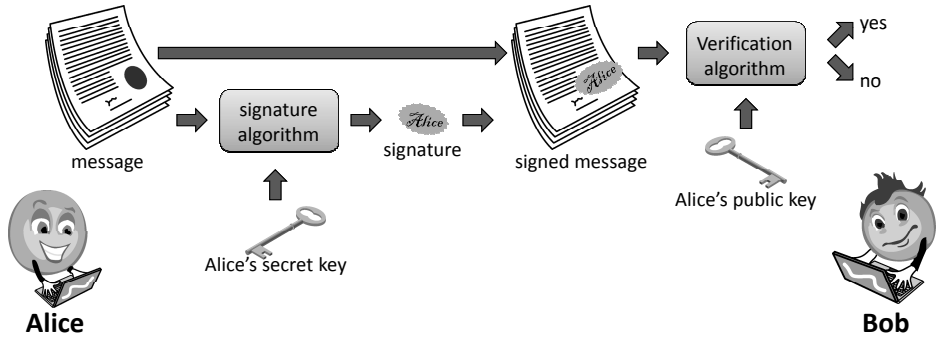


Figure 8.13: The digital signing process for Alice and the signature verification process for Bob.

Two important properties that we would like to have for a digital-signature scheme are the following:

- ***Nonforgeability***. It should be difficult for an attacker, Eve, to forge a signature, $S_{\text{Alice}}(M)$, for a message, M, as if it is coming from Alice.

- ***Nonmutability***. It should be difficult for an attacker, Eve, to take a signature, $S_{\text{Alice}}(M)$, for a message, M, and convert $S_{\text{Alice}}(M)$ into a valid signature on a different message, N.

If a digital-signature scheme achieves these properties, then it actually achieves one more, ***nonrepudiation***. It should be difficult for Alice to claim she didn't sign a document, M, once she has produced a digital signature, $S_{\text{Alice}}(M)$, for that document.

8.4.1 The RSA Signature Scheme

The first digital-signature scheme we study is the ***RSA signature*** scheme. Referring back to the discussion on the RSA cryptosystem from Section 8.2.2, recall that, in this cryptosystem, Bob creates a public key, (e, n), so that other parties can encrypt a message, M, as $C^e \bmod n$. In the RSA signature scheme, Bob instead encrypts a message, M, using his secret key, d, as follows:

$$S = M^d \bmod n.$$

Any third party can verify this signature by testing the following condition:

$$\text{Is it true that } M = S^e \bmod n?$$

The verification method follows from the fact that $de \bmod \phi(n) = 1$. Indeed, we have

$$S^e \bmod n = M^{de} \bmod n = M^{de \bmod \phi(n)} \bmod n = M^1 \bmod n = M.$$

In addition, the verification of the RSA signature scheme involves the same algorithm as RSA encryption and uses the same public key, (e, n), for Bob.

The nonforgeability of this scheme comes from the difficulty of breaking the RSA encryption algorithm. In order to forge a signature from Bob on a message, M, an attacker, Eve, would have to produce $M^d \bmod n$, but do so without knowing d. This amounts to being able to decrypt M as if it were an RSA encryption intended for Bob.

Strictly speaking, the RSA signature scheme does not achieve nonmutability, however. Suppose, for example, that an attacker, Eve, has two valid signatures,

$$S_1 = M_1^d \bmod n \quad \text{and} \quad S_2 = M_2^d \bmod n,$$

from Bob, on two messages, M_1 and M_2. In this case, Eve could produce a new signature,

$$S_1 \cdot S_2 \bmod n = (M_1 \cdot M_2)^d \bmod n,$$

which would validate as a verifiable signature from Bob on the message

$$M_1 \cdot M_2.$$

Fortunately, this issue is not a real problem in practice, for digital signatures are almost always used with cryptographic hash functions, as discussed in Section 8.4.3, which fixes this problem with the RSA signature scheme.

8.4.2 The Elgamal Signature Scheme

In the ***Elgamal signature*** scheme, document signatures are done through randomization, as in Elgamal encryption, but the details for Elgamal signatures are quite different from Elgamal encryption. Recall that in the setup for Elgamal encryption, Alice chooses a large random number, p, finds a generator for Z_p, picks a (secret) random number, x, computes $y = g^x \bmod p$, and publishes the pair (y, p) as her public key. To sign a message, M, Alice generates a fresh one-time-use random number, k, and computes the following two numbers:

$$\begin{aligned} a &= g^k \bmod p \\ b &= k^{-1}(M - xa) \bmod (p-1). \end{aligned}$$

The pair, (a, b), is Alice's signature on the message, M.

To verify the signature, (a, b), on M, Bob performs the following test:

$$\text{Is it true that } y^a a^b \bmod p \;=\; g^M \bmod p?$$

This is true because of the following:

$$\begin{aligned} y^a a^b \bmod p &= (g^x \bmod p)((g^k \bmod p)^{k^{-1}(M-xa) \bmod (p-1)} \bmod p) \\ &= g^{xa} g^{kk^{-1}(M-xa) \bmod (p-1)} \bmod p \\ &= g^{xa+M-xa} \bmod p \\ &= g^M \bmod p. \end{aligned}$$

The security of this scheme is based on the fact that the computation of b depends on both the random number, k, and Alice's secret key, x. Also, because k is random, its inverse is also random; hence, it is impossible for an adversary to distinguish b from a random number, unless she can solve the discrete logarithm problem to determine the number k from a (which equals $g^k \bmod p$). Thus, like Elgamal encryption, the security of the Elgamal signature scheme is based on the difficulty of computing discrete logarithms.

In addition, it is important that Alice never reuse a random number, k, for two different signatures. For instance, suppose she produces

$$b_1 = k^{-1}(M_1 - ax) \bmod (p-1) \quad \text{and} \quad b_2 = k^{-1}(M_2 - ax) \bmod (p-1),$$

with the same $a = g^k \bmod p$, for two different messages, M_1 and M_2. Then

$$(b_1 - b_2)k \bmod (p-1) \;=\; (M_1 - M_2) \bmod (p-1).$$

Thus, since $b_1 - b_2$ and $M_1 - M_2$ are easily computed values, an attacker, Eve, can compute k. And once Eve knows k, she can compute x from either b_1 or b_2, and from that point on, Eve knows Alice's secret key.

8.4.3 Using Hash Functions with Digital Signatures

For practical purposes, the above descriptions of the RSA and Elgamal digital-signature schemes are not what one would use in practice. For one thing, both schemes are inefficient if the message, M, being signed is very long. For instance, RSA signature creation involves an encryption of the message, M, using a private key, and Elgamal signature verification requires a modular exponentiation by M. For another, one can construct valid RSA signatures on combined messages from existing RSA signatures. Thus, for practical and security reasons, it is useful to be able to restrict digital signatures to messages that are digests.

For these reasons, real-world, digital-signature schemes are usually applied to cryptographic hashes of messages, not to actual messages. This approach significantly reduces the mutability risk for RSA signatures, for instance, since it is extremely unlikely that the product of two hash values, $H(M)$ and $H(N)$, would itself be equal to the hash of the product message, $M \cdot N$. Moreover, signing a hash value is more efficient than signing a full message.

Of course, the security of signing hash values depends on both the security of the signature scheme being used and the security of the cryptographic hash function being used as well. For instance, suppose an attacker, Eve, has found a collision between two inputs, M and N, with respect to a hash function, H, so that

$$H(M) = H(N).$$

If Eve can then get Alice to sign the hash, $H(M)$, of the message, M, then Eve has in effect tricked Alice into signing the message M. Thus, in the context of digital signatures of hash values, the risks of the birthday attack are heightened.

For example, Eve could construct a large collection of messages, M_1, M_2, ..., M_k, that are all various instances of a purchase agreement for Eve's guitar that Alice has agreed to buy for \$150. Because of the ambiguity of English, there are many different instances of the same essential message, so that each of the messages, M_i, means the same thing. But Eve could also construct a series of messages, N_1, N_2, ..., N_k, that are all variations of a purchase agreement for Eve's car that says Alice is agreeing to buy it for \$10,000. If Eve can find a collision between some M_i and N_j so that

$$H(M_i) = H(N_j),$$

then by getting Alice to sign the message M_i agreeing to buy Eve's guitar, Eve has also tricked Alice into signing the message N_j, agreeing to buy Eve's car.

8.5 Details of AES and RSA Cryptography

In this section, we give the details for the AES and RSA cryptosystems.

8.5.1 Details for AES

We provide a detailed description of the AES symmetric encryption algorithm for 128-bit keys. Recall that in Section 8.1.6, we discussed the ten rounds, built from four basic steps, of the 128-bit version of the AES algorithm. The algorithm starts with an AddRoundKey step applied directly to a 128-bit block of plaintext. It then performs the four steps repeatedly and in the order outlined in Section 8.1.6 for nine rounds, with the input of each step coming from the output of the previous step. Then, in the tenth round it performs the same set of steps, but with the MixColumns step missing, to produce a 128-bit block of ciphertext. As we discuss below, each of the steps is invertible, so the decryption algorithm essentially amounts to running this algorithm in reverse, to undo each of the transformations done by each step.

Matrix Representation

To provide some structure to the 128-bit blocks it operates on, the AES algorithm views each such block, starting with the 128-bit block of plaintext, as 16 bytes of 8 bits each,

$$(a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1}, a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2}, a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3}),$$

arranged in column-major order into a 4×4 matrix as follows:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}.$$

SubBytes Step

In the SubBytes step, each byte in the matrix is substituted with a replacement byte according to the S-box shown in Figure 8.14, resulting in the following transformation:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}.$$

This S-box is actually a lookup table for a mathematical equation on 8-bit binary words that operates in an esoteric number system known as $GF(2^8)$. Such an interpretation is not necessary for performing the SubBytes step, however, since we can perform this step with a simple lookup in the S-box table. So we omit the details of this equation here. Likewise, the inverse of this step, which is needed for decryption, can also be done with a fast and simple S-box lookup, which we also omit.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 8.14: The S-box used in the SubBytes step of AES. Each byte is shown in hexadecimal notation, which encodes each 4-bit string as a digit 0–9 or a–f. Each byte is indexed according to the first and second 4-bits in the byte to be transformed.

ShiftRows Step

The ShiftRows step is a simple permutation, which has the effect of mixing up the bytes in each row of the 4×4 matrix output from the SubBytes step. The permutation amounts to a cyclical shift of each row of the 4×4 matrix so that the first row is shifted left by 0, the second is shifted left by 1, the third is shifted left by 2, and the fourth is shifted left by 3, as follows:

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{bmatrix}$$

$$= \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}.$$

MixColumns Step

The MixColumns Step mixes up the information in each column of the 4×4 matrix output from the ShiftRows step. It does this mixing by applying what amounts to a Hill-cipher matrix-multiplication transformation applied to each column, using the esoteric number system $GF(2^8)$, which was used to generate the S-box for the SubBytes step.

In the $GF(2^8)$ number system, the bits in a byte, $b_7b_6b_5b_4b_3b_2b_1b_0$, are interpreted to be the coefficients of the polynomial

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0,$$

where all the arithmetic used to evaluate this polynomial is modulo 2. In other words, this is a Boolean polynomial where the addition used to evaluate it is the same as the XOR operation and multiplication is the same as the AND operation. But these polynomials are not used here for the sake of evaluating them. Instead, in the $GF(2^8)$ number system, we are interested in operations performed on the underlying Boolean polynomials, not on their evaluations. For example, to add two such polynomials, we sum their respective matching coefficients, modulo 2:

$$\begin{aligned} & (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) \\ + \quad & (c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0) \\ = \quad & (b_7 + c_7)x^7 + (b_6 + c_6)x^6 + (b_5 + c_5)x^5 + (b_4 + c_4)x^4 \\ + \quad & (b_3 + c_3)x^3 + (b_2 + b_2)x^2 + (b_1 + b_1)x + (b_0 + c_0). \end{aligned}$$

In other words, to add two bytes, b and c, in the $GF(2^8)$ number system, we compute the exclusive-or $b \oplus c$.

The multiplication of two bytes, b and c, in the $GF(2^8)$ number system, amounts to a representation of the product of the two underlying polynomials for b and c respectively. We can't take this product without modification, however, as it would, in general, be a degree-14 Boolean polynomial, which would require more than 8 bits to represent. So we define this product to be modulo the special polynomial

$$x^8 + x^4 + x^3 + x + 1.$$

That is, to compute the product of two bytes, b and c, in $GF(2^8)$, we compute the Boolean polynomial for the product of the Boolean polynomials for b and c, and then determine the remainder polynomial that results from dividing the result by $x^8 + x^4 + x^3 + x + 1$, using a polynomial analogue of the long division algorithm we learned in grade school. As complicated as this seems, there is a method for multiplying two bytes b and c in $GF(2^8)$ that is surprisingly simple to program and is almost as fast to compute as regular integer multiplication. We omit the details of this multiplication algorithm here, however.

Given this interpretation of arithmetic as being done as described above in the number system $GF(2^8)$, the MixColumns step of the AES encryption algorithm is performed as follows:

$$\begin{bmatrix} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{bmatrix} \cdot \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}$$
$$= \begin{bmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{bmatrix}.$$

As in the Hill cipher, this operation is invertible in the $GF(2^8)$ number system. In fact, the inverse matrix to be used during the reverse MixColumns step for decryption is as follows:

$$\begin{bmatrix} 00001110 & 00001011 & 00001101 & 00001001 \\ 00001001 & 00001110 & 00001011 & 00001101 \\ 00001101 & 00001001 & 00001110 & 00001011 \\ 00001011 & 00001101 & 00001001 & 00001110 \end{bmatrix}.$$

AddRoundKey Step

In the AddRoundKey step, we exclusive-or the result from previous steps with a set of keys derived from the 128-bit secret key. The operation of the AddRoundKey step, therefore, can be expressed as follows:

$$\begin{bmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} = \begin{bmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{bmatrix}.$$

Of course, the critical part of performing this step is determining how the matrix of keys, $k_{i,j}$, for this round, are derived from the single 128-bit secret key, K.

AES Key Schedule

The key schedule for AES encryption is determined using a type of pseudo-random number generator. The first 4×4 key matrix, which is applied to the plaintext directly before any of the steps in round 1, is simple. It is just the secret key, K, divided into 16 bytes and arranged into a 4×4 matrix in a column-major ordering. For the sake of numbering, let us call this the round 0 key matrix, and let us refer to these columns as $W[0]$, $W[1]$, $W[2]$, and $W[3]$, so that the round 0 key matrix can be viewed as

$$\begin{bmatrix} W[0] & W[1] & W[2] & W[3] \end{bmatrix}.$$

Given this starting point, we determine the columns, $W[4i]$, $W[4i+1]$, $W[4i+2]$, and $W[4i+3]$, for the round i key matrix from the columns, $W[4i-4]$, $W[4i-3]$, $W[4i-2]$, and $W[4i-1]$, of the round $i-1$ key matrix.

The first column we compute, $W[4i]$, is special. It is computed as

$$W[4i] = W[4i-4] \oplus T_i(W[4i-1]),$$

where T_i is a special transformation that we will describe shortly. Given this first column, the other three columns are computed as follows, and in this order. (See Figure 8.15.)

$$\begin{aligned} W[4i+1] &= W[4i-3] \oplus W[4i] \\ W[4i+2] &= W[4i-2] \oplus W[4i+1] \\ W[4i+3] &= W[4i-1] \oplus W[4i+2]. \end{aligned}$$

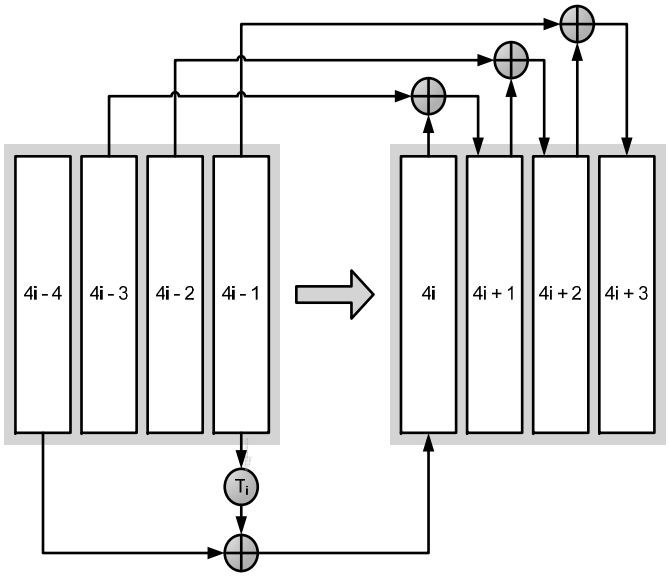


Figure 8.15: The key schedule for AES encryption.

T_i Transformation

The transformation, $T_i(W[4i-1])$, which is performed as a part of the computation of $W[4i]$, involves a number of elements. Let w_0, w_1, w_2, and w_3 denote the 4 bytes of $W[4i-1]$, in order. For each w_j, $j = 0, 1, 2, 3$, let $\mathcal{S}(w_i)$ denote the substitution transformation determined by the S-box used in the SubBytes step (see Figure 8.14) applied to w_j. In addition, let $R(i)$ denote an 8-bit ***round constant***, which is defined recursively, so that $R(1) = 00000001$, and, for $i \geq 2$,

$$R(i) = R(i-1) \cdot 00000010,$$

computed in the $GF(2^8)$. That is, $R(i)$ is an 8-bit representation of the Boolean polynomial

$$x^{i-1} \mod (x^8 + x^4 + x^3 + x + 1).$$

The round constant $R(i)$ is used in the computation of the key matrix for Round i. In hexadecimal, the first ten round constants are 01, 02, 04, 08, 10, 20, 40, 80, 1b, and 36, which are all that is needed for AES encryption with 128-bit keys. Given all these elements, the transformation $T_i(W[4i-1])$ is defined as follows:

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix} \rightarrow \begin{bmatrix} \mathcal{S}(w_1) \oplus R(i) \\ \mathcal{S}(w_2) \\ \mathcal{S}(w_3) \\ \mathcal{S}(w_0) \end{bmatrix}.$$

That is, to compute T_i on $W[4i-1]$, we do a cyclical left shift of the bytes in $W[4i-1]$, perform an S-box transformation of each shifted byte, and then exclusive-or the first byte with the round constant, $R(i)$. It is admittedly somewhat complicated, but each of these elements are relatively fast to perform in either software or hardware. Thus, since each step of the AES encryption involves these fast operations, and the number of rounds in the AES encryption algorithm is relatively small, the entire AES encryption algorithm can be performed relatively quickly. And, just as importantly, each step of the AES can be reversed, so as to allow for AES decryption. Moreover, this amounts to a symmetric encryption scheme, since we use the same key for both encryption and decryption.

8.5.2 Details for RSA

To understand the details of the RSA algorithm, we need to review some relevant facts of number theory.

Fermat's Little Theorem

We begin our number theory review with ***Fermat's Little Theorem*** .

Theorem 8.1: *Let p be a prime number and g be any positive integer less than p. Then*

$$g^{p-1} \bmod p = 1.$$

Proof: Because arithmetic in this case is done modulo p, this means that we are working in the number system Z_p. Moreover, since p is prime, every nonzero number less than p has a multiplicative inverse in Z_p. Therefore, if $ag \bmod p = bg \bmod p$, for $a, b \in Z_p$, then $a = b$. So the numbers $1g \bmod p$, $2g \bmod p$, $3g \bmod p$, ..., $(p-1)g \bmod p$ must all be distinct. That is, they are the numbers 1 through $p-1$ in some order. Thus,

$$(1g) \cdot (2g) \cdot (3g) \cdots ((p-1)g) \bmod p \;=\; 1 \cdot 2 \cdot 3 \cdots (p-1) \bmod p.$$

In other words,

$$(1 \cdot 2 \cdots (p-1))g^{p-1} \bmod p \;=\; (1 \cdot 2 \cdots (p-1)) \bmod p.$$

Therefore,

$$g^{p-1} \bmod p = 1.$$

■

Euler's Theorem

An important generalization to Fermat's Little Theorem is based on a function known as ***Euler's Totient Function***, $\phi(n)$. For any positive integer, n, the function $\phi(n)$ is equal to the number of positive integers that are relatively prime with n. Thus, for example, if p is prime, then $\phi(p) = p - 1$, and if n is the product of two primes, p and q, then $\phi(n) = (p-1)(q-1)$. The generalization to Fermat's Little Theorem is known as ***Euler's Theorem***, which is as follows.

Theorem 8.2: *Let x be any positive integer that is relatively prime to the integer $n > 0$, then*

$$x^{\phi(n)} \bmod n = 1.$$

Proof: The proof of Euler's Theorem is similar to that of Fermat's Little Theorem. Let Z_n^* denote the set of positive integers that are relatively prime to n, so that the number of integers in Z_n^* is $\phi(n)$. Also, note that each integer in Z_n^* has a multiplicative inverse in Z_n^*. So multiplying each member of Z_n^* modulo n by x will give all the members of Z_n^* back again in some order. Thus, the product of all the xi values, modulo n, for $i \in Z_n^*$ is the same as the product of the same i values. Therefore, cancelling out matching terms implies this theorem. ■

As a corollary of this fact, we have the following:

Corollary 8.3: *Let x be a positive integer relatively prime to n, and k be any positive integer. Then*

$$x^k \bmod n = x^{k \bmod \phi(n)} \bmod n.$$

Proof: Write $k = q\phi(n) + r$, so that $r = k \bmod \phi(n)$. Then

$$\begin{aligned} x^k \bmod n &= x^{q\phi(n)+r} \bmod n \\ &= x^{q\phi(n)} \cdot x^r \bmod n \\ &= (x^{q\phi(n)} \bmod n) \cdot (\cdot x^r \bmod n) \\ &= 1 \cdot (x^r \bmod n) \\ &= x^r \bmod n \end{aligned}$$

$x^{k \bmod \phi(n)} \bmod n.$

■

Euclid's GCD Algorithm

One of the key algorithms for dealing with the types of large numbers that are used in modern cryptography is one invented by the ancient Greek mathematician Euclid. In fact, it is quite remarkable that the cryptographic methods that allow for secure transactions on the Internet trace their roots to a time before algebra even existed. Nevertheless, we will take advantage of this more recent invention in describing how Euclid's algorithm works and how it can be used to facilitate arithmetic in Z_n.

Euclid's algorithm computes the ***greatest common divisor*** (***GCD***) of two numbers, a and b. That is, Euclid's algorithm computes the largest number, d, that divides both a and b (evenly with no remainder). The algorithm itself is remarkably simple, but before we can describe it in detail we need to discuss some background facts. The first fact is as follows.

Theorem 8.4: *The GCD d of two numbers, $a > 0$ and $b \geq 0$, is the smallest positive integer d such that*

$$d = ia + jb,$$

for integers i and j.

Proof: Let e be the GCD of a and b. We show that $d = e$ by first arguing why $d \geq e$ and then showing that $d \leq e$. Note first that, since e divides both a and b evenly, it divides d as well. That is,

$$d/e = (ia + jb)/e = i(a/e) + j(b/e),$$

which must be an integer. Thus, $d \geq e$.

Next, let $f = \lfloor a/d \rfloor$, and note that f satisfies the following:

$$\begin{aligned} a \bmod d &= a - fd \\ &= a - f(ia + jb) \\ &= (1 - fi)a + (-fj)b. \end{aligned}$$

In other words, the number, $a \bmod d$, can be written as the sum of a multiple of a and a multiple of b. But, by definition, $a \bmod d$ must be strictly less than d, which is the smallest positive integer that can be written as the sum of a multiple of a and a multiple of b. Thus, the only possibility is that $a \bmod d = 0$. That is, d is a divisor of a. Also, by a similar argument, $b \bmod d = 0$, which implies that d is also a divisor of b. Therefore, d is a common divisor of a and b; hence, $d \leq e$, since e is the greatest common divisor of a and b. ■

Note that an immediate consequence of this theorem is that the GCD of any number a and 0 is a itself. Given the theorem above and this little observation, we are ready to present Euclid's algorithm. We describe it so that it takes two integers a and b, with a being the larger, and returns a triple, (d, i, j), such that d is the GCD of a and b. The key idea behind Euclid's algorithm is that if d is the GCD of a and b, and $b > 0$, then d is also the GCD of b and the value, $a \bmod b$; hence, we can repeat this process to find the GCD of a and b. For example, consider the following illustration of this process:

$$\begin{aligned}
\text{GCD}(546, 198) &= \text{GCD}(198, 546 \bmod 198) = \text{GCD}(198, 150) \\
&= \text{GCD}(150, 198 \bmod 150) = \text{GCD}(150, 48) \\
&= \text{GCD}(48, 150 \bmod 48) = \text{GCD}(48, 6) \\
&= \text{GCD}(6, 48 \bmod 6) = \text{GCD}(6, 0) \\
&= 6.
\end{aligned}$$

Thus, the greatest common divisor of 546 and 198 is 6.

The Extended Euclidean Algorithm

To compute the GCD of a and b, we first test if b is zero, in which case the GCD of a and b is simply a; hence, we return the triple $(a, 1, 0)$ as the result of our algorithm. Otherwise, we recursively call the algorithm, getting the triple, (d, k, l), resulting from a call to our algorithm with arguments b and $a \bmod b$. Let us write $a = qb + r$, where $r = a \bmod b$. Thus,

$$\begin{aligned}
d &= kb + lr \\
&= kb + l(a - qb) \\
&= la + (k - lq)b.
\end{aligned}$$

Therefore, d is also the sum of a multiple of a and a multiple of b. So, in this case, we return the triple $(d, l, k - lq)$. This algorithm, which is known as the ***extended Euclidean algorithm***, is shown in Figure 8.16.

Algorithm GCD(a, b):
  if b = 0 **then** {we assume $a > b$}
    return $(a, 1, 0)$
  Let $q = \lfloor a/b \rfloor$
  Let $(d, k, l) =$ GCD(b, $a \bmod b$)
  return $(d, l, k - lq)$

Figure 8.16: The extended Euclidean algorithm.

Let us argue why this algorithm is correct. Certainly, if $b = 0$, then a is the GCD of a and b; hence, the triple returned by extended Euclid's algorithm is correct. Suppose, for the sake of an inductive argument, that the recursive call, GCD(a, $a \bmod b$), returns the correct value, d, as the GCD of a and $a \bmod b$. Let e denote the GCD of a and b. We have already argued how d can be written as the sum of a multiple of a and a multiple of b. So if we can show that $d = e$, then we know that the triple returned by the algorithm is correct. First, let us write $a = qb + r$, where $r = a \bmod b$, and note that

$$(a - qb)/e = (a/e) - q(b/e),$$

which must be an integer. Thus, e is a common divisor of a and $r = a - qb = a \bmod b$. Therefore, $e \leq d$. Next, note that, by definition, d divides b and $a - qb$. That is, the following is an integer:

$$(a - qb)/d = (a/d) - q(b/d).$$

Moreover, since b/d must be an integer, this implies that a/d is an integer. Thus, d is a divisor of both a and b. Therefore, $d \leq e$. That is, d is the greatest common divisor of a and b.

Modular Multiplicative Inverses

As it turns out, computing the GCD of pairs of integers is not the main use of the extended Euclidean algorithm. Instead, its main use is for computing modular multiplicative inverses.

Suppose we have a number, $x < n$, and we are interested in computing a number, y, such that

$$yx \bmod n = 1,$$

provided such a number exists. In this case, we say that y is the ***multiplicative inverse*** of x in Z_n, and we write $y = x^{-1}$ to indicate this relationship. To compute the value of y, we call the extended Euclidean algorithm to compute the GCD of x and n. The best case is when x and n are ***relatively prime***, that is, their greatest common divisor is 1. For when x and n are relatively prime, then the multiplicative inverse of x in Z_n exists. In this case, calling the extended Euclidean algorithm to compute GCD(n, x) returns the triple $(1, i, j)$, such that

$$1 = ix + jn.$$

Thus,

$$(ix + jn) \bmod n = ix \bmod n = 1.$$

Therefore, i is the multiplicative inverse, x^{-1}, in Z_n, in this case. Moreover, if our call to the extended Euclidean algorithm to compute GCD(n, x)

returns a greatest common divisor greater than 1, then we know that the multiplicative inverse of x does not exist in Z_n.

The Efficiency of the Extended Euclidian Algorithm

The other nice thing about the extended Euclidean algorithm is that it is relatively fast. It is easy to show that every two consecutive recursive calls made during the extended Euclidean algorithm will halve the magnitude of the first argument (recall, for instance, the example we gave of the process that forms the basis of Euclid's algorithm). Thus, the running time of the extended Euclidean algorithm is proportional to

$$\lceil \log a \rceil,$$

which is equal to the number of bits needed to represent a. Therefore, the extended Euclidean algorithm runs in linear time with respect to the size of its input; hence, computing multiplicative inverses in Z_n can be done in linear time.

Modular Exponentiation

Another important computational tool used in modern cryptography is ***modular exponentiation***. In this instance, we are given three positive integers, g, n, and p, which are represented in binary, and we want to compute

$$g^n \bmod p.$$

Of course, one way to calculate this value is to initialize a running product, q, to 1, and iteratively multiply q with g, modulo p, for n iterations. Such a straightforward algorithm would clearly use n modular multiplications. Unfortunately, if n is relatively large, then this is an expensive way to do modular exponentiation. Indeed, since the number n is represented in binary using

$$\lceil \log n \rceil$$

bits, this straightforward way of performing modular exponentiation requires a number of multiplications that are exponential in the input size. This algorithm is therefore way too slow for practical use in cryptographic computations.

Repeated Squaring

Fortunately, there is a better algorithm that runs much faster. The main purpose of this algorithm is to compute g^n using ***repeated squaring***. That is, using multiplications modulo p, we compute g, $g^2 = g \cdot g$, $g^4 = g^2 \cdot g^2$, $g^8 = g^4 \cdot g^4$, and so on. This approach allows us to iteratively build up powers of g with exponents that are powers of 2. Then, given the binary representation of a number, n, we can compute g^n from these powers of g based on this binary representation. For example, we could compute g^{25} as

$$g^{25} = g^{16+8+1} = g^{16} \cdot g^8 \cdot g^1,$$

since $25 = 11,000$ in binary. Or we could compute g^{46} as

$$= g^{32+8+4+2} = g^{32} \cdot g^8 \cdot g^4 \cdot g^2,$$

since $46 = 101,000$ in binary. We give a pseudo-code description of the repeated squaring algorithm in Figure 8.17.

Algorithm ModularExponentiation(g, n, p):
  $q = 1$ {The running product}
  $m = n$ {A copy of n that is destroyed during the algorithm}
  $s = g$ {The current square}
  while $m \geq 1$ **do**
    if m is odd **then**
      $q = q \cdot s \bmod p$
    $s = s \cdot s \bmod p$ {Compute the next square}
    $m = \lfloor m/2 \rfloor$ {This can be done by a right shift}

Figure 8.17: The repeated squaring algorithm for computing $g^n \bmod p$.

Note that this algorithm uses a number of multiplications that are proportional to the number of bits used to represent n. Thus, this algorithm uses a linear number of multiplications, which is clearly much better than an exponential number. The take-away message, therefore, is that modular exponentiation is a tool that can be used effectively in modern cryptography. It is not as fast as a single multiplication or even symmetric encryption methods though, so we should try not to overuse modular exponentiation when other faster methods are available.

Primality Testing

Yet another important computation that is often used in modern cryptography is ***primality testing***. In this instance, we are given a positive integer, n, and we want to determine if n is prime or not. That is, we want to determine if the only factors of n are 1 and n itself. Fortunately, there are efficient methods for performing such tests. Even so, the details of these methods are fairly complicated; hence, they are beyond the scope of this book.

One thing we mention, however, is that none of these methods actually factor n. They just indicate whether n is prime or not. Moreover, the fact that no primality testing algorithm actually factors n has given rise to a general belief in cryptographic circles that the problem of factoring a large number, n, is computationally difficult. Indeed, there are several cryptographic methods, including the RSA cryptosystem we discuss in the next section, whose security is based on the difficulty of factoring large numbers.

Given an efficient way of performing primality testing, actually generating a random prime number is relatively easy. This simplicity is due to an important fact about numbers, which is that the number of prime numbers between 1 and any number n is at least $n/\ln n$, for $n \geq 4$, which is a property derived from the Prime Number Theorem, whose exact statement and proof are beyond the scope of this book. In any case, simply knowing that the number of primes between 1 and n is at least $n/\ln n$ is sufficient for cryptographic purposes, because it means that if we generate a random odd number q between $n/2$ and n, then q will be prime with probability at least $1/\ln n$. Thus, if we repeat this process a logarithmic number of times, testing each number generated for primality, then one of our generated numbers is expected be prime.

How RSA is Typically Used

Even with an efficient implementation, the RSA cryptosystem is orders-of-magnitude slower than the AES symmetric cryptosystem (Section 8.1.6). Thus, a standard approach to encryption is as follows:

1. Encrypt a secret key, K, with the RSA cryptosystem for the AES symmetric cryptosystem.
2. Encrypt with AES using key K.
3. Transmit the RSA-encrypted key together with the AES-encrypted document.

The above method illustrates a common use of public-key cryptography in conjunction with a symmetric cryptosystem.

8.6 Exercises

For help with exercises, please visit **securitybook.net**.

Reinforcement

R-8.1 Eve has tricked Alice into decrypting a bunch of ciphertexts that Alice encrypted last month but forgot about. What type of attack is Eve employing?

R-8.2 Eve has an antenna that can pick up Alice's encrypted cell phone conversations. What type of attack is Eve employing?

R-8.3 Eve has given a bunch of messages to Alice for her to sign using the RSA signature scheme, which Alice does without looking at the messages and without using a one-way hash function. In fact, these messages are ciphertexts that Eve constructed to help her figure out Alice's RSA private key. What kind of attack is Eve using here?

R-8.4 Eve has bet Bob that she can figure out the AES secret key he shares with Alice if he will simply encrypt 20 messages for Eve using that key. For some unknown reason, Bob agrees. Eve gives him 20 messages, which he then encrypts and emails back to Eve. What kind of attack is Eve using here?

R-8.5 What is the encryption of the following string using the Caesar cipher: THELAZYFOX.

R-8.6 What are the substitutions for the (decimal) numbers 12, 7, and 2 using the S-box from Figure 8.3?

R-8.7 What are the next three numbers in the pseudo-random number generator $3x_i + 2 \bmod 11$, starting from 5?

R-8.8 What is the Hill cipher that corresponds to the permutation cipher

$$\pi : (1,2,3,4,5,6,7,8) \rightarrow (2,6,8,1,3,7,5,4)?$$

R-8.9 In the inverse of the S-box from Figure 8.14, what is the substitution for `e3`, in hexadecimal?

R-8.10 What would be the transformation done by three consecutive applications of the ShiftRows step in the AES encryption algorithm?

R-8.11 How many keys can be used with each of the three key lengths for the AES cryptosystem.

R-8.12 Bob is arguing that if you use Electronic Codebook (ECB) mode twice in a row to encrypt a long message, M, using the same key each time, that it will be more secure. Explain why Bob is wrong in the case of using a binary one-time pad encryption scheme.

R-8.13 Show the steps and intermediate results of applying the extended Euclidean algorithm to compute the GCD of 412 and 200.

R-8.14 Compute the multiplicative inverse of 5 in Z_{21}.

R-8.15 What is 7^{16} mod 11?

R-8.16 Roughly how many times would you have to call a primality tester to find a prime number between 1,000,000 and 2,000,000?

R-8.17 What is 7^{120} mod 143?

R-8.18 Show the result of encrypting $M = 4$ using the public key $(e, n) = (3, 77)$ in the RSA cryptosystem.

R-8.19 Why can't Bob use the pair $(1, n)$ as an RSA public key, even if $n = pq$, for two large primes, p and q?

R-8.20 Alice is telling Bob that he should use a pair of the form $(3, n)$ or $(16385, n)$ as his RSA public key, where, as usual, $n = pq$, for two large primes, p and q, if he wants people to encrypt messages for him from their cell phones. What is the justification for Alice's advice?

R-8.21 Show the result of an Elgamal encryption of the message $M = 8$ using $k = 4$ for the public key $(p, g, y) = (59, 2, 25)$.

R-8.22 Demonstrate that the hash function

$$H(x) = 5x + 11 \bmod 19$$

is not weakly collision resistant, for $H(4)$, by showing how easy it is to find such a collision.

R-8.23 Demonstrate that the hash function

$$H(x) = 5x + 11 \bmod 23$$

is not strongly collision resistant, by showing how easy it is to find such a collision.

R-8.24 Explain why nonforgeability and nonmutability imply nonrepudiation for digital signatures.

R-8.25 Explain the strengths and weaknesses of using symmetric encryption, like AES, versus a public-key cryptosystem, like RSA.

R-8.26 Name two things that the RSA and ElGamal cryptosystems have in common, other than the fact that they are both public-key cryptosystems?

Creativity

C-8.1 What is the plaintext for the following ciphertext, which was encrypted using a simple substitution cipher:
CJBT COZ NPON ZJV FTTK TWRTUYTFGT NJ DTN O XJL. Y COZ ZJV CPJVIK DTN O XJL MYUCN.

C-8.2 ROT13 is a cyclic shift cipher that substitutes each English letter with one that is 13 away in the alphabet. It is used today not for security, but as a simple obfuscation device, because the same algorithm is used for both encryption and decryption. People wishing to encrypt or decrypt a message, M (such as a spoiler paragraph in a movie review), just cut-and-paste M to a ROT13 converter and click a button "APPLY" to do the encryption or decryption. Give an example of another ROTi transformation that could be used for both encryption and decryption in a similar way.

C-8.3 In a special case of a permutation cipher, we take a message, M, and write its letters in an $s \times t$ table, in a row-major fashion, and then let the ciphertext be a column-major listing of the the entries in the table. For example, to encrypt the message `ATTACKATDAWN`, using a 3×4 table, we would write the message as
`ATTA`
`CKAT`
`DAWN`
and then write down the ciphertext as `ACDTKATAWATN`. The secret key in this cryptosystem is the pair (s, t). How is decryption done in this cryptosystem? Also, how hard would it be to attack this cryptosystem using a ciphertext-only attack?

C-8.4 How many valid English plaintexts are there for the ciphertext message CJU using a length-3, one-time pad of cyclic shifts, (i, j, k)?

C-8.5 Alice is using a linear congruential generator, $ax_i + b \bmod 13$, to generate pseudo-random numbers. Eve sees three numbers in a row, 7, 6, 4, that are generated from Alice's function. What are the values of a and b?

C-8.6 Bob is arguing that if you use output feedback (OFB) mode twice in a row to encrypt a long message, M, using the same key each time, it will be more secure. Explain why Bob is wrong, no matter what encryption algorithm he is using for block encryption.

C-8.7 Why can't Bob use the pair $(6, n)$ as an RSA public key, where $n = pq$, for two large primes, p and q?

C-8.8 Use Euler's Theorem, not repeated squaring, to compute

$$20^{10203} \bmod 10403.$$

Show your work.

C-8.9 Suppose we use the AES algorithm with a fixed key, K, to implement a cryptographic hash function. That is, we define

$$H(M) = AES_K(M).$$

Argue why this algorithm is likely to be weakly collision resistant.

C-8.10 Alice wants to send Bob a message, M, that is the price she is willing to pay for his used car (M is just an integer in binary). She uses the RSA algorithm to encrypt M into the ciphertext, C, using Bob's public key, so only he can decrypt it. But Eve has intercepted C and she also knows Bob's public key. Explain how Eve can alter the ciphertext C to change it into C_0 so that if she sends C_0 to Bob (with Eve pretending to be Alice), then, after Bob has decrypted C_0, he will get a plaintext message that is twice the value of M.

C-8.11 An Internet game show has asked if Alice is willing to commit today to whether she will marry Bob, who is either an ex-con with a dragon tattoo on his face or a former male model who just won the tristate lottery. Next week, the real identity of Bob will be revealed, at which time Alice must also reveal her answer (which she has already committed to). Explain a secure and confidential way that Alice can commit to her answer now that prevents her from forging her response next week when she learns who Bob really is.

C-8.12 Suppose the primes p and q used in the RSA algorithm to define $n = pq$ are in the range $[\sqrt{n} - 100, \sqrt{n} + 100]$. Explain how you can efficiently factor n using this information. Also, explain how this knowledge breaks the security of the RSA encryption algorithm.

C-8.13 Bob is stationed as a spy in Cyberia for a week and wants to prove that he is alive every day of this week and has not been captured. He has chosen a secret random number, x, which he memorized and told to no one. But he did tell his boss the value $y = H(H(H(H(H(H(H(x)))))))$, where H is a one-way cryptographic hash function. Unfortunately, he knows that the Cyberian Intelligence Agency (CIA) was able to listen in on that message; hence, they also know the value of y. Explain how he can send a single message every day that proves he is still alive and has not been captured. Your solution should not allow anyone to replay any previous message from Bob as a (false) proof he is still alive.

C-8.14 Bob has modulus n and exponent e as his RSA public key, (e, n). He has told Eve that she can send him any message $M < n$ and he is willing to sign it using a simple RSA signature method to compute $S = M^d \bmod n$, where d is his private RSA exponent, and he will return the signature S to Eve. Unfortunately for Bob, Eve has captured a ciphertext C that Alice encrypted for Bob from her plaintext P using his RSA public key. (Bob never actually got C.) Eve wants to trick Bob into decrypting C for her and she doesn't want Bob to see the original plaintext P that goes with C. So Eve asks Bob to sign the message $M = r^e C \bmod n$ using his private RSA exponent, and send her back the signature S for M, where r is a random number that Eve chose to be relatively prime to n. Explain how Eve can use Bob's signature, S, on M, to discover the plaintext, P, for C.

C-8.15 Let p be a prime. Give an efficient alternative algorithm for computing the multiplicative inverse of an element of Z_p that is not based on the extended Euclidean algorithm. (Hint: Use Fermat's Little Theorem.)

Projects

P-8.1 Write a program that can implement arbitrary substitution ciphers. The substitution should be specified by a conversion table for letters, which should be the same for both uppercase and lowercase letters.

P-8.2 Write a program that can perform AES encryption and decryption.

P-8.3 Write a program that can implement RSA setup, encryption, and decryption.

P-8.4 Write a program that can implement ElGamal setup, encryption, and decryption.

P-8.5 Write a program that can implement ElGamal digital signatures.

Chapter Notes

A more detailed coverage of cryptography can be found in the books by Ferguson, Schneier and Konho [30], Menezes *et al.* [58], Stinson [97], and Trappe and Washington [103]. Simon Singh gives a historical perspective on cryptography [95] in his best-selling title "The Code Book". The Hill cipher was published in 1929 by Lester Hill [39]. The first known description of the one-time pad algorithm was given in a patent issued in 1919 to Gilbert Vernam [106]. This cryptosystem was proven secure in 1949 by Claude Shannon [92]. Declassified details about the Venona Project can be found at a web site [66] of the U.S. National Security Agency (NSA). Additional details about AES, the Advanced Encryption Standard, are contained in a book by its designers, Daemen and Rijmen [22]. The concept of public-key cryptography is credited (in unclassified circles) to Diffie and Hellman [26]. The RSA public-key cryptosystem and digital signature scheme were discovered by Rivest, Shamir, and Adleman [82]. The Elgamal cryptosystem and signature scheme are due to Taher Elgamal [28]. Additional details about cryptographic hash functions can be found in a survey by Preneel [77]. The Merkle-Damgård construction is described in [23]. Chosen-prefix collisions attacks on the MD5 hash function wre found by Stevens, Lenstra and de Weger [53].