CHAPTER 5

Cryptography

ZHQM ZMGM ZMFM
—G. JULIUS CAESAR

XYAWO GAOOA GPEMO HPQCW IPNLG RPIXL TXLOA NNYCS YXBOY MNBIN YOBTY QYNAI
—JOHN F. KENNEDY

5.1 Introduction

Cryptography is where security engineering meets mathematics. It provides us with the tools that underlie most modern security protocols. It is probably the key enabling technology for protecting distributed systems, yet it is surprisingly hard to do right. As we've already seen in Chapter 2, "Protocols," cryptography has often been used to protect the wrong things, or used to protect them in the wrong way. We'll see plenty more examples when we start looking in detail at real applications.

Unfortunately, the computer security and cryptology communities have drifted apart over the last 20 years. Security people don't always understand the available crypto tools, and crypto people don't always understand the real-world problems. There are a number of reasons for this, such as different professional backgrounds (computer science versus mathematics) and different research funding (governments have tried to promote computer security research while suppressing cryptography). It reminds me of a story told by a medical friend. While she was young, she worked for a few years in a country where, for economic reasons, they'd shortened their medical degrees and concentrated on producing specialists as quickly as possible. One day, a patient who'd had both kidneys removed and was awaiting a transplant needed her dialysis shunt redone. The surgeon sent the patient back from the theater on the grounds that there was no urinalysis on file. It just didn't occur to him that a patient with no kidneys couldn't produce any urine.

Just as a doctor needs to understand physiology as well as surgery, so a security engineer needs to be familiar with cryptology as well as computer security (and much else). This chapter is aimed at people without a training in cryptology; cryptologists will find little in it which they don't already know. As I only have a few dozen pages, and a proper exposition of modern cryptography would run into thousands, I won't go into much of the mathematics (there are plenty books that do that; see the end of the chapter for further reading). I'll just explain the basic intuitions and constructions that seem to cause the most confusion. If you have to use cryptography in anything resembling a novel way, then I strongly recommend that you read a lot more about it.

Computer security people often ask for non-mathematical definitions of cryptographic terms. The basic terminology is that *cryptography* refers to the science and art of designing ciphers; *cryptanalysis* to the science and art of breaking them; while *cryptology*, often shortened to just *crypto*, is the study of both. The input to an encryption process is commonly called the *plaintext*, and the output the *ciphertext.* Thereafter, things get somewhat more complicated. There are a number of *cryptographic primitives*—basic building blocks, such as *block ciphers, stream ciphers*, and *hash functions.* Block ciphers may either have one key for both encryption and decryption, in which case they're called *shared key* (also *secret key* or *symmetric*), or have separate keys for encryption and decryption, in which case they're called *public key* or *asymmetric.* A *digital signature scheme* is a special type of asymmetric crypto primitive.

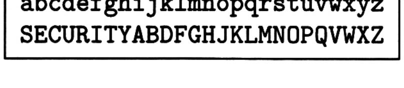
In the rest of this chapter, I will first give some simple historical examples to illustrate the basic concepts. I'll then try to fine-tune definitions by introducing the *random oracle model*, which many cryptologists use. Finally, I'll show how some of the more important cryptographic algorithms actually work, and how they can be used to protect data.

5.2 Historical Background

Suetonius tells us that Julius Caesar enciphered his dispatches by writing D for A, E for B and so on [742]. When Augustus Caesar ascended the throne, he changed the imperial cipher system so that C was now written for A, D for B, and so on. In modern terminology, we would say that he changed the key from *D* to *C.*

The Arabs generalized this idea to the monoalphabetic substitution, in which a keyword is used to permute the cipher alphabet. We will write the plaintext in lowercase letters, and the ciphertext in uppercase, as shown in Figure 5.1.

CYAN RWSGKFR AN AH RHTFANY MSOYRM OYSH SMSEAC NCMAKO; but breaking ciphers of this kind is a straightforward pencil and paper puzzle, which you may have done in primary school. The trick is that some letters, and combinations of letters, are much more common than others; in English the most common letters are e, t, a, i, o, n, s, h, r, d, l, u in that order. Artificial intelligence researchers have shown some interest in writing programs to solve monoalphabetic substitutions; using letter and digraph (letter-pair) frequencies alone. They typically succeed with about 600 letters of ciphertext, while-smarter strategies, such as guessing probable words, can cut this to about 150 letters. A human cryptanalyst will usually require much less.



```
abcdefghijklmnopqrstuvwxyz
SECURITYABDFGHJKLMNOPQVWXZ
```

Figure 5.1 Monoalphabetic substitution cipher.

There are basically two ways to make a stronger cipher: the *stream cipher* and the *block cipher*. In the former, you make the encryption rule depend on a plaintext symbol's position in the stream of plaintext symbols, while in the latter you encrypt several plaintext symbols at once in a block. Let's look at early examples.

5.2.1 An Early Stream Cipher: The Vigenère

An early stream cipher is commonly ascribed to the Frenchman Blaise de Vigenère, a diplomat who served King Charles IX. It works by adding a key repeatedly into the plaintext using the convention that A = 0, B = 1, . . . , Z = 25; and addition is carried out modulo 26—that is, if the result is greater than 25, we subtract as many multiples of 26 as are needed to bring us into the range [0, . . . ,25], that is, [A, . . . ,Z]. Mathematicians write this as:

$$C = P + K \bmod 26$$

For example, when we add P(15) to U(20) we get 35, which we reduce to 9 by subtracting 26; 9 corresponds to J, so the encryption of P under the key U (and of U under the key P) is J. In this notation, Julius Caesar's system used a fixed key, $K = D$ (modulo 23, as the alphabet Caesar used wrote U as V, J as I, and had no W), while Augustus Caesar's used $K = C$, and Vigenère used a repeating key, also known as a *running key*. Various means were developed to do this addition quickly, including printed tables and, for field use, cipher wheels. Whatever the implementation technology, the encryption using a repeated keyword for the key would look as shown in Figure 5.2.

A number of people appear to have worked out how to solve polyalphabetic ciphers, from the notorious womanizer Casanova to computing pioneer Charles Babbage. However, the first published solution was in 1863 by Friedrich Kasiski, a Prussian infantry officer [441]. He noticed that given a long enough piece of ciphertext, repeated patterns will appear at multiples of the keyword length.

In Figure 5.2, for example, we see "KIOV" repeated after nine letters, and "NU" after six. Since three divides both six and nine, we might guess a keyword of three letters. It follows that ciphertext letters one, four, seven, and so on all enciphered under the same keyletter; so we can use frequency analysis techniques to guess the most likely values of this letter, then repeat the process for the second and third letters of the key.

Plain:	`tobeornottobethatisthequestion`
Key:	`runrunrunrunrunrunrunrunrunrun`
Cipher:	`KIOVIEEIGKIOVNURNVJNUVKHVMGZIA`

Figure 5.2 A Vigenère polyalphabetic substitution cipher.

5.2.2 The One-Time Pad

One way to make a stream cipher of this type proof against attacks is for the key sequence to be as long as the plaintext, and to never repeat. This was proposed by Gilbert Vernam during World War I [428]; its effect is that given any ciphertext, and any plaintext of the same length, there is a key that decrypts the ciphertext to the plaintext. Regardless of the amount of computation that opponents can do, they are none the wiser, as all possible plaintexts are just as likely. This system is known as the *one-time pad.* Leo Marks' engaging book on cryptography in the Special Operations Executive in World War II [523] relates how one-time key material was printed on silk, which agents could conceal inside their clothing; whenever a key had been used, it was torn off and burned.

An example should explain all this. Suppose you had intercepted a message from a wartime German agent, which you knew started with "Heil Hitler," and that the first 10 letters of ciphertext were DGTYI BWPJA. This means that the first 10 letters of the one-time pad were wclnb tdefj, as shown in Figure 5.3.

Once he had burned the piece of silk with his key material, the spy could claim that he was actually a member of the anti-Nazi underground resistance, and that the message actually said "Hang Hitler." This is quite possible, as the key material could just as easily have been wggsb tdefj, as shown in Figure 5.4.

Now, we rarely get anything for nothing in cryptology, and the price of the perfect secrecy of the one-time pad is that it fails completely to protect message integrity. Suppose that you wanted to get this spy into trouble; you could change the ciphertext to DCYTI BWPJA, as shown in Figure 5.5.

During the World War II, Claude Shannon proved that a cipher has perfect secrecy if and only if there are as many possible keys as possible plaintexts, and if every key is equally likely; therefore, the one-time pad is the only kind of system that offers perfect secrecy [694, 695].

Plain:	**heilhitler**
Key:	**wclnbtdefj**
Cipher:	**DGTYIBWPJA**

Figure 5.3 A spy's message.

Cipher:	**DGTYIBWPJA**
Key:	**wggsbtdefj**
Plain:	**hanghitler**

Figure 5.4 What the spy claimed he said.

Cipher:	**DCYTIBWPJA**
Key:	**wclnbtdefj**
Plain:	**hanghitler**

Figure 5.5 Manipulating the message in Figure 5.3 to entrap the spy.

The one-time pad is still used for high-level diplomatic and intelligence traffic, but it consumes as much key material as there is traffic, hence is too expensive for most applications. It's more common for stream ciphers to use a suitable pseudorandom number generator to expand a short key into a long keystream. The data is then encrypted by exclusive-or'ing the keystream, one bit at a time, with the data. It's not enough for the keystream to appear "random" in the sense of passing the standard series randomness tests; it also must have the property that an opponent who gets their hands on even a number of keystream bits should not be able to predict any more of them. I'll formalize this more tightly in the next section.

Stream ciphers are commonly used nowadays in hardware applications where the number of gates has to be minimized to save power. We'll look at some actual designs in later chapters, including the A5 algorithm used to encipher GSM mobile phone traffic (in Chapter 17, "Telecom System Security"), and the multiplex shift register system used in pay-per-view TV (in Chapter 20, "Copyright and Privacy Protection"). However, block ciphers are more suited for many applications where encryption is done in software, so let's look at them next.

5.2.3 An Early Block Cipher: Playfair

One of the best-known early block ciphers is the Playfair system. It was invented in 1854 by Sir Charles Wheatstone, a telegraph pioneer who also invented the concertina and the Wheatstone bridge. The reason it's not called the Wheatstone cipher is that he demonstrated it to Baron Playfair, a politician; Playfair in turn demonstrated it to Prince Albert and to Lord Palmerston (later Prime Minister) on a napkin after dinner.

This cipher uses a 5 by 5 grid, in which the alphabet is placed, permuted by the keyword, and omitting the letter J (see Figure 5.6).

The plaintext is first conditioned by replacing J with I wherever it occurs, then dividing it into letter pairs, preventing double letters occurring in a pair by separating them with an x, and finally adding a z if necessary to complete the last letter pair. The example Playfair wrote on his napkin was "Lord Granville's letter," which becomes `"lo rd gr an vi lx le sl et te rz"`.

It is then enciphered two letters at a time using the following rules:

- If two letters are in the same row or column, they are replaced by the succeeding letters. For example, "`am`" enciphers to "`LE`."
- Otherwise, the two letters stand at two of the corners of a rectangle in the table, and we replace them with the letters at the other two corners of this rectangle. For example, "`lo`" enciphers to "`MT`."

P	A	L	M	E
R	S	T	O	N
B	C	D	F	G
H	I	K	Q	U
V	W	X	Y	Z

Figure 5.6 The Playfair enciphering tableau.

Plain:	lo	rd	gr	an	vi	lx	le	sl	et	te	rz
Cipher:	MT	TB	BN	ES	WH	TL	MP	TA	LN	NL	NV

Figure 5.7 Example of Playfair enciphering.

We can now encipher our specimen text as shown in Figure 5.7.

Variants of this cipher were used by the British army as a field cipher in World War I, and by the Americans and Germans in World War II. It's a substantial improvement on Vigenère, as the statistics an analyst can collect are of *digraphs* (letter pairs) rather than single letters, so the distribution is much flatter, and more ciphertext is needed for an attack.

Again, it's not enough for the output of a block cipher to just look intuitively "random." Playfair ciphertexts do look random, but they have the property that if you change a single letter of a plaintext pair, then often only a single letter of the ciphertext will change. Thus, using the key in Figure 5.7, `rd` enciphers to `TB` while `rf` enciphers to `OB` and `rg` enciphers to `NB`. One consequence is that, given enough ciphertext or a few probable words, the table (or an equivalent one) can be reconstructed [326]. We will want the effects of small changes in a block cipher's input to diffuse completely through its output: changing one input bit should, on average, cause half of the output bits to change. I'll tighten these ideas up in the next section.

The security of a block cipher can be greatly improved by choosing a longer block length than two characters. For example, the *Data Encryption Standard* (DES), which is widely used in banking, has a block length of 64 bits, which equates to eight ASCII characters and the Advanced Encryption Standard (AES), which is replacing it in many applications, has a block length of twice this. I discuss the internal details of DES and AES below; for the time being, I'll just remark that an eight byte or sixteen byte block size is not enough of itself. For example, if a bank account number always appears at the same place in a transaction format, then it's likely to produce the same ciphertext every time a transaction involving it is encrypted with the same key. This could allow an opponent who can eavesdrop on the line to monitor a customer's transaction pattern; it might also be exploited by an opponent to cut and paste parts of a ciphertext in order to produce a seemingly genuine but unauthorized transaction. Unless the block is as large as the message, the ciphertext will contain more than one block, and we will look later at ways of binding them together.

5.2.4 One-Way Functions

The third classical type of cipher is the *one-way function.* This evolved to protect the integrity and authenticity of messages, which as we've seen is not protected at all by many simple ciphers, where it is often easy to manipulate the ciphertext in such a way as to cause a predictable change in the plaintext.

After the invention of the telegraph in the mid-nineteenth century, banks rapidly became its main users, and developed systems for transferring money electronically. Of course, it isn't the money itself that is "wired," but a payment instruction, such as:

> *To Lombard Bank, London. Please pay from our account with you no. 1234567890 the sum of £1000 to John Smith of 456 Chesterton Road, who has an account with HSBC Bank Cambridge no. 301234 4567890123, and notify him that this was for "wedding*

present from Doreen Smith." From First Cowboy Bank of Santa Barbara, CA, USA. Charges to be paid by us.

Since telegraph messages were relayed from one office to another by human operators, it was possible for an operator to manipulate a payment message.

Banks, telegraph companies, and shipping companies developed *code books*, which not only could protect transactions, but also shorten them—which was very important given the costs of international telegrams at the time. A code book was essentially a block cipher that mapped words or phrases to fixed-length groups of letters or numbers. Thus, "Please pay from our account with you no" might become "AFVCT." A competing technology was *rotor machines*, mechanical cipher devices that produce a very long sequence of pseudorandom numbers, and combine them with plaintext to get ciphertext; these were independently invented by a number of people, many of whom dreamed of making a fortune selling them to the banking industry. Banks weren't in general interested, but rotor machines became the main high-level ciphers used by the combatants in World War II.

The banks realized that neither mechanical stream ciphers nor code books protected message authenticity. If, for example, the codeword for 1000 is mauve and for 1,000,000 is magenta, then the crooked telegraph clerk who can compare the coded traffic with known transactions should be able to figure this out and substitute one for the other.

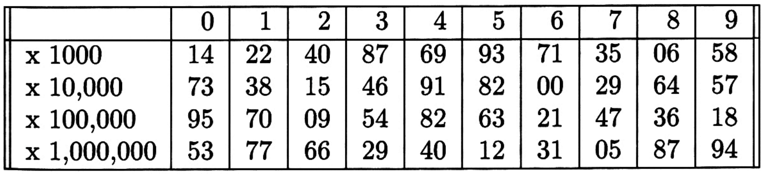
The critical innovation was to use a code book, but make the coding one-way by adding the code groups together into a number called a *test key.* (Modern cryptographers would describe it as a *hash value* or *message authentication code*, terms I'll define more carefully later.)

Here is a simple example. Suppose that the bank has a code book with a table of numbers corresponding to payment amounts, as in Figure 5.8. In order to authenticate a transaction for $376,514, we add 53 (no millions), 54 (300,000), 29 (70,000) and 71 (6,000). (It's common to ignore the less significant digits of the amount.) This gives us a test key of 207.

Most real systems were more complex than this; they usually had tables for currency codes, dates, and even recipient account numbers. In the better systems, the code groups were four digits long rather than two; and to make it harder for an attacker to reconstruct the tables, the test keys were compressed: a key of 7549 might become 23 by adding the first and second digits, and the third and fourth digits, and ignoring the carry.

Test keys are not strong by the standards of modern cryptography. Given somewhere between a few dozen and a few hundred tested messages, depending on the design details, a patient analyst could reconstruct enough of the tables to forge a transaction. With a few carefully chosen messages inserted into the banking system by an accomplice, it's even easier still. But the banks got away with it: test keys worked fine from the late nineteenth century through the 1980s. In several years working as a bank security consultant, and listening to elderly bank auditors' tales over lunch, I only heard of two cases of fraud that exploited it: one external attempt involving cryptanalysis, which failed because the attacker didn't understand bank procedures, and one successful but small fraud involving a crooked staff member. I'll explain the systems that replaced test keys, and cover the whole issue of how to tie cryptographic authentication mechanisms to procedural protection such as dual control, in Chapter 9, "Banking and

Bookkeeping." For now, test keys are the classic example of a one-way function used for authentication.



	0	1	2	3	4	5	6	7	8	9
x 1000	14	22	40	87	69	93	71	35	06	58
x 10,000	73	38	15	46	91	82	00	29	64	57
x 100,000	95	70	09	54	82	63	21	47	36	18
x 1,000,000	53	77	66	29	40	12	31	05	87	94

Figure 5.8 A simple test key system.

Later examples included functions for applications discussed in the previous chapters, such as storing passwords in a one-way encrypted password file, and computing a response from a challenge in an authentication protocol.

5.2.5 Asymmetric Primitives

Finally, some modern cryptosystems are asymmetric, in that different keys are used for encryption and decryption. For example, I publish on my Web page a *public key* with which people can encrypt messages to send to me; I can then decrypt them using the corresponding *private key.*

There are some precomputer examples of this too; perhaps the best is the postal service. You can send me a private message simply by addressing it to me and dropping it into a post box. Once that's done, I should be the only person who'll be able to read it. There are, of course, many things that can go wrong. You might get my address wrong (whether by error or as a result of deception); the police might get a warrant to open my mail; the letter might be stolen by a dishonest postman; a fraudster might redirect my mail without my knowledge; or a thief might steal the letter from my mailbox. Similar things can go wrong with public key cryptography. False public keys can be inserted into the system; computers can be hacked; people can be coerced; and so on. We'll look at these problems in more detail in later chapters.

Another asymmetric application of cryptography is the *digital signature.* The idea here is that I can sign a message using a *private signature key*, then anybody can check this using my *public signature verification key.* Again, there are precomputer analogues in the form of manuscript signatures and seals; and again, there is a remarkably similar litany of things that can go wrong, both with the old way of doing things and with the new.

5.3 The Random Oracle Model

Before delving into the detailed design of modern ciphers, I want to take a few pages to refine the definitions of the various types of cipher. (Readers who are phobic about theoretical computer science should skip this section at a first pass; I've included it because a basic understanding of random oracles is needed to understand many recent research papers on cryptography.)

The random oracle model seeks to formalize the idea that a cipher is "good" if, when viewed in a suitable way, it is indistinguishable from a random function of a certain type. I will call a cryptographic primitive *pseudorandom* if it passes all the statistical and other tests that a random function of the appropriate type would pass, in whichever model of computation we are using. Of course, the cryptographic primitive will actually be an algorithm, implemented as an array of gates in hardware or a program in software; but the outputs should "look random" by being indistinguishable from a suitable random oracle given the type and the number of tests that our computation model permits.

In this way, we can hope to separate the problem of designing ciphers from the problem of using them correctly. Mathematicians who design ciphers can provide evidence that their cipher is pseudorandom. Quite separately, a computer scientist who has designed a cryptographic protocol can try to prove that it is secure on the assumption that the crypto primitives used to implement it are pseudorandom. The process isn't infallible, as we saw with proofs of protocol correctness. Theorems can have bugs, just like programs; the problem could be idealized wrongly; or the mathematicians might be using a different model of computation from the computer scientists. But at least some progress can be made.

You can visualize a random oracle as an elf sitting in a black box with a source of physical randomness and some means of storage (see Figure 5.9)—represented in the figure by the dice and the scroll. The elf will accept inputs of a certain type, then look in the scroll to see whether this query has ever been answered before. If so, it will give the answer it finds there; if not, it will generate an answer at random by throwing the dice. We'll further assume that there is some kind of bandwidth limitation—that the elf will answer only so many queries every second. This ideal will turn out to be useful as a way of refining our notions of a stream cipher, a hash function, a block cipher, a public key encryption algorithm and a digital signature scheme.

Finally, we can get a useful simplification of our conceptual model by noting that encryption can be used to protect data across time as well as across distance. A good example is when we encrypt data before storing it with a third-party backup service, and may decrypt it later if we have to recover from a disk crash. In this case, we need only a single encryption/decryption device, rather than one at each end of a communications link. This is the sort of application we will be modelling here. The user takes a diskette to the cipher machine, types in a key, issues an instruction, and the data get transformed in the appropriate way.

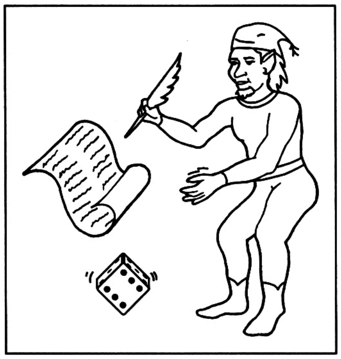


Figure 5.9 The random oracle.

Let's look at this model in more detail for these different cryptographic primitives.

5.3.1 Random Functions: Hash Functions

The first type of random oracle is the *random function.* A random function accepts an input string of any length, and outputs a random string of fixed length, say *n* bits long. So the elf just has a simple list of inputs and outputs, which grows steadily as it works. (We'll ignore any effects of the size of the scroll and assume that all queries are answered in constant time.)

Random functions are our model for *one-way functions* or *cryptographic hash functions*, which have many practical uses. They were first used in computer systems for one-way encryption of passwords in the 1960s and—as mentioned in Chapter 2—are used today in a number of authentication systems. They are also used to compute *message digests;* given a message *M*, we can pass it through a pseudorandom function to get a digest, say *h*(*M*), which can stand in for the message in various applications. One example is a digital signature: signature algorithms tend to be slow if the message is long, so it's usually convenient to sign a message digest rather than the message itself.

Another application is timestamping. If we want evidence that we possessed a given electronic document by a certain date, we might submit it to an online timestamping service. However, if the document is still secret—for example an invention that we plan to patent, and for which we merely want to establish a priority date—then we might not send the timestamping service the whole document, but just the message digest.

The output of the hash function is known as the *hash value* or *message digest;* an input corresponding to a given hash value is its *preimage;* the verb *to hash* is used to refer to computation of the hash value. Colloquially, the *hash* is also used as a noun to refer to the hash value.

5.3.1.1 Properties

The first main property of a random function is *one-wayness.* Given knowledge of an input *x*, we can easily compute the hash value *h*(*x*); but it is very difficult given the hash value *h*(*x*) to find a corresponding preimage *x* if one is not already known. (The elf will only pick outputs for given inputs, not the other way round.) As the output is random, the best an attacker who wants to invert a random function can do is to keep on feeding in more inputs until he or she gets lucky. A pseudorandom function will have the same property; or this could be used to distinguish it from a random function, contrary to our definition. It follows that a pseudorandom function will also be a *one-way function*, provided there are enough possible outputs that the opponent can't find a desired target output by chance. This means choosing the output to be an *n*-bit number where the opponent can't do anything near 2” computations.

A second property of pseudorandom functions is that the output will not give any information at all about even part of the input. Thus, one-way encryption of the value *x* can be accomplished by concatenating it with a secret key *k* and computing *h*(*x*, *k*). If the hash function isn't random enough though, using it for one-way encryption in this manner is asking for trouble. A topical example comes from the authentication in GSM mobile phones, where a 16-byte challenge from the base station is concatenated with a 16-byte secret key known to the phone into a 32-byte number, and passed through a hash function to give an 11-byte output [138]. The idea is that the phone company also knows k and can check this computation, while someone who eavesdrops on the radio link can only get a number of values of the random challenge x and corresponding output from $h(x, k)$. The eavesdropper must not be able to get any information about k or be able to compute $h(y, k)$ for a new input y. But the one-way function used by most phone companies isn't one-way enough, with the result that an eavesdropper who can pretend to be a base station and send a phone about 60,000 suitable challenges and get the responses can compute the key. I'll discuss this failure in more detail in Chapter 17, Section 17.3.3.

A third property of pseudorandom functions with sufficiently long outputs is that it is hard to find *collisions*, that is, different messages $M_1 \neq M_2$ with $h(M_1)=h(M_2)$. Unless the opponent can find a shortcut attack (which would mean the function wasn't really pseudorandom), then the best way of finding a collision is to collect a large set of messages M_i and their corresponding hashes $h(M_i)$, sort the hashes, and look for a match. If the hash function output is an n-bit number, so that there are 2^n possible hash values, then the number of hashes the enemy will need to compute before he or she can expect to find a match will be about the square root of this, namely $2^{n/2}$ hashes. This fact is of major importance in security engineering, so let's look at it more closely.

5.3.1.2 The Birthday Theorem

The *birthday theorem*, first known as *capture-recapture statistics*, was invented in the 1930s to count fish [679]. Suppose there are N fish in a lake, and you catch m of them, tag them, and throw them back; then when you first catch a fish you've tagged already, m should be "about" the square root of N. The intuitive reason this holds is that once you have $\sqrt{N}$ samples, each could potentially match any of the others, so the number of possible matches is about $\sqrt{N}\sqrt{N}$ or N, which is what you need.[1]

The birthday theorem has many applications for the security engineer. For example, if we have a biometric system that can authenticate a person's claim to identity with a probability of only one in a million that two randomly selected subjects will be falsely identified as the same person, this doesn't mean that we can use it as a reliable means of identification in a university with a user population of twenty thousand staff and students. This is because there will be almost two hundred million possible pairs. In fact, you can expect to find the first *collision*—the first pair of people who can be mistaken for each other by the system—once you have somewhat over a thousand people enrolled.

In some applications collision search attacks aren't a problem, such as in challenge response protocols where an attacker would have to be able to find the answer to the challenge just issued, and where you can prevent challenges repeating. (For example, the challenge might not be really random but generated by encrypting a counter.) In identify-friend-or-foe (IFF) systems, for example, common equipment has a response length of 48 to 80 bits.

[1] More precisely, the probability that m fish chosen randomly from N fish are different is $□ = N(N - 1) \cdots (N - m + 1)/N^m$ which is asymptotically solved by $N \approx m^2/2 \log(1/ □)$[451].

However, there are other applications in which collisions are unacceptable. In a digital signature application, if it were possible to find collisions with $h(M_1) = h(M_2)$ but $M_1 \neq M_2$, then a Mafia-owned bookstore's Web site might get you to sign a message M_1 saying something like, "I hereby order a copy of Rubber Fetish volume 7 for $32.95," and then present the signature together with an M_2, saying something like, "I hereby mortgage my house for $75,000; and please make the funds payable to Mafia Holdings Inc., Bermuda."

For this reason, hash functions used with digital signature schemes generally have n large enough to make them collision-free, that is, that $2^{n/2}$ computations are impractical for an opponent. The two most common are MD5, which has a 128-bit output and will thus require about 2^{64} computations to break, and SHA1 with a 160-bit output and a work factor for the cryptanalyst of about 2^{80}. MD5, at least, is starting to look vulnerable: already in 1994, a design was published for a $10 million machine that would find collisions in 24 days, and SHA1 will also be vulnerable in time. So the U.S. National Institute of Standards and Technology (NIST) has recently introduced still wider hash functions—SHA256 with a 256-bit output, and SHA512 with 512 bits. In the absence of cryptanalytic *shortcut attacks*—that is, attacks requiring less computation than brute force search—these should require 2^{128} and 2^{256} effort respectively to find a collision. This should keep Moore's Law at bay for a generation or two. In general, a prudent designer will use a longer hash function where this is possible, and the use of the MD series hash functions in new systems should be avoided (MD5 had a predecessor MD4 which turned out to be cryptanalytically weak, with collisions and preimages being found).

Thus, a pseudorandom function is also often referred to as being *collision-free* or *collision-intractable.* This doesn't mean that collisions don't exist—they must, as the set of possible inputs is larger than the set of possible outputs—just that you will never find any of them. The (usually unstated) assumption is that the output must be long enough.

5.3.2 Random Generators: Stream Ciphers

The second basic cryptographic primitive is the *random generator*, also known as a *keystream generator* or *stream cipher.* This is also a random function, but unlike in the hash function case it has a short input and a long output. (If we had a good pseudorandom function whose input and output were a billion bits long, and we never wanted to handle any objects larger than this, we could turn it into a hash function by throwing away all but a few hundred bits of the output, and a stream cipher by padding all but a few hundred bits of the input with a constant.) At the conceptual level, however, it's common to think of a stream cipher as a random oracle whose input length is fixed while the output is a very long stream of bits, known as the *keystream.* It can be used quite simply to protect the confidentiality of backup data: we go to the keystream generator, enter a key, get a long file of random bits, and exclusive-or it with our plaintext data to get ciphertext, which we then send to our backup contractor. We can think of the elf generating a random tape of the required length each time he is presented with a new key as input, giving it to us and keeping a copy of it on his scroll for reference in case he's given the same input again. If we need to recover the data, we go back to the generator, enter the same key, get the same long file of random data, and exclusive-or it with our ciphertext to get our plaintext data back again. Other people with access to the keystream generator won't be able to generate the same keystream unless they know the key.

I mentioned the one-time pad, and Shannon's result that a cipher has perfect secrecy if and only if there are as many possible keys as possible plaintexts, and every key is equally likely. Such security is called *unconditional* (or *statistical*) security, as it doesn't depend either on the computing power available to the opponent or on there being no future advances in mathematics that provide a shortcut attack on the cipher.

One-time pad systems are a very close fit for our theoretical model, except that they are typically used to secure communications across space rather than time: there are two communicating parties who have shared a copy of the randomly generated keystream in advance. Vernam's original telegraph cipher machine used punched paper tape; of which two copies were made in advance, one for the sender and one for the receiver. A modern diplomatic system might use optical tape, shipped in a tamper-evident container in a diplomatic bag. Various techniques have been used to do the random generation. Marks describes how SOE agents' silken keys were manufactured in Oxford by little old ladies shuffling counters.

One important problem with keystream generators is that we want to prevent the same keystream being used more than once, whether to encrypt more than one backup tape or to encrypt more than one message sent on a communications channel. During World War II, the amount of Russian diplomatic traffic exceeded the quantity of one-time tape they had distributed in advance to their embassies, so it was reused. This was a serious blunder. If $M_1 + K = C_1$, and $M_2 + K = C_2$, then the opponent can combine the two ciphertexts to get a combination of two messages: $C_1 - C_2 = M_1 - M_2$; and if the messages M_i have enough redundancy, then they can be recovered. Text messages do in fact contain enough redundancy for much to be recovered; and in the case of the Russian traffic, this led to the Venona project in which the United States and United Kingdom decrypted large amounts of wartime Russian traffic afterward and broke up a number of Russian spy rings. The saying is: "Avoid the two-time tape!"

Exactly the same consideration holds for any stream cipher, and the normal engineering practice when using an algorithmic keystream generator is to have a *seed* as well as a key. Each time the cipher is used, we want it to generate a different keystream, so the key supplied to the cipher should be different. So, if the long-term key that two users share is K, they may concatenate it with a seed that is a message number N (or some other nonce), then pass it through a hash function to form a working key $h(K, N)$. This working key is the one actually fed to the cipher machine.

5.3.3 Random Permutations: Block Ciphers

The third type of primitive, and the most important in modern commercial cryptography, is the *block cipher*, which we model as a *random permutation*. Here, the function is invertible, and the input plaintext and the output ciphertext are of a fixed size. With Playfair, both input and output are two characters; with DES, they're both bit strings of 64 bits. Whatever the number of symbols and the underlying alphabet, encryption acts on a block of fixed length. (If you want to encrypt a shorter input, you have to pad it, as with the final z in our Playfair example.)

We can visualize block encryption as follows. As before, we have an elf in a box with dice and a scroll. On the left is a column of plaintexts, and on the right is a column of ciphertexts. When we ask the elf to encrypt a message, it checks in the left-hand column to see if has a record of it. If not, it uses the dice to generate a random ciphertext of the appropriate size (and one that doesn't appear yet in the right-hand column of the scroll), and writes down the plaintext/ciphertext pair in the scroll. If it does find a record, it gives us the corresponding ciphertext from the right-hand column.

When asked to decrypt, the elf does the same, but with the function of the columns reversed: he takes the input ciphertext, checks it (this time on the right-hand scroll); and if he finds it, he gives the message with which it was previously associated. If not, he generates a message at random (which does not already appear in the left column) and notes it down.

A block cipher is a keyed family of pseudorandom permutations. For each key, we have a single permutation that is independent of all the others. We can think of each key as corresponding to a different scroll. The intuitive idea is that, given the plaintext and the key, a cipher machine should output the ciphertext; and given the ciphertext and the key, it should output the plaintext; but given only the plaintext and the ciphertext, it should output nothing.

Let's write a block cipher using the notation established for encryption in Chapter 2:

$$C = \{M\}_K$$

The random permutation model also allows us to define different types of attack on block ciphers. In a *known plaintext attack*, the opponent is just given a number of randomly chosen inputs and outputs from the oracle corresponding to a target key. In a *chosen plaintext attack*, the opponent is allowed to put a certain number of plaintext queries and get the corresponding ciphertexts. In a *chosen ciphertext attack*, he gets to make a number of ciphertext queries. In a *chosen plaintext/ciphertext attack*, he is allowed to make queries of either type. Finally, in a *related key attack*, the opponent can make queries that will be answered using keys related to the target key K (such as K + 1 and K + 2).

In each case, the objective of the attacker may be either to deduce the answer to a query he hasn't already made (a *forgery attack*), or to recover the key (unsurprisingly known as a *key recovery attack*).

This precision about attacks is important. When someone discovers a vulnerability in a cryptographic primitive, it may or may not be relevant to your application. Often, it won't be, but it will be hyped by the media, so you will need to be able to explain clearly to your boss and your customers why it's not a problem. To do this, you have to look carefully at what kind of attack has been found, and what the parameters are. For example, the first major attack announced on the DES algorithm requires 2^{47} chosen plaintexts to recover the key, while the next major attack improved this to 2^{43} known plaintexts. While these attacks were of great scientific importance, their practical engineering effect was zero, as no practical systems make that much known (let alone chosen) text available to an attacker. Such attacks are often referred to as *certificational.* They can have a commercial effect, though: the attacks on DES undermined confidence in it, and started moving people to other ciphers. In some other cases, an attack that started off as certificational has been developed by later ideas into an exploit.

Which sort of attacks you should be worried about depends very much on your application. With a broadcast entertainment system, for example, you can buy a decoder, observe a lot of material, and compare it with the enciphered broadcast signal; so a known-plaintext attack is the main threat to worry about. But there are surprisingly many applications where chosen plaintext attacks are possible. Obvious ones include ATMs, where, if you allow customers to change their PINs at will, they can change them through a range of possible values and observe the enciphered equivalents using a wiretap on the line from the ATM to the bank. A more traditional example is diplomatic messaging systems, where it has been known for a host government to give an ambassador a message to transmit to her capital that has been specially designed to help the local cryptanalysts fill out the missing gaps in the ambassador's code book [428]. In general, if the opponent can insert any kind of message into your system, it's chosen plaintext attacks you should worry about.

The other attacks are more specialized. Chosen plaintext/ciphertext attacks may be a worry where the threat is a *lunchtime attacker*, someone who gets temporary access to some cryptographic equipment while its authorized user is out. Related key attacks are of concern where the block cipher is used as a building block in the construction of a hash function (which I discuss later).

5.3.4 Public Key Encryption and Trapdoor One-Way Permutations

A *public key encryption* algorithm is a special kind of block cipher in which the elf will perform the encryption corresponding to a particular key for anyone who requests it, but will do the decryption operation only for the key's owner. To continue with our analogy, the user might give a secret name to the scroll, which only she and the elf know, use the elf's public one-way function to compute a hash of this secret name, publish the hash, and instruct the elf to perform the encryption operation for anybody who quotes this hash.

This means that a principal, say Alice, can publish a key; and if Bob wants to, he can now encrypt a message and send it to her, even if they have never met. All that is necessary is that they have access to the oracle. There are some more details that have to be taken care of, such as how Alice's name can be bound to the key, and indeed whether it means anything to Bob. We'll deal with these later.

A common way of implementing public key encryption is the *trapdoor one-way permutation*. This is a computation that anyone can perform, but that can be reversed only by someone who knows a *trapdoor* such as a secret key. This model is like the one-way function model of a cryptographic hash function, but I state it formally nonetheless: a public key encryption primitive consists of a function that, given a random input R, will return two keys, KR (the public encryption key) and KR^{-1} (the private decryption key) with the properties that:

- Given KR, it is unfeasible to compute KR^{-1} (so it's not possible to compute R either).
- There is an encryption function {...} that, applied to a message M using the encryption key KR, will produce a ciphertext $C = \{M\}_{KR}$.
- There is a decryption function that, applied to a ciphertext C, using the decryption key KR^{-1}, will produce the original message $M = \{C\}_{KR}{}^{-1}$.

For practical purposes, we will want the oracle to be replicated at both ends of the communications channel, and this means either using tamper-resistant hardware or (more commonly) implementing its functions using mathematics rather than metal. That's why the second of our models, which is somewhat less abstract than the first, can be more useful. Anyway, we'll look at implementation details later.

5.3.5 Digital Signatures

The final cryptographic primitive I'll define here is the *digital signature.* The basic idea is that a signature on a message can be created by only one person, but checked by anyone. It can thus perform the sort of function in the electronic world that ordinary signatures do in the world of paper.

Signature schemes can be *deterministic* or *randomized*: in the first, computing a signature on a message will always give the same result; in the second, it will give a different result each time you compute it. (The latter is more like handwritten signatures; no two are ever alike but the bank has a means of deciding whether a given specimen is genuine or forged). Also, signature schemes may or may not support *message recovery.* If they do, then, given the signature, anyone can recover the message on which it was generated; if they don't, then the verifier needs to know or guess the message before he can perform the verification. (There are further, more specialized, signature schemes, such as blind signatures and threshold signatures, but I'll postpone discussion of them for now.)

Formally, a signature scheme, like public key encryption scheme, has a keypair generation function that, given a random input R will return two keys, □ R (the private signing key) and VR (the public signature verification key) with the properties that:

- Given the public signature verification key VR, it is infeasible to compute, the private signing key □ R.
- There is a digital signature function that, given a message M and a private signature key □ R, will produce a signature $Sig_{□R}$ (M).
- There is a signature verification function that, given the signature $Sig_{□R}$ (M) and the public signature verification key VR, will output TRUE if the signature was computed correctly with □ R; otherwise, it will output FALSE.

In our random oracle model, we can model a digital signature algorithm as a random function that reduces any input message to a one-way hash value of fixed length, followed by a special kind of block cipher in which the elf will perform the operation in one direction, known as *signature*, for only one principal, while in the other direction, it will perform verification for anybody.

Signature verification can take two forms. In the basic scheme, the elf (or the signature verification algorithm) outputs only TRUE or FALSE, depending on whether the signature is good. But in a scheme with *message recovery*, anyone can input a signature and get back the message corresponding to it. In our elf model, this means that if the elf has seen the signature before, it will give the message corresponding to it on the scroll; otherwise, it will give a random value (and record the input and the random output as a signature and message pair). This is sometimes desirable: when sending short messages over a low-bandwidth channel, it can save space if only the signature has to be sent rather than the signature plus the message. An example is in the machine-printed postage stamps, or *indicia*, being brought into use in the United States and other countries: the stamp consists of a 2-d barcode with a digital signature made by the postal meter and that must contain all sorts of information, such as the value, the date, and the sender's and recipient's post codes. There's more detail about this at the end of Chapter 12, "Security Printing and Seals."

However, in the general case, we do not need message recovery, as the message to be signed may be of arbitrary length; so we will first pass it through a hash function and then sign the hash value. As hash functions are one-way, the resulting compound signature scheme does not have message recovery—although if the underlying signature scheme does, then the hash of the message can be recovered from the signature.

5.4 Symmetric Crypto Primitives

Now that we have defined the basic crypto primitives, we will look under the hood to see how they can be implemented in practice. While most explanations are geared toward graduate mathematics students, the presentation I'll give here is based on one I've developed over several years with computer science students. So I hope it will let the non-mathematician grasp the essentials. In fact, even at the research level, most of cryptography is as much computer science as mathematics. Modern attacks on ciphers are put together from guessing bits, searching for patterns, sorting possible results, and so on, rather than from anything particularly highbrow.

We'll focus in this section on block ciphers, then in the next section move on to how you can make hash functions and stream ciphers from them, and vice versa. (In later chapters, we'll also look at some special-purpose ciphers.)

5.4.1 SP-Networks

Shannon suggested in the 1940s that strong ciphers could be built by combining substitution with transposition repeatedly. For example, one might add some key material to a block of input text, then shuffle subsets of the input, and continue in this way a number of times. He described the properties of a cipher as being *confusion* and *diffusion*—adding unknown key values will confuse an attacker about the value of a plaintext symbol, while diffusion means spreading the plaintext information through the ciphertext. Block ciphers need diffusion as well as confusion.

The earliest block ciphers were simple networks that combined substitution and permutation circuits, and so were called SP-networks. The diagram in Figure 5.10 shows an SP-network with 16 inputs, which we can imagine as the bits of a 16-bit number, and two layers of 4-bit invertible substitution boxes (or *S-boxes*), each of which can be visualized as a lookup table containing some permutation of the numbers 0 to 15.

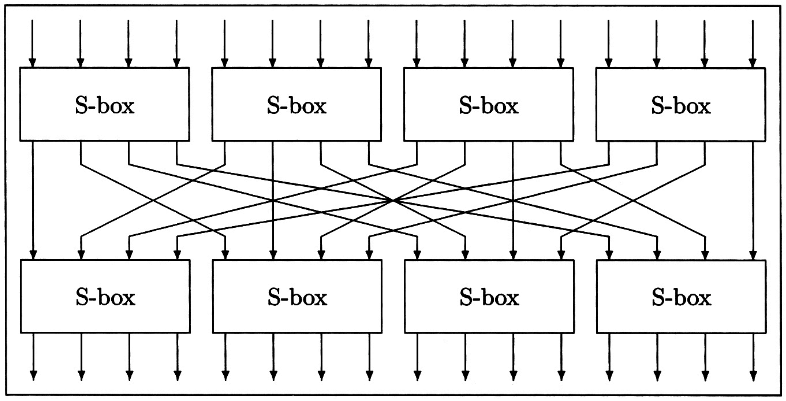


Figure 5.10 A simple 16-bit SP-network block cipher.

The point of this arrangement is that if we were to implement an arbitrary 16-bit to 16-bit function in digital logic, we would need 2^{20} bits of memory—one lookup table of 2^{16} bits for each single output bit. That's hundreds of thousands of gates, while a 4-bit to 4-bit function takes only 64 bits of memory. One might hope that with suitable choices of parameters, the function produced by iterating this simple structure would be indistinguishable from a random 16-bit to 16-bit function to an opponent who didn't know the value of the key. The key might consist of some choice of a number of 4-bit S-boxes, or it might be added at each round to provide confusion, and the resulting text fed through the S-boxes to provide diffusion.

Three things need to be done to make such a design secure:

1. The cipher needs to be "wide" enough.
2. The cipher needs to have enough rounds.
3. The S-boxes need to be suitably chosen.

5.4.1.1 Block Size

First, a block cipher that operated on 16-bit blocks would have rather limited applicability, as an opponent could just build a dictionary of plaintext and ciphertext blocks as he or she observed them. The birthday theorem tells us that even if the input plaintexts were random, the opponent would expect to find a match as soon as she had seen a little over 2^8 blocks. So a practical block cipher will usually deal with plaintexts and ciphertexts of 64 bits, 128 bits, or even more. If we are using 4-bit to 4-bit S-boxes, we may have 16 of them (for a 64-bit block size) or 32 of them (for a 128 bit block size).

5.4.1.2 Number of Rounds

Second, we must have enough rounds. The two rounds in Figure 5.10 are completely inadequate, as an opponent can deduce the values of the S-boxes by tweaking input bits in suitable patterns. For example, she could hold the rightmost 12 bits constant and try tweaking the leftmost 4 bits, to deduce the values in the top left S-box. (The attack is slightly more complicated than this, as sometimes a tweak in an input bit to an S-box won't produce a change in any output bit, so we have to change one of its other inputs and tweak again. But implementing it is still a simple student exercise.)

The number of rounds we require depends on the speed with which data diffuse through the cipher. In the above simple example, diffusion is very slow because each output bit from one round of S-boxes is connected to only one input bit in the next round. Instead of having a simple permutation of the wires, it is more efficient to have a linear transformation in which each input bit in one round is the exclusive-or of several output bits in the previous round. Of course, if the block cipher is to be used for decryption as well as encryption, this linear transformation will have to be invertible. We'll see some concrete examples below in the sections on AES and Serpent.

5.4.1.3 Choice of S-Boxes

The design of the S-boxes also affects the number of rounds required for security, and studying bad choices gives us our entry into the deeper theory of block ciphers. Suppose that the S-box were the permutation that maps the inputs (0, 1, 2,..., 15) to the outputs (5, 7, 0, 2, 4, 3, 1, 6, 8, 10, 15, 12, 9, 11, 14, 13). Then the most significant bit of its input would come through unchanged as the most significant bit of its output. If the same S-box were used in both rounds in the preceding cipher, then the most significant bit of the input block would pass through to become the most significant bit of the output block. So we certainly couldn't pretend that our cipher was pseudorandom.

5.4.1.4 Linear Cryptanalysis

Attacks on real block ciphers are usually harder to spot than in this artificial example, but they use the same ideas. It might turn out that the S-box had the property that bit 1 of the input was equal to bit 2, plus bit 4 of the output; more commonly, there will be linear approximations to an S-box, which hold with a certain probability. *Linear cryptanalysis* [526] proceeds by collecting a number of relations such as "bit 2 plus bit 5 of the input to the first S-box is equal to bit 1 plus bit 8 of the output, with probability 13/16," then searching for ways to glue them together into an algebraic relation between input bits, output bits, and keybits that holds with a probability different from one-half. If we can find a linear relationship that holds over the whole cipher with probability $p = 0.5 + 1/M$, then according to probability theory, we can expect to start recovering keybits once we have about M^2 known texts. If the best linear relationship has an M^2 greater than the total possible number of known texts (namely 2^n where the inputs and outputs are n-bits wide), then we consider the cipher to be secure against linear cryptanalysis.

5.4.1.5 Differential Cryptanalysis

Differential cryptanalysis [102] is similar but is based on the probability that a given change in the input to an S-box will give rise to a certain change in the output. A typical observation on an 8-bit S-box might be that "if we flip input bits 2, 3, and 7 at once, then with probability 11/16 the only output bits that will flip are 0 and 1." In fact, with any nonlinear Boolean function, tweaking some combination of input bits will cause some combination of output bits to change with a probability different from one half. The analysis procedure is to look at all possible input difference patterns and look for those values $\square_i$, $\square_o$ such that an input change of $\square_i$ will produce an output change of $\square_o$ with particularly high (or low) probability.

As in linear cryptanalysis, we then search for ways to join things up so that an input difference that we can feed into the cipher will produce a known output difference with a useful probability over a number of rounds. Given enough chosen inputs, we will see the expected output and be able to make deductions about the key. As in linear cryptanalysis, it's common to consider the cipher to be secure if the number of texts required for an attack is greater than the total possible number of different texts for that key. (We have to be careful of pathological cases, such as if we had a cipher with a 32-bit block and a 128-bit key with a differential attack whose success probability given a single pair was 2^{-40}. Given a lot of text under a number of keys, we'd eventually solve for the current key.)

There are a quite a few variants on these two themes. For example, instead of looking for high-probability differences, we can look for differences that can't happen (or that happen only rarely). This has the charming name of *impossible cryptanalysis*, even though it definitely possible against many systems [101]. There are also various specialized attacks on particular ciphers.

Block cipher design involves a number of trade-offs. For example, we can reduce the per-round information leakage, and thus the required number of rounds, by designing the rounds carefully. However, a complex design might be slow in software, or need a lot of gates in hardware, so using simple rounds but more of them might be better. Simple rounds may also be easier to analyze. A prudent designer will also use more rounds than are strictly necessary to block the attacks known today, in order to give some margin of safety against improved mathematics in the future. We may be able to show that a cipher resists all the attacks we know of, but this says little about whether it will resist the attacks we don't know of yet. (A general security proof for a block cipher would appear to imply a proof about an attacker's computational powers, which might entail a result such as $P \neq NP$ that would revolutionize computer science.)

The point that the security engineer should remember is that block cipher cryptanalysis is a complex subject about which we have a fairly extensive theory, so it is better to use an off-the-shelf design that has been thoroughly scrutinized by experts, rather than rolling your own.

5.4.1.6 Serpent

As a concrete example, the encryption algorithm Serpent is an SP-network with input and output block sizes of 128 bits. These are processed through 32 rounds, in each of which we first add 128 bits of key material, then pass the text through 32 S-boxes of 4-bits width, then perform a linear transformation that takes each output of one round to the inputs of a number of S-boxes in the next round. Rather than each input bit in one round coming from a single output bit in the last, it is the exclusive-or of between two and seven of them. This means that a change in an input bit propagates rapidly through the cipher—a so-called *avalanche* effect that makes both linear and differential attacks harder. After the final round, a further 128 bits of key material are added to give the ciphertext. The 33 times 128 bits of key material required are computed from a user-supplied key of up to 256 bits.

This is a real cipher using the structure of Figure 5.10, but modified to be “wide” enough and to have enough rounds. The S-boxes are chosen to make linear and differential analysis hard; they have fairly tight bounds on the maximum linear correlation between input and output bits, and on the maximum effect of toggling patterns of input bits. Each of the 32 S-boxes in a given round is the same; this means that bit-slicing techniques can be used to give a very efficient software implementation on 32-bit processors.

Its simple structure makes Serpent easy to analyze, and it can be shown that it withstands all the currently known attacks. (A full specification of Serpent is given in [40] and can be downloaded, together with implementations in a number of languages, from [41].)

5.4.2 The Advanced Encryption Standard (AES)

This discussion has prepared us to describe the Advanced Encryption Standard, an algorithm also known as Rijndael after its inventors Vincent Rijmen and Joan Daemen.[2] This algorithm acts on 128-bit blocks and can use a key of 128, 192 or 256 bits in length. It is an SP-network; in order to specify it, we need to fix the S-boxes, the linear transformation between the rounds, and the way in which the key is added into the computation.

Rijndael uses a single S-box which acts on a byte input to give a byte output. For implementation purposes it can be regarded simply as a lookup table of 256 bytes; it is actually defined by the equation $S(x) = M(1/x) + b$ over the field GF(2^8) where M is a suitably chosen matrix and b is a constant. This construction gives tight differential and linear bounds.

The linear transformation is based on arranging the 16 bytes of the value being enciphered in a square and then doing bytewise shuffling and mixing operations. (Rijndael is descended from an earlier cipher called Square, which introduced this technique.)

The first step in the linear transformation is the shuffle in which the top row of four bytes is left unchanged, while the second row is shifted one place to the left, the third row by two places and the fourth row by three places. The second step is a column mixing step in which the four bytes in a column are mixed using a matrix multiplication. This is illustrated in Figure 5.11 which shows, as an example, how a change in the value of the third byte in the first column is propagated. The effect of this combination is that a change in the input to the cipher can potentially affect all of the output after just two rounds.

[2] If you’re from Holland, Belgium or South Africa, Rijndael is pronounced just as you would expect; if you’re not a Dutch speaker, it is something like ‘rain-dahl.’ The ‘J’ is not a consonant in Dutch, so Rijndael is not pronounced anything like ‘Region Deal,’ and Rijmen is pronounced as ‘Raymen’ not ‘Ridgemen.’

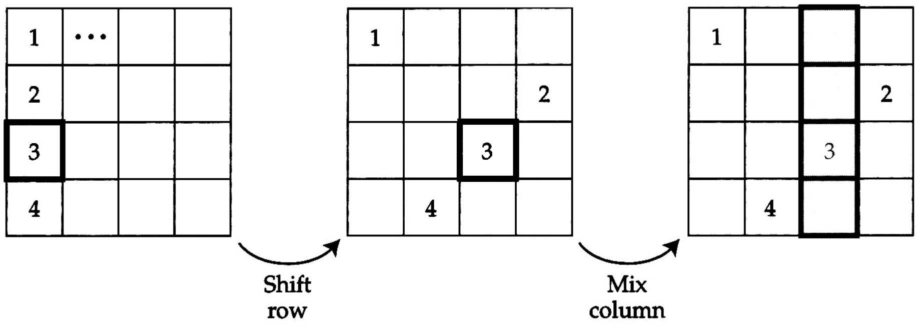


Figure 5.11 The Rijndael linear transformation, illustrated by its effect on byte 3 of the input.

The key material is added byte by byte after the linear transformation. This means that 16 bytes of key material are needed per round; they are derived from the user supplied key material by means of a recurrence relation.

The algorithm uses 10 rounds with 128-bit keys, 12 rounds with 192-bit keys and 14 rounds with 256-bit keys. These give about a 50% margin of safety; the best shortcut attacks known at the time of writing can tackle 6 rounds for 128 bit keys, 7 rounds for 192 bit keys and 9 rounds for 256 bit keys. The general belief in the block cipher community is that even if advances in the state of the art do permit attacks on Rijndael with the full number of rounds, they will be purely certificational attacks in that they will require infeasibly large numbers of texts. (Rijndael's margin of safety against attacks that require only feasible numbers of texts is about 100%.) Although there is no proof of security—whether in the sense of pseudorandomness, or in the weaker sense of an absence of shortcut attacks—there is now a high level of confidence that Rijndael is secure for all practical purposes.

Even although I was an author of Serpent which was an unsuccessful finalist in the AES competition (Rijndael got 86 votes, Serpent 59 votes, Twofish 31 votes, RC6 23 votes and MARS 13 votes at the last AES conference), and although Serpent was designed to have an even larger security margin than Rijndael, I recommend to my clients that they use Rijndael where a general purpose block cipher is required. I recommend the 256-bit-key version, and not because I think that the 10 rounds of the 128-bit-key variant will be broken anytime soon. Longer keys are better because some key bits often leak in real products, as we'll discuss at some length in Chapters 14 and 15. It does not make any sense to implement Serpent as well, 'just in case Rijndael is broken': the risk of a fatal error in the algorithm negotiation protocol is orders of magnitude greater than the risk that anyone will come up with a production attack on Rijndael. (We'll see a number of examples later where using multiple algorithms, or using an algorithm like DES multiple times, caused something to break horribly.)

The definitive specification of Rijndael will be published sometime in 2001 as a Federal Information processing Standard. Meanwhile, the algorithm is described in papers on the Rijndael home page [647]; there are also a number of implementations available both there and elsewhere on the net. The paper describing Rijndael's predecessor, Square, is at [213].

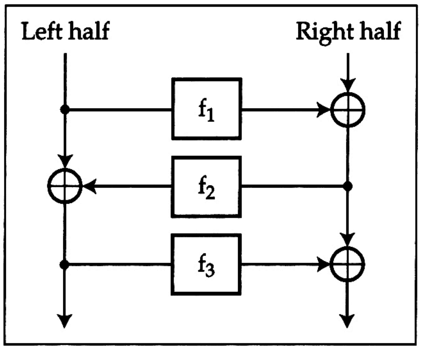


Figure 5.12 The Feistel cipher structure.

5.4.3 Feistel Ciphers

Most block ciphers use a more complex structure, which was invented by Horst Feistel's technicians while they were developing cryptographic protection for IFF in the 1950s and early 1960s. Feistel then moved to IBM and founded a research group that produced the Data Encryption Standard, (DES) algorithm, which is the mainstay of financial transaction processing security.

A Feistel cipher has the ladder structure shown in Figure 5.12. The input is split up into two blocks, the left half and the right half. A *round function* f_1 of the left half is computed and combined with the right half, using exclusive-or (binary addition without carry), though in some Feistel ciphers addition with carry is also used. (We use the notation ⊕ for exclusive-or.) Then, a function f_2 of the right half is computed and combined with the left half, and so on. Finally (if the number of rounds is even), the left half and right half are swapped.

A notation you may see for the Feistel cipher is □ $(f, g, h, \ldots)$ where $f, g, h, \ldots$ are the successive round functions. Under this notation, the preceding cipher is □ (f_1, f_2, f_3). The basic result that enables us to decrypt a Feistel cipher—and, indeed, the whole point of his design—is that:

$$\square^{-1}(f_1, f_2, \ldots, f_{2k-1}) = \square(f_{2k-1}, \ldots, f_2, f_1)$$

In other words, to decrypt, we just use the round functions in the reverse order. Thus, the round functions f_i do not have to be invertible, and the Feistel structure lets us turn any one-way function into a block cipher. This means that we are less constrained in trying to choose a round function with good diffusion and confusion properties; it can more easily satisfy any other design constraints such as code size, table size, software speed, hardware gate count, and so on.

5.4.3.1 The Luby-Rackoff Result

The seminal theoretical result on Feistel ciphers was proved by Mike Luby and Charlie Rackoff in 1988. They showed that if f_i were random functions, then □ (f_1, f_2, f_3) was indistinguishable from a random permutation under chosen plaintext attack. This result was soon extended to show that □ (f_1, f_2, f_3, f_4) was indistinguishable under chosen plaintext/ciphertext attack—in other words, it was a pseudorandom permutation.

I am omitting a number of technicalities. In engineering terms, the effect is that, given a really good round function, four rounds of Feistel are enough. So if we have a hash function in which we have confidence, it is straightforward to construct a block cipher from it.

5.4.3.2 DES

The DES algorithm is widely used in banking, government, and embedded applications. For example, it is the standard in automatic teller machine networks.

The DES algorithm is a Feistel cipher, with a 64-bit block and 56-bit key. Its round function operates on 32-bit half-blocks, and consists of four operations:

- First, the block is expanded from 32 bits to 48.
- Next, 48 bits of round key are mixed using exclusive-or.
- The result is passed through a row of eight S-boxes, each of which takes a 6-bit input and provides a 4-bit output.
- Finally, the bits of the output are permuted according to a fixed pattern.

The effect of the expansion, key mixing, and S-boxes is shown in the diagram in Figure 5.13.

The round keys are derived from the user-supplied key by using each user keybit in about 14 different rounds according to a slightly irregular pattern. (A full specification of DES is given in [575]; code can be found in [681] or downloaded from many places on the Web.)

DES was introduced in 1974 and caused some controversy. The most telling criticisms was that the key is too short. Someone who wants to find a 56-bit key using brute force—that is by trying all possible keys—will have a *total exhaust time* of 2^{56} encryptions and an *average solution time* of half that, namely 2^{55} encryptions. Diffie and Hellman pointed out that a DES keysearch machine could be built with a million chips, each testing a million keys a second; as a million is about 2^{20}, this would take on average 2^{15} seconds, or just over nine hours, to find the key. They argued that such a machine could be built for $20 million in 1977 [249]. IBM, whose scientists invented DES, retorted that they would charge the U.S. government $200 million to build such a machine. (Perhaps both were right.)

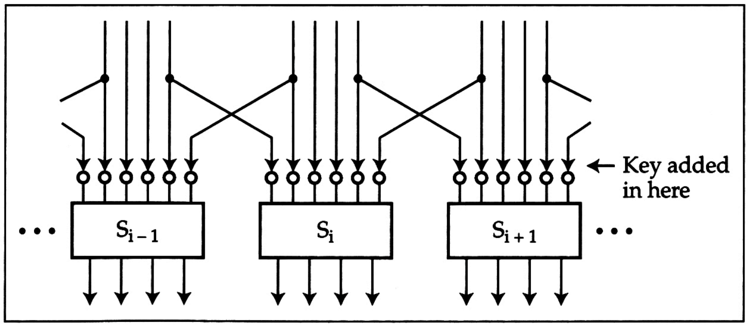


Figure 5.13 The DES round function.

During the 1980s, there were persistent rumors of DES keysearch machines being built by various intelligence agencies, but the first successful public keysearch took place in 1997. In a distributed effort organized over the Net, 14,000 Pentium-level computers took more than four months to find the key to a challenge. In 1998, the Electronic Frontier Foundation (EFF) built a DES keysearch machine for under $250,000; it broke a DES challenge in three days. It contained 1,536 chips run at 40 MHz, each chip containing 24 search units which each took 16 cycles to do a test decrypt. The search rate was thus 2.5 million test decryptions per second per search unit, or 60 million keys per second per chip. (The design of the cracker is public and can be found at [265].) Clearly, the key length of DES is now definitely inadequate for protecting data against a capable motivated opponent, and banks are upgrading their payment systems.

Another criticism of DES was that, since IBM kept its design principles secret at the request of the U.S. government, perhaps there was a trapdoor that would give them easy access. However, the design principles were published in 1992 after differential cryptanalysis was developed and published [205]. Their story was that IBM had discovered these techniques in 1972, and the NSA even earlier. IBM kept the design details secret at the NSA's request. We'll discuss the political aspects of all this in Chapter 21.

We now have a fairly thorough analysis of DES. The best-known *shortcut attack*—is a linear attack using 2^{42} known texts. DES would be secure with more than 20 rounds, but for practical purposes its security appears to be limited by its keylength. I know of no applications where an attacker might get hold of even 2^{40} known texts, so the known shortcut attacks are not an issue in practice. However, its growing vulnerability to keysearch cannot be ignored. If Moore's law continues, then by 2015 or 2020 it will be possible to find a DES key on a single PC in a few months, which means even low-grade systems such as taximeters will be vulnerable to attack using brute-force cryptanalysis. (Your reaction at this point might be "Give me one reason to attack a taximeter!" I will, in Chapter 10, "Monitoring Systems.")

One way of preventing keysearch is *whitening*. In addition to the 56-bit key, say k_0, we choose two 64-bit whitening keys k_1 and k_2, xor'ing the first with the plaintext before encryption and the second with the output of the encryption to get the ciphertext afterward. This composite cipher is known as DESX, and is used in the Win2K encrypting file system. Formally:

$$DESX(k_0, k_1, k_2; M) = DES(k_0; \mathrm{M} \oplus k_1) \oplus k_2$$

It can be shown that, on reasonable assumptions, DESX has the properties you'd expect: it inherits the differential strength of DES, but its resistance to keysearch is increased by the amount of the whitening [457].

Another way of dealing with DES keysearch is to use the algorithm multiple times with different keys. This is being introduced in banking networks, and the *triple-DES* algorithm banks use is now a draft U.S. government standard [575]. Triple-DES does an encryption, then a decryption, then a further encryption, all with independent keys. Formally:

$$3DES(k_0, k_1, k_2; M) = DES(k_2; DES^{-1}(k_1; DES(k_0; M)))$$

The reason for this design is that, by setting the three keys equal, one gets the same result as a single DES encryption, thus giving a backward-compatibility mode with legacy equipment. (Some systems use *two-key triple-DES*, which sets $k_2 = k_0$; this gives an intermediate step between single- and triple-DES).

5.5 Modes of Operation

In practice, how you use an encryption algorithm is more important than which one you pick. An important factor is the *mode of operation*, which specifies how a block cipher with a fixed block size (8 bytes for DES, 16 for AES) can be extended to process messages of arbitrary length.

There are several modes of operation for using a block cipher on multiple blocks. Understanding them, and choosing the right one for the job, is an important factor in using a block cipher securely.

5.5.1 Electronic Code Book

In *electronic code book* (ECB), we just encrypt each succeeding block of plaintext with our block cipher to get ciphertext, as with the Playfair cipher given as an example earlier. This is adequate for many simple operations, such as challenge-response and some key management tasks; it's also used to encrypt PINs in cash machine systems. However, if we use it to encrypt redundant data, the patterns will show through, letting an opponent deduce information about the plaintext. For example, if a word processing format has lots of strings of nulls, then the ciphertext will have a lot of blocks whose value is the encryption of null characters under the current key.

In one popular corporate email system from the late 1980s, the encryption used was DES ECB with the key derived from an eight-character password. If you looked at a ciphertext generated by this system, you saw that a certain block was far more common than the others—the one that corresponded to a plaintext of nulls. This enabled one of the simplest attacks on a fielded DES encryption system: just encrypt a null block with each password in a dictionary, and sort the answers. You can now break on sight any ciphertext whose password was one of those in your dictionary.

In addition, using ECB mode to encrypt messages of more than one block length and that have an authenticity requirement—such as bank payment messages—would be foolish, as messages could be subject to a *cut-and-splice* attack along the block boundaries. For example, if a bank message said, "Please pay account number *X* the sum *Y*, and their reference number is *Z*," then an attacker might initiate a payment designed so that some of the digits of *X* could be replaced with some of the digits of *Z*.

5.5.2 Cipher Block Chaining

Most commercial applications that encrypt more than one block use *cipher block chaining*, or CBC, mode. In it, we exclusive-or the previous block of ciphertext to the current block of plaintext before encryption (see Figure 5.14).

This mode is effective at disguising any patterns in the plaintext: the encryption of each block depends on all the previous blocks. The input IV is an *initialization vector*, a random number that performs the same function as a seed in a stream cipher, and ensures that stereotyped plaintext message headers won't leak information by encrypting to identical ciphertext blocks.

However, an opponent who knows some of the plaintext may be able to cut and splice a message (or parts of several messages encrypted under the same key), so the integrity protection is not total.

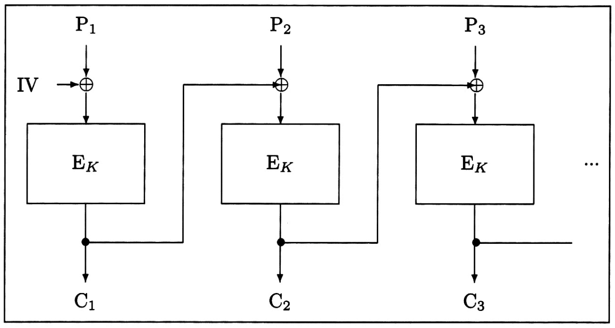


Figure 5.14 Cipher block chaining (CBC) mode.

5.5.3 Output Feedback

Output feedback (OFB) mode consists of repeatedly encrypting an initial value and using this as a keystream in a stream cipher of the kind discussed earlier. Writing "IV" for the initialization vector, or seed, the *i*-th block of keystream will be given by $K_1 = \{IV\}_K$, $K_i = \{K_{i-1}\}$ or:

$$K_i = \{\ldots\{\{IV\}_K\}_K \ldots \textit{total of i times}\}$$

This is the standard way of turning a block cipher into a stream cipher. The key K is expanded into a long stream of blocks K_i of *keystream.* Keystream is typically combined with the blocks of a message M_i using exclusive-or to give ciphertext $C_i = M_i \oplus K_i$; this arrangement is sometimes called an *additive stream cipher*, as exclusive-or is just addition modulo 2 (and some old hand systems used addition modulo 26).

Sometimes, a specialist keystream generator is used; for example, the A5 algorithm, which is covered in Chapter 17, has a much lower gate count than DES, and is thus used in mobile applications where battery power is the critical design paramater. However, in the absence of a constraint like this, it is common to use a block cipher in OFB mode to provide the keystream.

All additive stream ciphers have an important vulnerability: they fail to protect message integrity. We mentioned this in the context of the one-time pad in Section 5.2.2, but it's important to realize that this doesn't just affect "perfectly secure" systems but "real life" stream ciphers, too. Suppose, for example, that a stream cipher were used to encipher fund transfer messages. These messages are very highly structured; you might know, for example, that bytes 37–42 contained the amount of money being transferred. You could then carry out the following attack. You cause the data traffic from a local bank to go via your computer (whether by physically splicing into the line, or more simply by using one of the standard routing attacks discussed in Part 2). You go into the bank and send a modest sum (say, $500) to an accomplice. The ciphertext $C_i = M_i \oplus K_i$, duly arrives in your machine. Because you know M_i for bytes 37–42, you know K_i and can easily construct a modified message that instructs the receiving bank to pay not $500 but $500,000! This is an example of an *attack in depth*; it is the price not just of the perfect secrecy we get from the one-time pad, but of much more humble stream ciphers too.

5.5.4 Counter Encryption

One possible drawback of output feedback mode, and in fact of all feedback modes of block cipher encryption, is latency; feedback modes are hard to parallelize. With CBC, a whole block of the cipher must be computed between each block input and each block output; with OFB, we can precompute keystream but storing it requires memory. This can be inconvenient in very high-speed applications, such as protecting traffic on 155 Mbit/s backbone links. There, as silicon is cheap, we would rather pipeline our encryption chip, so that it encrypts a new block (or generates a new block of keystream) in as few clock ticks as possible.

The simplest solution is often is to generate a keystream by just encrypting a counter: $K_i = \{IV + i\}_K$. As before, this is then added to the plaintext to get ciphertext (so it's also vulnerable to attacks in depth).

Another problem that this mode solves when using a 64-bit block cipher such as DES or triple-DES on a very high-speed link is cycle length. An n-bit block cipher in OFB mode will typically have a cycle length of $2^{n/2}$ blocks, after which the birthday theorem will see to it that the keystream will start to repeat. (Once we have a little more than 2^{32} 64-bit values, the odds are that two of them will match.) In CBC mode, too, the birthday theorem ensures that after about $2^{n/2}$ blocks, we will start to see repeats. Counter-mode encryption, however, has a guaranteed cycle length of 2^n rather than $2^{n/2}$.

5.5.5 Cipher Feedback

Cipher feedback, or CFB, mode is another kind of stream cipher. It was designed to be self-synchronizing, in that even if we get a burst error and drop a few bits, the system will recover synchronization after one block length. This is achieved by using our block cipher to encrypt the last n-bits of ciphertext, then adding one of the output bits to the next plaintext bit.

With decryption, the reverse operation is performed, with ciphertext feeding in from the right, as shown in Figure 5.15. Thus, even if we get a burst error and drop a few bits, as soon as we've received enough ciphertext bits to fill up the shift register, the system will resynchronize.

Cipher feedback is not used much any more. It is a specialized mode of operation for applications such as military HF radio links, which are vulnerable to fading, in the days when digital electronics were relatively expensive. Now that silicon is cheap, people use dedicated link-layer protocols for synchronization and error correction rather than trying to combine them with the cryptography.

5.5.6 Message Authentication Code

The next official mode of operation of a block cipher is not used to encipher data, but to protect its integrity and authenticity. This is the *message authentication code*, or MAC. To compute a MAC on a message using a block cipher, we encrypt it using CBC mode and throw away all the output ciphertext blocks except the last one; this last block is the MAC. (The intermediate results are kept secret in order to prevent splicing attacks.)

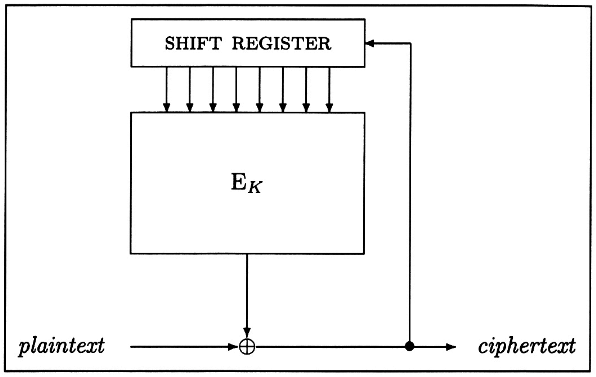


Figure 5.15 Ciphertext feedback mode (CFB).

This construction makes the MAC depend on all the plaintext blocks as well as on the key. It is secure provided the message length is fixed; it can be shown that any attack on a MAC under these circumstances would give an attack on the underlying block cipher [87]. (If the message length is variable, you have to ensure that a MAC computed on one string can't be used as the IV for computing a MAC on a different string, so that an opponent can't cheat by getting a MAC on the composition of the two strings.)

In applications needing both integrity and privacy, the procedure is to first calculate a MAC on the message using one key, then CBC-encrypt it using a different key. If the same key is used for both encryption and authentication, the security of the latter is no longer guaranteed; cut-and-splice attacks are still possible.

There are other possible constructions of MACs: a common one is to use a hash function with a key, which we'll look at in more detail in Section 5.6.2. Before we do that, let's revisit hash functions.

5.6 Hash Functions

Section 5.4.3.1 showed how the Luby-Rackoff theorem enables us to construct a block cipher from a hash function. It's also possible to construct a hash function from a block cipher. (In fact, we can also construct hash functions and block ciphers from stream ciphers—therefore, subject to some caveats described in the next section, given any one of these three primitives, we can construct the other two.)

The trick is to feed the message blocks one at a time to the key input of our block cipher, and use it to update a hash value (which starts off at, say, $H_0 = 0$). In order to make this operation noninvertible, we add feedforward: the $(i - 1)$st hash value is exclusive-or'ed with the output of round i. This is our final mode of operation of a block cipher (Figure 5.16).

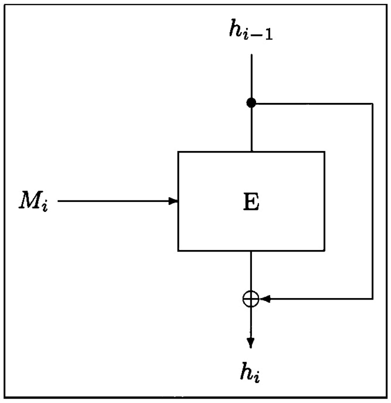


Figure 5.16 Feedforward mode (hash function).

5.6.1 Extra Requirements on the Underlying Cipher

The birthday effect makes another appearance here, in that if a hash function h is built using an n-bit block cipher, it is possible to find two messages $M_1 \neq M_2$ with $h(M_1) = h(M_2)$ (hash about $2^{n/2}$ messages M_i and look for a match). So, a 64-bit block cipher is not adequate, as the cost of forging a message would be of the order of 2^{32} messages, which is quite practical.

This is not the only way in which the hash function mode of operation is more demanding on the underlying block cipher than a mode such as CBC designed for confidentiality. A good illustration comes from a cipher called Treyfer, which was designed to encrypt data using as little memory as possible in the 8051 microcontrollers commonly found in consumer electronics and domestic appliances [819]. (It takes only 30 bytes of ROM.)

Treyfer "scavenges" its S-box by using 256 bytes from the ROM, which may be code, or may even—to make commercial cloning riskier—contain a copyright message. At each round, it acts on eight bytes of text with eight bytes of key by adding a byte of text to a byte of key, passing it through the S-box, adding it to the next byte, then rotating the result by one bit (Figure 5.17). This rotation deals with some of the problems that might arise if the S-box has uneven randomness across its bitplanes (for example, if it contains ascii text such as a copyright message). Finally, the algorithm makes up for its simple round structure and probably less-than-ideal S-box by having a large number of rounds (32 of them, in fact).

No attacks are known on Treyfer that prevent its use for confidentiality and for computing MACs. However, the algorithm does have a weakness that prevents its use in hash functions. It suffers from a *fixed-point attack.* Given any input, there is a fair chance we can find a key that will leave the input unchanged. We just have to look to see, for each byte of input, whether the S-box assumes the output that, when added to the byte on the right, has the effect of rotating it one bit to the right. If such outputs exist for each of the input bytes, then it's easy to choose key values that will leave the data unchanged after one round, and thus after 32 rounds. The probability that we can do this depends on the S-box.[3] This means that we can easily find collisions if Treyfer is used as a hash function. (In effect, hash functions have to be based on block ciphers that withstand *chosen key attacks*).

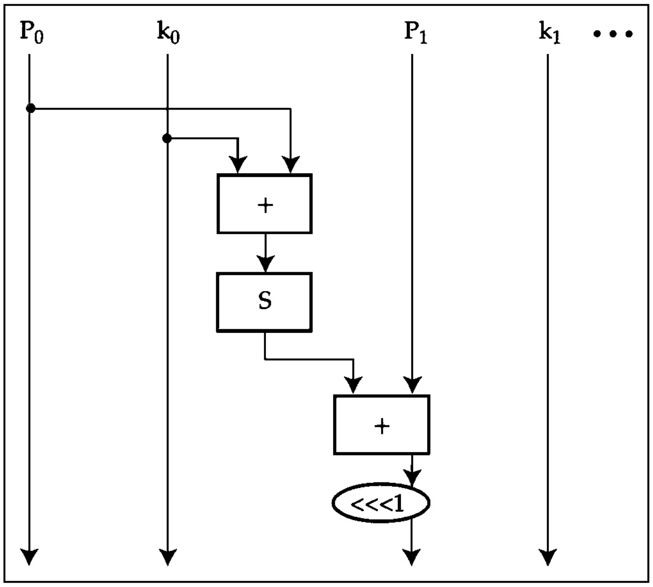


Figure 5.17 The basic component of the Treyfer block cipher.

5.6.2 Common Hash Functions and Applications

Algorithms similar to Treyfer have been used in hash functions in key management protocols in some pay-TV systems, but typically they have a modification to prevent fixed-point attacks, such as a procedure to add in the round number at each round, or to mix up the bits of the key in some way (a *key-scheduling* algorithm).

The three hash functions most commonly used in applications are all related, and are based on variants of a block cipher with a 512-bit key and a block size of either 128 or 160 bits. MD4 has three rounds and a 128-bit hash value; and a collision has recently been found for it [255]. MD5 has four rounds and a 128-bit hash value; while the U.S. Secure Hash Standard has five rounds and a 160-bit hash value. The block ciphers underlying these hash functions are similar; their round function is a complicated mixture of the register operations available on 32-bit processors [681]. It seems that SHA1 is a reasonable approximation to a pseudorandom function, as long as our opponents can't perform 2^{80} computations; because this number is coming within range of military intelligence agencies and large companies, the 256-bit and 512-bit versions of SHA have been introduced.

[3] Curiously, an S-box that is a permutation is always vulnerable, while a randomly selected one isn't quite so risky. In many cipher designs, S-boxes that are permutations are essential, or at least desirable. Treyfer is an interesting exception.

Hash functions have many uses. One is to compute MACs. A naive method would be to simply hash the message with a key: $\mathrm{MAC}_k(M) = h(k, M)$. However, the accepted way of doing this, called HMAC, uses an extra step in which the result of this computation is hashed again. The two hashing operations are done using variants of the key, derived by exclusive-or'ing them with two different constants. Thus, $\mathrm{HMAC}_k(M) = h(k \oplus A, h(k \oplus)B, M))$. A is constructed by repeating the byte `0 _ 36` as often as necessary, and B similarly from the byte `0 _ 5c`. The reason for this is that it makes collision finding much harder [474].

Another hash function use is to make commitments that are to be revealed later. For example, I might wish to timestamp a digital document in order to establish intellectual priority, but not be willing to reveal the contents yet. In that case, I can submit a hash of the document to a commercial timestamping service [364]. Later, when I reveal the document, the fact that its hash was timestamped at a given time establishes that I had written it by then.

Finally, before we go on to discuss asymmetric cryptography, there are two particular uses of hash functions that deserve mention: key updating and autokeying.

Key updating means that two or more principals who share a key pass it through a one-way hash function at agreed times: $K_i = h(K_{i-1})$. The point is that if an attacker compromises one of their systems and steals the key, he only gets the current key and is unable to decrypt back traffic. This property is known as *backward security.*

Autokeying means that two or more principals who share a key hash it at agreed times with the messages they have exchanged since the last key change: $K_{i+1} = h(K_i, M_{i1}, M \ldots.)$. The point is that if an attacker compromises one of their systems and steals the key. then as soon as they exchange a message which he doesn't observe or guess, security will be recovered in that he can no longer decrypt their traffic. This property is known as *forward security*. It is used, for example, in EFT payment terminals in Australia [83, 85]. The use of asymmetric crypto allows a slightly stronger form of forward security, namely that as soon as a compromised terminal exchanges a message with an uncompromised one that the opponent doesn't control, then security can be recovered even if the message is in plain sight. I'll describe how this trick works next.

5.7 Asymmetric Crypto Primitives

The commonly used building blocks in *asymmetric cryptography*, that is, public key encryption and digital signature, are based on number theory. I'll give only a brief overview here, then in Part 2 where we discuss applications, I'll describe in more detail some of the mechanisms used. (If you find the description assumes too much mathematics, skip the following two sections and read the material from a cryptography textbook.)

The technique of asymmetric cryptography is to make the security of the cipher depend on the difficulty of solving a certain mathematical problem. The two problems used in most fielded systems are factorization (used in most commercial systems) and discrete logarithm (used in many military systems).

5.7.1 Cryptography Based on Factoring

The *prime numbers* are the positive whole numbers with no proper divisors; that is, the only numbers that divide a prime number are 1 and the number itself. By definition, 1 is not prime; so the primes are {2, 3, 5, 7, 11, ...}. The *fundamental theorem of arithmetic* states that each natural number greater than 1 factors into prime numbers in a way that is unique up to the order of the factors. It is easy to find prime numbers and multiply them together to give a composite number, but much harder to resolve a composite number into its factors. The largest composite product of two large random primes to have been factorized to date was 512 bits (155 digits) long; when such a computation was first done, it took several thousand MIPS-years of effort. Recently, however, some Swedish students managed to factor a 512-bit number surreptitiously to solve a challenge cipher, so 512-bit composite numbers are now no more 'secure' than 56-bit DES keys. However, it is believed that a similar number of 1024 bits length could not be factored without an advance in mathematics.

The algorithm commonly used to do public key encryption and digital signatures based on factoring is RSA, named after its inventors Ron Rivest, Adi Shamir, and Len Adleman [649]. It uses *Fermat's* (*little*) *theorem*, which states that for all primes p not dividing a, $a^{p-1} \equiv 1$ modulo p. (Proof: take the set {1, 2, ... $p - 1$} and multiply each of them modulo p by a, then cancel out $(p - 1)!$ each side.) Euler's function $\square(n)$ is the number of positive integers less than n with which it has no divisor in common; so if n is the product of two primes pq then $\square(n) = (p - 1)(q - 1)$ (the proof is similar).

The encryption key is a modulus N which is hard to factor (take $N = pq$ for two large randomly chosen primes p and q), plus a public exponent e that has no common factors with either $p - 1$ or $q - 1$. The private key is the factors p and q, which are kept secret. Where M is the message and C is the ciphertext, encryption is defined by:

$$C \equiv M^e \text{ modulo } N$$

Decryption is the reverse operation:

$$M \equiv \sqrt[e]{\mathrm{C}} \text{ modulo } N$$

Whoever knows the private key—the factors p and q of N—can easily calculate $\sqrt[e]{\mathrm{C}}$ (mod N). As $\square(N) = (p - 1)(q - 1)$, and e has no common factors with $\square(N)$, the key's owner can find a number d such that $de \equiv 1$(modulo $\square(N)$)—she finds the value of d separately, modulo $p - 1$ and $q - 1$, and combines the answers. Thus, $\sqrt[e]{\mathrm{C}}$ (modulo N) is now computed as C^d (modulo N), and decryption works because of Fermat's theorem:

$$C^d \equiv \{M^e\}^d \equiv M^{ed} \equiv M^{1+k\square(N)} \equiv M\,.M^{k\square(N)} \equiv M \square 1 \equiv M \text{ modulo } N$$

Similarly, the owner of the private key can operate on the message with this to produce a digital signature:

$$Sig_d(M) \equiv M^d \text{ modulo } N$$

and this signature can be verified by raising it to the power e mod N (thus, using e and N as the public signature verification key) and checking that the message M is recovered:

$$M \equiv (Sig_d(M))^e \text{ modulo } N$$

Neither RSA encryption nor signature is generally safe to use on its own. The reason is that, encryption being an algebraic process, it preserves certain algebraic properties. For example, if we have a relation such as $M_1M_2 = M_3$ that holds among plaintexts, then the same relationship will hold among ciphertexts $C_1C_2 = C_3$ and signatures $Sig_1Sig_2 = Sig_3$. This property is known as a *multiplicative homomorphism* (mathematicians describe a function that preserves mathematical structure as a *homomorphism*). The homomorphic nature of raw RSA means that it doesn't meet the random oracle model definitions of public key encryption or signature.

There are a number of standards that try to stop attacks based on homomorphic mathematical structure by setting various parts of the input to the algorithm to fixed constants or to random values. Many of them have been broken. The better solutions involve processing the message using hash functions as well as random nonces and padding before the RSA primitive is applied. For example, in *optimal asymmetric encryption padding* (OAEP), we concatenate the message M with a random nonce N, and use a hash function h to combine them:

$$C_1 = M \oplus h(N)$$

$$C_2 = N \oplus h(C_1)$$

In effect, this is a two-round Feistel cipher, which uses h as its round function. The result, the combination C_1, C_2, is then encrypted with RSA and sent. The recipient then computes N as $C_2 \oplus h(C_1)$ and recovers M as $C_1 \oplus h(N)$ [88].

With signatures, things are slightly simpler. In general, it's often enough to just hash the message before applying the private key: $Sig_d = [h(M)]^d$ (mod N). However, in some applications, one might wish to include further data in the signature block, such as a timestamp.

5.7.2 Cryptography Based on Discrete Logarithms

While RSA is used in most Web browsers in the SSL protocol, there are other products (such as PGP) and many government systems that base public key operations on discrete logarithms. These come in a number of flavors, some using "normal" arithmetic, while others use mathematical structures called *elliptic curves*. I'll explain the normal case, as the elliptic variants use essentially the same idea but the implementation is more complex.

A *primitive root* modulo p is a number whose powers generate all the nonzero numbers mod p; for example, when working modulo 7, we find that $5^2 = 25$, which reduces to 4 (modulo 7), then we can compute 5^3 as 5^2 x 5 or 4 x 5, which is 20, which reduces to 6 (modulo 7), and so on, as shown in Figure 5.18.

Thus, 5 is a primitive root modulo 7. This means that given any y, we can always solve the equation $y = 5^x$ (mod 7); x is then called the discrete logarithm of y modulo 7. Small examples like this can be solved by inspection, but for a large random prime number p, we do not know how to do this computation. So the mapping $f : x$ □ g^x (mod p) is a one-way function, with the additional properties that $f(x + y) = f(x)f(y)$ and

$$
\begin{array}{llll}
5^1 & & = 5 & \pmod 7 \\
5^2 = & 25 & \equiv 4 & \pmod 7 \\
5^3 \equiv & 4 \text{ x } 5 & \equiv 6 & \pmod 7 \\
5^4 \equiv & 6 \text{ x } 5 & \equiv 2 & \pmod 7 \\
5^5 \equiv & 2 \text{ x } 5 & \equiv 3 & \pmod 7 \\
5^6 \equiv & 3 \text{ x } 5 & \equiv 1 & \pmod 7
\end{array}
$$

Figure 5.18 Example of discrete logarithm calculations.

$f(nx) = f(x)^n$. In other words, it is a *one-way homomorphism*. As such, it can be used to construct digital signature and public key encryption algorithms.

5.7.2.1 Public Key Encryption: The Diffie-Hellman Protocol

To understand how discrete logarithms can be used to build a public key algorithm, bear in mind that we want a cryptosystem that does not need the users to start off with a shared secret key. Consider the following “classical” scenario.

Imagine that Anthony wants to send a secret to Brutus, and the only communications channel available is an untrustworthy courier (say, a slave belonging to Caesar). Anthony can take the message, put it in a box, padlock it, and get the courier to take it to Brutus. Brutus could then put his own padlock on it, too, and have it taken back to Anthony. Anthony in turn would remove his padlock, and have it taken back to Brutus, who would now at last open it.

Exactly the same can be done using a suitable encryption function that *commutes*, that is, has the property that $\{\{M\}_{KA}\}_{KB} = \{\{M\}_{KB}\}_{KA}$. Alice can take the message M and encrypt it with her key $K A$ to get $\{M\}_{KA}$ which she sends to Bob. Bob encrypts it again with his key KB getting $\{\{M\}_{KA}\}_{KB}$. But the commutativity property means that this is just $\{\{M\}_{KB}\}_{KA}$, so Alice can decrypt it using her key KA getting $\{M\}_{KB}$. She sends this to Bob and he can decrypt it with KB, finally recovering the message M. The keys KA and KB might be long-term keys if this mechanism were to be used as a conventional public-key encryption system, or they might be transient keys if the goal were to establish a key with forward secrecy.

How can a suitable commutative encryption be implemented? The one-time pad does commute, but is not suitable here. Suppose Alice chooses a random key xA and sends Bob M $\oplus$ xA while Bob returns M $\oplus$ xB and Alice finally sends him M $\oplus$ $xA \oplus xB$, then an attacker can simply exclusive-or these three messages together; as X $\oplus$ X = 0 for all X, the two values of xA and xB both cancel our leaving as an answer the plaintext M.

The discrete logarithm problem comes to the rescue. If we have found values of g and p such that the discrete log problem to the base g modulo p is hard, then we can use discrete exponentiation as our encryption function. For example, Alice chooses a random number xA, calculates g^{xA} modulo p and sends it, together with p, to Bob. Bob likewise chooses a random number xB and forms g^{xAxB} modulo p, which he passes back to Alice. Alice can now remove her exponentiation: using Fermat’s theorem, she calculates $g^{xB} = (g^{xAxB})^{p-xA}$ modulo p and sends it to Bob. Bob can now remove his exponentiation, too, and so finally gets hold of g. The security of this scheme depends on the difficulty of the discrete logarithm problem.

In practice, it is tricky to encode a message to be a primitive root; but there is a much simpler means of achieving the same effect. The first public key encryption scheme to be published, by Whitfield Diffie and Martin Hellman in 1976, uses g^{xAxB} modulo p as the key to a shared key encryption system. The values xA and xB can be the private keys of the two parties.

Let's see how this might work to provide a public-key encryption system. The prime p and generator g are typically common to all users. Alice chooses a secret random number xA, calculates $yA = g^{xA}$ and publishes it opposite her name in the company phone book. Bob does the same, choosing a random number xB and publishing $yB = g^{xB}$. In order to communicate with Bob, Alice fetches y_B from the phone book, forms yB^{xA} which is of course g^{xAxB}, and uses this to encrypt the message to Bob. On receiving it, Bob looks up Alice's public key y_A and forms yA^{xB} which is also equal to g^{xAxB}, so he can decrypt her message

Slightly more work is needed to provide a full solution. Some care is needed when choosing the parameters p and g; and there are several other details that depend on whether we want properties such as forward security. Variants on the Diffie-Hellman theme include the U.S. government *key exchange algorithm* (KEA) [577], used in network security products such as the Fortezza card, and the so-called Royal Holloway protocol, which is used by the U.K. government [50].

The biggest problem with such systems is how to be sure that you've got a genuine copy of the phone book, and that the entry you're interested in isn't out of date. I'll discuss that in Section 5.7.4.

5.7.2.2 Key Establishment

Mechanisms for providing forward security in such protocols are of independent interest. As before, let the prime p and generator g be common to all users. Alice chooses a random number RA, calculates g^{RA} and sends it to Bob; Bob does the same, choosing a random number RB and sending g^{RB} to Alice; they then both form $g^{RA.RB}$, which they use as a session key.

Alice and Bob can now use the session key $g^{RA.RB}$ to encrypt a conversation. They have managed to create a shared secret "out of nothing." Even if an opponent had obtained full access to both their machines before this protocol was started, and thus knew all their stored private keys, then, provided some basic conditions were met (e.g., that their random-number generators were not predictable), the opponent still could not eaves-drop on their traffic. This is the strong version of the forward security property referred to in Section 5.6.2. The opponent can't work forward from knowledge of previous keys that he might have obtained. Provided that Alice and Bob both destroy the shared secret after use, they will also have backward security; an opponent who gets access to their equipment subsequently cannot work backward to break their old traffic.

But this protocol has a small problem: although Alice and Bob end up with a session key, neither of them has any idea with whom they share it.

Suppose that in our padlock protocol, Caesar has just ordered his slave to bring the box to him instead; he places his own padlock on it next to Anthony's. The slave takes the box back to Anthony, who removes his padlock, and brings the box back to Caesar who opens it. Caesar can even run two instances of the protocol, pretending to Anthony that he's Brutus and to Brutus that he's Anthony. One fix is for Anthony and Brutus to apply their seals to their locks.

The same idea leads to a middleperson attack on the Diffie-Hellman protocol unless transient keys are authenticated. Charlie intercepts Alice's message to Bob and replies to it; at the same time, he initiates a key exchange with Bob, pretending to be Alice. He ends up with a key $g^{RA.RC}$, which he shares with Alice, and another key $g^{RC.RB}$, which he shares with Bob. As long as he continues to sit in the middle of the network and translate the messages between them, they may have a hard time detecting that their communications are being compromised.

In one secure telephone product, the two principals would read out an eight-digit hash of the key they had generated, and check that they had the same value, before starting to discuss classified matters. A more general solution is for Alice and Bob to sign the messages that they send to each other.

Finally, discrete logarithms and their analogues exist in many other mathematical structures; thus, for example, *elliptic curve cryptography* uses discrete logarithms on an elliptic curve—a curve given by an equation such as $y^2 = x^3 + ax + b$. The algebra gets somewhat more complex, but the basic underlying ideas are the same.

5.7.2.3 Digital Signature

Suppose that the base p and the generator g (which may or may not be a primitive root) are public values chosen in some suitable way, and that each user who wishes to sign messages has a private signing key X and a public signature verification key $Y = g^X$. An *ElGamal signature scheme* works as follows: choose a message key k at random, and form $r = g^k$ (modulo p). Now form the signature s using a linear equation in k, r, the message M, and the private key X. There are a number of equations that will do; the particular one that happens to be used in ElGamal signatures is:

$$rX + sk = M \text{ modulo } p - 1$$

So s is computed as $s = (M - rX)/k$; this is done modulo $\square(p)$. When both sides are passed through our one-way homomorphism $f(x) = g^x$ modulo p we get:

$$g^{rX} g^{sk} \equiv g^M \text{ modulo } p$$

or

$$Y^r r^s \equiv g^M \text{ modulo } p$$

An ElGamal signature on the message M consists of the values r and s, and the recipient can verify it using the above equation.

A few details need to be sorted out to get a functional digital signature scheme. For example, bad choices of p or g can weaken the algorithm; and we will want to hash the message M using a hash function so that we can sign messages of arbitrary length, and so that an opponent can't use the algorithm's algebraic structure to forge signatures on messages that were never signed. Having attended to these details and applied one or two optimizations, we get the *Digital Signature Algorithm* (DSA) which is a U.S. standard and widely used in government applications.

DSA (also known as DSS, for Digital Signature Standard) assumes a prime p of typically 1024 bits, a prime q of 160 bits dividing $(p - 1)$, an element g of order q in the integers modulo p, a secret signing key x, and a public verification key $y = g^x$. The signature on a message M, $Sig_x(M)$, is (r, s), where:

$$r \equiv (g^k \text{ (modulo } p)) \text{ modulo } q$$
$$s \equiv (h(M) - xr)/k \text{ modulo } q$$

The hash function used here is SHA1.

DSA is the classic example of a randomized digital signature scheme without message recovery.

5.7.3 Special-Purpose Primitives

Researchers have discovered a large number of public key and signature primitives with special properties; I'll describe only the two that appear to have been fielded to date: threshold signatures and blind signatures.

Threshold signatures are a mechanism whereby a signing key (or for that matter a decryption key) can be split up among n principals so that any k out of n can sign a message (or decrypt one). For $k = n$, the construction is easy. With RSA, for example, we can split up the private decryption key d as $d = d_1+d_2+\ldots+d_n$. For $k < n$ it's slightly more complex (but not much) [246]. Threshold signatures are used in systems where a number of servers process transactions independently and vote independently on the outcome; they may also be used to implement business rules such as "a check may be signed by any two of the seven directors."

Blind signatures can be used to make a signature on a message without knowing what the message is. For example, if I am using RSA, I can take a random number R, form R^eM (mod n), and give it to the signer, who computes $(R^eM)^d = R.M^d$ (modulo n). When he or she gives this back to me, I can divide out R to get the signature M^d. The possible application is in *digital cash*; a bank might agree to honor for \$10 any string M with a unique serial number and a specified form of redundancy, bearing a signature that verified as correct using the public key (e,n). Such a string is known as a *digital coin*. The blind signature protocol shows how a customer can get a bank to sign a coin without the banker knowing its serial number. The effect is that the digital cash can be anonymous for the spender. (There are a few more details that need to be sorted out, such as how to detect people who spend the same coin twice; but these are fixable.) Blind signatures and digital cash were invented by Chaum [178], along with much other supporting digital privacy technology covered in Chapter 20 [177].

Researchers continue to suggest new applications for specialist public key mechanisms. A strong candidate is in online elections, where one requires a particular mixture of anonymity and accountability.

5.7.4 Certification

Now that we can do public key encryption and digital signature, we need some mechanism to bind users to keys. The approach proposed by Diffie and Hellman when they invented digital signatures was to have a directory of the public keys of a system's authorized users, such as a phone book. A more common solution, due to Loren Kohnfelder, is for a *certification authority* (CA) to sign the user's public encryption and/or signature verification keys giving certificates that contain the user's name, attributes such as authorizations, and public keys. The CA might be run by the local system administrator; or it might be a third-party service such as Verisign whose business is to sign public key certificates after checking that they belong to the principals named in them.

A certificate might be described symbolically as:

$$C_A = Sig_{K_S}(T_S, L, A, K_A, V_A)$$

where (using the same notation as with Kerberos) T_S is the certificate's starting date and time, L is the length of time for which it is valid, A is the user's name, K_A is her public encryption key, and V_A is her public signature verification key. In this way, only the administrator's public signature verification key needs to be communicated to all principals in a trustworthy manner.

Distributed system aspects of certification are covered in Chapter 6, "Distributed Systems"; e-commerce applications in Chapter 19, "Protecting E-Commerce Systems"; and the policy aspects in Chapter 21, "E-Policy." At this stage I'll merely point out that the protocol aspects are much harder than they look.

One of the first proposed public key protocols was due to Dorothy Denning and Giovanni Sacco, who in 1981 proposed that two users, say Alice and Bob, set up a shared DES key K_{AB} as follows. When Alice first wants to communicate with Bob, she goes to the certification authority and gets current copies of public key certificates for herself and Bob. She then makes up a key packet containing a timestamp T_A, a session key K_{AB} and a signature, which she computes on these items using her private signing key. She then encrypts this whole bundle under Bob's public encryption key and ships it off to him. Symbolically:

$$A \rightarrow B : C_A,\ C_B,\ \{T_A,\ K_{AB},\ Sig_{K_A}(T_A,\ K_{AB})\}_{K_B}$$

In 1994, Martín Abadi and Roger Needham pointed out that this protocol is fatally flawed [2]. Bob, on receiving this message, can masquerade as Alice for as long as Alice's timestamp T_A remains valid! To see how, suppose that Bob wants to masquerade as Alice to Charlie. He goes to Sam and gets a fresh certificate C_C for Charlie, then strips off the outer encryption $\{...\}_{K_B}$ from message 3 in the preceding protocol. He now re-encrypts the signed key packet $T_A,\ K_{AB},\ Sig_{K_A}(T_A,\ K_{AB})$ with Charlie's public key—which he gets from C_C—and makes up a bogus message 3:

$$B \rightarrow C : C_A,\ C_C,\ \{T_A,\ K_{AB},\ Sig_{K_A}(T_A,\ K_{AB})\}_{K_C}$$

It is actually quite alarming that such a simple protocol—essentially, a one-line program—should have such a serious flaw in it and remain undetected for so long. With a normal program of only a few lines of code, you might expect to find a bug in it by looking at it for a minute or two. In fact, public key protocols, if anything, are harder to design than protocols using shared key encryption, as they are prone to subtle and pernicious middleperson attacks. This further motivates the use of formal methods to prove that protocols are correct.

Often, the participants' names aren't the most important things which the authentication mechanism has to establish. In the STU-III secure telephone used by the U.S.

government and defense contractors, there is a protocol for establishing transient keys with forward and backward security; to exclude middleperson attacks, users have a *crypto ignition key*, a portable electronic device that they can plug into the phone to identify not just their names, but their security clearance levels. In general, books on the topic tend to talk about identification as the main goal of authentication and key management protocols; but in real life, it's usually authorization that matters. This is more complex, as it starts to introduce assumptions about the application into the protocol design. (In fact, the NSA security manual emphasizes the importance of always knowing whether there is an uncleared person in the room. The STU-III design is a natural way of extending this to electronic communications.)

One serious weakness of relying on public key certificates is the difficulty of getting users to understand all their implications and to manage them properly, especially where they are not an exact reimplementation of a familiar manual control system [224]. Many other things can go wrong with certification at the level of systems engineering as well, and we'll look at these in the next chapter.

5.7.5 The Strength of Asymmetric Cryptographic Primitives

To provide the same level of protection as a symmetric block cipher, asymmetric cryptographic primitives are believed to require at least twice the block length. Elliptic curve systems appear to achieve this; a 128-bit elliptic scheme could be about as hard to break as a 64-bit block cipher with a 64-bit key. The commoner schemes, based on factoring and discrete log, are less robust because there are shortcut attack algorithms such as the number field sieve, which exploit the fact that some integers are *smooth*, that is, they have a large number of small factors. At the time of writing, the number field sieve has been used to attack keys up to 512 bits, a task comparable in difficulty to performing keysearch on 56-bit DES keys. The current consensus is that private keys for RSA and for standard discrete log systems should be at least 1024 bits long, while 2048 bits gives some useful safety margin against mathematicians making significant improvements in algorithms.

There has been some publicity recently about *quantum computers*. These are devices that perform a large number of computations simultaneously using superposed quantum states. Peter Shor has shown that if a sufficiently large quantum computer can be built, then both factoring and discrete logarithm computations will become easy. So far only very small quantum computers can be built, and many people are sceptical about whether the technology can be made to work well enough to threaten real systems. In the event that it can, asymmetric cryptography may have to be abandoned. So it is fortunate that many of the things that are currently done with asymmetric mechanisms can also be done with symmetric ones; thus many authentication protocols can be redesigned to use variants on Kerberos.

5.8 Summary

Many ciphers fail because they're used improperly, so we need a clear model of what a cipher does. The random oracle model is useful here: we assume that each new value returned by the encryption engine is random in the sense of being statistically independent of all the outputs seen before.

Block ciphers for symmetric key applications can be constructed by the careful combination of substitutions and permutations; for asymmetric applications such as public key encryption and digital signature one uses number theory. In both cases, there is quite a large body of mathematics to guide us. Other kinds of ciphers—stream ciphers and hash functions—can be constructed from block ciphers by using them in suitable modes of operation. These have different error propagation, pattern concealment, and integrity protection properties.

The basic properties the security engineer needs to understand are not too difficult to grasp, though there are some subtle things that can go wrong. In particular, it is surprisingly hard to build systems that are robust even when components fail (or are encouraged to), and where the cryptographic mechanisms are well integrated with other measures such as access control and physical security. We'll return to this repeatedly in later chapters.

Research Problems

There are many active threads in cryptography research. Many of them are where crypto meets a particular branch of mathematics (number theory, algebraic geometry, complexity theory, combinatorics, graph theory, and information theory). The empirical end of the business is concerned with designing primitives for encryption, signature, and composite operations, and that perform reasonably well on available platforms. The two meet in the study of subjects ranging from linear and differential cryptanalysis to attacks on public key protocols. Research is more driven by the existing body of knowledge than by applications, though there are exceptions: copyright protection concerns have been a stimulus, and so has the recent competition to find an Advanced Encryption Standard.

The best way to get a flavor of what's going on is to read the last few years' proceedings of research conferences, such as Crypto, Eurocrypt, Asiacrypt and Fast Software Encryption, all published by Springer-Verlag in the *Lecture Notes on Computer Science* (LNCS) series.

Further Reading

The classic papers by Diffie and Hellman [248] and by Rivest, Shamir, and Adleman [649] are the closest to required reading on this subject. The most popular modern introduction is Bruce Schneier's *Applied Cryptography* [681], which covers a lot of ground at a level a nonmathematician can understand, and which has C source code for a number of algorithms. The *Handbook of Applied Cryptography*, by Alfred Menezes, Paul von Oorschot and Scott Vanstone [544], is the closest to a standard reference book on the mathematical detail.

More specialized texts include a book by Eli Biham and Adi Shamir [102], which is the standard reference on differential cryptanalysis; the best explanation of linear cryptanalysis may be in a textbook by Doug Stinson [738]; the modern theory of block ciphers can be found developing in the papers of the *Fast Software Encryption* conference series during the 1990s (the proceedings are published by Springer-Verlag in the LNCS series). The original book on modes of operation is Carl Meyer and Steve Matyas [548]. Neal Koblitz has a good basic introduction to the mathematics behind public key cryptography [463]; the number field sieve is described in [497]; while quantum factoring is described in [698].

There's a shortage of good books on the random oracle model and on theoretical cryptology in general; all the published texts I've seen are very technical and heavy going. Probably the most well-regarded source is a book being written by Oded Goldreich: the online fragments of this can be found at [342]. If you need something with an ISBN, try his lecture notes on 'Modern Cryptography, Probabilistic Proofs and Pseudorandomness' [343], which are pitched at the level of a postgraduate mathematics student. A less thorough but more readable introduction to randomness and algorithms is in [360]. Current research at the theoretical end of cryptology is found at the FOCS, STOC, Crypto, Eurocrypt, and Asiacrypt conferences.

The history of cryptology is fascinating, and so many old problems keep on recurring in modern guises that the security engineer should be familiar with it. The standard work is by David Kahn [428]; there are also compilations of historical articles from *Cryptologia* [229, 227, 228], as well as several books on the history of cryptology during World War II [188, 429, 523, 800]. The NSA Museum at Fort George Meade, Maryland, is also worth a visit, as is the one at Bletchley Park in England.

Finally, no chapter that introduces public key encryption would be complete without a mention that, under the name of 'non-secret encryption,' it was first discovered by James Ellis in about 1969. However, as Ellis worked for GCHQ—Britain's Government Communications Headquarters, the equivalent of the NSA—his work remained classified. The RSA algorithm was then invented by Clifford Cocks, and also kept secret. This story is told in [267]. One effect of the secrecy was that their work was not used: although it was motivated by the expense of Army key distribution, Britain's Ministry of Defence did not start building electronic key distribution systems for its main networks until 1992. It should also be noted that the classified community did not pre-invent digital signatures; they remain the achievement of Whit Diffie and Martin Hellman.