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.

`OYAN 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,

```
abcdefghijklmnopqrstuvwxyz
SECURITYABDFGHJKLMNOPQVWXZ
```

**Figure 5.1** Monoalphabetic substitution cipher.

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 digram (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.

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:* | runrunrunrunrunrunrunrunrun |
| *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 `wggsbtdefj`, 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, it 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 217.

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

| | 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.

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.

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 distinguishable 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

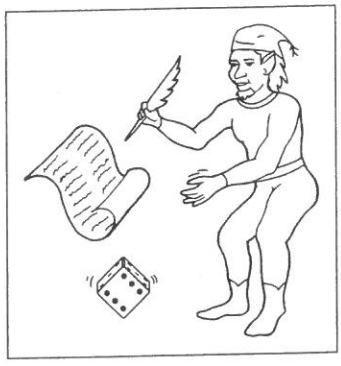

**Figure 5.9** The random oracle.

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.

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 $2^n$ 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 150,000 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 $\sqrt{N} \times \sqrt{N}$ 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

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

identify-friend-or-foe (IFF) systems, for example, common equipment has a response length of 48 to 80 bits.

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)