CHAPTER

5

Cryptography

ZHQM ZMGM ZMFM

— G Julius Caesar

KXJEY UREBE ZWEHE WRYTU HEYFS KREHE GOYFI WTTTU OLKSY CAJPO BOTEI ZONTX BYBWT GONEY CUZWR GDSON SXBOU YWRHE BAAHY USEDQ

— 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 3, '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 25 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 any training in cryptology; cryptologists will find little in it that 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 lots of 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 — and talk to some real experts. The security engineer Paul Kocher remarked, at a keynote speech at Crypto 2007, that you could expect to break any crypto product designed by 'any company that doesn't employ someone in this room'. There is a fair bit of truth in that.

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 [1232]. When Augustus Caesar ascended the throne, he changed the imperial cipher system so that 'C' was now written for

'A', 'D' for 'B' etcetera. In modern terminology, we would say that he changed the key from *'D'* to *'C'*. Remarkably, a similar code was used by Bernardo Provenzano, allegedly the *capo di tutti capi* of the Sicilian mafia, who wrote '4' for 'a, '5' for 'b' and so on. This led directly to his capture by the Italian police in 2006 after they intercepted and deciphered some of his messages [1034].

The Arabs generalised this idea to the *monoalphabetic substitution*, in which a keyword is used to permute the cipher alphabet. We will write the plaintext in lower case letters, and the ciphertext in upper case, as shown in Figure 5.1:

```
abcdefghijklmnopqrstuvwxyz
SECURITYABDFGHJKLMNOPQVWXZ
```

Figure 5.1: Monoalphabetic substitution cipher

`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,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

This 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 is into the range [0, ... , 25], that is, [A, ... , Z]. Mathematicians write this as

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

So, for example, when we add P (15) to U (20) we get 35, which we reduce to 9 by subtracting 26. 8 corresponds to J, so the encryption of P under the key U (and of U under the key P) is J. So in this notation, Julius Caesar's system used a fixed key $K = D$[1], 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:

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

Figure 5.2: Vigenère (polyalphabetic substitution cipher)

A number of people appear to have worked out how to solve polyalphabetic ciphers, from the womaniser Giacomo Casanova to the computing pioneer Charles Babbage. However the first published solution was in 1863 by Friedrich Kasiski, a Prussian infantry officer [695]. 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, and then repeat the process for the second and third letters of the key.

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 1 [676]; its effect is that given any ciphertext, and any plaintext of the same length, there is a key which 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 2 [836] 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 burnt.

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 the first ten letters of ciphertext were `DGTYI BWPJA`. This means that

[1]modulo 23, as the alphabet Caesar used wrote U as V, J as I, and had no W.

the first ten letters of the one-time pad were `wclnb tdefj`, as shown in Figure 5.3:

Plain	`heilhitler`
Key	`wclnbtdefj`
Cipher	`DGTYIBWPJA`

Figure 5.3: A spy's message

But once he's burnt the piece of silk with his key material, the spy can claim that he's actually a member of the anti-Nazi underground resistance, and 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:

Cipher	`DGTYIBWPJA`
Key	`wggsbtdefj`
Plain	`hanghitler`

Figure 5.4: What the spy claimed he said

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 for example that you wanted to get this spy into trouble, you could change the ciphertext to `DCYTI BWPJA` (Figure 5.5):

Cipher	`DCYTIBWPJA`
Key	`wclnbtdefj`
Plain	`hanghitler`

Figure 5.5: Manipulating the message to entrap the spy

During the Second World War, Claude Shannon proved 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; so the one-time pad is the only kind of system which offers perfect secrecy [1157, 1158].

The one-time pad is still used for some diplomatic and intelligence traffic, but it consumes as much key material as there is traffic and this 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 statistical randomness tests: it must also have the property that an opponent who gets his hands on even quite a lot of keystream bits should not be able to predict any more of them. I'll formalise this more tightly in the next section.

Stream ciphers are commonly used nowadays in hardware applications where the number of gates has to be minimised 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 the chapter on 'Telecom System Security'), and the shift register systems used in pay-per-view TV and DVD CSS (in the chapter on '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 Viscount Palmerston (later Prime Minister), on a napkin after dinner.

This cipher uses a 5 by 5 grid, in which we place the alphabet, permuted by the key word, and omitting the letter 'J' (see Figure 5.6):

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

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 the 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`'.

We can now encipher our specimen text as follows:

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

Variants of this cipher were used by the British army as a field cipher in World War 1, and by the Americans and Germans in World War 2. It's a substantial improvement on Vigenère as the statistics which 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 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 [512]. So 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. We'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, then it's likely to produce the same ciphertext every time a transaction involving it is encrypted with the same key.

This might allow an opponent to cut and paste parts of two different ciphertexts in order to produce a seemingly genuine but unauthorized transaction. Suppose a bad man worked for a bank's phone company, and could intercept their traffic. If he monitored an enciphered transaction that he knew said ''Pay IBM $10,000,000'' he might wire $1,000 to his brother causing the bank computer to insert another transaction saying ''Pay John Smith $1,000'', intercept this instruction, and make up a false instruction from the two ciphertexts that decrypted as ''Pay John Smith $10,000,000''. So unless the cipher block is as large as the message, the ciphertext will contain more than one block and we will usually need some way of binding the blocks 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-19th century, banks rapidly became its main users and developed systems for transferring money electronically. Of course, it isn't the money itself which 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.

In the nineteenth century, banks, telegraph companies and shipping companies developed *code books* that could not only 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 which mapped words or phrases to fixed-length groups of letters or numbers. So 'Please pay from our account with you no.' might become 'AFVCT'. A competing technology from the 1920s was *rotor machines*, mechanical cipher devices which 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 2.

The banks realised that neither mechanical stream ciphers nor code books protect 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, for the banks' purposes, was to use a code book but to 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 the bank has a code book with a table of numbers corresponding to payment amounts as in Figure 5.8:

	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

Now in order to authenticate a transaction for £376,514 we add together 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 in order 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.

This made such test key systems into *one-way functions* in that although given knowledge of the key it was possible to compute a test from a message, it was not possible to reverse the process and recover a message from a test — the test just did not contain enough information. Indeed, one-way functions had been around since at least the seventeenth century. The scientist Robert Hooke published in 1678 the sorted anagram 'ceiiinosssttuu' and revealed two years later that it was derived from 'Ut tensio sic uis' — 'the force varies as the tension', or what we now call Hooke's law for a spring. (The goal was to establish priority for the idea while giving him time to continue developing it.)

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 1980's. In several years working as a bank security consultant, and listening to elderly bank auditors' tales over lunch, I only ever 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 discuss the systems which replaced test keys, and the whole issue of how to tie cryptographic authentication mechanisms to procedural protection such as dual control, in the chapter on 'Banking and Bookkeeping'. For the meantime, 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. So, for example, many people publish on their web page a *public key* with which people can encrypt messages to send to them; the owner of the web page can then decrypt them using the corresponding *private key*.

There are some pre-computer examples of this too; perhaps the best is the postal service. You can send me a private message just as simply by addressing it to me and dropping it into a post box. Once that's done, I'm the only person who'll be able to read it. There are of course many things that can go wrong. You might get the wrong address for me (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. There are similar things that 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* and then anybody can check this using my public *signature verification key*. Again, there are pre-computer 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 grasp of the terminology of random oracles is needed to decipher 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 which a random function of the appropriate type would pass, in whatever 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' in that they're 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; and the mathematicians might be using a different model of computation from the computer scientists. In fact, there is a live debate among crypto researchers about whether formal models and proofs are valuable [724]. But crypto theory can help us sharpen our understanding of how ciphers behave and how they can safely be used.

We 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 our picture 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 only answer 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.

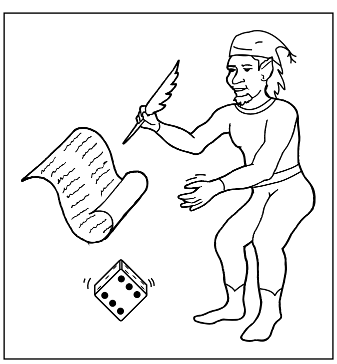

Figure 5.9: The random oracle

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 only need a single encryption/decryption device, rather than having one at each end of a communications link. For simplicity, let us assume it is this sort of application we are 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. A year later, she comes back to get the data decrypted and verified.

We shall now look at this model in more detail for various 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*, also known as *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 I mentioned in the chapter on security protocols, are used today in a number of authentication systems. They are used to compute checksums on files in forensic applications: presented with a computer seized from a suspect, you can compute hash values of the files to identify which files are already known (such as system files) and which are novel (such as user data). Hash values are also used as a means of checking the integrity of files, as they will change if a file is corrupted. In messaging applications, hashes are often known as *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 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 time-stamping service. However, if the document is still secret — for example an invention which 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.

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 gets lucky. A pseudorandom function will have the same properties, or they 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 a 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 [226]. 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 $h(x, k)$. So the eavesdropper must not be able to get any information about k, or compute $h(y, k)$ for a new input y. But the one-way function used by many 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 section 20.3.2.

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 can expect to find a match will be about the square root of this, namely $2^{n/2}$ hashes. This fact is of huge importance in security engineering, so let's look at it more closely.

5.3.1.2 The Birthday Theorem

The birthday theorem gets its name from the following problem. A maths teacher asks a typical class of 30 pupils what they think is the probability that two of them have the same birthday. Most pupils will intuitively think it's unlikely, and the maths teacher then asks the pupils to state their birthdays one after another. As the result seems unlikely to most people, it's also known as the 'birthday paradox'. The odds of a match exceed 50% once 23 pupils have been called.

The birthday theorem was first invented in the 1930's to count fish, which led to its also being known as *capture-recapture statistics* [1123]. Suppose there are N fish in a lake and you catch m of them, ring them and throw them back, then when you first catch a fish you've ringed already, m should be 'about' the square root of N. The intuitive reason why 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 $\sqrt{N}$ x $\sqrt{N}$ or N, which is what you need[2].

This theorem has many applications for the security engineer. For example, if we have a biometric system which 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 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.

There are some applications where 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 be generated by encrypting a counter.) So in 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

[2]More precisely, the probability that m fish chosen randomly from N fish are different is $\beta = N(N-1)\ldots(N-m+1)/N^m$ which is asymptotically solved by $N \simeq m^2/2log(1/\beta)$ [708].

are impractical for an opponent. The two most common are MD5, which has a 128-bit output and will thus require at most 2^{64} computations to break, and SHA1 with a 160-bit output and a work factor for the cryptanalyst of at most 2^{80}. However, collision search gives at best an upper bound on the strength of a hash function, and both these particular functions have turned out to be disappointing, with cryptanalytic attacks that I'll describe later in section 5.6.2. Collisions are easy to find for MD4 and MD5, while for SHA-1 it takes about 2^{60} computations to find a collision — something that a botnet of half a million machines should be able to do in a few days.

In any case, 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) assumptions are that the output must be long enough, and that the cryptographic design of the hash function must be sound.

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 turn it into 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 which provide a shortcut attack on the cipher.

One-time pad systems are a very close fit for our theoretical model, except in 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; a modern diplomatic system might use DVDs, 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 2, 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 US and UK decrypted large amounts of wartime Russian traffic afterwards 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 which two users share is K, they may concatenate it with a seed which is a message number N (or some other nonce) and 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. The nonce may be a separate pre-agreed key, or it may be generated at random and sent along with the ciphertext. However, the details of key management can be quite tricky, and the designer has to watch out for attacks in which a principal is tricked into synchronising on the wrong key. In effect, a protocol has to be designed to ensure that both parties can synchronise on the right working key even in the presence of an adversary.

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. (So 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. This has on the left a column of plaintexts and on the right a column of ciphertexts. When we ask the elf to encrypt a message, it checks in the left hand column to see if it has a record of it. If not, it uses the dice to generate a random ciphertext of the appropriate size (and which doesn't appear yet in the right hand column of the scroll), and then 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 which is independent of all the others. We can think of each key as corresponding to a different scroll. The intuitive idea is that a cipher machine should output the ciphertext given the plaintext and the key, and output the plaintext given the ciphertext and the key, but given only the plaintext and the ciphertext it should output nothing.

We will write a block cipher using the notation established for encryption in the chapter on protocols:

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

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* he 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 will have been 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. So you have to look carefully to find out exactly what kind of attack has been found, and what the parameters are. For example, the first major attack announced on the Data Encryption Standard 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 on your application. With a broadcast entertainment system, for example, a bad man 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. But there are surprisingly many applications where chosen-plaintext attacks are possible. Obvious ones include IFF, where the enemy can send challenges of his choice to any aircraft in range of one of his radars; and ATMs, where if you allow customers to change their PINs, an attacker can change his PIN through a range of possible values and observe the enciphered equivalents by wiretapping the line from the ATM to the bank. A more traditional example is diplomatic messaging systems, where it's been known for a host government to give an ambassador a message to transmit to his capital that's been specially designed to help the local cryptanalysts fill out the missing gaps in the ambassador's code book [676]. 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 a concern where the block cipher is used as a building block in the construction of a hash function (which we'll discuss below).

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 that 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; I'll deal with these later.

A common way of implementing public key encryption is the *trapdoor one-way permutation*. This is a computation which anyone can perform, but which 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. Let us state it formally nonetheless: a public key encryption primitive consists of a function which 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

1. Given KR, it is infeasible to compute KR^{-1} (so it's not possible to compute R either);
2. There is an encryption function $\{\ldots\}$ which, applied to a message M using the encryption key KR, will produce a ciphertext $C = \{M\}_{KR}$; and
3. There is a decryption function which, 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. There are several more demanding models than this, for example to analyze security in the case where the opponent can get ciphertexts of his choice decrypted, with the exception of the target ciphertext. But this will do for now.

5.3.5 Digital Signatures

The final cryptographic primitive which we'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. Applications include signing software updates, so that a PC can tell that an update to Windows was really produced by Microsoft rather than by a villain.

Signature schemes can be *deterministic* or *randomized*: in the first, computing a signature on a message will always give the same result and in the second, it will give a different result. (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 specialised, 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 which given a random input R will return two keys, σR (the private signing key) and VR (the public signature verification key) with the properties that

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

We can model a simple 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) only outputs 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 many countries: the stamp may consist of a 2-d barcode with a digital signature made by the postal meter and which contains information such as the value, the date and the sender's and recipient's post codes. We give some more detail about this at the end of section 14.3.2.

However, in the general case we do not need message recovery, as the message to be signed may be of arbitrary length and 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 towards 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, and then see in the next section 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

Claude Shannon suggested in the 1940's 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, and 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 which combined substitution and permutation circuits, and so were called SP-networks [681]. Figure 5.10 shows an SP-network with sixteen inputs, which we can imagine as the bits of a sixteen-bit number, and two layers of four-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.

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 four bit to four bit function takes only 4 x 2^4 or 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 four-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. it needs to have enough rounds, and
3. the S-boxes need to be suitably chosen.

5.4.1.1 Block Size

First, a block cipher which operated on sixteen bit blocks would have rather limited applicability, as an opponent could just build a dictionary of plaintext and ciphertext blocks as he observed them. The birthday theorem tells us that even if the input plaintexts were random, he'd expect to find a match as soon as he 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. So if we are using four-bit to four-but 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 have to 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, he could hold the rightmost 12 bits constant and try tweaking the leftmost four 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 Serpent and AES.

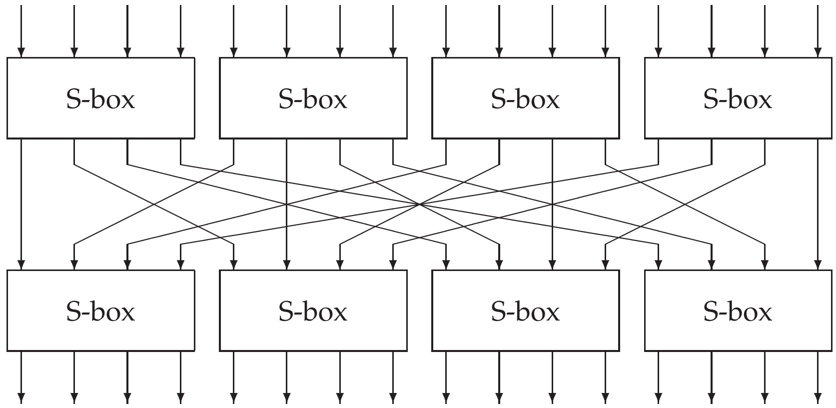

Figure 5.10: A simple 16-bit SP-network block cipher

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 the input would come through unchanged as the most significant bit of the output. If the same S-box were used in both rounds in the above cipher, then the most significant bit of the input would pass through to become the most significant bit of the output. This would usually be a bad thing; we certainly couldn't claim 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 one of the input was equal to bit two plus bit four of the output; more commonly, there will be linear approximations to an S-box which hold with a certain probability. *Linear cryptanalysis* [602, 843] 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' and then searching for ways to glue them together into an algebraic relation between input bits, output bits and key bits 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 value of M^2 for the best linear relationship is 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 [170, 602] 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 δ_i, δ_o such that an input change of δ_i will produce an output change of δ_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 which 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 though of pathological cases, such as if you 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*, but it is quite definitely possible against many systems [169]. There are also various specialised 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 have been 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. Use an off-the-shelf design that has been thoroughly scrutinized by experts, rather than rolling your own; and if there's a compelling reason to use a proprietary cipher (for example, if you want to use a patented design to stop other people copying a product) then get it reviewed by experts. Cipher design is not an amateur sport any more.

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, and 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 which makes both linear and differential attacks harder. After the final round, a further 128 bits of key material are added to give the plaintext. 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 [60] and can be downloaded, together with implementations in a number of languages, from [61].

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 [342]. 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.

AES 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. (AES 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.

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 a reasonable margin of safety; the best shortcut attacks known at the time of writing (2007) can tackle 7 rounds for 128-bit keys, and 9 rounds for 192- and 256-bit keys [16]. The general belief in the block cipher community is that even if advances in the state of the art do permit attacks on AES with the full number of rounds, they will be purely certificational attacks in that they will require infeasibly large numbers of texts. (AES'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 of known types — there is now a high level of confidence that AES is secure for all practical purposes. The NSA has since 2005 approved AES with 128-bit keys for protecting information up to SECRET and with 256-bit keys for TOP SECRET.

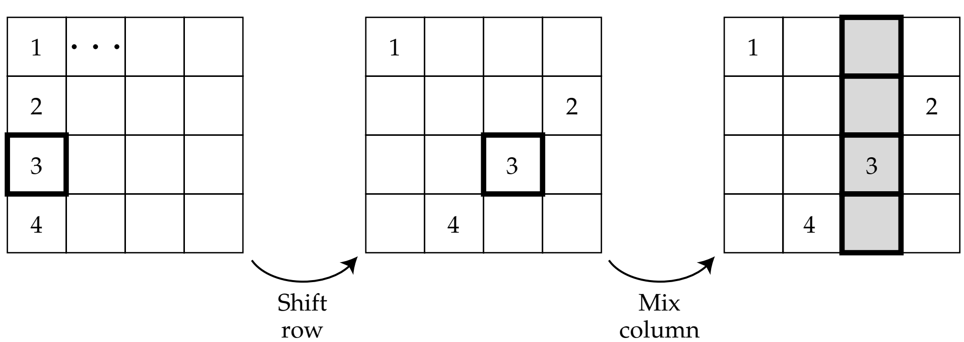

Figure 5.11: The AES linear transformation, illustrated by its effect on byte 3 of the input

Even although I was an author of Serpent which was an unsuccessful finalist in the AES competition (the winner 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 AES 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 I'll discuss at some length in the chapters on tamper-resistance and emission security. It does not make sense to implement Serpent as well, 'just in case AES 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 AES. (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 AES is Federal Information Processing Standard 197, and its inventors have written a book describing its design in detail [342]. Other information, from book errata to links to implementations, can be found on the AES Lounge web page [16].

One word of warning: the most likely practical attacks on a real implementation of AES include timing analysis and power analysis, both of which I discuss in Part II in the chapter on emission security. In timing analysis, the risk is that an opponent observes cache misses and uses them to work out the key. The latest versions of this attack can extract a key given the precise measurements of the time taken to do a few hundred cryptographic operations. In power analysis, an opponent uses measurements of the current drawn by the device doing the crypto — think of a bank smartcard that a customer places in a terminal in a Mafia-owned shop. The two overlap; cache misses cause a device like a smartcard to draw more power — and can also be observed on remote machines by an opponent who can measure the time taken to encrypt. The implementation details matter.

5.4.3 Feistel Ciphers

Many block ciphers use a more complex structure, which was invented by Feistel and his team while they were developing the Mark XII IFF in the late 1950's and early 1960's. Feistel then moved to IBM and founded a research group which produced the Data Encryption Standard, (DES) algorithm, which is still 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 $\oplus$ 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.

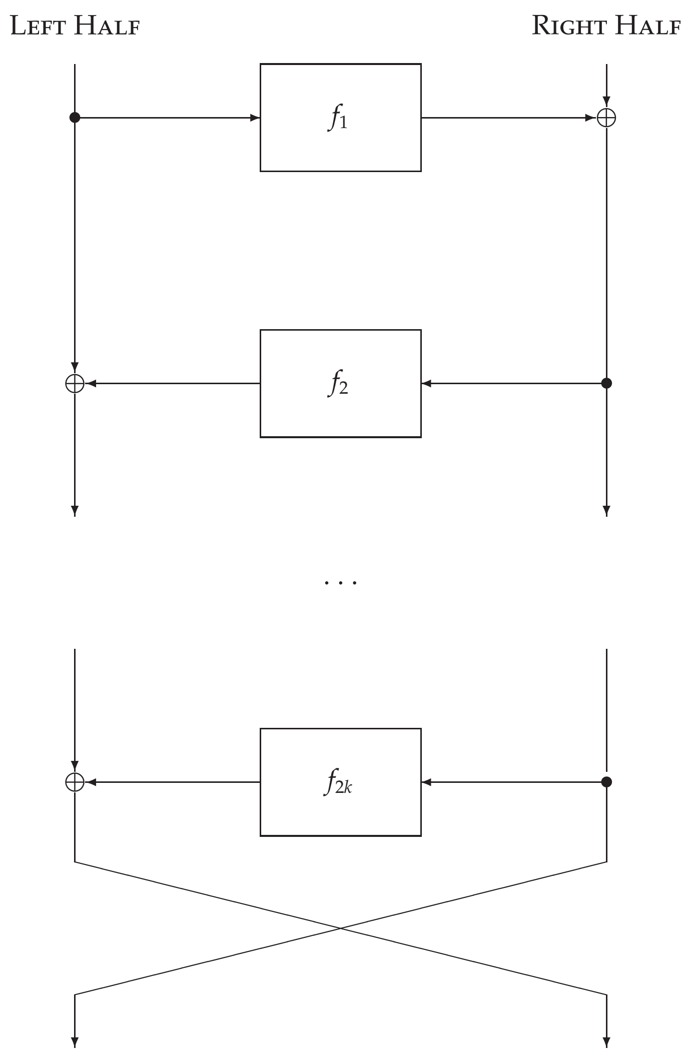

Figure 5.12: The Feistel cipher structure

A notation which you may see for the Feistel cipher is $\psi(f, g, h, \ldots)$ where f, g, h, ... are the successive round functions. Under this notation, the above cipher is $\psi(f_1, f_2, \ldots f_{2k-1}, f_{2k})$. The basic result that enables us to decrypt a Feistel cipher — and indeed the whole point of his design — is that:

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

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, and which also satisfies 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 key 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 $\psi(f_1, f_2, f_3)$ was indistinguishable from a random permutation under chosen plaintext attack, and this result was soon extended to show that $\psi(f_1, f_2, f_3, f_4)$ was indistinguishable under chosen plaintext/ciphertext attack — in other words, it was a pseudorandom permutation.

There are a number of technicalities we omit. 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: use four rounds of keyed hash in a Feistel network.

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. It 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 three operations:

- first, the block is expanded from 32 bits to 48;
- next, 48 bits of round key are mixed in using exclusive-or;
- the result is passed through a row of eight S-boxes, each of which takes a six-bit input and provides a four-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 Figure 5.13:

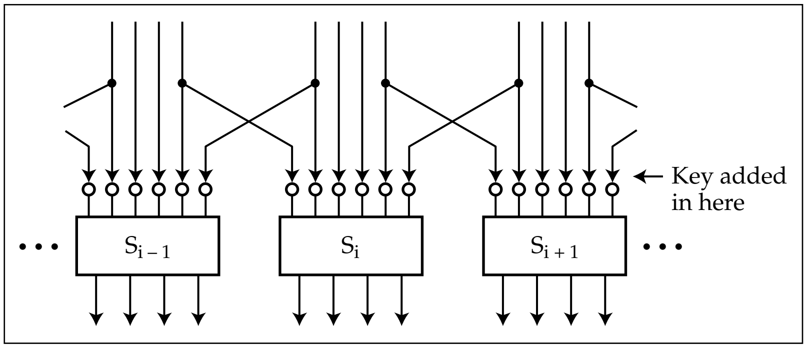

Figure 5.13: The DES round function

The round keys are derived from the user-supplied key by using each user key bit in twelve different rounds according to a slightly irregular pattern. A full specification of DES is given in [936]; code can be found in [1125] or downloaded from many places on the web.

DES was introduced in 1974 and caused some controversy. The most telling criticism 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. Whit Diffie and Martin Hellman argued in 1977 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 a bit over 9 hours, to find the key. They argued that such a machine could be built for $20 million dollars in 1977 [386]. IBM, whose scientists invented DES, retorted that they would charge the US government $200 million to build such a machine. (Perhaps both were right.)

During the 1980's, there were persistent rumors of DES keysearch machines being built by various intelligence agencies, but the first successful public keysearch attack took place in 1997. In a distributed effort organised over the net, 14,000 PCs 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 called Deep Crack for under $250,000 which broke a DES challenge in 3 days. It contained 1,536 chips run at 40MHz, 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 [423]. By 2006, Sandeep Kumar and colleagues at the universities of Bochum and Kiel built a machine using 120 FPGAs and costing $10,000, which could break DES in 7 days on average [755]. A modern botnet with half a million machines would take a few hours. So the key length of DES is now definitely inadequate, and banks have for some years been upgrading their payment systems.

Another criticism of DES was that, since IBM kept its design principles secret at the request of the US government, perhaps there was a 'trapdoor' which would give them easy access. However, the design principles were published in 1992 after differential cryptanalysis was invented and published [326]. Their story was that IBM had discovered these techniques in 1972, and the US National Security Agency (NSA) even earlier. IBM kept the design details secret at the NSA's request. We'll discuss the political aspects of all this in 24.3.9.1.

We now have a fairly thorough analysis of DES. The best known *shortcut attack*, that is, a cryptanalytic attack involving less computation than keysearch, is a linear attack using 2^{42} known texts. DES would be secure with more than 20 rounds, but for practical purposes its security is limited by its keylength. I don't know of any real applications where an attacker might get hold of even 2^{40} known texts. So the known shortcut attacks are not an issue. However, its growing vulnerability to keysearch makes DES unusable in its original form. If Moore's law continues, than by 2020 it might be possible to find a DES key on a single PC in a few months, so even low-grade systems such as taxi meters will be vulnerable to brute force-cryptanalysis. As with AES, there are also attacks based on timing analysis and power analysis, but because of DES's structure, the latter are more serious.

The usual way of dealing with the DES keysearch problem is to use the algorithm multiple times with different keys. Banking networks have largely moved to *triple-DES*, a standard since 1999 [936]. Triple-DES does an encryption, then a decryption, and then a further encryption, all done 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 backwards compatibility mode with legacy equipment. (Some banking systems use *two-key triple-DES* which sets $k_2 = k_0$; this gives an intermediate step between single and triple DES). New systems now use AES as of choice, but banking systems are deeply committed to using block ciphers with an eight-byte block size, because of the message formats used in the many protocols by which ATMs, point-of-sale terminals and bank networks talk to each other, and because of the use of block ciphers to generate and protect customer PINs (which I discuss in Chapter 10). Triple DES is a perfectly serviceable block cipher for such purposes for the foreseeable future.

Another way of preventing keysearch (and making power analysis harder) 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 afterwards. This composite cipher is known as DESX, and is used in the Win2K encrypting file system. Formally,

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

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 [717]. Whitened block ciphers are used in some applications.

5.5 Modes of Operation

In practice, how you use an encryption algorithm is often 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 standard modes of operation for using a block cipher on multiple blocks [944]. 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 I gave above as an example. 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 1980's, 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 corresponding to a plaintext of nulls. This gave 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 at 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 which 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 which 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. In fact, if an error is inserted into the ciphertext, it will affect only two blocks of plaintext on decryption, so if there isn't any integrity protection on the plaintext, an enemy can insert two-block garbles of random data at locations of his choice.

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 above.

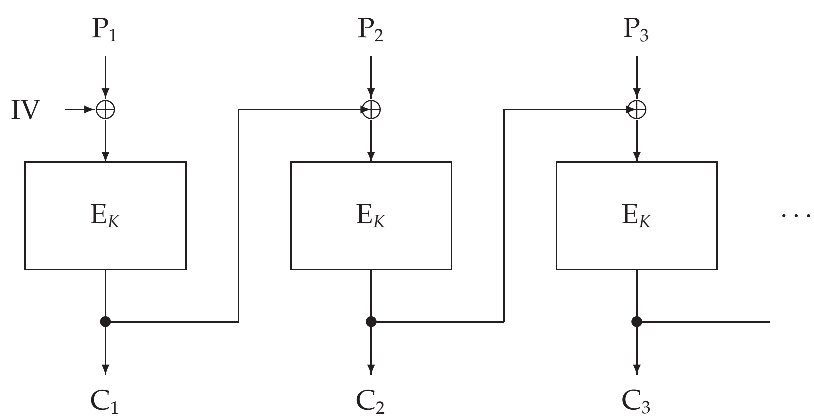

Figure 5.14: Cipher Block Chaining (CBC) mode

Writing IV for the initialization vector or seed, the i-th block of keystream will be given by

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

This is one 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 module 2 (and some old hand systems used addition modulo 26).

All additive stream ciphers have an important vulnerability: they fail to protect message integrity. I mentioned this in the context of the one-time pad in section 5.2.2 above, but it's important to realise 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, for example by a wiretap. 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. You know M_i for bytes 37–42, so you know K_i and can easily construct a modified message which 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 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 gigabit 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 this mode solves when using a 64-bit block cipher such as 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 starts to repeat. (Once we've a little over 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, and 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 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 much used any more. It was designed for use in 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.)

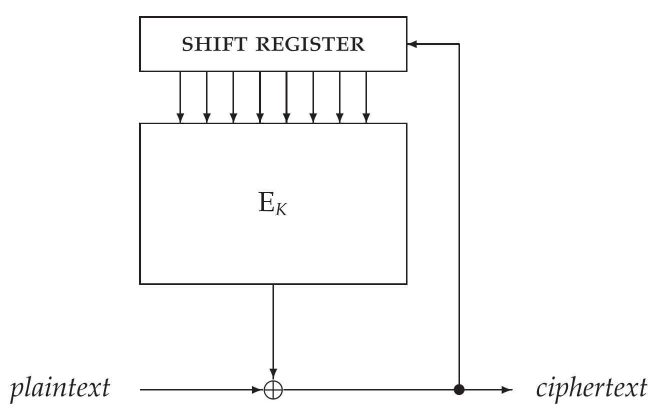

Figure 5.15: Ciphertext feedback mode (CFB)

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; Mihir Bellare, Joe Kilian and Philip Rogaway proved that any attack on a MAC under these circumstances would give an attack on the underlying block cipher [147].

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 order to fix this problem, NIST has standardised CMAC, in which a variant of the key is xor-ed in before the last encryption [945]. (CMAC is based on a proposal by Tetsu Iwata and Kaoru Kurosawa [649].)

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.

5.5.7 Composite Modes of Operation

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

Recently two further modes of operation have been tackled by NIST that combine encryption and authentication. The first is CCM, which combines counter-mode encryption with CBC-MAC authentication. The danger to watch for here is that the counter values used in encryption must not coincide with the initialisation vector used in the MAC; the standard requires that the formatting function prevent this [946].

The second combined mode is Galois Counter Mode (GCM), which has just been approved at the time of writing (2007). This interesting and innovative mode is designed to be parallelisable so that it can give high throughput on fast data links with low cost and low latency. As the implementation is moderately complex, and the algorithm was approved as this book was in its final edit, I don't include the details here, but refer you instead to the official specification [947]. The telegraphic summary is that the encryption is performed in a variant of counter mode; the resulting ciphertexts are also multiplied together with key material and message length information in a Galois field of 2^{128} elements to get an authenticator tag. The output is thus a ciphertext of the same length as the plaintext, plus a tag of typically 128 bits. The tag computation uses a *universal hash function* which comes from the theory of unconditionally-secure authentication codes; I'll describe this in Chapter 13, 'Nuclear Command and Control'.

Both CCM, and old-fashioned CBC plus CBC MAC, need a completely new MAC to be computed on the whole message if any bit of it is changed. However, the GCM mode of operation has an interesting incremental property: a new authenticator and ciphertext can be calculated with an amount of effort proportional to the number of bits that were changed. GCM is an invention of David McGrew and John Viega of Cisco; their goal was to create an authenticated encryption mode that is highly parallelisable for use in high-performance network hardware and that only uses one block cipher operation per block of plaintext, unlike CCM or the old-fashioned CBC plus CBC-MAC [862]. Now that GCM has been adopted as a standard, we might expect it to become the most common mode of operation for the encryption of bulk content.

5.6 Hash Functions

In section 5.4.3.1 I 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 — so, subject to some caveats I'll discuss 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 non-invertible, 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).

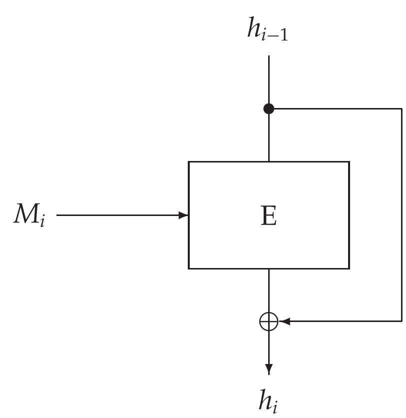

Figure 5.16: Feedforward mode (hash function)

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 $h(M_1) = h(M_2)$ with about $2^{n/2}$ effort (hash slightly more than that many 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. A 128-bit cipher such as AES may be just about adequate, and in fact the AACS content protection mechanism used in the next generation of DVDs uses 'AES-H', the hash function derived from AES in this way.

The birthday limit 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 [1371]. (It takes only 30 bytes of ROM.)

Treyfer 'scavenges' its S-box by using 256 bytes from the ROM, which may be code, or 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 and then rotating the result by one bit (see 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

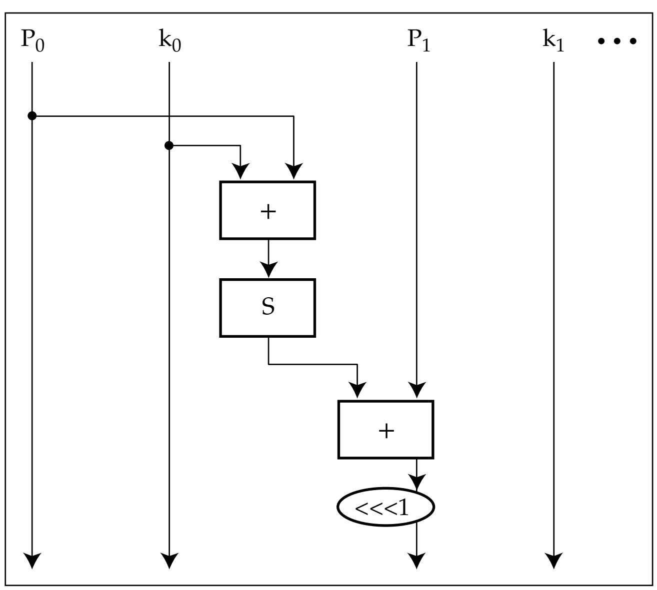

Figure 5.17: The basic component of the Treyfer block cipher

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).

No attacks are known on Treyfer which 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 which will leave the input unchanged. We just have to look to see, for each byte of input, whether the S-box assumes the output which, 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 which will leave the data unchanged after one round, and thus after 32. 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 which withstand *chosen-key attacks*.

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 most commonly used hash functions are all cryptographically suspect. They 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 was found for it in 1998 [394];
- MD5 has four rounds and a 128 bit hash value, and a collision was found for it in 2004 [1315, 1317];
- the US Secure Hash Standard has five rounds and a 160 bit hash value, and it was shown in 2005 that a collision can be found with a computational effort of 2^{69} steps rather than the 2^{80} that one would hope given its block size [1316].

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 [1125].

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

MD5 was broken by Xiaoyun Wang and her colleagues in 2004 [1315, 1317]; collisions can now be found easily, even between strings containing meaningful text and adhering to message formats such as those used for digital certificates. Wang seriously dented SHA the following year, providing an algorithm that will find collisions in only 2^{69} steps [1316]; and at the Crypto 2007 conference, the view was that finding a collision should cost about 2^{60}. Volunteers were being recruited for the task. So it appears that soon a collision will be found and SHA-1 will be declared 'broken'.

At the time of writing, the US National Institute of Standards and Technology (NIST) recommends that people use extended block-size versions of SHA, such as SHA-256 or SHA-512. The draft FIPS 180-3 allows, though discourages, the original SHA; it specifies SHA-256 and SHA-512, and also supports 224-bit and 384-bit hashes derived from SHA-256 and SHA-512 respectively by changing the initial values and truncating the output. The NSA specifies the use of SHA-256 or SHA-382 along with AES in its Suite B of cryptographic algorithms for defense use. NIST is also organising a competition to find a replacement hash function family [949].

Whether a collision-search algorithm that requires months of work on hundreds of machines (or a few days on a large botnet) will put any given application at risk can be a complex question. If bank systems would actually take a message composed by a customer saying 'Pay X the sum Y', hash it and sign it, then a weak hash function could indeed be exploited: a bad man could find two messages 'Pay X the sum Y' and 'Pay X the sum Z' that hashed to the same value, get one signed, and swap it for the other. But bank systems don't work like that. They typically use MACs rather than digital signatures on actual transactions, relying on signatures only in public-key certificates that bootstrap key-management protocols; and as the public-key certificates are generated by trusted CAs using fairly constrained algorithms, there isn't an opportunity to insert one text of a colliding pair. Instead you'd have to find a collision with an externally-given target value, which is a much harder cryptanalytic task.

Hash functions have many uses. One of them is to compute MACs. A naive method would be to simply hash the message with a key: $\text{MAC}_k(M) = h(k, M)$. 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 $\text{HMAC}_k(M) = h(k \oplus A, h(k \oplus B, M))$. A is constructed by repeating the byte `0x36` as often as necessary, and B similarly from the byte `0x5C`. Given a hash function that may be on the weak side, this is believed to make exploitable collisions harder to find [741]. HMAC is now FIPS 198, being replaced by FIPS 198-1.

Another use of hash functions 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 reveal the contents yet. In that case, I can submit a hash of the document to a commercial timestamping service [572]. 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. Again, an algorithm that generates colliding pairs doesn't break this, as you have to have the pair to hand when you do the timestamp. The moral, I suppose, is that engineers should be clear about whether a given application needs a hash function that's strongly collision-resistant.

But even though there may be few applications where the ability to find collisions could enable a bad guy to steal real money today, the existence of a potential vulnerability can still undermine a system's value. In 2005, a motorist accused of speeding in Sydney, Australia, was acquitted after the New South Wales Roads and Traffic Authority failed to find an expert to testify that MD5 was secure. The judge was ''not satisfied beyond reasonable doubt that the photograph [had] not been altered since it was taken'' and acquitted the motorist; this ruling was upheld on appeal the following year [964]. So even if a vulnerability doesn't present an engineering threat, it can still present a very real *certificational* threat.

Finally, before we go on to discuss asymmetric cryptography, there are two particular uses of hash functions which need 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. The chain of compromise is broken by the hash function's one-wayness. This property is also 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: $K_{+1}i = h(K_i, M_{i1}, M_{i2}, \ldots)$. 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. Again, the chain of compromise is broken. This property is known as *forward security*. It is used, for example, in EFT payment terminals in Australia [143, 145]. 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 which 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, and look in more detail at some of the mechanisms used in Part II where I discuss applications. (If you find the description assumes too much mathematics, I'd suggest you skip the following two sections and read up the material from a cryptography textbook.)

The technique is to make the security of the cipher depend on the difficulty of solving a certain mathematical problem. The two problems which are used in almost all fielded systems are factorization (used in most commercial systems) and discrete logarithm (used in many government 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 $\{2, 3, 5, 7, 11, \ldots\}$. 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 RSA-200, a 663-bit number (200 decimal digits), factored in 2005. This factorization was done on a number of PCs and took the equivalent of 75 years' work on a single 2.2GHz machine. It is possible for factoring to be done surreptitiously, perhaps using a botnet; in 2001, when the state of the art was factoring 512-bit numbers, such a challenge was set in Simon Singh's 'Code Book' and solved by five Swedish students using several hundred computers to which they had access [24]. By 2007, 512-bit factorization had entered into mainstream commerce. From 2003, Intuit had protected its Quicken files with strong encryption, but left a back door based on a 512-bit RSA key so that they could offer a key recovery service. Elcomsoft appears to have factored this key and now offers a competing recovery product.

It is believed that factoring an RSA modulus of 1024 bits would require a special-purpose machine costing in the range of \$10–50m and that would take a year for each factorization [781]; but I've heard of no-one seriously planning to build such a machine. Many physicists hope that a quantum computer could be built that would make it easy to factor even large numbers. So, given that Moore's law is slowing down and that quantum computers haven't arrived yet, we can summarise the state of the art as follows. 1024-bit products of two random primes are hard to factor and cryptographic systems that rely on them are at no immediate risk from low-to-medium budget attackers; NIST expects them to be secure until 2010, while an extrapolation of the history of factoring records suggests the first factorization will be published in 2018. So risk-averse organisations that want keys to remain secure for many years are already using 2048-bit numbers.

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. It uses *Fermat's (little) theorem*, which states that for all primes p not dividing a, $a^{p-1} \equiv 1 \pmod{p}$ (proof: take the set $\{1, 2, \ldots, p-1\}$ and multiply each of them modulo p by a, then cancel out $(p-1)!$ each side). Euler's function $\phi(n)$ 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 $\phi(n) = (p-1)(q-1)$ (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, say of 1024 bits each) 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

$$C \equiv M^e \pmod{N}$$

Decryption is the reverse operation:

$$M \equiv \sqrt[e]{C} \pmod{N}$$

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

$$C^d \equiv \{M^e\}^d \equiv M^{ed} \equiv M^{1+k\phi(N)} \equiv M.M^{k\phi(N)} \equiv Mx1 \equiv M \pmod{N}$$

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

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

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:

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

Neither RSA encryption nor signature is generally safe to use on its own. The reason is that, as encryption is 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*; a homomorphism is a function that preserves some mathematical structure. The homomorphic nature of raw RSA means that it doesn't meet the random oracle model definitions of public key encryption or signature.

Another problem with public-key encryption is that if the plaintexts are drawn from a small set, such as 'attack' or 'retreat', and the encryption process is known to the opponent, then he can precompute possible ciphertexts and recognise them when they appear. Specific algorithms also have specific vulnerabilities: with RSA, it's dangerous to use a small exponent e to encrypt the same message to multiple recipients, as this can lead to an algebraic attack. To stop the guessing attack, the low-exponent attack and attacks based on homomorphism, it's sensible to add in some randomness, and some redundancy, into a plaintext block before encrypting it. However, there are good ways and bad ways of doing this.

In fact, crypto theoreticians have wrestled for decades to analyze all the things that can go wrong with asymmetric cryptography, and to find ways to tidy it up. Shafi Goldwasser and Silvio Micali came up with formal models of *probabilistic encryption* in which we add randomness to the encryption process, and *semantic security*, which means that an attacker cannot get any information at all about a plaintext M that was encrypted to a ciphertext C, even if he is allowed to request the decryption of any other ciphertext C' not equal to C [536]. There are a number of constructions that give provable semantic security, but they tend to be too ungainly for practical use.

The common real-world solution is *optimal asymmetric encryption padding* (OAEP), where 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 that 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 $C_1 \oplus h(N)$ [148]. (This construction came with a security proof, in which a mistake was subsequently found [1167, 234], sparking a vigorous debate on the value of mathematical proofs in security engineering [724].) RSA Data Security, which for years licensed the RSA algorithm, developed a number of public-key cryptography standards; PKCS #1 describes OAEP [672].

With signatures, things are slightly simpler. In general, it's often enough to just hash the message before applying the private key: $Sig_d = [h(M)]^d \pmod{N}$; PKCS #7 describes simple mechanisms for signing a message digest [680]. However, in some applications one might wish to include further data in the signature block, such as a timestamp, or some randomness in order to make side-channel attacks harder.

Many of the things that have gone wrong with real implementations have to do with error handling. Some errors can affect cryptographic mechanisms directly. The most spectacular example was when Daniel Bleichenbacher found a way to break the RSA implementation in SSL v 3.0 by sending suitably chosen ciphertexts to the victim and observing any resulting error messages. If he can learn from the target whether a given c, when decrypted as c^d (mod n), corresponds to a PKCS #1 message, then he can use this to decrypt or sign messages [189]. Other attacks have depended on measuring the precise time taken to decrypt; I'll discuss these in the chapter on emission security. Yet others have involved stack overflows, whether by sending the attack code in as keys, or as padding in poorly-implemented standards. Don't assume that the only attacks on your crypto code will be doing cryptanalysis.

5.7.2 Cryptography Based on Discrete Logarithms

While RSA is used in most web browsers in the SSL protocol, and in the SSH protocol commonly used for remote login to computer systems, there are other products, and many government systems, which 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. 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 $5^2 \times 5$ or 4×5 which is 20, which reduces to 6 (modulo 7), and so on, as in Figure 5.18:

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

Figure 5.18: Example of discrete logarithm calculations

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 $f : x \rightarrow g^x \pmod{p}$ is a one-way function, with the additional properties that $f(x+y) = f(x)f(y)$ and $f(nx) = f(x)^n$. 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 — Diffie Hellman and ElGamal

To understand how discrete logarithms can be used to build a public-key encryption algorithm, bear in mind that we want a cryptosystem which 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. He 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 KA 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 the discrete log problem based on a primitive root modulo p is hard, then we can use discrete exponentiation as our encryption function. For example, Alice encodes her message as the primitive root g, 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 $g^{xB} = (g^{xAxB})^{(p-xA)} \pmod{p}$ 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, has a fixed primitive root g and 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 provide a public-key encryption system. The prime p and generator g are 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 x_B and publishing $yB = g^{xB}$. In order to communicate with Bob, Alice fetches yB from the phone book, forms yB^{xA} which is just 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 which depend on whether we want properties such as forward security. Variants on the Diffie-Hellman theme include the US government *key exchange algorithm* (KEA) [939], used in network security products such as the Fortezza card, and the so-called Royal Holloway protocol, which is used by the UK government [76].

Of course, one of the big problems with public-key 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 5.7.5.

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 R_A, calculates g^{R_A} and sends it to Bob; Bob does the same, choosing a random number R_B and sending g^{R_B} to Alice; they then both form $g^{R_A R_B}$, which they use as a session key (Figure 5.19).

$$
\begin{array}{ll}
A \rightarrow B: & g^{R_A} \pmod p \\
B \rightarrow A: & g^{R_B} \pmod p \\
A \rightarrow B: & \{M\}_{g^{R_A R_B}}
\end{array}
$$

Figure 5.19: The Diffie-Hellman key exchange protocol

Alice and Bob can now use the session key $g^{R_A R_B}$ 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 could still not eavesdrop on their traffic. This is the strong version of the forward security property to which I referred in section 5.6.2. The opponent can't work forward from knowledge of previous keys which 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 who they share it with.

Suppose that in our padlock protocol Caesar had just ordered his slave to bring the box to him instead, and placed 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.

With the vanilla Diffie-Hellman protocol, the same idea leads to a middleperson attack. 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 $g^{R_A R_C}$ which he shares with Alice, and another key $g^{R_B R_C}$ which he shares with Bob. So 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 compromised. The usual solution is to authenticate transient keys, and there are various possibilities.

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.

A few other details have to be got right, such as a suitable choice of the values p and g. There's some non-trivial mathematics behind this, which is best left to specialists. There are also many things that can go wrong in implementations — examples being software that will generate or accept very weak keys and thus give only the appearance of protection; programs that leak the key by the amount of time they take to decrypt; and software vulnerabilities leading to stack overflows and other nasties. Nonspecialists implementing public-key cryptography should consult up-to-date standards documents and/or use properly accredited toolkits.

5.7.2.3 Digital Signature

Suppose that the base p and the generator g 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 $r = g^k \pmod{p}$. 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

$$rX + sk = M$$

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

$$g^{rX} g^{sk} \equiv g^M$$

or

$$Y^r r^s \equiv g^M$$

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 more details need to be fixed up to get a functional digital signature scheme. As before, bad choices of p and g can weaken the algorithm. We will also 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 optimisations, we get the *Digital Signature Algorithm* (DSA) which is a US 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

$$r \equiv (g^k \pmod{p}) \pmod{q}$$

$$s \equiv (h(M) - xr)/k \pmod{q}$$

The hash function used here is SHA1.

DSA is the classic example of a randomized digital signature scheme without message recovery. The standard has changed somewhat with faster computers, as variants of the algorithm used to factor large numbers can also be used to compute discrete logarithms modulo bases of similar size[4]. Initially the prime p could be in the range 512–1024 bits, but this was changed to 1023–1024 bits in 2001 [941]; the proposed third-generation standard will allow primes p in the range 1024–3072 bits and q in the range 160–256 bits1 [942]. Further tweaks to the standard are also foreseeable after a new hash function standard is adopted.

[4]Discrete log efforts lag slightly behind, with a record set in 2006 of 440 bits.

5.7.3 Special Purpose Primitives

Researchers have discovered a large number of public-key and signature primitives with special properties. Two that have so far appeared in real products are threshold cryptography and blind signatures.

Threshold crypto is a mechanism whereby a signing key, or a decryption key, can be split up among n principals so that any k out of n can sign a message (or decrypt). For $k = n$ the construction is easy. With RSA, for example, you can split up the private key d as $d = d_1 + d_2 + \ldots + d_n$. For $k < n$ it's slightly more complex (but not much — you use the Lagrange interpolation formula) [382]. Threshold signatures are used in systems where a number of servers process transactions independently and vote independently on the outcome; they could also be used to implement business rules such as 'a check may be signed by any two of the seven directors'.

Blind signatures are a way of making a signature on a message without knowing what the message is. For example, if we are using RSA, I can take a random number R, form R^eM (mod n), and give it to the signer who computes $(R^eM)^d = R.M^d$ (mod n). When he gives this back to me, I can divide out R to get the signature M^d. Now you might ask why on earth someone would want to sign a document without knowing its contents, but there are indeed applications.

The first was in *digital cash*; a bank might want to be able to issue anonymous payment tokens to customers, and this has been done by getting it to sign 'digital coins' without knowing their serial numbers. In such a system, the bank might agree to honour 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). 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 technical details that need to be sorted out, such as how you detect people who spend the same coin twice; but these are fixable.) Blind signatures and digital cash were invented by Chaum [285], along with much other supporting digital privacy technology which I'll discuss later [284]. They were used briefly in pilot projects for road tolls in the Netherlands and for electronic purses in Brussels, but failed to take off on a broader scale because of patent issues and because neither banks nor governments really want payments to be anonymous: the anti-money-laundering regulations nowadays restrict anonymous payment services to rather small amounts. Anonymous digital credentials are now talked about, for example, in the context of 'identity management': the TPM chip on your PC motherboard might prove something about you (such as your age) without actually revealing your name.

Researchers continue to suggest new applications for specialist public key mechanisms. A popular candidate is in online elections, which require a particular mixture of anonymity and accountability. Voters want to be sure that their votes have been counted, but it's also desirable that they should not be able to prove which way they voted to anybody else; if they can, then vote-buying and intimidation become easier.

5.7.4 Elliptic Curve Cryptography

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 like $y^2 = x^3 + ax + b$. These curves have the property that you can define an addition operation on them and use it for cryptography; the algebra gets a bit complex and a general book like this isn't the place to set it out. However, elliptic curve cryptosystems are interesting for two reasons.

First, they give versions of the familiar primitives such as Diffie-Hellmann key exchange and the Digital Signature Algorithm that use less computation, and also have slightly shorter variables; both can be welcome in constrained environments such as smartcards and embedded processors. Elliptic curve cryptography is used, for example, in the rights-management mechanisms of Windows Media Player, and has been adopted as a standard by the NSA for use in defense systems.

Second, some elliptic curves have a *bilinear pairing* which Dan Boneh and Matt Franklin used to construct cryptosystems where your public key is your name [207]. Recall that in RSA and Diffie-Hellmann, the user chose his private key and then computed a corresponding public key. In a so-called *identity-based cryptosystem*, you choose your identity then go to a central authority that issues you with a private key corresponding to that identity. There is a global public key, with which anyone can encrypt a message to your identity; you can decrypt this using your private key. Earlier, Adi Shamir had discovered *identity-based signature schemes* that allow you to sign messages using a private key so that anyone can verify the signature against your name [1147]. In both cases, your private key is computed by the central authority using a system-wide private key known only to itself. Identity-based primitives could have interesting implications for specialist systems, but in the context of ordinary public-key and signature systems they achieve much the same result as the certification of public keys, which I'll discuss next.

5.7.5 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, like a phone book. A more common solution, due to Loren Kohnfelder, is for a *certification authority* (CA) to sign the users' public encryption and/or signature verification keys giving certificates that contain the user's name, attributed 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 keys after doing some due diligence about whether they belong to the principals named in them.

A certificate might be described symbolically as

$$C_A = Sig_{K_S}(T_S, L, A, K_A, V_A) \tag{5.1}$$

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.

Certification is hard, for a whole lot of reasons. I'll discuss different aspects later — naming in Chapter 6, 'Distributed Systems', public-key infrastructures in Chapter 21, 'Network Attack and Defense', and the policy aspects in Part III. Here I'll merely point out that the protocol design 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 1982 proposed that two users, say Alice and Bob, set up a shared 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 key and ships it off to him. Symbolically,

$$A \rightarrow B : C_A, C_B, \{T_A, K_{AB}, Sig_{K_A}(T_A, K_{AB})\}_{K_B} \tag{5.2}$$

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, and then strips off the outer encryption $\{\ldots\}_{K_B}$ from message 3 in the above 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:

$$B \rightarrow C : C_A, C_C, \{T_A, K_{AB}, Sig_{K_A}(T_A, K_{AB})\}_{K_C} \tag{5.3}$$

It is quite alarming that such a simple protocol — essentially, a one line program — should have such a serious flaw 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 are if anything harder to design than protocols that use 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 the authentication mechanism has to establish. In the STU-III secure telephone used by the US 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 plug into the phone to identify not just their names, but their security clearance level. In general, textbooks 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 emphasises 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 manage them properly, especially where they are not an exact reimplementation of a familiar manual control system [357]. There are many other things that can go wrong with certification at the level of systems engineering, which I'll start to look at in the next chapter.

5.7.6 The Strength of Asymmetric Cryptographic Primitives

In order to provide the same level of protection as a symmetric block cipher, asymmetric cryptographic primitives generally require at least twice the block length. Elliptic curve systems appear to achieve this bound; a 128-bit elliptic scheme could be about as hard to break as a 64-bit block cipher with a 64-bit key; and the only public-key encryption schemes used in the NSA's Suite B of military algorithms are 256- and 384-bit elliptic curve systems. The commoner schemes, based on factoring and discrete log, are less robust because there are shortcut attack algorithms such as the number field sieve that exploit the fact that some integers are *smooth*, that is, they have a large number of small factors. When I wrote the first edition of this book in 2000, the number field sieve had been used to attack keys up to 512 bits, a task comparable in difficulty to keysearch on 56-bit DES keys; by the time I rewrote this chapter for the second edition in 2007, 64-bit symmetric keys had been brute-forced, and the 663-bit challenge number RSA-200 had been factored. The advance in factoring has historically been due about equally to better hardware and better algorithms. I wrote in 2000 that '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'; now in 2007, 1024-bit RSA is widely believed to give about the same protection as 80-bit symmetric keys, and designers are starting to move to 2048 bits for keys intended to last many years. As I mentioned above, an extrapolation of recent factoring results suggests that it might be a decade before we see a 1024-bit challenge factored — although with Moore's law starting to slow down, it might take much longer. No-one really knows. (However I expect to see 768-bit RSA factored within a few years.)

There has been much research into *quantum computers* — 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 [1165]. So far only very small quantum computers can be built; factoring 15 is about the state of the art in 2007. Many people are sceptical about whether the technology can be scaled up to threaten real systems. But if it does, then asymmetric cryptography may have to change radically. So it is fortunate that many of the things we currently do with asymmetric mechanisms can also be done with symmetric ones; most authentication protocols in use could 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 provides a useful intuition: we assume that each new value returned by the encryption engine is random in the sense of being statistically independent of all the different 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. 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 that the security engineer needs to understand are not too difficult to grasp, though there are many 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. I'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 which perform reasonably well on available platforms. The two meet in the study of subjects ranging from cryptanalysis, through the search for primitives that combine provable security properties with decent performance, 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 and 'Trusted Computing' have been a stimulus in recent years, as was the US government's competition in the late 1990s 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, CHES and Fast Software Encryption — all published by Springer in their *Lecture Notes on Computer Science* series.

Further Reading

The classic papers by Whit Diffie and Martin Hellman [385] and by Ron Rivest, Adi Shamir and Len Adleman [1078] are the closest to required reading in this subject. The most popular introduction is Bruce Schneier's *Applied Cryptography* [1125] which covers a lot of ground at a level a non-mathematician can understand, but is slightly dated. Alfred Menezes, Paul van Oorshot and Scott Vanstone's *Handbook of Applied Cryptography* [872] is the closest to a standard reference book on the mathematical detail. For an appreciation of the recent history of cryptanalysis, try Mark Stamp and Richard Low's 'Applied Cryptanalysis' [1214]: this has recent attacks on fielded ciphers such as PKZIP, RC4, CMEA and MD5.

There are many more specialised references. The bible on differential cryptanalysis is a book by its inventors Eli Biham and Adi Shamir [170], while a good short tutorial on linear and differential cryptanalysis was written by Howard Heys [602]. A textbook by Doug Stinson has another detailed explanation of linear cryptanalysis [1226]; and the modern theory of block ciphers can be traced through the papers in the *Fast Software Encryption* conference series. The original book on modes of operation is by Carl Meyer and Steve Matyas [880]. Neal Koblitz has a good basic introduction to the mathematics behind public key cryptography [723]; and the number field sieve is described in [780].

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 regarded source is a book by Oded Goldreich [535] but this is pitched at the postgraduate maths student. A less thorough but more readable introduction to randomness and algorithms is in [564]. Current research at the theoretical end of cryptology is found at the FOCS, STOC, Crypto, Eurocrypt and Asiacrypt conferences.

Four of the simple block cipher modes of operation (ECB, CBC, OFB and CFB) date back to FIPS-81; their specification was reissued, with CTR mode added, in 2001 as NIST Special Publication 800-38A [944]. The compound modes of operation are described in subsequent papers in that series.

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 Kahn [676]; there are also compilations of historical articles from *Cryptologia* [363, 361, 362] as well as several books on the history of cryptology in World War 2 [296, 677, 836, 1336]. The NSA Museum at Fort George Meade, Md., 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 [427]. 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.