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 gives us the tools that underlie most modern security protocols. It is the key technology for protecting distributed systems, yet it is surprisingly hard to do right. As we've already seen in Chapter 4, "Protocols," cryptography has often been used to protect the wrong things, or to protect them in the wrong way. Unfortunately, the available crypto tools aren't always very usable.

But no security engineer can ignore cryptology. A medical friend once told me that while she was young, she worked overseas 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 at least the basics of crypto (and much else). There are, broadly speaking, three levels at which one can approach crypto. The first consists of the underlying intuitions; the second of the mathematics that we use to clarify these intuitions, provide security proofs where possible and tidy up the constructions that cause the most confusion; and the third is the cryptographic engineering – the tools we commonly use, and the experience of what can go wrong with them. In this chapter, I assume you have no training in crypto and set out to explain the basic intuitions. I illustrate them with engineering, and sketch enough of the mathematics to help give you access to the literature when you need it. One reason you need some crypto know-how is that many common constructions are confusing, and many tools offer unsafe defaults. For example, Microsoft's Crypto API (CAPI) nudges engineers to use electronic codebook mode; by the end of this chapter you should understand what that is, why it's bad, and what you should do instead.

Many crypto textbooks assume that their readers are pure maths graduates, so let me start off with non-mathematical definitions. *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* or *cleartext*, and the output the *ciphertext*. Thereafter, things get somewhat more complicated. There are a number of 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.

I will first give some historical examples to illustrate the basic concepts. I'll then fine-tune definitions by introducing the security models that cryptologists use, including perfect secrecy, concrete security, indistinguishability and the random oracle model. Finally, I'll show how the more important cryptographic algorithms actually work, and how they can be used to protect data. En route, I'll give examples of how people broke weak ciphers, and weak constructions using strong ciphers.

5.2 Historical Background

Suetonius tells us that Julius Caesar enciphered his dispatches by writing 'D' for 'A', 'E' for 'B' and so on [1843]. 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 [1535].

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 it's a pencil and paper puzzle to break ciphers of this kind. 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 experimented with programs to solve monoalphabetic substitutions. Using letter and digram (letter pair) frequencies alone, they typically need about 600 letters of ciphertext; smarter strategies such as guessing probable words can cut this to about 150 letters; and state-of-the-art systems that use neural networks and approach the competence of human analysts are also tested on deciphering ancient scripts such as Ugaritic and Linear B [1194].

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.

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 it 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. 9 corresponds to J, so the encryption of P under the key U (and of U under the key P) is J, or more simply U + P = J. In this notation, Julius Caesar's system used a fixed key K = D, while Augustus Caesar's used K = C and Vigenère used a repeating key, also known as a *running key*. Techniques were developed to do this quickly, ranging from printed tables to brass cipher wheels. Whatever the 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. But the first published solution was in 1863 by Friedrich Kasiski, a Prussian infantry officer [1020]. 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. Then ciphertext letters one, four, seven and so on were 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 remaining 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 is known as the *one-time pad* and was proposed by Gilbert Vernam during World War I [1001]; given any ciphertext, and any plaintext of the same length, there's a key that decrypts the ciphertext to the plaintext. So regardless of the amount of computation opponents can do, they're none the wiser, as given any ciphertext, all possible plaintexts of that length are equally likely. This system therefore has *perfect secrecy*.

Here's an example. 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`. So 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 underground resistance, and the message actually said 'Hang Hitler'. This is also 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 can claim 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. So if you wanted to get this spy into trouble, you could change the ciphertext to `DCYTI BWPJA` (Figure 5.4):

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

Figure 5.5 – Manipulating the message to entrap the spy

Leo Marks' engaging book on cryptography in the Special Operations Executive in World War II [1224] 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. In fact, during the 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 that offers perfect secrecy. He was finally allowed to publish this in 1948 [1714, 1715].

The one-time tape was used for top-level communications by both sides from late in World War II, then for strategic communications between NATO allies, and for the US-USSR hotline from 1963. Thousands of machines were produced in total, using paper tapes for key material, until they were eventually replaced by computers from the mid-1980s[1]. But such cryptography is too expensive for most applications as it consumes as much key material as there is traffic. It's more common for stream ciphers to use a pseudorandom number generator to expand a short key into a long keystream. The data is then encrypted by combining the keystream, one symbol 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 symbols should not be able to predict any more of them.

An early example was *rotor machines*, mechanical stream-cipher devices that produce a very long sequence of pseudorandom states[2] and combine them with plaintext to get ciphertext. These machines were independently invented by a number of people from the 1920s, many of whom tried to sell them to the banking industry. Banks weren't in general interested, for reasons we'll discuss below, but rotor machines were very widely used by the combatants in World War II to encipher radio traffic, and the efforts made by the Allies to decipher German traffic included the work by Alan Turing and others on Colossus, which helped kickstart the computer industry after the war.

Stream ciphers have been widely used in hardware applications where the number of gates had to be minimised to save power. However, block ciphers are more flexible and are more common in systems being designed now, so let's look at them next.

5.2.3 An early block cipher – Playfair

The Playfair cipher 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

[1] Information about the machines can be seen at the Crypto Museum, `https://www.cryptomuseum.com`.

[2] letters in the case of the Hagelin machine used by the USA, permutations in the case of the German Enigma and the British Typex

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 table

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 I, and by the Americans and Germans in World War II. It's a substantial improvement on Vigenère as the statistics that 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 [740]. In fact, the quote at the head of this chapter is a Playfair-encrypted message sent by the future President Jack Kennedy when he was a young lieutenant holed up on a small island with ten other survivors after his motor torpedo boat had been sunk in a collision with a Japanese destroyer. Had the Japanese intercepted it, they might possibly have decrypted it, and history could be different. For a stronger cipher, we will want the effects of small changes in the 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 also be greatly improved by choosing a longer block length than two characters. For example, the *Data Encryption Standard* (DES), which is widely used in payment systems, has a block length of 64 bits and the *Advanced Encryption Standard* (AES), which has replaced it in most other 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 we need more than just an adequate block size.

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 valid but unauthorised transaction. Suppose a crook worked for a bank's phone company, and 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'll need some way of binding the blocks together.

5.2.4 Hash functions

The third classical type of cipher is the *hash function.* This evolved to protect the integrity and authenticity of messages, where we don't want someone to be able 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. What's 'wired' is 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 important given the costs of international telegrams at the time. A code book was essentially a block cipher that mapped words or phrases to fixed-length groups of letters or numbers. So "Please pay from our account with you no." might become 'AFVCT'. Sometimes the codes were also enciphered.

The banks realised that neither 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 might add together 53 (no millions), 54 (300,000), 29 (70,000) and 71 (6,000) ignoring the less significant digits. 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, ignoring the carry.

This made such test key systems into *one-way functions* in that although it was possible to compute a test from a message, given knowledge of the key, it was not possible to reverse the process and recover either a message or a key from a single 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 do more work on it.)

Banking test keys are not strong by the standards of modern cryptography. Given 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. But the banks got away with it: test keys worked fine from the late nineteenth century through the

1980s. In several years working as a bank security consultant, and listening to elderly 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 that replaced test keys in the chapter on Banking and Bookkeeping.

However, test keys are our historical example of an algebraic function used for authentication. They have important modern descendants in the authentication codes used in the command and control of nuclear weapons, and also with modern block ciphers. The idea in each case is the same: if you can use a unique key to authenticate each message, simple algebra can give you ideal security. Suppose you have a message M of arbitrary length and want to compute an authentication code A of 128 bits long, and the property you want is that nobody should be able to find a different message M' whose authentication code under the same key will also be A, unless they know the key, except by a lucky guess for which the probability is 2^{-128}. You can simply choose a 128-bit prime number p and compute $A = k_1M + k_2 \pmod p$ where the key consists of two 128-bit numbers k_1 and k_2.

This is secure for the same reason the one-time pad is: given any other message M' you can find another key (k_1', k_2') that authenticates M' to A. So without knowledge of the key, the adversary who sees M and A simply has no information of any use in creating a valid forgery. As there are 256 bits of key and only 128 bits of tag, this holds even for an adversary with unlimited computing power: such an adversary can easily find the 2^{128} possible keys for each pair of message and tag but has no way to choose between them. I'll discuss how this *universal hash function* is used with block ciphers below, and how it's used in nuclear command and control in Part 2.

5.2.5 Asymmetric primitives

Finally, some modern cryptosystems are asymmetric, in that different keys are used for encryption and decryption. So, for example, most web sites nowadays have a certificate containing a *public key* with which people can encrypt their session using a protocol called TLS; the owner of the web page can decrypt the traffic using the corresponding *private key.* We'll go into the details later.

There are some pre-computer examples of this too; perhaps the best is the postal service. You can send me a private message 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. Of course, many things 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 doormat. Similar things can go wrong with public key cryptography: false public keys can be inserted into the system, computers can be hacked, people can be coerced and so on. We'll look at these problems in more detail in later chapters.

Another asymmetric application of cryptography is the *digital signature.* The idea here is that I can sign a message using a private *signature key* 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 Security Models

Before delving into the detailed design of modern ciphers, I want to look more carefully at the various types of cipher and the ways in which we can reason about their security.

Security models seek to formalise the idea that a cipher is “good”. We’ve already seen the model of *perfect secrecy*: given any ciphertext, all possible plaintexts of that length are equally likely. Similarly, an authentication scheme that uses a key only once can be designed so that the best forgery attack on it is a random guess, whose probability of success can be made as low as we want by choosing a long enough tag.

The second model is *concrete security*, where we want to know how much actual work an adversary has to do. At the time of writing, it takes the most powerful adversary in existence – the community of bitcoin miners, burning about as much electricity as the state of Denmark – about ten minutes to solve a 68-bit cryptographic puzzle and mine a new block. So an 80-bit key would take them 2^{12} times as long, or about a month; a 128-bit key, the default in modern systems, is 2^{48} times harder again. So even in 1000 years the probability of finding the right key by chance is 2^{-35} or one in many billion. In general, a system is (t, ϵ)-secure if an adversary working for time t succeeds in breaking the cipher with probability at most ϵ.

The third model, which many theoreticians now call the standard model, is about *indistinguishability.* This enables us to reason about the specific properties of a cipher we care about. For example, most cipher systems don’t hide the length of a message, so we can’t define a cipher to be secure by just requiring that an adversary not be able to distinguish ciphertexts corresponding to two messages; we have to be more explicit and require that the adversary not be able to distinguish between two messages $M1$ and $M2$ of the same length. This is formalised by having the cryptographer and the cryptanalyst play a game in which the analyst wins by finding an efficient discriminator of something she shouldn’t be able to discriminate with more than negligible probability. If the cipher doesn’t have perfect security this can be *asymptotic*, where we typically want the effort to grow faster than any polynomial function of a security parameter n – say the length of the key in bits. A security proof typically consists of a *reduction* where we show that if there exists a randomised (i.e. probabilistic) algorithm running in time polynomial in n that learns information it shouldn’t with non-negligible probability, then this would give an efficient discriminator for an underlying cryptographic primitive that we already trust. Finally, a construction is said to have *semantic security* if there’s no efficient distinguisher for the plaintext regardless of any side information the analyst may have about it; even if she knows all but one bit of it, and even if she can get a decryption of any other ciphertext, she can't learn anything more from the target ciphertext. This skips over quite a few mathematical details, which you can find in a standard text such as Katz and Lindell [1022].

The fourth model is the random oracle model, which is not as general as the standard model but which often leads to more efficient constructions. We call a cryptographic primitive *pseudorandom* if there's no efficient way of distinguishing it from a random function of that type, and in particular it passes all the statistical and other randomness tests we apply. 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 model of computation permits.

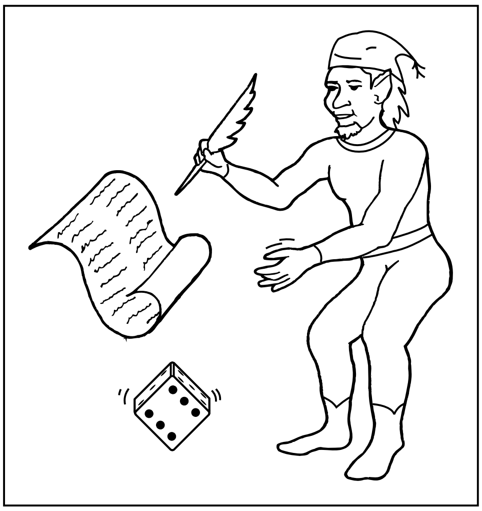


Figure 5.9: – the random oracle

To visualise a random oracle, we might imagine 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, and keep a record for future reference. We'll further assume finite bandwidth – the elf will only answer so many queries every second. What's more, our oracle can operate according to several different rules.

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 string of fixed length, say n bits long. The same input gives the same output, but the set of outputs appears random. So the elf just has a simple list of inputs and outputs, which grows steadily as it works.

Random functions are our model for *cryptographic hash functions*. These were first used in computer systems for one-way encryption of passwords in the 1960s and have many more uses today. For example, if the police seize your laptop, the standard forensic tools will compute checksums on all the files, to identify which files are already known (such as system files) and which are novel (such as user data). These hash values will change if a file is corrupted and so can assure the court that the police haven't tampered with evidence. And 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 or have it mined into the Bitcoin blockchain. However, if the document is still secret – for example an invention for which we want to establish a priority date – then we would not upload the whole document, but just the message hash. This is the modern equivalent of Hooke's anagram that we discussed in section 5.2.4 above.

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 $h(x)$ to find x if such an input 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 can do to invert a random function is to keep on feeding in more inputs until he gets lucky; with an n-bit output this will take about 2^{n-1} guesses on average. A pseudorandom function will have the same properties, or they could be used to distinguish it from a random function, contrary to our definition. So a pseudorandom function will also be a *one-way function*, provided there are too many possible outputs for the opponent to guess an input that has a desired target output by chance. This means choosing n so that the opponent can't do anything near 2^n computations. If we claim, for example, that SHA256 is a pseudorandom function, then we're saying that there's no practical way to find an input that hashes to a given 256-bit value, unless you knew it already and used it to compute that value.

A second property of pseudorandom functions is that the output will not give any information at all about even part of the input. So we can get a one-way encryption of the value x 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. (I'll discuss an example later in section 22.2.1: the hash function used by many phone companies in the 1990s and early 2000s to authenticate mobile phone users wasn't random enough, which led to attacks.)

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 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 class of 30 pupils what they think is the probability that two of them have the same birthday. Most pupils intuitively think it's unlikely, and the maths teacher then asks the pupils to state their birthdays one after another. The odds of a match exceed 50% once 23 pupils have been called. As this surprises most people, it's also known as the 'birthday paradox'.

The birthday theorem was first used in the 1930's to count fish, so it's also known as *capture-recapture statistics* [1665]. 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[3].

This theorem has many applications for the security engineer. For example, if we have a biometric system that can authenticate a person's claim to identity with a probability of only one in a million that two randomly selected subjects will be falsely identified as the same person, this doesn't mean that we can use it as a reliable means of identification in a university with a user population of twenty thousand staff and students. This is because there will be almost two hundred million possible pairs. In fact, you 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. It may well, however, be OK to use it to verify a claimed identity (though many other things can go wrong; see the chapter on Biometrics in Part 2 for a discussion).

There are some applications where collision-search attacks aren't a problem, such as in challenge-response protocols where an attacker has to find the answer to the challenge just issued, and where you can prevent challenges repeating. In identify-friend-or-foe (IFF) systems, for example, common equipment has a response length of 48 to 80 bits. You can't afford much more than that, as it costs radar accuracy.

But there are other applications in which collisions are unacceptable. When we design digital signature systems, we typically pass the message M through a cryptographic hash function first, and then sign the hash $h(M)$, for a number of reasons we'll discuss later. In such an 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 precalculate suitable pairs M_1, M_2, get you to sign an 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 send the funds to Mafia Holdings Inc., Bermuda."

For this reason, hash functions used with digital signature schemes have n

[3] More precisely, the probability that m fish chosen randomly from N fish are different is $\beta = N(N-1)\dots(N-m+1)/N^m$ which is asymptotically solved by $N \simeq m^2/2log(1/\beta)$ [1037].

large enough to make them collision-free. Historically, the two most common hash functions have been 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.

To sum up: if you need a cryptographic hash function to be collision resistant, then you'd better choose a function with an output of at least 256 bits, such as SHA-2 or SHA-3. However if you only need to be sure that nobody will find a second preimage for an existing, externally given hash, then you can perhaps make do with less.

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 it's the reverse of the hash function in that it has a short input and a long output. If we had a good pseudorandom function whose input and output were long enough, 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 and using the output as a keystream.

It can be used to protect the confidentiality of our backup data as follows: 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 service in the cloud. (This is also called an *additive stream cipher* as exclusive-or is addition modulo 2.) We can think of the elf generating a random tape of the required length each time he is presented with a new key, giving it to us and keeping a copy on his scroll for reference in case he's given the same input key again. If we need to recover the data, we go back to the generator, enter the same key, get the same keystream, and exclusive-or it with our ciphertext to get our plaintext back again. Other people with access to the keystream generator won't be able to generate the same keystream unless they know the key. Note that this would not give us any guarantee of file integrity; as we saw in the discussion of the one-time pad, adding a keystream to plaintext can protect confidentiality, but it can't detect modification of the file. For that, we might make a hash of the file and keep that somewhere safe. It may be easier to protect the hash from modification than the whole file.

One-time pad systems are a close fit for our theoretical model, except in that they are used to secure communications across space rather than time: the two communicating parties have shared a copy of a keystream in advance. Vernam's original telegraph cipher machine used punched paper tape; Marks describes how SOE agents' silken keys were manufactured in Oxford by retired ladies shuffling counters; we'll discuss modern hardware random number generators in the chapter on Physical Security.

A real problem with keystream generators is to prevent the same keystream being used more than once, whether to encrypt more than one backup tape or to encrypt more than one message sent on a communications channel. During World War II, the amount of Russian diplomatic traffic exceeded the quantity of one-time tape they had distributed in advance to their embassies, so it was reused. But 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; 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 from 1943 onwards and broke up a number of Russian spy rings. In the words of one former NSA chief scientist, it became a "two-time tape".

To avoid this, the normal engineering practice is to have not just a key but also a *seed* (also known as an *initialisation vector* or IV) so we start the keystream at a different place each time. The seed N may be a sequence number, or generated from a protocol in a more complex way. Here, you need to ensure that both parties synchronise on the right working key even in the presence of an adversary who may try to get you to reuse old keystream.

5.3.3 Random permutations – block ciphers

The third type of primitive, and the most important in modern 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 visualise 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 rolls 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, looks for it 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 new message at random, notes it down and gives it to us.

A *block cipher* is a keyed family of pseudorandom permutations. For each key, we have a single permutation that's 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. Furthermore, nobody should be able to infer any information about plaintexts or ciphertexts that it has not yet produced.

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 (differential cryptanalysis) required 2^{47} chosen plaintexts to recover the key, while the next major attack (linear cryptanalysis) improved this to 2^{43} known plaintexts. While these attacks were of huge scientific importance, their practical engineering effect was zero, as no practical systems make that much known text (let alone chosen text) available to an attacker. Such impractical attacks are often referred to as *certificational* as they affect the cipher's security certification rather than providing a practical exploit. They can have a commercial effect, though: the attacks on DES undermined confidence 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 hacker can buy a decoder, watch a lot of movies and compare them with the enciphered broadcast signal; so a *known-plaintext attack* might be the main threat. But there are surprisingly many applications where *chosen-plaintext attacks* are possible. A historic example is from World War II, where US analysts learned of Japanese intentions for an island 'AF' which they suspected meant Midway. So they arranged for Midway's commander to send an unencrypted message reporting problems with its fresh water condenser, and then intercepted a Japanese report that 'AF is short of water'. Knowing that Midway was the Japanese objective, Admiral Chester Nimitz was waiting for them and sank four Japanese carriers, turning the tide of the war [1001].

The other attacks are more specialised. *Chosen plaintext/ciphertext* attacks may be a worry where the threat is a *lunchtime attack*: someone who gets temporary access to a cryptographic device while its authorised user is out, and tries out the full range of permitted operations for a while with data of their choice. *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). To exclude all such attacks, the goal is semantic security, as discussed above; the cipher should not allow the inference of unauthorised information (whether of plaintexts, ciphertexts or keys) other than with negligible probability.

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.

The simplest variation is the *trapdoor one-way permutation*. This is a computation that 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.

In most real systems, the encryption is randomised, so that every time someone uses the same public key to encrypt the same message, the answer is different; this is necessary for semantic security, so that an opponent cannot check whether a guess of the plaintext of a given ciphertext is correct. There are even more demanding models than this, for example to analyse security in the case where the opponent can get ciphertexts of their choice decrypted, with the exception of the target ciphertext. But this will do for now.

5.3.5 Digital signatures

The final cryptographic primitive we'll define here is the *digital signature.* The basic idea is that a signature on a message can be created by only one principal, but checked by anyone. It can thus perform the same 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 foreign intelligence agency.

Signature schemes, too, can be deterministic or randomised: 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 they can perform the verification.

Formally, a signature scheme, like a 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 verification function which, given a 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.

Where we don't need message recovery, 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. In the other direction, it will perform verification for anybody.

For this simple scheme, signature verification means that 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 application that uses message recovery is machine-printed postage stamps, or *indicia*: the stamp consists 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 discuss this at the end of section 16.3.2.

In the general case we do not need message recovery; the message to be signed may be of arbitrary length, so we first pass it through a hash function and then sign the hash value. We need the hash function to be not just one-way, but also collision resistant.

5.4 Symmetric crypto algorithms

Now that we've tidied up the definitions, we'll 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 developed over the years with computer science undergraduates, to help the non-specialist 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, and require ingenuity and persistence rather than anything particularly highbrow.

5.4.1 SP-networks

Claude Shannon suggested in the 1940s that strong ciphers could be built by combining substitution with transposition repeatedly. For example, one might add some key material to a block of input text, 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 [1009]. 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 visualised 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

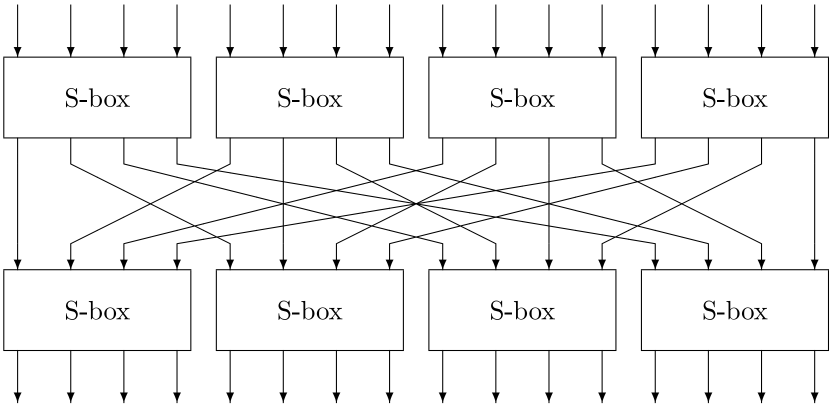


Figure 5.10: – a simple 16-bit SP-network block cipher

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 be rather limited, as an opponent could just build a dictionary of plaintext and ciphertext blocks as they were observed. 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 few hundred 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-bit S-boxes, we may have 16 of them (for a 64 bit block size) or 32 of them (for a 128 bit block size).

5.4.1.2 Number of rounds

Second, we 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 it is still a basic student exercise.)

The number of rounds we need depends on the speed with which data diffuse through the cipher. In our 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. If the block cipher is to be used for decryption as well as encryption, this linear transformation will have to be invertible. We'll see some concrete examples below in the sections on AES and DES.

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. 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 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* [895, 1244] proceeds by collecting a number of relations such as "bit 2 plus bit 5 of the input to the first S-box is equal to bit 1 plus bit 8 of the output, with probability 13/16", then searching for ways to glue them together into an algebraic relation between input bits, output bits and 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 the sampling theorem in 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 [245, 895] 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 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 many variations 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 [242][4].

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. But 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 analyse. A prudent designer will also use more rounds than are strictly necessary to block the attacks known today, in order to give some safety margin, as attacks only ever get better. But while we may be able to show that a cipher resists all the attacks we know of, and with some safety margin, this says little about whether it will resist novel types of attack. (A general security proof for a block cipher would appear to imply a result such as $P \neq NP$ that would revolutionise computer science.)

5.4.2 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is an algorithm originally known as Rijndael after its inventors Vincent Rijmen and Joan Daemen [507]. It 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 that acts on a byte input to give a byte output. For implementation purposes it can be regarded simply as a lookup table of

[4] This may have been used first at Bletchley in World War II where a key insight into breaking the German Enigma machine was that no letter ever enciphered to itself.

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. The first step 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 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 – an *avalanche* effect that makes both linear and differential attacks harder.

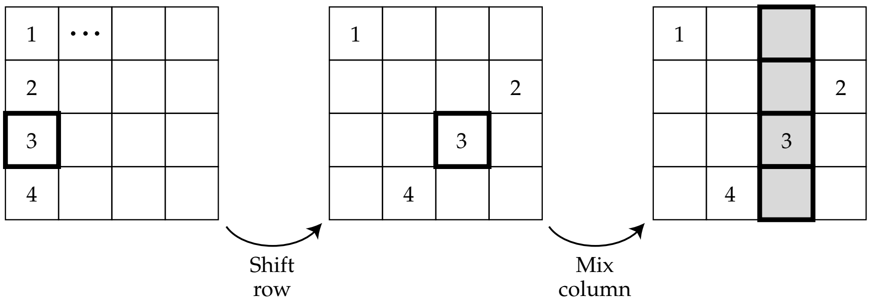


Figure 5.11: – the AES linear transformation, illustrated by its effect on byte 3 of the input

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

The algorithm uses 10 rounds with 128-bit keys, 12 rounds with 192-bit keys and 14 rounds with 256-bit keys. These are enough to give practical, but not certficational, security – as indeed we expected at the time of the AES competition, and as I described in earlier editions of this chapter. The first key-recovery attacks use a technique called biclique cryptanalysis and were discovered in 2009 by Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger [273]; they give only a very small advantage, with complexity now estimated at 2^{126} for 128-bit AES and $2^{254.3}$ for 256-bit AES, as opposed to 2^{127} and 2^{255} for brute-force search. Faster shortcut attacks are known for the case where we have related keys. But none of these attacks make any difference in practice, as they require infeasibly large numbers of texts or very special combinations of related keys.

Should we trust AES? The governments of Russia, China and Japan try to get firms to use local ciphers instead, and the Japanese offering, Camellia, is found in a number of crypto libraries alongside AES and another AES competition finalist, Bruce Schneier's Twofish. (Camellia was designed by a team whose own AES candidate was knocked out at the first round.) Conspiracy theorists note that the US government picked the weakest of the five algorithms that were finalists in the AES competition. Well, I was one of the designers of the AES finalist Serpent [94], which came second in the competition: the winner Rijndael got 86 votes, Serpent 59 votes, Twofish 31 votes, RC6 23 votes and MARS 13 votes. Serpent has a simple structure that makes it easy to analyse – the structure of Figure 5.10, but modified to be wide enough and to have enough rounds – and was designed to have a much larger security margin than Rijndael in anticipation of the attacks that have now appeared. Yet the simple fact is that while Serpent is more secure, Rijndael is faster; industry and crypto researchers voted for it at the last AES conference, and NIST approved it as the standard.

Having been involved in the whole process, and having worked on the analysis and design of shared-key ciphers for much of the 1990s, I have a high level of confidence that AES is secure against practical attacks based on mathematical cryptanalysis. And even though AES is less secure than Serpent, practical security is all about implementation, and we now have enormous experience at implementing AES. Practical attacks include timing analysis and power analysis. In the former, the main risk is that an opponent observes cache misses and uses them to work out the key. In the latter, 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. I discuss both in detail in Part 2, in the chapter on Emission Security; countermeasures include special operations in many CPUs to do AES, which are available precisely because the algorithm is now a standard. It does not make sense to implement Serpent as well, 'just in case AES is broken': having swappable algorithms is known as *pluggable cryptography*, yet 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 caused something to break horribly.)

The back story is that, back in the 1970s, the NSA manipulated the choice and parameters of the previous standard block cipher, the *Data Encryption Standard* (DES) in such a way as to deliver a cipher that was good enough for US industry at the time, while causing foreign governments to believe it was insecure, so they used their own weak designs instead. I'll discuss this in more detail below, once I've described the design of DES. AES seems to have followed this playbook; by selecting an algorithm that was only just strong enough mathematically and whose safe implementation requires skill and care, the US government saw to it that firms in Russia, China, Japan and elsewhere will end up using systems that are less secure because less skill and effort has been invested in the implementation. However, this was probably luck rather than Machiavellian cunning: the relevant committee at NIST would have had to have a lot of courage to disregard the vote and choose another algorithm instead. Oh, and the NSA has since 2005 approved AES with 128-bit keys for protecting information up to SECRET and with 192-bit or 256-bit keys for TOP SECRET. So I recommend that you use AES instead of GOST, or Camellia, or even Serpent. The definitive specification of AES is Federal Information Processing Standard 197, and its inventors have written a book describing its design in detail [507].

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 1950s and early 1960s. Feistel then moved to IBM and founded a research group that produced the Data Encryption Standard (DES) algorithm, which is still a mainstay of payment system 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.

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

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

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, software speed or hardware gate count.

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. (I omit a number of technicalities.)

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

5.4.3.2 DES

The DES algorithm is widely used in banking and other payment applications. The ‘killer app’ that got it widely deployed was ATM networks; from there

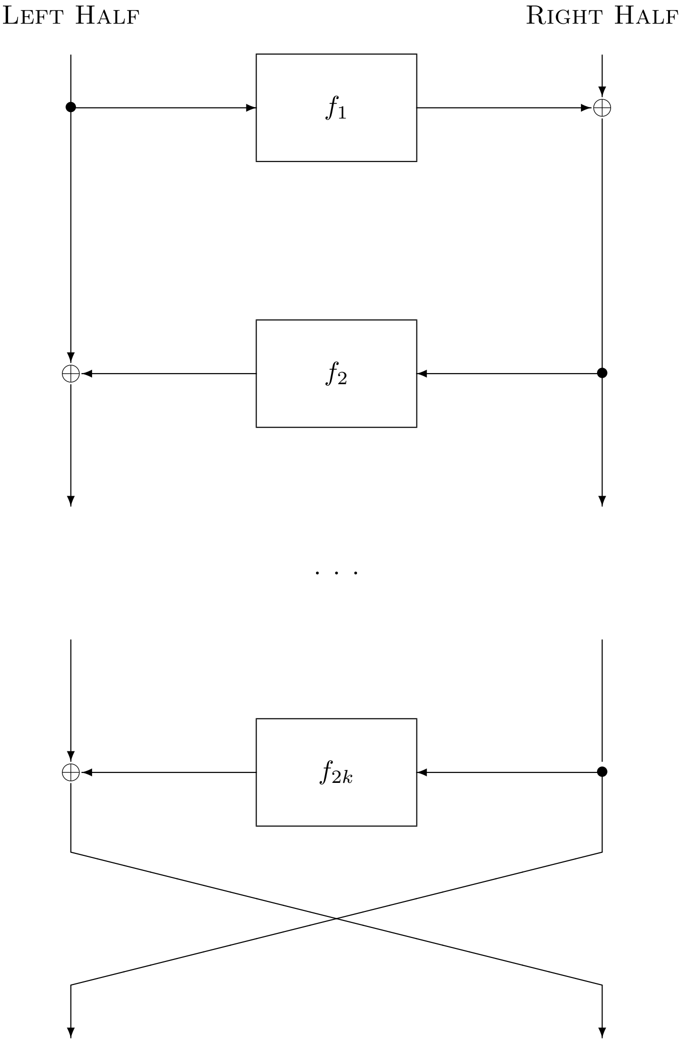


Figure 5.12: – the Feistel cipher structure

it spread to prepayment meters, transport tickets and much else. In its classic form, 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:

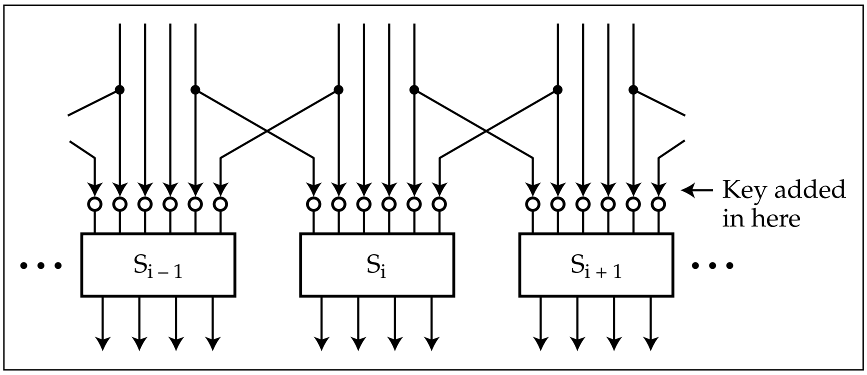


Figure 5.13: – the DES round function

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 [1397].

DES was introduced in 1974 and immediately caused 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 in 1977 [557]. IBM, whose scientists invented DES, retorted that they would charge the US government $200 million to build such a machine. (In hindsight, 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 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 [619]. 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 [1108]. A modern botnet with 100,000 machines would take a few hours. So the key length of single DES is now inadequate.

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 [473]. The 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 26.2.7.1.

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 vulnerability to keysearch makes single DES unusable in most applications. As with AES, there are also attacks based on timing analysis and power analysis.

The usual way of dealing with the DES key length problem is to use the algorithm multiple times with different keys. Banking networks have largely moved to *triple-DES*, a standard since 1999 [1397]. Triple-DES does an encryption, then a decryption, and then a further encryption, all with independent keys. Formally:

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

By setting the three keys equal, you get 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.) Most new systems use AES as the default choice, but many banking systems are committed to using block ciphers with an eight-byte block, 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 the chapter on Banking and Bookkeeping). 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. 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 [1047]. Whitened block ciphers are used in some applications, most specifically in the XTS mode of operation which I discuss below. Nowadays, it's usually used with AES, and AESX is defined similarly, with the whitening keys used to make each block encryption operation unique – as we shall see below in section 5.5.7.

5.5 Modes of Operation

A common failure is that cryptographic libraries enable or even encourage developers to use an inappropriate *mode of operation*. This 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.

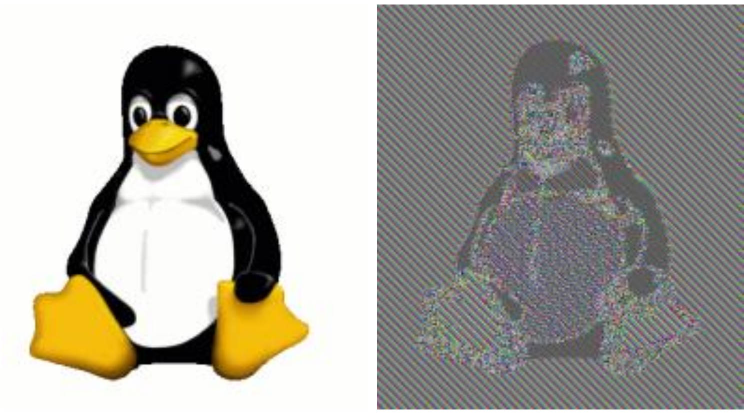
There are several standard modes of operation for using a block cipher on multiple blocks [1404]. It is vital to understand them, so you can choose the right one for the job, especially as some common tools provide a weak one by default. This weak mode is electronic code book (ECB) mode, which we discuss next.

5.5.1 How not to use a block cipher

In electronic code book mode, we just encrypt each succeeding block of plaintext with our block cipher to get ciphertext, as with the Playfair example above. This is adequate for protocols using single blocks such as challenge-response and some key management tasks; it's also used to encrypt PINs in cash machine systems. But if we use it to encrypt redundant data the patterns will show through, giving an opponent information about the plaintext. For example, figure 5.14 shows what happens to a cartoon image when encrypted using DES in ECB mode. Repeated blocks of plaintext all encrypt to the same ciphertext, leaving the image quite recognisable.

In one popular corporate email system from the last century, 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 ever 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 require authenticity – such as bank payment messages – is particularly foolish, as it opens you 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 are replaced with some of the digits of Z.



(a) Plaintext (b) ECB ciphertext

Figure 5.14: The Linux penguin, in clear and ECB encrypted (from Wikipedia, derived from images created by Larry Ewing).

5.5.2 Cipher block chaining

Most commercial applications which encrypt more than one block used to use cipher block chaining, or CBC, mode. Like ECB, this was one of the original modes of operation standardised with DES. In it, we exclusive-or the previous block of ciphertext to the current block of plaintext before encryption (see Figure 5.15).

This mode disguises patterns in the plaintext: the encryption of each block depends on all the previous blocks. The input initialisation vector (IV) ensures that stereotyped plaintext message headers won't leak information by encrypting to identical ciphertexts, just as with a stream cipher.

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). 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 their choice. For that reason, CBC encryption usually has to be used with a separate authentication code.

More subtle things can go wrong, too; systems have to pad the plaintext to a multiple of the block size, and if a server that decrypts a message and finds incorrect padding signals this fact, whether by returning an 'invalid padding' message or just taking longer to respond, then this opens a *padding oracle attack* in which the attacker tweaks input ciphertexts, one byte at a time, watches the error messages, and ends up being able to decrypt whole messages. This was discovered by Serge Vaudenay in 2002; variants of it were used against SSL, IPSEC and TLS as late as 2016 [1949].

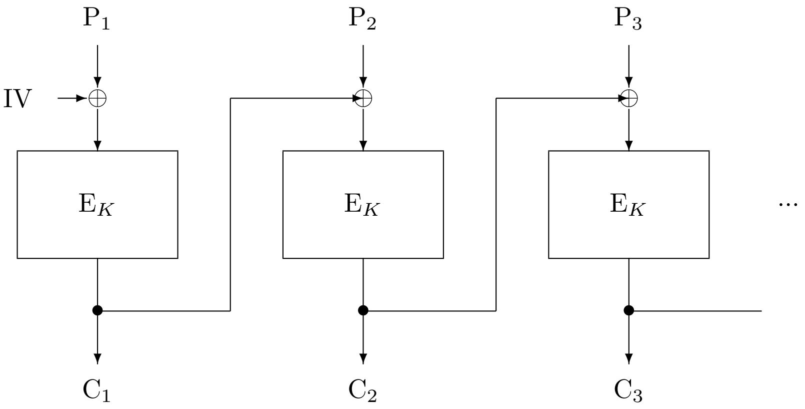


Figure 5.15: – Cipher Block Chaining (CBC) mode

5.5.3 Counter encryption

Feedback modes of block cipher encryption are falling from fashion, and not just because of cryptographic issues. They are hard to parallelise. With CBC, a whole block of the cipher must be computed between each block input and each block output. This can be inconvenient in high-speed applications, such as protecting traffic on backbone links. 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 to use AES as a stream cipher. We generate a keystream by encrypting a counter starting at an initialisation vector: $K_i = \{IV + i\}_K$, thus expanding the key K into a long stream of blocks K_i of keystream, which is typically combined with the blocks of a message M_i using exclusive-or to give ciphertext $C_i = M_i \oplus K_i$.

Additive stream ciphers have two systemic vulnerabilities, as we noted in section 5.2.2 above. The first is an attack in depth: if the same keystream is used twice, then the xor of the two ciphertexts is the xor of the two plaintexts, from which plaintext can often be deduced, as with Venona. The second is that they fail to protect message integrity. Suppose that a stream cipher were used to encipher fund transfer messages. These messages are highly structured; you might know, for example, that bytes 37–42 contain the sum being transferred. You could then cause the data traffic from a local bank to go via your computer, for example by an SS7 exploit. You go into the bank and send $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 can recover K_i and 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.

The usual way of dealing with this is to add an authentication code, and the most common standard uses a technique called Galois counter mode, which I describe later.

5.5.4 Legacy stream cipher modes

You may find two old stream-cipher modes of operation, output feedback mode (OFB) and less frequently ciphertext feedback mode (CFB).

Output feedback mode consists of repeatedly encrypting an initial value and using this as a keystream in a stream cipher. Writing IV for the initialization vector, we will have $K1 = \{IV\}_K$ and $Ki = \{IV\}_{K(i-1)}$. However 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 we loop back to the IV. So we may have a cycle-length problem if we use a 64-bit block cipher such as triple-DES on a high-speed link: once we've called a little over 2^{32} pseudorandom 64-bit values, the odds favour a 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 $2^{n/2}$, and as we noted above is easy to parallelise. Despite this OFB is still used, as counter mode only became a NIST standard in 2002.

Cipher feedback mode is another kind of stream cipher, designed for use in radio systems that have to resist jamming. 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, adding the last output bit to the next plaintext bit, and shifting the ciphertext along one bit. But this costs one block cipher operation per bit and has very bad error amplification properties; nowadays people tend to use dedicated link layer protocols for synchronization and error correction rather than trying to combine them with the cryptography at the traffic layer.

5.5.5 Message Authentication Code

Another 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.)

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 [211].

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 [1405]. (CMAC is based on a proposal by Tetsu Iwata and Kaoru Kurosawa [965].) You may see legacy systems in which the MAC consists of only half of the last output block, with the other half thrown away, or used in other mechanisms.

There are other possible constructions of MACs: the most common one is HMAC, which uses a hash function with a key; we'll describe it in section 5.6.2.

5.5.6 Galois Counter Mode

The above modes were all developed for DES in the 1970s and 1980s (although counter mode only became an official US government standard in 2002). They are not efficient for bulk encryption where you need to protect integrity as well as confidentiality; if you use either CBC mode or counter mode to encrypt your data and a CBC-MAC or CMAC to protect its integrity, then you invoke the block cipher twice for each block of data you process, and the operation cannot be parallelised.

The modern approach is to use a mode of operation designed for authenticated encryption. Galois Counter Mode (GCM) has taken over as the default since being approved by NIST in 2007 [1407]. It uses only one invocation of the block cipher per block of text, and it's parallelisable so you can get high throughput on fast data links with low cost and low latency. Encryption is performed in a variant of counter mode; the resulting ciphertexts are also used as coefficients of a polynomial which is evaluated at a key-dependent point over a Galois field of 2^{128} elements to give an authenticator tag. The tag computation is a universal hash function of the kind I described in section 5.2.4 and is provably secure so long as keys are never reused. The supplied key is used along with a random IV to generate both a unique message key and a unique authenticator key. The output is thus a ciphertext of the same length as the plaintext, plus an IV and a tag of typically 128 bits each.

GCM also 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 was invented by David McGrew and John Viega of Cisco; their goal was to create an efficient authenticated encryption mode suitable for use in high-performance network hardware [1268]. GCM is the sensible default for authenticated encryption of bulk content. (There's an earlier composite mode, CCM, which you'll find used in Bluetooth 4.0 and later; this combines counter mode with CBC-MAC, so it costs about twice as much effort to compute, and cannot be parallelised or recomputed incrementally [1406].)

5.5.7 XTS

GCM and other authenticated encryption modes expand the plaintext by adding a message key and an authenticator tag. This is very inconvenient in applications such as hard disk encryption, where we prefer a mode of operation that preserves plaintext length. Disk encryption systems used to use CBC with the sector number providing an IV, but since Windows 10, Microsoft has been using a new

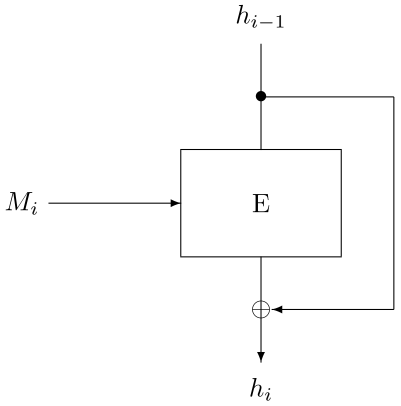


Figure 5.16: – feedforward mode (hash function)

mode of operation, XTS-AES, inspired by GCM and standardised in 2007. This is a codebook mode but with the plaintext whitened by a *tweak key* derived from the disk sector. Formally, the message M_i encrypted with the key K at at block j is

$$AESX(KT_j, K, KT_j; M)$$

where the tweak key KT_j is derived by encrypting the IV using a different key and then multiplying it repeatedly with a suitable constant so as to give a different whitener for each block. This means that if an attacker swaps two encrypted blocks, all 256 bits will decrypt to randomly wrong values. You still need higher-layer mechanisms to detect ciphertext manipulation, but simple checksums will be sufficient.

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[5]. 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 *Davies-Meyer construction* gives our final mode of operation of a block cipher (Figure 5.16).

[5] 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 birthday theorem 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 forging a message would cost of the order of 2^{32} messages, which is just too easy. A 128-bit cipher such as AES used to be just about adequate, and in fact the AACS content protection mechanism in Blu-ray DVDs used 'AES-H', the hash function derived from AES in this way.

5.6.1 Common hash functions

The hash functions most commonly used through the 1990s and 2000s evolved as variants of a block cipher with a 512 bit key and a block size increasing from 128 to 512 bits. The first two were designed by Ron Rivest and the others by the NSA:

- MD4 has three rounds and a 128 bit hash value, and a collision was found for it in 1998 [568];
- MD5 has four rounds and a 128 bit hash value, and a collision was found for it in 2004 [1979, 1981];
- SHA-1, released in 1995, has five rounds and a 160 bit hash value. A collision was found in 2017 [1828], and a more powerful version of the attack in 2020 [1146];
- SHA-2, which replaced it in 2002, comes in 256-bit and 512-bit versions (called SHA256 and SHA512) plus a number of variants.

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 [1667]. Cryptanalysis has advanced steadily. MD4 was broken by Hans Dobbertin in 1998 [568]; MD5 was broken by Xiaoyun Wang and her colleagues in 2004 [1979, 1981]; 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-1 the following year, providing an algorithm to find collisions in only 2^{69} steps [1980]; it now takes about 2^{60} computations. In February 2017, scientists from Amsterdam and Google published one, to prove the point and help persuade people to move to stronger hash functions such as SHA-2 [1828] (and from earlier versions of TLS to TLS 1.3). In 2020, Gaëtan Leurent and Thomas Peyrin developed an improved attack that computes chosen-prefix collisions, enabling certificate forgery at a cost of several tens of thousands of dollars [1146].

In 2007, the US National Institute of Standards and Technology (NIST) organised a competition to find a replacement hash function family [1409]. The winner, Keccak, has a quite different internal structure, and was standardised as SHA-3 in 2015. So we now have a choice of SHA-2 and SHA-3 as standard hash functions.

A lot of deployed systems still use hash functions such as MD5 for which there's an easy collision-search algorithm. Whether a collision will break any given application can be a complex question. I already mentioned forensic systems, which keep hashes of files on seized computers, to reassure the court that the police didn't tamper with the evidence; a hash collision would merely signal that someone had been trying to tamper, whether the police or the defendant, and trigger a more careful investigation. If bank systems actually took a message composed by a customer saying 'Pay X the sum Y', hashed it and signed it, then a crook 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, and logs are kept by all the parties to a transaction, so it's not easy to sneak in one of a colliding pair. And in both cases you'd probably have to find a preimage of an existing hash value, which is a much harder cryptanalytic task than finding a collision.

5.6.2 Hash function applications – HMAC, commitments and updating

But even though there may be few applications where a collision-finding algorithm could let a bad guy to steal real money today, the existence of a vulnerability can still undermine a system's value. Some people doing forensic work continue to use MD5, as they've used it for years, and its collisions don't give useful attacks. This is probably a mistake. 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 in this application. The judge was "not satisfied beyond reasonable doubt that the photograph [had] not been altered since it was taken" and acquitted the motorist; his strange ruling was upheld on appeal the following year [1432]. So even if a vulnerability doesn't present an engineering threat, it can still present a *certificational* threat.

Hash functions have many other uses. One of them is to compute MACs. A naïve method would be to hash the message with a key: $\mathrm{MAC}_k(M) = h(k, M)$. However the accepted way of doing this, called HMAC, uses an extra step in which the result of this computation is hashed again. The two hashing operations are done using variants of the key, derived by exclusive-or'ing them with two different constants. Thus $\mathrm{HMAC}_k(M) = h(k \oplus B, h(k \oplus A, M))$. A is constructed by repeating the byte `0x36` as often as necessary, and B similarly from the byte `0x5C`. If a hash function is on the weak side, this construction can make exploitable collisions harder to find [1089]. HMAC is now 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 publish a hash of the document, or send it to a commercial timestamping service, or have it mined into the Bitcoin blockchain. Later, when I reveal the document, the timestamp on its hash 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.

Merkle trees hash a large number of inputs to a single hash output. The inputs are hashed to values that form the leaves of a tree; each non-leaf node contains the hash of all the hashes at its child nodes, so the hash at the root is a hash of all the values at the leaves. This is a fast way to hash a large data structure; it's used in code signing, where you may not want to wait for all of an application's files to have their signatures checked before you open it. It's also widely used in blockchain applications; in fact, a blockchain is just a Merkle tree. It was invented by Ralph Merkle, who first proposed it to calculate a short hash of a large file of public keys [1296], particularly for systems where public keys are used only once. For example, a Lamport digital signature can be constructed from a hash function: you create a private key of 512 random 256-bit values k_i and publish the verification key V as their Merkle tree hash. Then to sign $h = \mathrm{SHA256}(M)$ you would reveal k_{2i} if the i-th bit of h is zero, and otherwise reveal k_{2i+1}. This is secure if the hash function is, but has the drawback that each key can be used only once. Merkle saw that you could generate a series of private keys by encrypting a counter with a master secret key, and then use a tree to hash the resulting public keys. However, for most purposes, people use signature algorithms based on number theory, which I'll describe in the next section.

One security-protocol use of hash functions is worth a 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.* A variant is *autokeying* where the principals update a key by hashing it with the messages they have exchanged since the last key change: $K_{i+1} = h(K_i, M_{i1}, M_{i2}, \ldots)$. If an attacker now compromises one of their systems and steals the key, then as soon as they exchange a message which he can't observe or guess, security will be recovered; again, the chain of compromise is broken. This property is known as *forward security.* It was first used in banking in EFT payment terminals in Australia [207, 209]. The use of asymmetric cryptography 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, security can be recovered even if the message is in plain sight. I'll describe how this works next.

5.7 Asymmetric crypto primitives

The commonly used building blocks in *asymmetric cryptography*, public-key encryption and digital signature are based on number theory. I'll give a brief overview here, and look in more detail at some of the mechanisms in Part II when I discuss applications.

The basic idea is to make the security of the cipher depend on the difficulty of solving a mathematical problem that's known to be hard, in the sense that a lot of people have tried to solve it and failed. The two problems used in almost all real systems are factorization and discrete logarithm.

5.7.1 Cryptography based on factoring

The *prime numbers* are the positive whole numbers with no proper divisors: the only numbers that divide a prime number are 1 and the number itself. By definition, 1 is not prime; so the primes are {2, 3, 5, 7, 11, ...}. The *fundamental theorem of arithmetic* states that each natural number greater than 1 factors into prime numbers in a way that is unique up to the order of the factors. It is easy to find prime numbers and multiply them together to give a composite number, but much harder to resolve a composite number into its factors. And lots of smart people have tried really hard since we started using cryptography based on factoring. The largest composite product of two large random primes to have been factorized in 2020 was RSA-250, an 829-bit number (250 decimal digits). This took the equivalent of 2700 years' work on a single 2.2GHz core; the previous record, RSA-240 in 2019, had taken the equivalent of 900 years [301]. 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 [43]. As for 1024-bit numbers, I expect the NSA can factor them already, and I noted in the second edition that 'an extrapolation of the history of factoring records suggests the first factorization will be published in 2018.' Moore's law is slowing down, and we're a year late. Anyway, organisations that want keys to remain secure for many years are already using 2048-bit numbers at least.

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 $\{1, 2, ..., p-1\}$ and multiply each of them modulo p by a, then cancel out $(p-1)!$ each side). For a general integer n, $a^{\phi(n)} \equiv 1 \pmod{p}$ where Euler's function $\phi(n)$ is the number of positive integers less than n with which it has no divisor in common (the proof is similar). So if n is the product of two primes pq then $\phi(n) = (p-1)(q-1)$.

In RSA, 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:

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

Similarly, the owner of a private key can operate on a message with it 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 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 general 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 deterministic (as RSA is), then an attacker might just precompute the possible ciphertexts and recognise them when they appear. With RSA, it's also 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. Every time we encrypt the same short message, say 'attack', we want to get a completely different ciphertext, and for these to be indistinguishable from each other as well as from the ciphertexts for 'retreat'. And there are good ways and bad ways of doing this.

Crypto theoreticians have wrestled for decades to analyse 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 we mentioned already; in this context it 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 [777]. In other words, we want the encryption to resist chosen-ciphertext attack as well as chosen-plaintext attack. There are a number of constructions that give semantic security, but they tend to be too ungainly for practical use.

The usual 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 $C_1 \oplus h(N)$ [212]. This was eventually proven to be secure. There are a number of public-key cryptography standards; PKCS #1 describes OAEP [993]. These block a whole lot of attacks that were discovered in the 20th century and about which people have mostly forgotten, such as the fact that an opponent can detect if you encrypt the same message with two different RSA keys. In fact, one of the things we learned in the 1990s was that randomisation helps make crypto protocols more robust against all sorts of attacks, and not just the mathematical ones. Side-channel attacks and even physical probing of devices take a lot more work.

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 [1008]. However, in some applications one might wish to include further data in the signature block, such as a timestamp, or some randomness to make side-channel attacks harder.

Many of the things that have gone wrong with real implementations have to do with side channels and error handling. One 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 could learn from the target whether a given c, when decrypted as $c^d \pmod n$, corresponds to a PKCS #1 message, then he could use this to decrypt or sign messages [264]. There have been many more side-channel attacks on common public-key implementations, typically via measuring the precise time taken to decrypt. RSA is also mathematically fragile; you can break it using homomorphisms, or if you have the same ciphertext encrypted under too many different small keys, or if the message is too short, or if two messages are related by a known polynomial, or in several other edge cases. Errors in computation can also give a result that's correct modulo one factor of the modulus and wrong modulo the other, enabling the modulus to be factored; errors can be inserted tactically, by interfering with the crypto device, or strategically, for example by the chipmaker arranging for one particular value of a 64-bit multiply to be computed incorrectly. Yet other attacks have involved stack overflows, whether by sending the attack code in as keys, or as padding in poorly-implemented standards.

5.7.2 Cryptography based on discrete logarithms

While RSA was the first public-key encryption algorithm deployed in the SSL and SSH protocols, the most popular public-key algorithms now are based on discrete logarithms. There are a number of flavors, some using normal modular arithmetic while others use *elliptic curves*. I'll explain the normal case first.

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 5.17:

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

Figure 5.17 – 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 $y = 5^x \pmod 7$; 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 efficiently. 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 One-way commutative encryption

Imagine we're back in ancient Rome, 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.

How can a suitable commutative encryption be implemented? The one-time pad does indeed 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 out, leaving 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 can be tricky to encode a message as a primitive root; but there's a simpler way to achieve the same effect.

5.7.2.2 Diffie-Hellman key establishment

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 walk through this. 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.

Alternatively, Alice and Bob can use transient keys, and get a mechanism for providing forward security. 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 (see Figure 5.19).

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

Figure 5.18 – the Diffie-Hellman key exchange protocol

Alice and Bob can now use the session key $g^{R_A R_B}$ to encrypt a conversation. If they used transient keys, rather than long-lived ones, they have managed to create a shared secret 'out of nothing'. Even if an opponent had inspected both their machines before this protocol was started, and knew all their stored private keys, then provided some basic conditions were met (e.g., that their random number generators were not predictable and no malware was left behind) 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. In what follows, we may write the Diffie-Hellman key derived from R_A and R_B as $DH(R_A, R_B)$ when we don't have to be explicit about which group we're working in and don't need to write out explicitly which is the private key R_A and which is the public key g^{R_A}.

Slightly more work is needed to provide a full solution. Some care is needed when choosing the parameters p and g; we can infer from the Snowden disclosures, for example, that the NSA can solve the discrete logarithm problem for commonly-used 1024-bit prime numbers[6]. And there are several other details which depend on whether we want properties such as forward security.

But this protocol has a small problem: although Alice and Bob end up with a session key, neither of them has any real 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 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 $DH(R_A, R_C)$ which he shares with Alice, and another key $DH(R_B, R_C)$ 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 the STU-2 telephone, which is now obsolete but which you can see in the NSA museum at Fort Meade, 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. Something similar is implemented in Bluetooth versions 4 and later, but is complicated by the many versions that the protocol has evolved to support devices with different user interfaces. The protocol has suffered from multiple attacks, most recently the key negotiation of Bluetooth (KNOB) attack, which allows a middleperson to force one-byte keys that are easily brute forced; all devices produced before 2018 are vulnerable [123]. The standard allows for key lengths between one and sixteen bytes;

[6] The likely discrete log algorithm, NFS, involves a large computation for each prime number followed by a smaller computation for each discrete log modulo that prime number. The open record is 795 bits, which took 3,100 core-years in 2019 [?], using a version of NFS that's three times more efficient than ten years ago. There have been persistent rumours of a further NSA improvement and in any case the agency can throw a lot more horsepower at an important calculation.

as the keylength negotiation is performed in the clear, an attacker can force the length to the lower limit. All standards-compliant chips are vulnerable; this may be yet more of the toxic waste from the Crypto Wars, which I discuss in section 26.2.7. Earlier versions of Bluetooth are more like the 'just-works' mode of the HomePlug protocol described in section 14.3.3.3 in that they were principally designed to help you set up a pairing key with the right device in a benign environment, rather than defending against a sophisticated attack in a hostile one. The more modern ones appear to be better, but it's really just theatre.

So many things go wrong: protocols that will generate or accept very weak keys and thus give only the appearance of protection; programs that leak keys via side channels such as the length of time they take to decrypt; and software vulnerabilities leading to stack overflows and other hacks. If you're implementing public-key cryptography you need to consult up-to-date standards, use properly accredited toolkits, and get someone knowledgeable to evaluate what you've done. And please don't write the actual crypto code on your own – doing it properly requires a lot of different skills, from computational number theory to side-channel analysis and formal methods. Even using good crypto libraries gives you plenty of opportunities to shoot your foot off.

5.7.2.3 ElGamal digital signature and DSA

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 with 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 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 assumes a prime p of typically 2048 bits[7], a prime q of 256 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 by default is SHA256[8].

DSA is the classic example of a randomised digital signature scheme without message recovery. The most commonly-used version nowadays is ECDSA, a variant based on elliptic curves, which we'll discuss now – this is for example the standard for cryptocurrency and increasingly also for certificates in bank smartcards.

5.7.3 Elliptic curve cryptography

Discrete logarithms and their analogues exist in many other mathematical structures. *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 the resulting *Mordell group* can be used for cryptography. The algebra gets a bit complex and this book isn't the place to set it out[9]. However, elliptic curve cryptosystems are interesting for at least two reasons.

First is performance; 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 shorter variables; both are welcome in constrained environments. Elliptic curve cryptography is used in applications from the latest versions of EMV payment cards to Bitcoin.

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 [286]. 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

[7] In the 1990s p could be in the range 512–1024 bits and q 160 bits; this was changed to 1023–1024 bits in 2001 [1402] and 1024–3072 bits in 2009, with q in the range 160–256 bits [1403].

[8] The default sizes of p are chosen to be 2048 bits and q 256 bits in order to equalise the work factors of the two best known cryptanalytic attacks, namely the number field sieve whose running speed depends on the size of p and Pollard's rho which depends on the size of q. Larger sizes can be chosen if you're anxious about Moore's law or about progress in algorithms.

[9] See Katz and Lindell [1022] for an introduction.

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 [1704]. 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 have been used in a few specialist systems: in Zcash for the payment privacy mechanisms, and in a UK government key-management protocol called Mikey-Sakke. Computing people's private keys from their email addresses or other identifiers may seem a neat hack, but it can be expensive when government departments are reorganised or renamed [115]. Most organisations and applications use ordinary public-key systems with certification of public keys, which I'll discuss next.

5.7.4 Certification authorities

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 authorised 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 keys or their signature verification keys giving certificates that contain a user's name, one or more of their public keys, and attributes such as authorisations. The CA might be run by the local system administrator; but it is most commonly a third party service such as Verisign whose business is to sign public keys after doing some due diligence about whether they are controlled by 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 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. Naming is hard, for starters; we discuss this in the following chapter on Distributed Systems. But often names aren't really what the protocol has to establish, as in the real world it's often about authorisation rather than authentication. Government systems are often about establishing not just a user's name or role but their security clearance level. In banking systems, it's about your balance, your available credit and your authority to spend it. In commercial systems, it's often about linking remote users to role-based access control. In user-facing systems, there is a tendency to dump on the customer as many of the compliance costs as possible [524]. There are many other things that can go wrong with certification at the level of systems engineering. At the level of politics, there are hundreds of certification authorities in a typical browser, they are all more or less equally trusted, and many nation states can coerce at least one of them[10]. The revocation of bad certificates is usually flaky, if it works at all. There will be much more on these topics later. With these warnings, it's time to look at the most commonly used public key protocol, TLS.

5.7.5 TLS

I remarked above that a server could publish a public key KS and any web browser could then send a message M containing a credit card number to it encrypted using KS: $\{M\}_{KS}$. This is in essence what the TLS protocol (then known as SSL) was designed to do, at the start of e-commerce. It was developed by Paul Kocher and Taher ElGamal in 1995 to support encryption and authentication in both directions, so that both `http` requests and responses can be protected against both eavesdropping and manipulation. It's the protocol that's activated when you see the padlock on your browser toolbar.

Here is a simplified description of the basic version of the protocol in TLS v1:

1. the client sends the server a *client hello* message that contains its name C, a transaction serial number $C\#$, and a random nonce N_C;

2. the server replies with a *server hello* message that contains its name S, a transaction serial number $S\#$, a random nonce N_S, and a certificate CS containing its public key KS. The client now checks the certificate CS, and if need be checks the key that signed it in another certificate, and so on back to a root certificate issued by a company such as Verisign and stored in the browser;

3. the client sends a *key exchange* message containing a *pre-master-secret* key, K_0, encrypted under the server public key KS. It also sends a *finished* message with a message authentication code (MAC) computed on all the messages to date. The key for this MAC is the *master-secret*, K_1. This key is computed by hashing the pre-master-secret key with the nonces sent by the client and server: $K_1 = h(K_0, N_C, N_S)$. From this point onward, all the traffic is encrypted; we'll write this as $\{...\}_{KCS}$ in the client-server direction and $\{...\}_{KSC}$ from the server to the client. These keys are generated in turn by hashing the nonces with K_1.

4. The server also sends a *finished* message with a MAC computed on all the messages to date. It then finally starts sending the data.

[10] The few that can't, try to cheat. In 2011 Iran hacked the CA Diginotar, and in 2019 Kazakhstan forced its citizens to add a local police certificate to their browser. In both cases the browser vendors pushed back fast and hard: Diginotar failed after it was blacklisted, while the Kazakh cert was blocked even if its citizens installed it manually. This of course raises issues of sovereignty.

$$\begin{aligned}
C \rightarrow S: &\quad C, C\#, N_C \\
S \rightarrow C: &\quad S, S\#, N_S, CS \\
C \rightarrow S: &\quad \{K_0\}_{KS} \\
C \rightarrow S: &\quad \{finished, MAC(K_1, everythingtodate)\}_{KCS} \\
S \rightarrow C: &\quad \{finished, MAC(K_1, everythingtodate)\}_{KSC}, \{data\}_{KSC}
\end{aligned}$$

Once a client and server have established a pre-master-secret, no more public-key operations are needed as further master secrets can be obtained by hashing it with new nonces.

5.7.5.1 TLS uses

The full protocol is more complex than this, and has gone through a number of versions. It has supported a number of different ciphersuites, initially so that export versions of software could be limited to 40 bit keys – a condition of export licensing that was imposed for many years by the US government. This led to downgrade attacks where a middleperson could force the use of weak keys. Other ciphersuites support signed Diffie-Hellman key exchanges for transient keys, to provide forward and backward secrecy. TLS also has options for bidirectional authentication so that if the client also has a certificate, this can be checked by the server. In addition, the working keys KCS and KSC can contain separate subkeys for encryption and authentication, as is needed for legacy modes of operation such as CBC plus CBC MAC.

As well as being used to encrypt web traffic, TLS has also been available as an authentication option in Windows from Windows 2000 onwards; you can use it instead of Kerberos for authentication on corporate networks. I will describe its use in more detail in the chapter on network attack and defence.

5.7.5.2 TLS security

Although early versions of SSL had a number of bugs [1973], SSL 3.0 and later appear to be sound; the version after SSL 3.0 was renamed TLS 1.0. It was formally verified by Larry Paulson in 1998, so we know that the idealised version of the protocol doesn't have any bugs [1502].

However, in the more than twenty years since then, there have been over a dozen serious attacks. Even in 1998, Daniel Bleichenbacher came up with the first of a number of attacks based on measuring the time it takes a server to decrypt, or the error messages it returns in response to carefully-crafted protocol responses [264]. TLS 1.1 appeared in 2006 with protection against exploits of CBC encryption and of padding errors; TLS 1.2 followed two years later, upgrading the hash function to SHA256 and supporting authenticated encryption; and meanwhile there were a number of patches dealing with various attacks that had emerged. Many of these patches were rather inelegant because of the difficulty of changing a widely-used protocol; it's difficult to change both the server and client ends at once, as any client still has to interact with millions of servers, many running outdated software, and most websites want to be able to deal with browsers of all ages and on all sorts of devices. This has been dealt with by the big service firms changing their browsers to reject obsolete ciphersuites, and to add features like *strict transport security* (STS) whereby a website can instruct browsers to only interact with it using https in future (to prevent downgrade attacks). The browser firms have also mandated a number of other supporting measures, from shorter certificate lifetimes to certificate transparency, which we'll discuss in the chapter on network attack and defence.

5.7.5.3 TLS 1.3

The most recent major upgrade to the core protocol, TLS 1.3, was approved by the IETF in January 2019 after two years of discussion. It has dropped backwards compatibility in order to end support for many old ciphers, and made it mandatory to establish end-to-end forward secrecy by means of a Diffie-Hellman key exchange at the start of each session. This has caused controversy with the banking industry, which routinely intercepts encrypted sessions in order to do monitoring for compliance purposes. This will no longer be possible, so banks will have to bear the legal discomfort of using obsolete encryption or the financial cost of redeveloping systems to monitor compliance at endpoints instead[11].

5.7.6 Other public-key protocols

Dozens of other public-key protocols have found wide use, including the following, most of which we'll discuss in detail later. Here I'll briefly mention code signing, PGP and QUIC.

5.7.6.1 Code signing

Code signing was introduced in the 1990s when people started downloading software rather than getting it on diskettes. It is now used very widely to assure the provenance of software. You might think that having a public signature-verification key in your software so that version N can verify an update to version $N + 1$ would be a simple application of public-key cryptography but this is far from the case. Many platforms sign their operating-system code, including updates, to prevent persistent malware; the mechanisms often involve trusted hardware such as TPMs and I'll discuss them in the next chapter in section 6.2.5. Some platforms, such as the iPhone, will only run signed code; this not only assures the provenance of software but enables platform owners to monetise apps, as I will discuss in section 22.3.2; games consoles are similar. As some users go to great lengths to jailbreak their devices, such platforms typically have trustworthy hardware to store the verification keys. Where that isn't available, verification may be done using code that is obfuscated to make it harder for malware (or customers) to tamper with it; this is a constant arms race, which I discuss in section 24.3.3. As for the signing key, the developer may keep it in a hardware security module, which is expensive and breaks in subtle ways discussed in section 20.5; there may be a chain of trust going back a commercial CA, but then have to worry about legal coercion by government

[11] The COVID-19 pandemic has given some respite: Microsoft had been due to remove support for legacy versions of TLS in spring 2020 but has delayed this.

agencies, which I discuss in section 26.2.7; you might even implement your own CA for peace of mind. In short, code signing isn't quite as easy as it looks, particularly when the user is the enemy.

5.7.6.2 PGP/GPG

During the 'Crypto Wars' in the 1990s, cyber-activists fought governments for the right to encrypt email, while governments pushed for laws restricting encryption; I'll discuss the history and politics in section 26.2.7. The crypto activist Phil Zimmermann wrote an open-source encryption product *Pretty Good Privacy* (PGP) and circumvented U.S. export controls by publishing the source code in a paper book, which could be posted, scanned and compiled. Along with later compatible products such as GPG, it has become fairly widely used among geeks. For example, sysadmins, Computer Emergency Response Teams (CERTs) and malware researchers use it to share information about attacks and vulnerabilities. It has also been built into customised phones sold to criminal gangs to support messaging; I'll discuss this later in section 25.4.1.

PGP has a number of features but, in its most basic form, each user generates private/public keypairs manually and shares public keys with contacts. There are command-line options to sign a message with your signature key and/or encrypt it using the public key of each of the intended recipients. Manual key management avoids the need for a CA that can be cracked or coerced. Many things were learned from the deployment and use of PGP during the 1990s. As I described in section 3.2.1, Alma Whitten and Doug Tygar wrote the seminal paper on security usability by assessing whether motivated but cryptologically unsophisticated users could understand it well enough to drive the program safely. Only four of twelve subjects were able to correctly send encrypted email to the other subjects, and every subject made at least one significant error.

5.7.6.3 QUIC

QUIC is a new UDP-based protocol designed by Google and promoted as an alternative to TLS that allows quicker session establishment and thus cutting latency in the ad auctions that happen as pages load; sessions can persist as people move between access points. This is achieved by a cookie that holds the client's last IP address, encrypted by the server. It appeared in Chrome in 2013 and now has about 7% of Internet traffic; it's acquired a vigorous standardisation community. Google claims it reduces search latency 8% and YouTube buffer time 18%. Independent evaluation suggests that the benefit is mostly on the desktop rather than mobile [1007], and there's a privacy concern as the server can use an individual public key for each client, and use this for tracking. As a general principle, one should be wary of corporate attempts to replace open standards with proprietary ones, whether IBM's EBCDIC coding standard of the 1950s and SNA in the 1970s, or Microsoft's attempts to 'embrace and extend' both mail standards and security protocols since the 1990s, or Facebook's promotion of Internet access in Africa that kept users largely within its walled garden. I'll discuss the monopolistic tendencies of our industry at greater length in Chapter 8.

5.7.7 Special-purpose primitives

Researchers have invented 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) [554]. Threshold signatures were first used in systems where a number of servers process transactions independently and vote independently on the outcome; they have more recently been used to implement business rules on cryptocurrency wallets such as 'a payment must be authorised by any two of the seven company 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 $R^e M \pmod{n}$, and give it to the signer who computes $(R^e M)^d = R.M^d \pmod{n}$. 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 some applications.

The first was in *digital cash*; you might want to be able to issue anonymous payment tokens to customers, and the earliest idea, due to David Chaum, was a way to sign 'digital coins' without knowing their serial numbers [411]. A 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 ensures a customer can get a bank to sign a coin without the banker knowing its serial number, and it was used in prototype road toll systems. The effect is that the digital cash can be anonymous for the spender. The main problem with digital cash was to detect people who spend the same coin twice, and this was eventually fixed using blockchains or other ledger mechanisms, as I discuss in section 20.7. Digital cash failed to take off because neither banks nor governments really want payments to be anonymous: anti-money-laundering regulations since 9/11 restrict anonymous payment services to small amounts, while both banks and bitcoin miners like to collect transaction fees.

Anonymous digital credentials are now used in attestation: the TPM chip on your PC motherboard might prove something about the software running on your machine without identifying you. Unfortunately, this led to designs for attestation in SGX (and its AMD equivalent) which mean that a single compromised device breaks the whole ecosystem. Anonymous signatures are also found in prototype systems for conducting electronic elections, to which I will return in section 25.5.

5.7.8 How strong are 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 256-bit elliptic scheme could be about as hard to break as a 128-bit block cipher with a 128-bit key; and the only public-key encryption schemes used in the NSA's Suite B of military algorithms are 384-bit elliptic curve systems. The traditional schemes, based on factoring and discrete log, now require 3072-bit keys to protect material at Top Secret, as there are shortcut attack algorithms such as the number field sieve. As a result, elliptic curve cryptosystems are faster.

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. By the third edition in 2019, bitcoin miners are finding 68-bit hash collisions every ten minutes, RSA-768 has been factored and Ed Snowden has as good as told us that the NSA can do discrete logs for a 1024-bit prime modulus.

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 could be built, then both factoring and discrete logarithm computations will become easy [1725]. So far only very small quantum devices have been built; although there are occasional claims of 'quantum supremacy' – of a quantum computer performing a task sufficiently faster than a conventional one to convince us that quantum superposition or entanglement is doing any real work – they seem to lead nowhere. I am sceptical (as are many physicists) about whether the technology will ever threaten real systems. I am even more sceptical about the value of quantum cryptography; it may be able to re-key a line encryption device that uses AES for bulk encryption, but we already know how to do that.

What's more, I find the security proofs offered for entanglement-based quantum cryptography to be unconvincing [311]. Theoretical physics has been stuck since the early 1970s when Gerard 't Hooft completed the Standard Model by proving the renormalisability of Yang-Mills. Since then, a series of ideas have come and gone, such as string theory [2032]. Quantum information theory is the latest enthusiasm. Its proponents talk up the mystery of the Bell tests, which are supposed to demonstrate that physics cannot be simultaneously local and causal. But alternative interpretations such as 't Hooft's cellular automaton model [916] and Grisha Volovik's superfluid model [1967] suggest that the Bell tests merely demonstrate the existence of long-range order in the quantum vacuum, like the order parameter of a superfluid. And there are now classical mechanical experiments that demonstrate the quantum-mechanical properties relevant to the Bell tests [1557]. When I point out such loopholes in the proofs claimed for quantum cryptosystems, the proponents' only answer appears to be personal abuse.

Personally I think it more likely that a major challenge to public-key cryptography would come in the form of a better algorithm for computing discrete logarithms on elliptic curves. These curves have a lot of structure; they are studied intensively by some of the world's smartest pure mathematicians; better discrete-log algorithms for curves of small characteristic were discovered in 2013 [168]; and the NSA is apparently moving away from using elliptic-curve crypto.

If quantum computers ever work, we have other 'post-quantum' algorithms ready to go for which quantum computers give no obvious advantage. In 2020, NIST began the third round of public review of submissions for the Post-Quantum Cryptography Standardization Process. The 65 initial submissions have been cut to 15 through two rounds of review[12]. One or more algorithms will now be chosen and standardised, so ciphersuites using them could be dropped into protocols such as TLS as upgrades. Many protocols in use could even be redesigned to use variants on Kerberos. If elliptic logarithms become easy, we have these resources and can also fall back to discrete logs in prime fields, or to RSA. But if elliptic logs become easy, bitcoins will become trivial to forge, and the cryptocurrency ecosystem would probably collapse, putting an end to the immensely wasteful mining operations I describe in section 20.7. So mathematicians who care about the future of the planet might do worse than to think about the elliptic logarithm problem.

5.7.9 What else goes wrong

Very few attacks on systems nowadays involve cryptanalysis in the sense of a mathematical attack on the encryption algorithm or key. There have indeed been attacks on systems designed in the 20th century, mostly involving keys that were kept too short by export-control rules, clueless designs or both. I already discussed in section 4.3.1 how weak crypto has facilitated a wave of car theft, as all the devices used for remote key entry were defeated one after another in 2005–15. In later chapters, I give examples of how the crypto wars and their export control rules resulted in attacks on door locks (section 13.2.5), mobile phones (section 22.2.1) and copyright enforcement (section 24.2.5).

Most attacks nowadays exploit the implementation. In chapter 2, I mentioned the scandal of NIST standardising a complicated random number generator based on elliptic curves that turned out to contain an NSA backdoor; see section 2.2.1.5. Poor random number generators have led to many other failures: RSA keys with common factors [1140], predictable seeds for discrete logs [1676], etc. These vulnerabilities have continued; thanks to the Internet of Things, the proportion of RSA certs one can find out there on the Internet that share a common factor with other RSA keys has actually risen between 2012 and 2020; 1 in 172 IoT certs are trivially vulnerable [1046].

Many of the practical attacks on cryptographic implementations that have forced significant changes over the past 20 years have exploited side channels such as timing and power analysis; I devote Chapter 19 to these.

[12] One of them, the McEliece cryptosystem, has been around since 1978; we've had digital signatures based on hash functions for about as long, and some of us used them in the 1990s to avoid paying patent royalties on RSA.

In Chapter 20, I'll discuss a number of systems that use public-key mechanisms in intricate ways to get interesting emergent properties, including the Signal messaging protocol, the TOR anonymity system, and cryptocurrencies. I'll also look at the crypto aspects of SGX enclaves. These also have interesting failure modes, some but not all of them relating to side channels.

In Chapter 21, I'll discuss protocols used in network infrastructure such as DKIM, DNSSec versus DNS over HTTP, and SSH.

5.8 Summary

Many ciphers fail because they're used badly, so the security engineer needs a clear idea of what different types of cipher do. This can be tackled at different levels; one is at the level of crypto theory, where we can talk about the random oracle model, the concrete model and the semantic security model, and hopefully avoid using weak modes of operation and other constructions. The next level is that of the design of individual ciphers, such as AES, or the number-theoretic mechanisms that underlie public-key cryptosystems and digital signature mechanisms. These also have their own specialised fields of mathematics, namely block cipher cryptanalysis and computational number theory. The next level involves implementation badness, which is much more intractable and messy. This involves dealing with timing, error handling, power consumption and all sorts of other grubby details, and is where modern cryptosystems tend to break in practice.

Peering under the hood of real systems, we've discussed how 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. A lot of systems fail because popular crypto libraries encourage programmers to use inappropriate modes of operation by exposing unsafe defaults. Never use ECB mode unless you really understand what you're doing.

There are many other things that can go wrong, from side channel attacks to poor random number generators. 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.

The moral is: Don't roll your own! Don't design your own protocols, or your own ciphers; and don't write your own crypto code unless you absolutely have to. If you do, then you not only need to read this book (and then read it again, carefully); you need to read up the relevant specialist material, speak to experts, and have capable motivated people try to break it. At the very least, you need to get your work peer-reviewed. Designing crypto is a bit like juggling chainsaws; it's just too easy to make fatal errors.

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, to the search for primitives that combine provable security properties with decent performance.

The best way to get a flavor of what's going on at the theoretical end of things is to read the last few years' proceedings of research conferences such as Crypto, Eurocrypt and Asiacrypt; work on cipher design appears at Fast Software Encryption; attacks on implementations often appear at CHES; while attacks on how crypto gets used in systems can be found in the top systems security conferences such as IEEE Security and Privacy, CCS and Usenix.

Further Reading

The classic papers by Whit Diffie and Martin Hellman [556] and by Ron Rivest, Adi Shamir and Len Adleman [1607] are the closest to required reading in this subject. Bruce Schneier's *Applied Cryptography* [1667] covers a lot of ground at a level a non-mathematician can understand, and got crypto code out there in the 1990s despite US export control laws, but is now slightly dated. Alfred Menezes, Paul van Oorshot and Scott Vanstone's *Handbook of Applied Cryptography* [1289] is one reference book on the mathematical detail. Katz and Lindell is the book we get our students to read for the math. It gives an introduction to the standard crypto theory plus the number theory you need for public-key crypto (including elliptic curves and index calculus) but is also dated: they don't mention GCM, for example [1022].

There are many more specialist books. The bible on differential cryptanalysis is by its inventors Eli Biham and Adi Shamir [245], while a good short tutorial on linear and differential cryptanalysis was written by Howard Heys [895]. Doug Stinson's textbook has another detailed explanation of linear cryptanalysis [1829]; 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 [1301]. Neal Koblitz has a good basic introduction to the mathematics behind public key cryptography [1060]; and the number field sieve is described by Arjen and Henrik Lenstra [1141]. For the practical attacks on TLS over the past twenty years, see the survey paper by Christopher Meyer and Joerg Schwenk [1302] as well as the chapter on Side Channels later in this book.

If you want to work through the mathematical detail of theoretical cryptology, there's an recent graduate textbook by Dan Boneh and Victor Shoup [287].

A less thorough but more readable introduction to randomness and algorithms is in [835]. Research at the theoretical end of cryptology is found at the FOCS, STOC, Crypto, Eurocrypt and Asiacrypt conferences.

The history of cryptology is fascinating, and so many old problems keep on recurring. The standard work is Kahn [1001]; there are also compilations of historical articles from *Cryptologia* [531, 529, 530] as well as several books on the history of cryptology in World War II by Kahn, Marks, Welchman and others [438, 1002, 1224, 2007]. The NSA Museum at Fort George Meade, Md., is also worth a visit, but perhaps the best is the museum 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, his work remained classified. The RSA algorithm was then invented by Clifford Cocks, and also kept secret. This story is told in [625]. 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. And the classified community did not pre-invent digital signatures; they remain the achievement of Whit Diffie and Martin Hellman.