Chapter 2

Block Ciphers

Block ciphers are the central tool in the design of protocols for shared-key cryptography. They are the main available "technology" we have at our disposal. This chapter will take a look at these objects and describe the state of the art in their construction.

It is important to stress that block ciphers are just tools—raw ingredients for cooking up something more useful. Block ciphers don't, by themselves, do something that an end-user would care about. As with any powerful tool, one has to learn to use this one. Even a wonderful block cipher won't give you security if you use don't use it right. But used well, these are powerful tools indeed. Accordingly, an important theme in several upcoming chapters will be on how to use block ciphers well. We won't be emphasizing how to design or analyze block ciphers, as this remains very much an art. The main purpose of this chapter is just to get you acquainted with what typical block ciphers look like. We'll look at two examples, DES and AES. DES is the "old standby." It is currently (year 2001) the most widely-used block cipher in existence, and it is of sufficient historical significance that every trained cryptographer needs to have seen its description. AES is a modern block cipher, and it is expected to supplant DES in the years to come.

2.1 What is a block cipher?

A block cipher is a function $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ that takes two inputs, a k-bit key K and an n-bit "plaintext" M, to return an n-bit "ciphertext" $C = E(K, M)$. The *key-length* k and the *block-length* n are parameters associated to the block cipher. They vary from block cipher to block cipher, as of course does the design of the algorithm itself. For each key $K \in \{0,1\}^k$ we let $E_K\colon \{0,1\}^l \to \{0,1\}^l$ be the function defined by $E_K(M) = E(K, M)$. For any block cipher, and any key K, it is required that the function E_K be a *permutation* on $\{0,1\}^n$. This means that it is a bijection (ie., a one-to-one and onto function) of $\{0,1\}^n$ to $\{0,1\}^n$.

Accordingly E_K has an inverse, and we can denote it E_K^{-1}. This function also maps $\{0,1\}^n$ to $\{0,1\}^n$, and of course we have $E_K^{-1}(E_K(M)) = M$ and $E_K(E_K^{-1}(C)) = C$ for all $M, C \in \{0,1\}^n$. We let $E^{-1}\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be defined by $E^{-1}(K, C) = E_K^{-1}(C)$; this is the inverse block cipher to E.

The block cipher E is a public and fully specified algorithm. Both the cipher E and its inverse E^{-1} should be easily computable, meaning given K, M we can readily compute $E(K, M)$, and given K, C we can readily compute $E^{-1}(K, C)$.

In typical usage, a random key K is chosen and kept secret between a pair of users. The function E_K is then used by the two parties to process data in some way before they send it to each other. Typically, the adversary will be able to see input-output examples for E_K, meaning pairs of the form (M, C) where $C = E_K(M)$. But, ordinarily, the adversary will not be shown the key K. Security relies on the secrecy of the key. So, as a first cut, you might think of the adversary's goal as recovering the key K given some input-output examples of E_K. The block cipher should be designed to make this task computationally difficult. Later we will refine this (fundamentally incorrect) view.

We emphasize that we've said absolutely nothing about what properties a block cipher should have. A function like $E_K(M) = M$ is a block cipher (the "identity block cipher"), but we shall not regard it as a "good" one. Only in the next chapter do we start to take up what "goodness" means for a block cipher.

How do real block ciphers work? Lets take a look at some of them to get a sense of this.

2.2 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is the quintessential block cipher. Even though it is now quite old, and on the way out, no discussion of block ciphers can really omit mention of this construction. DES is a remarkably well-engineered algorithm which has had a powerful influence on cryptography. It is in very widespread use, and probably will be for some years to come. Every time you use an ATM machine, you are using DES.

2.2.1 A brief history

In 1972 the NBS (National Bureau of Standards, now NIST, the National Institute of Standards and Technology) initiated a program for data protection and wanted as part of it an encryption algorithm that could be standardized. They put out a request for such an algorithm. In 1974, IBM responded with a design based on their "Lucifer" algorithm. This design would eventually evolve into the DES.

DES has a key-length of $k = 56$ bits and a block-length of $n = 64$ bits. It consists of 16 rounds of what is called a "Feistel network." We will describe more details shortly.

After NBS, several other bodies adopted DES as a standard, including ANSI (the American National Standards Institute) and the American Bankers Association.

The standard was to be reviewed every five years to see whether or not it should be re-adopted. Although there were claims that it would not be re-certified, the algorithm was re-certified again and again. Only recently did the work for finding a replacement begin in earnest, in the form of the AES (Advanced Encryption Standard) effort.

DES proved remarkably secure. There has, since the beginning, been one primary concern, and that was the threat of key-search. But for a fairly long time, the key size of 56 bits was good enough against all but very well-funded organizations. Interesting attacks emerged only in the nineties, and even then they don't break DES in a sense more significant than the threat of exhaustive key search. But with today's technology, 56 bits is just too small a key size for many security applications, as we will see.

2.2.2 Construction

The canonical description is FIPS 46 [7], but the construction is also described in most books on cryptography. We intend, eventually, to have a full description with pictures here, but meanwhile we suggest that you have in hand some description such as Susan Landau's article on DES [19], which is available from the course web page, and use that to follow along.

The DES algorithm takes a 56 bit key and a 64 bit plaintext input. (Sometimes it is said that the key is 64 bits long, but actually every eighth bit is ignored. It is often mandated to be the xor of the previous seven bits.) Notice the algorithm is public. You operate with a hidden key, but nothing about the algorithm is hidden. The first thing that happens is that the 64-bit input is hit with something called the initial permutation, or IP. This just shuffles bit positions. That is, each bit is moved to some other position. How? In a fixed and specified way, indicated in the standard. Similarly, right at the end, notice they apply the inverse of the same permutation. From now on, ignore these. They do not affect security (as far as anyone knows).

The essence of DES is in the round structure. There are 16 rounds. Each round i has an associated subkey K_i which is 48 bits long. The subkeys $K_1, \ldots, K_{16}$ are derived from the main key K.

In each round, the input is viewed as a pair (L_i, R_i) of 32 bit blocks, and these are transformed into the new pair (L_{i+1}, R_{i+1}), via a certain function f that depends on a subkey K_i associated to round i. The structure of this transformation is important: it is called the Feistel transformation.

The Feistel transformation, in general, is like this. For some function g known to the party computing the transformation, it takes input (L, R) and returns (L', R') where $L' = R$ and $R' = g(R) \oplus L$. A central property of this transformation is that it is a permutation, and moreover if you can compute g then you can also easily invert the transformation. Indeed, given (L', R') we can recover (L, R) via $R = L'$

and $L = g(R) \oplus R'$. For DES, the role of g in round i is played by $f(K_i, \cdot)$, the round function specified by the subkey K_i. Since $\mathrm{DES}_K(\cdot)$ is a sequence of Feistel transforms, each of which is a permutation, the whole algorithm is a permutation, and knowledge of the key K permits computation of $\mathrm{DES}_K^{-1}(\cdot)$.

Up to now the structure has been quite generic, and indeed many block-ciphers use this high level design: a sequence of Feistel rounds. For a closer look we need to see how the function $f(K_i, \cdot)$ works. This function maps 32 bits to 32 bits. See the picture on page 90 of the FIPS document. Here K_i is a 48-bit subkey, derived from the 56-bit key (just by selecting particular bits) in a way depending on the round number. The 32-bit R_i is first expanded into 48 bits. How? In a precise, fixed way, indicated by the table on the same page, saying E-bit selection table. It has 48 entries. Read it as which inputs bits are output. Namely, output bits 32, 1, 2, 3, 4, 5, then 4, 5 again, and so on. It is NOT random looking! In fact barring that 1 and 32 have been swapped (see top left and bottom right) it looks almost sequential. Why did they do this? Who knows. That's the answer to most things about DES.

Now K_i is XORed with the output of the E-box and this 48 bit input enters the famous S-boxes. There are eight S-boxes. Each takes 8 bits to 6 bits. Thus we get out 32 bits. Finally, there is a P-box, a permutation applied to these 32 bits to get another 32 bits.

What are the S-boxes? Each is a fixed, tabulated function, which the algorithm stores as tables in the code or hardware. You can see them on page 93. How to read them? Take the 6 bit input $b_1, b_2, b_3, b_4, b_5, b_6$. Interpret the first and last bits as a row number (row 0, 1, 2, or 3). Interpret the rest as a column number (column 0, 1, ..., 15). Now look up what you get in the table and write down those four bits.

Well now you know how DES works. Of course, the main questions about the design are: why, why and why? What motivated these design choices? We don't know too much about this, although we can guess a little. And one of the designers of DES, Don Coppersmith, has written a short paper which gives information on why (thought what Don wrote was information which had effectively been reverse-engineered out of the algorithm in the previous years).

2.2.3 Speed

One of the design goals of DES was that it would have fast implementations relative to the technology of its time. How fast can you compute DES? In roughly current technology (well, nothing is current by the time one writes it down!) one can get well over 1 Gbit/sec on high-end VLSI. Specifically at least 1.6 Gbits/sec, maybe more. That's pretty fast. Perhaps a more interesting figure is that one can implement each DES S-boxes with at most 50?? two-input gates, where the circuit has depth of only 3??. Thus one can compute DES by a combinatorial circuit of about $8 \cdot 16 \cdot 50 = 640$ gates and depth of $3 \cdot 16 = 48$ gates.

In software, on a fairly modern processor, DES takes something like 80(?) cycles per byte. This is disappointingly slow—not surprisingly, since DES was optimized for hardware and was designed before the days in which software implementations were considered feasible or desirable.

2.3 Advanced Encryption Standard (AES)

In 1998 the National Institute of Standards and Technology (NIST/USA) announced a "competition" for a new block cipher. The new block cipher would, in time, replace DES. The relatively short key length of DES was the main problem that motivated the effort: with the advances in computing power, a key space of 2^{56} keys was just too small. With the development of a new algorithm one could also take the opportunity to address the modest software speed of DES, making something substantially faster, and to increase the block size from 64 to 128 bits (the choice of 64 bits for the block size can lead to security difficulties, as we shall later see. Unlike the design of DES, the new algorithm would be designed in the open and by the public.

Fifteen algorithms were submitted to NIST. They came from around the world. A second round narrowed the choice to five of these algorithms. In the summer of 2001 NIST announced their choice: an algorithm called Rijndael. The algorithm should be embodied in a NIST FIPS (Federal Information Processing Standard) any day now; right now, there is a draft FIPS. Rijndael was designed by Joan Daemen and Vincent Rijmen (from which the algorithm gets its name), both from Belgium. It is descendent of an algorithm called Square.

In this section we shall describe AES.

A word about notation. Purists would prefer to reserve the term "AES" to refer to the standard, using the word "Rijndael" or the phrase "the AES algorithm" to refer to the algorithm itself. (The same naming pundits would have us use the acronym DEA, Data Encryption Algorithm, to refer to the algorithm of the DES, the Data Encryption Standard.) We choose to follow common convention and refer to both the standard and the algorithm as AES. Such an abuse of terminology never seems to lead to any misunderstandings. (Strictly speaking, AES is a special case of Rijndael. The latter includes more options for block lengths than AES does.)

The AES has a block length of $n = 128$ bits, and a key length k that is variable: it may be 128, 192 or 256 bits. So the standard actually specifies three different block ciphers: AES128, AES192, AES256. These three block ciphers are all very similar, so we will stick to describing just one of them, AES128. For simplicity, in the remainder of this section, AES means the algorithm AES128. We'll write $C = \mathrm{AES}_K(M)$ where $|K| = 128$ and $|M| = |C| = 128$.

We're going to describe AES in terms of four additional mappings: *expand*, S, *shift-rows*, and *mix-cols*. The function *expand* takes a 128-bit string and produces a vector of eleven keys, $(K_0, \ldots, K_{10})$. The remaining three functions bijectively map 128-bits to 128-bits. Actually, we'll be more general for S, letting git be a map on $((\{0,1\})^8)^+$. Let's postpone describing all of these maps and start off with the high-level structure of AES, which is given in Figure 2.1.

function $\text{AES}_K(M)$
begin
 $(K_0, \ldots, K_{10}) \leftarrow \mathit{expand}(K)$
 $s \leftarrow M \oplus K_0$
 for $r = 1$ to 10 **do**
  $s \leftarrow S(s)$
  $s \leftarrow \mathit{shift\text{-}rows}(s)$
  if $r \leq 9$ **then** $s \leftarrow \mathit{mix\text{-}cols}(s)$ **fi**
  $s \leftarrow s \oplus K_r$
 endfor
 return s
end

Figure 2.1: The function AES128. See the accompanying text and figures for definitions of the maps *expand*, S, *shift-rows*, *mix-cols*.

Refer to Figure 2.1. The value s is called the *state*. One initizlizes the state to M and the final state is the ciphertext C on gets by enciphering M. What happens in each of lines 1–10 is called a *round*. So AES (remember this means AES128 in this section) consists of ten rounds. The rounds are identical except that each uses a different subkey K_i and, also, round 10 omits the call to *mix-cols*.

To understand what goes on in S and *mix-cols* we will need to review a bit of algebra. Let us make a pause to do that. We describe a way to do arithmetic on bytes. Identify each byte $a = a_7a_6a_5a_4a_3a_2a_1a_0$ with the formal polynomial $a_7\mathtt{x}^7 + a_6\mathtt{x}^6 + a + 5\mathtt{x}^5 + a_4\mathtt{x}^4 + a_3\mathtt{x}^3 + a_2\mathtt{x}^2 + a_1\mathtt{x} + a_0$. We can add two bytes by taking their bitwise xor (which is the same as the mod-2 sum the corresponding polynomials). We can multiply two bytes to get a degree 14 (or less) polynomial, and then take the remainder of this polynomial by the fixed irreducible polynomial

$$m(\mathtt{x}) = \mathtt{x}^8 + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1 \;.$$

This remainder polynomial is a polynomial of degree at most seven which, as before, can be regarded as a byte. In this way can add and multiply any two bytes. The resulting algebraic structure has all the properties necessary to be called a *finite field.* In particular, this is one representation of the finite field known as $\text{GF}(2^8)$—the Galois field on $2^8 = 256$ points. As a finite field, you can find the inverse of any nonzero field point (the zero-element is the zero byte) and you can distribute addition over multiplication, for example.

There are some useful tricks when you want to multiply two bytes. Since $m(\mathtt{x})$ is another name for zero, $\mathtt{x}^8 = \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1 = \{1\text{b}\}$. (Here the curly brackets simply indicate a hexadecimal number.) So it is easy to multiply a byte a by the byte $\mathtt{x} = \{02\}$: namely, shift the 8-bit byte a one position to the left, letting the first bit "fall off" (but remember it!) and shifting a zero into the last bit position.

We write this operation $a \lll 1$. If that first bit of a was a 0, we are done. If the first bit was a 1, we need to add in (that is, xor in) $\mathtt{x}^8 = \{1\text{b}\}$. In summary, for a a byte, $a \cdot \mathtt{x} = a \cdot \{02\}$ is $a \lll 1$ if the first bit of a is 0, and it is $(a \lll 1) \oplus \{1\text{b}\}$ if the first bit of a is 1.

Knowing how to multiply by $\mathtt{x} = \{02\}$ let's you conveniently multiply by other quantities. For example, to compute $\{\text{a}1\} \cdot \{03\}$ compute $\{\text{a}1\} \cdot (\{02\} \oplus \{01\}) = \{\text{a}1\} \cdot \{02\} \oplus \{\text{a}1\} \cdot \{01\} = \{42\} \oplus \{1\text{b}\} \oplus a1 = \{\text{f}8\}$. Try some more examples on your own.

As we said, each nonzero byte a has a multiplicative inverse, $\mathit{inv}(a) = a^{-1}$, The mapping we will denote $S : \{0,1\}^8 \to \{0,1\}^8$ is obtained from the map $\mathit{inv} : a \mapsto a^{-1}$. First, patch this map to make it total on $\{0,1\}^8$ by setting $\mathit{inv}(\{00\}) = \{00\}$. Then, to compute $S(a)$, first replace a by $\mathit{inv}(a)$, number the bits of a by $a = a_7a_6a_5a_4a_3a_2a_1a_0$, and return the value a', where $a' = a'_7a'_6a'_5a'_4a'_3a'_2a'_1a'_0$ where

$$\begin{pmatrix} a'_7 \\ a'_6 \\ a'_5 \\ a'_4 \\ a'_3 \\ a'_2 \\ a'_1 \\ a'_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

All arithmetic is in GF(2), meaning that addition of bits is their xor and multiplication of bits is the conjunction (and).

All together, the map S is give by Figure 2.2, which lists the values of

$$S(0), S(1), \ldots, S(255)\ .$$

In fact, one could forget how this table is produced, and just take it for granted. But the fact is that it is made in the simple way we have said.

Now that we have the function S, let us extend it (without bothering to change the name) to a function with domain $\{\{0,1\}^8\}^+$. Namely, given an m-byte string $A = A[1] \ldots A[m]$, set $S(A)$ to be $S(A[1]) \ \ldots S(A[m])$. In other words, just apply S bytewise.

Now we're ready to understand the first map, $S(s)$. One takes the 16-byte state s and applies the 8-bit lookup table to each of its bytes to get the modified state s.

Moving on, the *shift-rows* operation works like this. Imagine plastering the 16 bytes of $s = s_0s_1 \ldots s_{15}$ going top-to-bottom, then left-to-right, to make a 4×4 table:

$$\begin{matrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{matrix}$$

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

Figure 2.2: The AES S-box, which is a function $S:\{0,1\}^8 \to \{0,1\}^8$ specified by the following list. All values in hexadecimal. The meaning is: $S(00) = 63$, $S(01) = 7c$, $\ldots$, $S(\mathrm{ff}) = 16$.

For the *shift-rows* step, left circularly shift the second row by one position; the third row by two positions; and the the fourth row by three positions. The first row is not shifted at all. Somewhat less colorfully, the mapping is simply

$$\textit{shift-rows}(s_0 s_1 s_2 \cdots s_{15}) \quad = \quad s_0 s_5 s_{10} s_{15} s_4 s_9 s_{14} s_3 s_8 s_{13} s_2 s_7 s_{12} s_1 s_6 s_{11}$$

Using the same convention as before, the *mix-cols* step takes each of the four columns in the 4×4 table and applies the (same) transformation to it. Thus we define $\textit{mix-cols}(s)$ on 4-byte words, and then extend this to a 16-byte quantity wordwise. The value of $\textit{mix-cols}(a_0 a_1 a_2 a_3) = a'_0 a'_1 a'_2 a'_3$ is defined by:

$$\begin{pmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 02 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

An equivalent way to explain this step is to say that we are multiplying $a(\mathtt{x}) = a_3\mathtt{x}^3 + a_2\mathtt{x}^2 + a_1\mathtt{x}^1 + a_0$ by the fixed polynomial $c(\mathtt{x}) = \{03\}\mathtt{x}^3 + \{01\}\mathtt{x}^2 + \{01\}\mathtt{x} + \{02\}$ and taking the result modulo $\mathtt{x}^4 + 1$.

At this point we have described everything but the key-expansion map, *expand*. That map is given in Figure 2.3.

We have now completed the definition of AES. One key property is that AES *is* a block cipher: the map is invertible. This follows because every round is invertible.

function *expand*(K)
begin
  $K_0 \leftarrow K$
  for $i \leftarrow 1$ **to** 10 **do**
    $K_i[0] \leftarrow K_{i-1}[0] \oplus S(K_{i-1}[3] \lll 8) \oplus \mathrm{C}_i$
    $K_i[1] \leftarrow K_{i-1}[1] \oplus K_i[0]$
    $K_i[2] \leftarrow K_{i-1}[2] \oplus K_i[1]$
    $K_i[3] \leftarrow K_{i-1}[3] \oplus K_i[2]$
  od
  return $(K_0, \ldots, K_{10})$
end

Figure 2.3: The AES128 key-expansion algorithm maps a 128-bit key K into eleven 128-bit subkeys, $K_0, \ldots, K_{10}$. Constants $(\mathrm{C}_1, \ldots, \mathrm{C}_{10})$ are ({02000000}, {04000000}, {08000000}, {10000000}, {20000000}, {40000000}, {80000000}, {1B000000}, {36000000}, {6C000000}). All other notation is described in the accompanying text.

That a round is invertible follows from each of its steps being invertible, which is a consequence of S being a permutation and the matrix used in *mix-cols* having an inverse .

In the case of DES, the rationale for the design were not made public. Some explanation for different aspects of the design have become more apparent over time as we have watched the effects on DES of new attack strategies, but fundamentally, the question of why the design is as it is has not received a satisfying cipher. In the case of AES there was significantly more documentation of the rationale for design choices. (See the book *The design of Rijndael* by the designers [9]).

Nonetheless, the security of block ciphers, including DES and AES, eventually comes down to the statement that "we have been unable to find effective attacks, and we have tried attacks along the following lines ...." If people with enough smarts and experience utter this statement, then it suggests that the block cipher is good. Beyond this, it's hard to say much. Yet, by now, our community has become reasonably experienced designing these things. It wouldn't even be that hard a game were it not for the fact we tend to be agressive in optimizing the block-cipher's speed. (Some may come to the opposite opinion, that it's a very hard game, seeing just how many reasonable-looking block ciphers have been broken.) Later we give some vague sense of the sort of cleverness that people muster against block ciphers.

2.4 Some modes of operation

Fix a block cipher E, and assume two parties share a key K for this block cipher. This gives them the ability to compute the functions $E_K(\cdot)$ and $E_K^{-1}(\cdot)$. These functions can be applied to an input of n-bits. An application of E_K is a called *enciphering* and an application of E_K^{-1} is called *deciphering*.

Typically the block size n is 64 or 128. Yet, in practice, we may want to process much larger inputs, say text files to encrypt. To do this one uses a block cipher in some *mode of operation.* There are several well-known modes of operation. We will illustrate by describing three of them, all intended for message privacy. We look at ECB (Electronic Codebook), CBC (Cipher Block Chaining) and CTR (Counter). In each case there is an encryption process which takes an nm-bit string M, usually called the plaintext, and returns a string C, usually called the ciphertext. (If the length of M is not a positive multiple of n then some appropriate padding can be done to make it so. We're not going to worry about that here; we'll simply assume that each plaintext M has a length which is some positive multiple of n.) An associated decryption process recovers M from C.

If M is a string whose length is a multiple of n then we view it as divided into a sequence of n-bit blocks, and let $M[i]$ denote the i-th block, for $i = 1, \ldots, |M|/n$. That is, $M = M[1] \ldots M[m]$ where $m = |M|/n$.

2.4.1 Electronic codebook mode

Each plaintext block is individually enciphered into an associated ciphertext block.

Algorithm $\mathcal{E}_K(M[1] \cdots M[m])$	Algorithm $\mathcal{D}_K(C[1] \ldots C[m])$
For $i = 1, \ldots, m$ do	For $i = 1, \ldots, m$ do
$C[i] \leftarrow E_K(M[i])$	$M[i] \leftarrow E_K^{-1}(C[i])$
Return $C[1] \ldots C[m]$	Return $M[1] \cdots M[m]$

2.4.2 Cipher-block chaining mode

CBC mode processes the data based on some *initialization vector* IV which is an l-bit string, as follows.

Algorithm $\mathcal{E}_K(\mathrm{IV},\ M[1] \cdots M[m])$	Algorithm $\mathcal{D}_K(C[0]C[1] \ldots C[n])$
$C[0] \leftarrow \mathrm{IV}$	For $i = 1, \ldots, n$ do
For $i = 1, \ldots, n$ do	$M[i] \leftarrow E_K^{-1}(C[i]) \oplus C[i-1]$
$C[i] \leftarrow E_K(C[i-1] \oplus M[i])$	Return $M[1] \ldots M[n]$
Return $C[0]C[1] \cdots C[m]$	

Unlike ECB encryption, this operation is not length preserving: the output is n-bits longer than the input. The initialization vector is used for encryption, but it is then made part of the ciphertext, so that the receiver need not be assumed to know it a priori.

Different specific modes result from different ways of choosing the initialization vector. Unless otherwise stated, it is assumed that before applying the above encryption operation, the encryptor chooses the initialization vector at random, anew for each message M to be encrypted. Other choices however can also be considered, such as letting IV be a counter that is incremented by one each time the algorithm is applied. The security attributes of these different choices are discussed later.

CBC is the most popular mode, used pervasively in practice.

2.4.3 Counter mode

CTR mode also uses an auxiliary value, an "initial value" IV which is an integer in the range $0, 1, \ldots, 2^n - 1$. In the following, addition is done modulo 2^n, and $[j]_n$ denotes the binary representation of integer j as an n-bit string.

Algorithm $\mathcal{E}_K(\mathrm{IV},\ M[1] \cdots M[m])$	Algorithm $\mathcal{D}_K([\mathrm{IV}]_n C[1] \ldots C[m])$
For $i = 1, \ldots, m$ do	For $i = 1, \ldots, m$ do
$C[i] \leftarrow E_K([\mathrm{IV} + i]_n) \oplus M[i]$	$M[i] \leftarrow E_K([\mathrm{IV} + i]_n) \oplus C[i]$
Return $[\mathrm{IV}]_n\ C[1] \cdots C[n]$	Return $M[1] \ldots M[m]$

Notice that in this case, decryption did not require computation of E_K^{-1}, and in fact did not even require that E_K be a permutation. Also notice the efficiency advantage over CBC: the encryption is parallelizable.

Again, there are several choices regarding the initial vector. It could be a counter maintained by the sender and incremented by $m = |M|/n$ after message M has been encrypted. Or, it could be chosen anew at random each time the algorithm is invoked. And there are still other possibilities.

2.5 Key recovery attacks on block ciphers

Historically, cryptanalysis of block ciphers $E\colon\ \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ has always focused on key-recovery. The cryptanalyst may think of the problem to be solved as something like this. A k-bit key K is chosen at random. Let $q \geq 0$ be some integer parameter.

Given: The adversary has a sequence of q input-output examples of E_K, say

$$(M_1, C_1), \ldots, (M_q, C_q)$$

where $C_i = E_K(M_i)$ for $i = 1, \ldots, q$ and $M_1, \ldots, M_q$ are all distinct n-bit strings.

Find: The adversary must find the key K.

Some typical kinds of "attack" that are considered within this framework:

Known-message attack: $M_1, \ldots, M_q$ are any distinct points; the adversary has no control over them, and must work with whatever it gets.

Chosen-message attack: $M_1, \ldots, M_q$ are chosen by the adversary, perhaps even adaptively. That is, imagine it has access to an "oracle" for the function E_K. It can feed the oracle M_1 and get back $C_1 = E_K(M_1)$. It can then decide on a value M_2, feed the oracle this, and get back C_2, and so on.

Clearly a chosen-message attack gives the adversary much more power, but is also less realistic in practice.

The most obvious attack is exhaustive key search.

Exhaustive key search: Go through all possible keys $K' \in \{0,1\}^k$ until you find one that explains the input/output pairs. Here is the attack in detail, using $q = 2$, meaning two input-output examples. For $i = 1, \ldots, 2^k$ let K_i denote the i-th k-bit string (in lexicographic order).

For $i = 1, \ldots, 2^k$ do
    If $E(K_i, M_1) = C_1$
        then if $E(K_i, M_2) = C_2$ then return K_i

The key returned by the above procedure is not necessarily equal to the target key K under which the input-output examples were generated; it might be that we found a key K' that happened to agree with K on the target examples but was different from K. In practice, it turns out that this is very rare, meaning that the key returned by the above is almost always the right one.

How long does exhaustive key-search take? In the worst case, 2^k computations of the block cipher. For the case of DES, even if you use the above mentioned 1.6 Gbits/sec chip to do these computations, the search takes about 6,000 years. So key search appears to be infeasible.

Yet, this conclusion is actually too hasty. We will return to key search and see why later.

Differential and linear cryptanalysis: For DES, the discovery of theoretically superior attacks (assuming one has massive amount of plaintext/ciphertext pairs) waited until 1990. Differential cryptanalysis is capable of finding a DES key using about 2^{47} input-output examples (that is, it requires $q = 2^{47}$). However, differential cryptanalysis required a chosen-message attack.

Linear cryptanalysis [**?**] improved differential in two ways. The number of input-output examples required is reduced to 2^{44}, but also only a known-message attack is required. (An alternative version uses 2^{42} chosen plaintexts [24].)

These were major breakthroughs in cryptanalysis. Yet, their practical impact is small. Why? Ordinarily it would be impossible to obtain 2^{44} input-output examples. Furthermore, simply storing all these examples requires about 280 terabytes.

Linear and differential cryptanalysis were however more devastating when applied to other ciphers, some of which succumbed completely to the attack.

So what's the best possible attack against DES? The answer is exhaustive key search. What we ignored above is *parallelism*.

Key search machines: A few years back it was argued that one can design a \$1 million machine that does the exhaustive key search for DES in about 3.5 hours. More recently, a DES key search machine was actually built, at a cost of \$250,000. It finds the key in 56 hours, or about 2.5 days. The builders say it will be cheaper to build more machines now that this one is built.

Thus DES is feeling its age. Yet, it would be a mistake to take away from this discussion the impression that DES is weak. Rather, what the above says is that it is an impressively strong algorithm. After all these years, the best practical attack known is still exhaustive key search. That says a lot for its design and its designers.

Later we will see that we would like security properties from a block cipher that go beyond resistance to key-recovery attacks. It turns out that from that point of view, a limitation of DES is its block size. Birthday attacks "break" DES with about $q = 2^{32}$ input output examples. (The meaning of "break" here is very different from above.) Here 2^{32} is the square root of 2^{64}, meaning to resist these attacks we must have bigger block size. The next generation of ciphers—things like AES—took this into account.

2.6 Limitations of key-recovery based security

As discussed above, classically, the security of block ciphers has been looked at with regard to key recovery. That is, analysis of a block cipher E has focused primarily on the following question: given some number q of input-output examples $(M_1, C_1)), \ldots, (M_q, C_q)$, where K is a random, unknown key and $C_i = E_K(M_i)$, how hard is it for an attacker to find K? A block cipher is viewed as "secure" if the best key-recovery attack is computationally infeasible, meaning requires a value of q or a running time t that is too large to make the attack practical. In the sequel, we refer to this as *security against key-recovery*

However, as a notion of security, security against key-recovery is quite limited. A good notion should be sufficiently strong to be useful. This means that if a block cipher is secure, then it should be possible to use the block cipher to make worthwhile constructions and be able to have some guarantee of the security of these constructions. But even a cursory glance at common block cipher usages shows that good security in the sense of key recovery is not sufficient for security of the usages of block ciphers.

Take for example the CTR mode of operation discussed in Section 2.4. Suppose that the block cipher had the following weakness: Given $C, F_K(C+1), F_K(C+2)$, it is possible to compute $F_K(C+3)$. Then clearly the encryption scheme is not secure, because if an adversary happens to know the first two message blocks, it can figure out the third message block from the ciphertext. (It is perfectly reasonable to assume the adversary already knows the first two message blocks. These might, for example, be public header information, or the name of some known recipient.) This means that if CTR mode encryption is to be secure, the block cipher must have the property that given $C, F_K(C+1), F_K(C+2)$, it is computationally infeasible to compute $F_K(C+3)$. Let us call this property SP1, for "security property one".

Of course, anyone who knows the key K can easily compute $F_K(C+3)$ given $C, F_K(C+1), F_K(C+2)$. And it is hard to think how one can do it without knowing the key. But there is no guarantee that someone *cannot* do this without knowing the key. That is, confidence in the security of F against key recovery does *not* imply that SP1 is true.

This phenomenon continues. As we see more usages of ciphers, we build up a longer and longer list of security properties SP1, SP2, SP3, ... that are necessary for the security of some block cipher based application.

Furthermore, even if SP1 is true, CTR mode encryption may still be weak. SP1 is not *sufficient* to guarantee the security of CTR mode encryption. Similarly with other security properties that one might naively come up with.

This long list of necessary but not sufficient properties is no way to treat security. What we need is *one single* "MASTER" property of a block cipher which, if met, *guarantees* security of *lots of* natural usages of the cipher.

A good example to convince oneself that security against key recovery is not enough is to consider the block cipher E: $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ defined for all keys $K \in \{0,1\}^k$ and plaintexts $x \in \{0,1\}^n$ by $F(K,x) = x$. That is, each instance F_K of the block cipher is the identity function. Is this a "good" block cipher? Surely not. Yet, it is exceedingly secure against key-recovery. Indeed, given any number of input-output examples of F_K, an adversary cannot even test whether a given key is the one in use.

This might seem like an artificial example. Many people, on seeing this, respond by saying: "But, clearly, DES and AES are *not* designed like this." True. But that is missing the point. The point is that security against key-recovery *alone* does not make a "good" block cipher. We must seek a better notion of security. Chapter 3 on pseudorandom functions does this.

2.7 Problems

Exercise 2.1 Show that for all $K \in \{0,1\}^{56}$ and all $x \in \{0,1\}^{64}$

$$\mathrm{DES}_K(x) = \overline{\mathrm{DES}_{\overline{K}}(\overline{x})} .$$

This is called the key-complementation property of DES.

Exercise 2.2 Explain how to use the key-complementation property of DES to speed up exhaustive key search by about a factor of two. Explain any assumptions that you make.

Exercise 2.3 Find a key K such that $\mathrm{DES}_K(\cdot) = \mathrm{DES}_K^{-1}(\cdot)$. Such a key is sometimes called a "weak" key.

Exercise 2.4 As with AES, suppose we are working in the finite field with 2^8 elements, representing field points using the irreducible polynomial $m(\mathtt{x}) = \mathtt{x}^8 + \mathtt{x}^4 + \mathtt{x}^3 + \mathtt{x} + 1$. Compute the byte that is the result of multiplying bytes:

$$\{\mathrm{e1}\} \cdot \{05\}$$

Exercise 2.5 For AES, we have given two different descriptions of *mix-cols*: one using matric multiplication (in $\mathrm{GF}(2^8)$) and one based on multiplying by a fixed polynomial $c(\mathtt{x})$ modulo a second fixed polynomial, $d(\mathtt{x}) = \mathtt{x}^4 + 1$. Show that these two methods are equivalent.

Exercise 2.6 Verify that the matrix used for *mix-cols* has as its inverse the matrix

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix}$$

Explain why all entries in this matrix begin with a zero-byte.

Exercise 2.7 How many different permutations are there from 128 bits to 128 bits? How man different functions are then from 128 bits to 128 bits?

Exercise 2.8 Upper and lower bound, as best you can, the probability that a random function from 128 bits to 128 bits is actually a permutation.

Problem 2.1 Without consulting any of the numerous public-domain implementations available, implement AES, on your own, from the spec or from the description provided by this chapter. Then test your implementation according to the test vectors provided in the AES documentation.

Problem 2.2 Justify and then refute (both) the following proposition: enciphering under AES can be implemented faster than deciphering.