Information Security: Principles and Practice, Second Edition
by Mark Stamp

Chapter 3

Symmetric Key Crypto

The chief forms of beauty are order and symmetry...
— Aristotle

"You boil it in sawdust: you salt it in glue:
You condense it with locusts and tape:
Still keeping one principal object in view—
To preserve its symmetrical shape."
— Lewis Carroll, *The Hunting of the Snark*

3.1 Introduction

In this chapter, we discuss the two branches of the symmetric key crypto family tree: stream ciphers and block ciphers. Stream ciphers generalize the idea of a one-time pad, except that we trade provable security for a relatively small (and manageable) key. The key is stretched into a long stream of bits, which is then used just like a one-time pad. Like their one-time pad brethren, stream ciphers employ (in Shannon's terminology) confusion only.

Block ciphers can be viewed as the modern successors to the classic codebook ciphers, where the key determines the codebook. The internal workings of block cipher algorithms can be fairly intimidating, so it is useful to keep in mind that a block cipher is really just an "electronic" version of a codebook. Internally, block ciphers employ both confusion and diffusion.

We'll take a fairly close look at two stream cipher algorithms, A5/1 and RC4, both of which have been widely deployed. The A5/1 algorithm (used in GSM cell phones) is a good representative of a large class of stream ciphers that are based in hardware. RC4 is used in many places, including the SSL and WEP protocols. RC4 is virtually unique among stream ciphers since it is designed for efficient implementation in software.

In the block cipher realm, we'll look closely at DES, since it's relatively simple (by block cipher standards) and it's the granddaddy of them all, making it the block cipher to which all others are compared. We'll also take a brief look at a few other popular block ciphers. Then we'll examine some of the many ways that block ciphers are used for confidentiality and we'll consider the role of block ciphers in the equally important area of data integrity.

Our goal in this chapter is to introduce symmetric key ciphers and gain some familiarity with their inner workings and their uses. That is, we'll focus more on the "how" than the "why." To understand why block ciphers are designed the way they are, some aspects of advanced cryptanalysis are essential. We cover the ideas behind such cryptanalysis in Chapter 6.

3.2 Stream Ciphers

A stream cipher takes a key K of n bits in length and stretches it into a long *keystream*. This keystream is then XORed with the plaintext P to produce ciphertext C. Through the magic of the XOR, the same keystream is used to recover the plaintext P from the ciphertext C. Note that the use of the keystream is identical to the use of the pad (or key) in a one-time pad cipher. An excellent introduction to stream ciphers can be found in Rueppel's book [254], and for leads into some very challenging research problems in the field, see [153].

The function of a stream cipher can be viewed simply as

$$\texttt{StreamCipher}(K) = S,$$

where K is the key and S represents the resulting keystream. Remember, the keystream is not ciphertext, but is instead simply a string of bits that we use like a one-time pad.

Now, given a keystream $S = s_0, s_1, s_2 \ldots$, and plaintext $P = p_0, p_1, p_2 \ldots$ we generate the ciphertext $C = c_0, c_1, c_2 \ldots$ by XOR-ing the corresponding bits, that is,

$$c_0 = p_0 \oplus s_0,\ c_1 = p_1 \oplus s_1,\ c_2 = p_2 \oplus s_2, \ldots.$$

To decrypt ciphertext C, the keystream S is again used, that is,

$$p_0 = c_0 \oplus s_0,\ p_1 = c_1 \oplus s_1,\ p_2 = c_2 \oplus s_2, \ldots.$$

Provided that both the sender and receiver have the same stream cipher algorithm and that both know the key K, this system provides a practical generalization of the one-time pad. However, the resulting cipher is not provably secure (as discussed in the problems at the end of the chapter), so we have traded provable security for practicality.

3.2.1 A5/1

The first stream cipher that we'll examine is A5/1, which is used for confidentiality in GSM cell phones (GSM is discussed in Chapter 10). This algorithm has an algebraic description, but it also can be illustrated via a relatively simple wiring diagram. We give both descriptions here.

A5/1 employs three linear feedback *shift registers* [126], or LFSRs, which we'll label X, Y, and Z. Register X holds 19 bits, $(x_0, x_1, \ldots, x_{18})$. The register Y holds 22 bits, $(y_0, y_1, \ldots, y_{21})$, and Z holds 23 bits, $(z_0, z_1, \ldots, z_{22})$. Of course, all computer geeks love powers of two, so it's no accident that the three LFSRs hold a total of 64 bits.

Not coincidentally, the A5/1 key K is also 64 bits. The key is used as the *initial fill* of the three registers, that is, the key is used as the initial values in the three registers. After these three registers are filled with the key,[1] we are ready to generate the keystream. But before we can describe how the keystream is generated, we need to say a little more about the registers X, Y, and Z.

When register X *steps*, the following series of operations occur:

$$\begin{aligned} t &= x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \\ x_i &= x_{i-1} \text{ for } i = 18, 17, 16, \ldots, 1 \\ x_0 &= t \end{aligned}$$

Similarly, for registers Y and Z, each step consists of

$$\begin{aligned} t &= y_{20} \oplus y_{21} \\ y_i &= y_{i-1} \text{ for } i = 21, 20, 19 \ldots, 1 \\ y_0 &= t \end{aligned}$$

and

$$\begin{aligned} t &= z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22} \\ z_i &= z_{i-1} \text{ for } i = 22, 21, 20, \ldots, 1 \\ z_0 &= t \end{aligned}$$

respectively.

Given three bits x, y, and z, define $\text{maj}(x, y, z)$ to be the majority vote function, that is, if the majority of x, y, and z are 0, the function returns 0; otherwise it returns 1. Since there are an odd number of bits, there cannot be a tie, so this function is well defined.

[1] We've simplified things a little. In reality, the registers are filled with the key, and then there is an involved run up (i.e., initial stepping procedure) that is used before we generate any keystream bits. Here, we ignore the runup process.

In A5/1, for each keystream bit that we generate, the following takes place. First, we compute

$$m = \text{maj}(x_8, y_{10}, z_{10}).$$

Then the registers X, Y, and Z step (or not) as follows:

- If $x_8 = m$ then X steps.
- If $y_{10} = m$ then Y steps.
- If $z_{10} = m$ then Z steps.

Finally, a single keystream bit s is generated as

$$s = x_{18} \oplus y_{21} \oplus z_{22},$$

which can then be XORed with the plaintext (if encrypting) or XORed with the ciphertext (if decrypting). We then repeat the entire process to generate as many key stream bits as are required.

Note that when a register steps, its fill changes due to the bit shifting. Consequently, after generating one keystream bit, the fills of at least two of the registers X, Y, Z have changed, which implies that new bits are in positions x_8, y_{10}, and z_{10}. Therefore, we can repeat this process and generate a new keystream bit.

Although this may seem like a complicated way to generate a single keystream bit, A5/1 is easily implemented in hardware and can generate bits at a rate proportional to the clock speed. Also, the number of keystream bits that can be generated from a single 64-bit key is virtually unlimited—although eventually the keystream will repeat. The wiring diagram for the A5/1 algorithm is illustrated in Figure 3.1. See, for example, [33] for a more detailed discussion of A5/1.

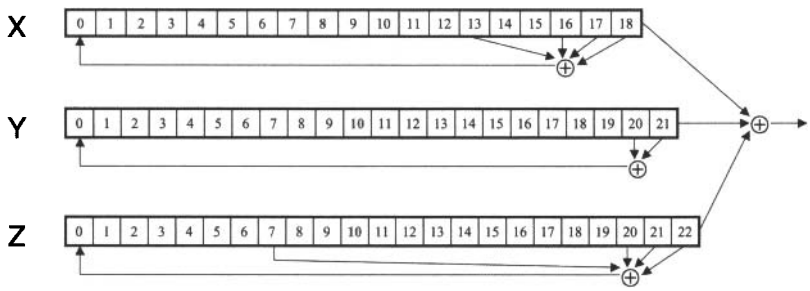

Figure 3.1: A5/1 Keystream Generator

The A5/1 algorithm is our representative example of a large class of stream ciphers that are based on shift registers and implemented in hardware. These systems were once the kings of symmetric key crypto, but in recent years the block cipher has clearly taken the crown. And where a stream cipher is needed today, it is likely to be RC4, which we'll discuss below.

Why has there been a mass migration away from stream ciphers towards block ciphers? In the bygone era of slow processor speeds, shift register based stream ciphers were necessary to keep pace with relatively high data-rate systems (such as audio). In the past, software-based crypto could not generate bits fast enough for such applications. Today, however, there are few applications for which software-based crypto is not appropriate. In addition, block ciphers are relatively easy to design and they can do everything stream ciphers can do, and more. These are the primary reasons why block ciphers are on the ascendancy.

3.2.2 RC4

RC4 is a stream cipher, but it's a completely different beast than A5/1. The RC4 algorithm is optimized for software implementation, whereas A5/1 is designed for hardware, and RC4 produces a keystream byte at each step, whereas A5/1 only produces a single keystream bit. All else being equal (which, of course, it never is), generating a byte at each step is much better than generating a single bit.

The RC4 algorithm is remarkably simple, because it is essentially just a lookup table containing a permutation of all possible 256 byte values. The crucial trick that makes it a strong cipher is that each time a byte of keystream is produced, the lookup table is modified in such a way that the table always contains a permutation of $\{0, 1, 2, \ldots, 255\}$. Because of this constant updating, the lookup table—and hence the cipher itself—presents the cryptanalyst with a moving target.

The entire RC4 algorithm is byte based. The first phase of the algorithm initializes the lookup table using the key. We'll denote the key as key$[i]$, for $i = 0, 1, \ldots, N-1$, where each key$[i]$ is a byte. We denote the lookup table as $S[i]$, where each $S[i]$ is also a byte. Pseudo-code for the initialization of the permutation S appears in Table 3.1. One interesting feature of RC4 is that the key can be of any length from 1 to 256 bytes. And again, the key is only used to initialize the permutation S. Note that the 256-byte array K is filled by simply repeating the key until the array is full.

After the initialization phase, each keystream byte is generated following the algorithm that appears in Table 3.2. The output, which we've denoted here as `keystreamByte`, is a single byte that can be XORed with plaintext (to encrypt) or XORed with ciphertext (to decrypt). We'll mention another possible application for RC4 keystream bytes in Chapter 5.

Table 3.1: RC4 Initialization

```
for i = 0 to 255
    S[i] = i
    K[i] = key[i mod N]
next i
j = 0
for i = 0 to 255
    j = (j + S[i] + K[i]) mod 256
    swap(S[i],S[j])
next i
i = j = 0
```

The RC4 algorithm—which can be viewed as a self-modifying lookup table—is elegant, simple, and efficient in software. However, there is an attack that is feasible against certain uses of RC4 [112, 195, 294], but the attack is infeasible if we discard the first 256 keystream bytes that are generated. This could be achieved by simply adding an extra 256 steps to the initialization phase, where each additional step generates—and discards—a keystream byte following the algorithm in Table 3.2. As long as Alice and Bob both implement these additional steps, they can use RC4 to communicate securely.

Table 3.2: RC4 Keystream Byte

```
i = (i + 1) mod 256
j = (j + S[i]) mod 256
swap(S[i], S[j])
t = (S[i] + S[j]) mod 256
keystreamByte = S[t]
```

RC4 is used in many applications, including SSL and WEP. However, the algorithm is fairly old and is not optimized for 32-bit processors (in fact, it's optimized for ancient 8-bit processors). Nevertheless, RC4 is sure to be a major player in the crypto arena for many years to come.

Stream ciphers were once king of the hill, but they are now relatively rare, in comparison to block ciphers. Some have even gone so far as to declare the death of stream ciphers [74] and, as evidence, they point to the fact that there has been almost no serious effort to develop new stream ciphers in recent years. However, today there are an increasing number of

significant applications where dedicated stream ciphers are more appropriate than block ciphers. Examples of such applications include wireless devices, severely resource-constrained devices, and extremely high data-rate systems. Undoubtedly, the reports of the death of stream ciphers have been greatly exaggerated.

3.3 Block Ciphers

An iterated block cipher splits the plaintext into fixed-sized blocks and generates fixed-sized blocks of ciphertext. In most designs, the ciphertext is obtained from the plaintext by iterating a function F over some number of *rounds*. The function F, which depends on the output of the previous round and the key K, is known as the *round function*, not because of its shape, but because it is applied over multiple rounds.

The design goals for block ciphers are security and efficiency. It's not too difficult to develop a reasonably secure block cipher or an efficient block cipher, but to design one that is secure and efficient requires a high form of the cryptographer's art.

3.3.1 Feistel Cipher

A *Feistel cipher*, named after block cipher pioneer Horst Feistel, is a general cipher design principle, not a specific cipher. In a Feistel cipher, the plaintext block P is split into left and right halves,

$$P = (L_0, R_0),$$

and for each round $i = 1, 2, \ldots, n$, new left and right halves are computed according to the rules

$$L_i = R_{i-1} \tag{3.1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \tag{3.2}$$

where K_i is the *subkey* for round i. The subkey is derived from the key K according to a specified *key schedule* algorithm. Finally, the ciphertext C is the output of the final round, namely,

$$C = (L_n, R_n).$$

Instead of trying to memorize equations (3.1) and (3.2), it's much easier to simply remember how each round of a Fiestel cipher works. Note that equation (3.1) tells us that the "new" left half is the "old" right half. On the other hand, equation (3.2) says that the new right half is the old left half XORed with a function of the old right half and the key.

Of course, it's necessary to be able to decrypt the ciphertext. The beauty of a Feistel cipher is that we can decrypt, regardless of the particular round function F. Thanks to the magic of the XOR, we can solve equations (3.1) and (3.2) for R_{i-1} and L_{i-1}, respectively, which allows us to run the process backwards. That is, for $i = n, n-1, \ldots, 1$, the decryption rule is

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus F(R_{i-1}, K_i). \end{aligned}$$

The final result of this decryption process is the plaintext $P = (L_0, R_0)$, as desired.

Again, any round function F will work in a Feistel cipher, provided that the output of F produces the correct number of bits. It is particularly nice that there is no requirement that the function F be invertible. However, a Feistel cipher will not be secure for all possible choices of F. For example, the round function

$$F(R_{i-1}, K_i) = 0 \text{ for all } R_{i-1} \text{ and } K_i \tag{3.3}$$

is a legitimate round function since we can encrypt and decrypt with this F. However, Trudy would be very happy if Alice and Bob decide to use a Feistel cipher with the round function in (3.3).

Note that all questions about the security of a Feistel cipher boil down to questions about the round function and the key schedule. The key schedule is usually not a major issue, so most of the analysis can be focused on F.

3.3.2 DES

Now there was an algorithm to study;
one that the NSA said was secure.
— Bruce Schneier, in reference to DES

The Data Encryption Standard, affectionately known as DES,[2] was developed way back in the computing dark ages of the 1970s. The design is based on the so-called Lucifer cipher, a Feistel cipher developed by a team at IBM. DES is a surprisingly simple block cipher, but the story of how Lucifer became DES is anything but simple.

By the mid 1970s, it was clear even to U.S. government bureaucrats that there was a legitimate commercial need for secure crypto. At the time, the

[2]People "in the know" pronounce DES so as to rhyme with "fez" or "pez," not as the three letters D-E-S. Of course, you can say Data Encryption Standard, but that would be very uncool.

computer revolution was underway, and the amount—and sensitivity—of digital data was rapidly increasing.

In the mid 1970s, crypto was poorly understood outside of classified military and government circles, and they weren't talking (and, for the most part, that's still the case). The upshot was that businesses had no way to judge the merits of a crypto product and the quality of most such products was very poor.

Into this environment, the National Bureau of Standards, or NBS (now known as NIST) issued a request for cipher proposals. The winning submission would become a U.S. government standard and almost certainly a *de facto* industrial standard. Very few reasonable submissions were received, and it quickly became apparent that IBM's Lucifer cipher was the only serious contender.

At this point, NBS had a problem. There was little crypto expertise at NBS, so they turned to the government's crypto experts, the super-secret National Security Agency, or NSA.[3] The NSA designs and builds the crypto that is used by the U.S. military and government for highly sensitive information. However, the NSA also wears a black hat, since it conducts signals intelligence, or SIGINT, where it tries to obtain intelligence information from foreign sources.

The NSA was reluctant to get involved with DES but, under pressure, eventually agreed to study the Lucifer design and offer an opinion, provided its role would not become public. When this information came to public light [273] (as is inevitable in the United States[4]) many were suspicious that NSA had placed a backdoor into DES so that it alone could break the cipher. Certainly, the black hat SIGINT mission of NSA and a general climate of distrust of government fueled such fears. In the defense of NSA, it's worth noting that 30 years of intense cryptanalysis has revealed no backdoor in DES. Nevertheless, this suspicion tainted DES from its inception.

Lucifer eventually became DES, but not before a few subtle—and a few not so subtle—changes were made. The most obvious change was that the key length was apparently reduced from 128 bits to 64 bits. However, upon careful analysis, it was found that 8 of the 64 key bits were effectively discarded, so the actual key length is a mere 56 bits. As a result of this modification, the expected work for an exhaustive key search was reduced from 2^{127} to 2^{55}. By this measure, DES is 2^{72} times easier to break than Lucifer.

[3]NSA is so super-secret that its employees joke that the acronym NSA stands for No Such Agency.

[4]Your secretive author once attended a public talk by the Director of NSA, aka DIRNSA. At this talk the DIRNSA made a comment to the effect, "Do you want to know what problems we're working on now?" Of course, the audience gave an enthusiastic "Yes!" hoping that they might be about to hear the deepest darkest secrets of the super-secret spy agency. The DIRNSA responded, "Read the front page of the New York Times."

Understandably, the suspicion was that NSA had had a hand in purposely weakening DES. However, subsequent cryptanalysis of the algorithm has revealed attacks that require slightly less work than trying 2^{55} keys and, as a result, DES is probably just about as strong with a key of 56 bits as it would be with the longer Lucifer key.

The subtle changes to Lucifer involved the substitution boxes, or *S-boxes*, which are described below. These changes in particular fueled the suspicion of a backdoor. But it has become clear over time that the modifications to the S-boxes actually strengthened the algorithm by offering protection against cryptanalytic techniques that were unknown (at least outside of NSA, and they're not talking) until many years later. The inescapable conclusion is that whoever modified the Lucifer algorithm (NSA, that is) knew what they were doing and, in fact, significantly strengthened the algorithm. See [215, 273] for more information on the role of NSA in the development of DES.

Now it's time for the nitty gritty details of the DES algorithm. DES is a Feistel cipher with the following numerology:

- 16 rounds
- 64-bit block length
- 56-bit key
- 48-bit subkeys

Each round of DES is relatively simple—at least by the standards of block cipher design. The DES S-boxes are one of its most important security features. We'll see that S-boxes (or similar) are a common feature of modern block cipher designs. In DES, each S-box maps 6 bits to 4 bits, and DES employs eight distinct S-boxes. The S-boxes, taken together, map 48 bits to 32 bits. The same S-boxes are used at each round of DES and each S-box is implemented as a lookup table.

Since DES is a Feistel cipher, encryption follows the formulas given in equations (3.1) and (3.2). A single round of DES is illustrated in the wiring diagram in Figure 3.2, where each number indicates the number of bits that follow a particular "wire."

Unravelling the diagram in Figure 3.2, we see that the DES round function F can be written as

$$F(R_{i-1}, K_i) = \text{P-box}(\text{S-boxes}(\text{Expand}(R_{i-1}) \oplus K_i)). \tag{3.4}$$

With this round function, DES can be seen to be a Feistel cipher as defined in equations (3.1) and (3.2). As required by equation (3.1), the new left half is simply the old right half. The round function F is the composition of the expansion permutation, addition of subkey, S-boxes, and P-box, as given in equation (3.4).

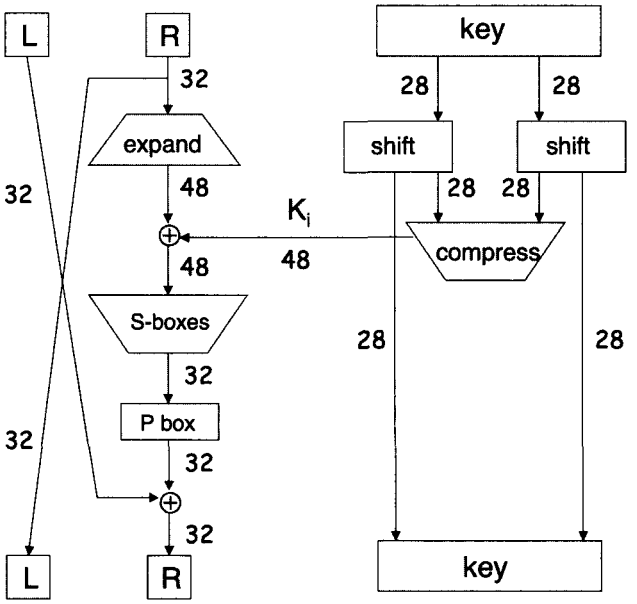

Figure 3.2: One Round of DES

The expansion permutation expands its input from 32 to 48 bits, and the subkey is XORed with the result. The S-boxes then compress these 48 bits down to 32 bits before the result is passed through the P-box. The P-box output is XORed with the old left half to obtain the new right half.

Next, we'll describe each of the components of F in precise detail, as well as the algorithm used to calculate the subkey K_i. But it's important to keep the big picture in mind and to realize that the overall structure of DES is actually fairly simple. In fact, some of the DES operations are of no security benefit whatsoever, and if these were stripped away to reveal the essential security features, the algorithm becomes even simpler.

Throughout this discussion—and elsewhere in this book—we'll adopt the convention that bits are numbered from left to right, beginning with the index zero.[5] The 48-bit output of the DES expansion permutation consists of the following bits.

31	0	1	2	3	4	3	4	5	6	7	8
7	8	9	10	11	12	11	12	13	14	15	16
15	16	17	18	19	20	19	20	21	22	23	24
23	24	25	26	27	28	27	28	29	30	31	0

[5]Your author is not a dinosaur (i.e., FORTRAN programmer), so the indexing starts at 0, not 1.

where the 32-bit input is, according to our convention, numbered as

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Each of the eight DES S-boxes maps 6 bits to 4 bits, and, consequently, each can be viewed as an array of 4 rows and 16 columns, with one nibble (4-bit value) stored in each of the 64 positions. When viewed in this way, each S-box has been constructed so that each of its four rows is a permutation of the hexadecimal digits $0, 1, 2, \ldots, E, F$. The DES S-box number 1 appears in Table 3.3, where the six-bit input to the S-box is denoted $b_0b_1b_2b_3b_4b_5$. Note that the first and last input bits are used to index the row, while the middle four bits index the column. Also note that we've given the output in hex. For those who just can't get enough of S-boxes, all eight DES S-boxes can be found on the textbook website.

Table 3.3: DES S-box 1

	$b_1b_2b_3b_4$															
b_0b_5	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7
1	0	F	7	4	E	2	D	1	A	6	C	B	9	5	3	8
2	4	1	E	8	D	6	2	B	F	C	9	7	3	A	5	0
3	F	C	8	2	4	9	1	7	5	B	3	E	A	0	6	D

The DES permutation box, or P-box, contributes little to the security of the cipher and its real purpose seems to have been lost to the mists of history. One plausible explanation is that the designers wanted to make DES more difficult to implement in software since the original design called for hardware-based implementation. It was apparently hoped that DES would remain a hardware-only algorithm, perhaps in the belief that this would allow the algorithm to remain secret. In fact, the S-boxes themselves were originally classified, so undoubtedly the goal was to keep them secret. But, predictably, the DES S-boxes were reverse engineered and they became public knowledge almost immediately. For the record, the P-box permutation is

15	6	19	20	28	11	27	16	0	14	22	25	4	17	30	9
1	7	23	13	31	26	2	8	18	12	29	5	21	10	3	24

The only significant remaining part of DES is the key schedule algorithm, which is used to generate the subkeys. This is a somewhat convoluted process, but the ultimate result is simply that 48 of the 56 bits of key are selected at

each round. The details are relevant, since block cipher designs have been attacked due flawed key schedule algorithms.

As usual, we'll number the 56-bit DES key from left-to-right, beginning with 0. We first extract 28 of the DES key bits, permute them, and call the result LK. Initially, LK consists of the following DES key bits:

$$\begin{array}{rrrrrrr} 49 & 42 & 35 & 28 & 21 & 14 & 7 \\ 0 & 50 & 43 & 36 & 29 & 22 & 15 \\ 8 & 1 & 51 & 44 & 37 & 30 & 23 \\ 16 & 9 & 2 & 52 & 45 & 38 & 31 \end{array}$$

The remaining 28 bits of the DES key are permuted and assigned to the variable RK. Initially, RK consists of the following DES key bits:

$$\begin{array}{rrrrrrr} 55 & 48 & 41 & 34 & 27 & 20 & 13 \\ 6 & 54 & 47 & 40 & 33 & 26 & 19 \\ 12 & 5 & 53 & 46 & 39 & 32 & 25 \\ 18 & 11 & 4 & 24 & 17 & 10 & 3 \end{array}$$

Before we can precisely state the key schedule algorithm, we need a few more items. Define the permutation LP as

$$\begin{array}{rrrrrrrrrrrr} 13 & 16 & 10 & 23 & 0 & 4 & 2 & 27 & 14 & 5 & 20 & 9 \\ 22 & 18 & 11 & 3 & 25 & 7 & 15 & 6 & 26 & 19 & 12 & 1 \end{array}$$

and RP as

$$\begin{array}{rrrrrrrrrrrr} 12 & 23 & 2 & 8 & 18 & 26 & 1 & 11 & 22 & 16 & 4 & 19 \\ 15 & 20 & 10 & 27 & 5 & 24 & 17 & 13 & 21 & 7 & 0 & 3 \end{array}$$

Finally, define

$$r_i = \begin{cases} 1 & \text{if } i \in \{1, 2, 9, 16\} \\ 2 & \text{otherwise.} \end{cases}$$

The DES key schedule algorithm, which is used to generate the 48-bit subkeys, appears in Table 3.4.

Note that when writing code to implement DES, we would probably not want to implement the key schedule algorithm as it appears in Table 3.4. It would be more efficient to use the key schedule algorithm to determine each K_i (in terms of the original DES key) and simply hardcode these values into our program.

For completeness, there are two other features of DES that we should mention. An initial permutation is applied to the plaintext before round one, and its inverse is applied after the final round. Also, when encrypting, the halves are swapped after last round, so the actual ciphertext is (R_{16}, L_{16}), not (L_{16}, R_{16}). Neither of these quirks serve any security purpose and we'll

Table 3.4: DES Key Schedule Algorithm

for each round $i = 1, 2, \ldots, n$
LK = cyclically left shift LK by r_i bits
RK = cyclically left shift RK by r_i bits
The left half of subkey K_i consists of bits LP of LK
The right half of subkey K_i consists of bits RP of RK
next i

ignore them in the remaining discussion. However, these are part of the DES algorithm, so they must be implemented if you want to call the resulting cipher DES.

A few words on the security of DES may be enlightening. First, mathematicians are very good at solving linear equations, and the only part of DES that is not linear is the S-boxes. Due to those annoying mathematicians, linear ciphers are inherently weak, so the S-boxes are fundamental to the security of DES. Actually, the expansion permutation has an important security role to play and, to a lesser extent, so does the key schedule. All of this will become clearer after we discuss linear and differential cryptanalytic attacks in Chapter 6. For more details on the design of the DES cipher, see [258].

Despite the concern over the design of DES—particularly the role of the NSA in the process—DES has clearly stood the test of time [181]. Today, DES is vulnerable simply because the key is too small, not because of any noteworthy shortcut attack. Although some attacks have been developed that, in theory, require somewhat less work than an exhaustive key search, all practical DES crackers[6] built to date simply try all keys until they stumble across the correct one, that is, an exhaustive key search. The inescapable conclusion is that the designers of DES knew what they were doing.

We'll have more to say about DES when we study advanced cryptanalysis in Chapter 6. In fact, the historic importance of DES is hard to overstate. DES can be viewed as the impetus behind the development of modern symmetric crypto, which makes it all the more ironic that NSA was the unwilling godfather of DES.

Next, we describe triple DES, which is often used to effectively extend the key length of DES. We'll follow this with a quick overview of a few other block ciphers. Then we discuss one truly simple block cipher in a bit more detail.

[6]Not to be confused with Ritz crackers.

3.3.3 Triple DES

Before moving on to other block ciphers, we discuss a popular variant of DES known as triple DES, or 3DES. But before that, we need some notation. Let P be a block of plaintext, K a key, and C the corresponding block of ciphertext. For DES, C and P are each 64 bits, while K is 56 bits, but our notation applies in general. The notation that we'll adopt for the encryption of P with key K is

$$C = E(P, K)$$

while the corresponding decryption is denoted

$$P = D(C, K).$$

Note that for the same key, encryption and decryption are inverse operations, that is,

$$P = D(E(P, K), K) \quad \text{and} \quad C = E(D(C, K), K).$$

However, in general,

$$P \neq D(E(P, K_1), K_2) \quad \text{and} \quad C \neq E(D(C, K_1), K_2),$$

when $K_1 \neq K_2$.

At one time, DES was nearly ubiquitous, but its key length is insufficient today. But for DES-philes, all is not lost—there is a clever way to use DES with a larger key length. Intuitively, it seems that double DES might be the thing to do, that is,

$$C = E(E(P, K_1), K_2). \tag{3.5}$$

This would seem to offer the benefits of a 112 bit key (two 56-bit DES keys), with the only drawback being a loss of efficiency due to the two DES operations.

However, there is a meet-in-the-middle attack on double DES that renders it more or less equivalent to single DES. Although the attack may not be entirely practical, it's too close for comfort. This attack is a chosen plaintext attack, meaning that we assume the attacker can always choose a specific plaintext P and obtain the corresponding ciphertext C.

So, suppose Trudy selects a particular plaintext P and obtains the corresponding ciphertext C, which for double DES is $C = E(E(P, K_1), K_2)$. Trudy's goal is to find the keys K_1 and K_2. Toward this goal, Trudy first pre-computes a table of size 2^{56} containing the pairs $E(P, K)$ and K for all possible key values K. Trudy sorts this table on the values $E(P, K)$. Now using her table and the ciphertext value C, Trudy decrypts C with keys $\tilde{K}$ until she finds a value $X = D(C, \tilde{K})$ that is in table. Then, because of the way the table was constructed, we have $X = E(P, K)$ for some K and Trudy now has

$$D(C, \tilde{K}) = E(P, K),$$

where $\tilde{K}$ and K are known. That Trudy has found the 112-bit key can be seen by encrypting both sides with the key $\tilde{K}$, which gives

$$C = E(E(P, K), \tilde{K}),$$

that is, in equation (3.5), we have $K_1 = K$ and $K_2 = \tilde{K}$.

This attack on double DES requires that Trudy pre-compute, sort, and store an enormous table of 2^{56} elements. But the table computation is one-time work,[7] so if we use this table many times (by attacking double DES many times) the work for computing the table can be amortized over the number of attacks. Neglecting the work needed to pre-compute the table, the work consists of computing $D(C, K)$ until we find a match in the table. This has an expected work of 2^{55}, just as in an exhaustive key search attack on single DES. So, in a sense, double DES is no more secure than single DES.

Since double DES isn't secure, will triple DES fare any better? Before worrying about attacks, we need to define triple DES. It seems that the logical approach to triple DES would be

$$C = E(E(E(P, K_1), K_2), K_3)$$

but this is not the way it's done. Instead, triple DES is defined as

$$C = E(D(E(P, K_1), K_2), K_1).$$

Note that triple DES only uses two keys, and encrypt-decrypt-encrypt, or EDE, is used instead of encrypt-encrypt-encrypt, or EEE. The reason for only using two keys is that 112 bits is sufficient, and three keys does not add much security (see Problem 42). But why EDE instead of EEE? Surprisingly, the answer is backwards compatibility—if 3DES is used with $K_1 = K_2 = K$ then it collapses to single DES, since

$$C = E(D(E(P, K), K), K) = E(P, K).$$

Now, what about attacks on triple DES? We can say with certainty that a meet-in-the-middle attack of the type used against double DES is impractical since the table pre-computation is infeasible or the per attack work is infeasible—see Problem 42 for more details.

Triple DES remains fairly popular today. However, with the coming of the Advanced Encryption Standard and other modern alternatives, triple DES should, like any old soldier, slowly fade away.

[7]The pre-computation work is one time, provided that chosen plaintext is available. If we only have known plaintext, then we would need to compute the table each time we conduct the attack—see Problem 18.

3.3.4 AES

By the 1990s it was apparent to everyone—even the U.S. government—that DES had outlived its usefulness. The crucial problem with DES is that the key length of 56 bits is susceptible to an exhaustive key search. Special-purpose DES crackers have been built that can recover DES keys in a matter of hours, and distributed attacks using volunteer computers on the Internet have succeeded in finding DES keys [98].

In the early 1990s, NIST, which is the present incarnation of NBS, issued a call for crypto proposals for what would become the Advanced Encryption Standard, or AES. Unlike the DES call for proposals of 20 years earlier, NIST was inundated with quality proposals. The field of candidates was eventually reduced to a handful of finalists, and an algorithm known a Rijndael (pronounced something like "rain doll") was ultimately selected. See [182] for information on the AES competition and [75] for the details on the Rijndael algorithm.

The AES competition was conducted in a completely open manner and, unlike the DES competition, the NSA was openly involved as one of the judges. As a result, there are no plausible claims of a backdoor having been inserted into the AES. In fact, AES is highly regarded in the cryptographic community. Shamir has stated that he believes data encrypted with a 256-bit AES key will be "secure forever," regardless of any conceivable advances in computing technology [73].

Like DES, the AES is an iterated block cipher. Unlike DES, the AES algorithm is not a Feistel cipher. The major implication of this fact is that in order to decrypt, the AES operations must be invertible. Also unlike DES, the AES algorithm has a highly mathematical structure. We'll only give a quick overview of the algorithm—large volumes of information on all aspects of AES are readily available—and we'll largely ignore the elegant mathematical structure. In any case, it is a safe bet that no crypto algorithm in history has received as much scrutiny in as short of a period of time as the AES. See [7, 75] for more details on the Rijndael algorithm.

Some of the pertinent facts of AES are as follows.

- The block size is 128 bits.[8]
- Three key lengths are available: 128, 192, or 256 bits.
- The number of rounds varies from 10 to 14, depending on the key length.
- Each round consists of four functions, in three layers—the functions are listed below, with the layer in parentheses.

[8]The Rijndael algorithm actually supports block sizes of 128, 192, or 256 bits, independent of the key length. However, the larger block sizes are not part of the official AES.

- `ByteSub` (nonlinear layer)
- `ShiftRow` (linear mixing layer)
- `MixColumn` (nonlinear layer)
- `AddRoundKey` (key addition layer)

AES treats the 128-bit block as a 4×4 byte array:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

The `ByteSub` operation is applied to each byte a_{ij}, that is, $b_{ij} = \texttt{ByteSub}(a_{ij})$. The result is the array of b_{ij} as illustrated below:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \texttt{ByteSub} \longrightarrow \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

`ByteSub`, which is roughly the AES equivalent of the DES S-boxes, can be viewed as a nonlinear—but invertible—composition of two mathematical functions, or it can be viewed simply as a lookup table. We'll take the latter view. The `ByteSub` lookup table appears in Table 3.5. For example, `ByteSub(3c)` = `eb` since `eb` appears in row `3` and column `c` of Table 3.5.

Table 3.5: AES `ByteSub`

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

The ShiftRow operation is a cyclic shift of the bytes in each row of the 4×4 byte array. This operation is given by

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \texttt{ShiftRow} \longrightarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{11} & a_{12} & a_{13} & a_{10} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{33} & a_{30} & a_{31} & a_{32} \end{bmatrix},$$

that is, the first row doesn't shift, the second row circular left-shifts by one byte, the third row left-shifts by two bytes, and the last row left-shifts three bytes. Note that ShiftRow is inverted by simply shifting in the opposite direction.

Next, the MixColumn operation is applied to each column of the 4×4 byte array as indicated below:

$$\begin{bmatrix} a_{0i} \\ a_{1i} \\ a_{2i} \\ a_{3i} \end{bmatrix} \longrightarrow \texttt{MixColumn} \longrightarrow \begin{bmatrix} b_{0i} \\ b_{1i} \\ b_{2i} \\ b_{3i} \end{bmatrix} \quad \text{for } i = 0, 1, 2, 3.$$

MixColumn consists of shift and XOR operations, and it's most efficiently implemented as a lookup table. The overall operation is nonlinear but invertible, and, as with ByteSub, it serves a similar purpose as the DES S-boxes.

The AddRoundKey operation is straightforward. Similar to DES, a key schedule algorithm is used to generate a subkey for each round. Let k_{ij} be the 4×4 subkey array for a particular round. Then the subkey is XORed with the current 4×4 byte array a_{ij} as illustrated below:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \oplus \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix} = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}.$$

We'll ignore the AES key schedule but, as with any block cipher, it's a significant part of the security of the algorithm. Finally, as we noted above, the four functions, ByteSub, ShiftRow, MixColumn, and AddRoundKey, are all invertible. As a result, the entire algorithm is invertible, and consequently AES can decrypt as well as encrypt.

3.3.5 Three More Block Ciphers

In this section, we briefly consider three well-known block cipher algorithms, namely, the International Data Encryption Algorithm (IDEA), Blowfish, and RC6. Each of these has some particular noteworthy design feature. In the

next section we'll take a closer look at the Tiny Encryption Algorithm, or TEA [323].

IDEA is the handiwork of James L. Massey, one of the great—if somewhat lesser-known—cryptographers of modern times. The most innovative feature of IDEA is its use of *mixed mode arithmetic*. The algorithm combines addition modulo two (also known as XOR) with addition modulo 2^{16} and the Lai-Massey multiplication, which is "almost" multiplication modulo 2^{16}. These operations together produce the necessary nonlinearity, and as a result no explicit S-box is required. Massey was apparently the first to use this approach, which is common today. See [201] for more details on the design of IDEA.

Blowfish is one of Bruce Schneier's favorite crypto algorithms, no doubt because he invented it. Schneier is a well-known cryptographer and an entertaining writer on all things security-related. The interesting quirk of Blowfish is its use of *key dependent S-boxes*—instead of having fixed S-boxes, Blowfish generates its S-boxes based on the key. It can be shown that typical Blowfish S-boxes are strong. See [262] for more information on Blowfish.

RC6 is due to Ron Rivest, whose crypto accomplishments are truly remarkable, including the public key system RSA and the previously mentioned RC4 stream cipher, as well as one of the most popular hash functions, MD5. The unusual aspect of RC6 is its use of data-dependent rotations [247]. It is highly unusual to rely on the data as an essential part of the operation of a crypto algorithm. RC6 was one of the AES finalists, although it ultimately lost out to Rjindael.

These three ciphers illustrate a small sample of the many variations that have been used in the quest for the ideal balance between security and performance in block ciphers. In Chapter 6 we discuss linear and differential cryptanalysis, which makes the fundamental trade-offs inherent in block cipher design more explicit.

3.3.6 TEA

The final block cipher that we'll consider is the Tiny Encryption Algorithm (TEA). The wiring diagrams that we've displayed so far might lead you to conclude that block ciphers are necessarily complex. TEA nicely illustrates that such is not the case.

TEA uses a 64-bit block length and a 128-bit key. The algorithm assumes a computing architecture with 32-bit words—all operations are implicitly modulo 2^{32} and any bits beyond the 32nd position are automatically truncated. The number of rounds is variable but must be relatively large. The conventional wisdom is that 32 rounds is secure. However, each round of TEA is more like two rounds of a Feistel cipher (such as DES), so this is roughly equivalent to 64 rounds of DES. That's a lot of rounds.

In block cipher design, there is an inherent trade-off between the complexity of each round and the number of rounds required. Ciphers such as DES try to strike a balance between these two, while AES reduces the number of rounds as much as possible, at the expense of having a more complex round function. In a sense, TEA can be seen as living at the opposite extreme of AES, since TEA uses a very simple round function. But as a consequence of its simple rounds, the number of rounds must be large to achieve a high level of security. Pseudo-code for TEA encryption—assuming 32 rounds are used—appears in Table 3.6, where "$\ll$" is a left (non-cyclic) shift and "$\gg$" is a right (non-cyclic) shift.

Table 3.6: TEA Encryption

$(K[0], K[1], K[2], K[3]) =$ 128 bit key
$(L, R) =$ `plaintext` (64-bit block)
`delta` $= 0$x9e3779b9
`sum` $= 0$
`for` $i = 1$ `to` 32
    `sum` $=$ `sum` $+$ `delta`
    $L = L + (((R \ll 4) + K[0]) \oplus (R +$ `sum`$) \oplus ((R \gg 5) + K[1]))$
    $R = R + (((L \ll 4) + K[2]) \oplus (L +$ `sum`$) \oplus ((L \gg 5) + K[3]))$
`next` i
`ciphertext` $= (L, R)$

One interesting thing to notice about TEA is that it's not a Feistel cipher, and so we need separate encryption and decryption routines. However, TEA is about as close to a Feistel cipher as it is possible to be without actually being one—TEA uses addition and subtraction instead of XOR. But the need for separate encryption and decryption routines is a minor concern with TEA, since so few lines of code are required, and the algorithm is reasonably efficient even with the large number of rounds. The TEA decryption algorithm, assuming 32 rounds, appears in Table 3.7.

There is a somewhat obscure related key attack on TEA [163]. That is, if a cryptanalyst knows that two TEA messages are encrypted with keys that are related to each other in some very special way, then the plaintext can be recovered. This is a low-probability attack that in most circumstances can probably safely be ignored. But in case you are worried about this attack, there is a slightly more complex variant of TEA, known as extended TEA, or XTEA [218], that overcomes this potential problem. There is also a simplified version of TEA, known as STEA, that is extremely weak and is used to illustrate certain types of attacks [208].

Table 3.7: TEA Decryption

$(K[0], K[1], K[2], K[3]) =$ 128 bit key
$(L, R) =$ `ciphertext` (64-bit block)
`delta` $= 0x9e3779b9$
`sum` $=$ `delta` $\ll 5$
`for` $i = 1$ `to` 32
    $R = R - (((L \ll 4) + K[2]) \oplus (L + \texttt{sum}) \oplus ((L \gg 5) + K[3]))$
    $L = L - (((R \ll 4) + K[0]) \oplus (R + \texttt{sum}) \oplus ((R \gg 5) + K[1]))$
    `sum` $=$ `sum` $-$ `delta`
`next` i
`plaintext` $= (L, R)$

3.3.7 Block Cipher Modes

Using a stream cipher is easy—you generate a keystream that is the same length as the plaintext (or ciphertext) and XOR. Using a block cipher is also easy, provided that you have exactly one block to encrypt. But how should multiple blocks be encrypted with a block cipher? It turns out that the answer is not as straightforward as it might seem.

Suppose we have multiple plaintext blocks, say,

$$P_0, P_1, P_2, \ldots.$$

For a fixed key K, a block cipher is a codebook, since it creates a fixed mapping between plaintext and ciphertext blocks. Following the codebook idea, the obvious thing to do is to use a block cipher in so-called *electronic codebook mode*, or ECB. In ECB mode, we encrypt using the formula

$$C_i = E(P_i, K) \text{ for } i = 0, 1, 2, \ldots.$$

Then we can decrypt according to

$$P_i = D(C_i, K) \text{ for } i = 0, 1, 2, \ldots.$$

This approach works, but there are serious security issues with ECB mode and, as a result, it should never be used in practice.

Suppose ECB mode is used, and an attacker observes that $C_i = C_j$. Then the attacker knows that $P_i = P_j$. Although this may seem innocent enough, there are cases where the attacker will know part of the plaintext, and any match with a known block reveals another block. But even if the attacker does not know P_i or P_j, some information has been revealed, namely, that these two plaintext blocks are the same, and we don't want to give the cryptanalyst anything for free—especially if there is an easy way to avoid it.

Massey [196] gives a dramatic illustration of the consequences of this seemingly minor weakness. We give a similar example in Figure 3.3, which shows an (uncompressed) image of Alice next to the same image encrypted in ECB mode. Every block of the right-hand image in Figure 3.3 has been encrypted,

Figure 3.3: Alice and ECB Mode

but the blocks that were the same in the plaintext are the same in the ECB-encrypted ciphertext. Note that it does not matter which block cipher is used—the curious result in Figure 3.3 only depends on the fact that ECB mode was used, not on the details of the algorithm. In this case, it's not difficult for Trudy to guess the plaintext from the ciphertext.

The ECB mode problem illustrated in Figure 3.3 is the basis for the "new ciphertext-only attack" discussed in [95]. The purveyors of this "new" version of a well-known attack have created a video in which they provide a demonstration of the results, along with a large dose of marketing hype [239].

Fortunately, there are better ways to use a block cipher, which avoid the weakness of ECB mode. We'll discuss the most common method, *cipher block chaining mode*, or CBC. In CBC mode, the ciphertext from a block is used to obscure the plaintext of the next block before it is encrypted. The encryption formula for CBC mode is

$$C_i = E(P_i \oplus C_{i-1}, K) \text{ for } i = 0, 1, 2, \ldots, \tag{3.6}$$

which is decrypted via

$$P_i = D(C_i, K) \oplus C_{i-1} \text{ for } i = 0, 1, 2, \ldots. \tag{3.7}$$

The first block requires special handling since there is no ciphertext block C_{-1}. An *initialization vector*, or IV, is used to take the place of the mythical C_{-1}. Since the ciphertext is not secret, and since the IV plays a role analogous to a ciphertext block, it need not be secret either. But the IV should be randomly selected.

Using the IV, the first block is CBC encrypted as

$$C_0 = E(P_0 \oplus \text{IV}, K),$$

with the formula in equation (3.6) used for the remaining blocks. The first block is decrypted as

$$P_0 = D(C_0, K) \oplus \text{IV},$$

with the formula in equation (3.7) used to decrypt all remaining blocks. Since the IV need not be secret, it's usually randomly generated at encryption time and sent (or stored) as the first "ciphertext" block. In any case, when decrypting, the IV must be handled appropriately.

The benefit of CBC mode is that identical plaintext will not yield identical ciphertext. This is dramatically illustrated by comparing Alice's image encrypted using ECB mode—which appears in Figure 3.3—with the image of Alice encrypted in CBC mode, which appears in Figure 3.4.

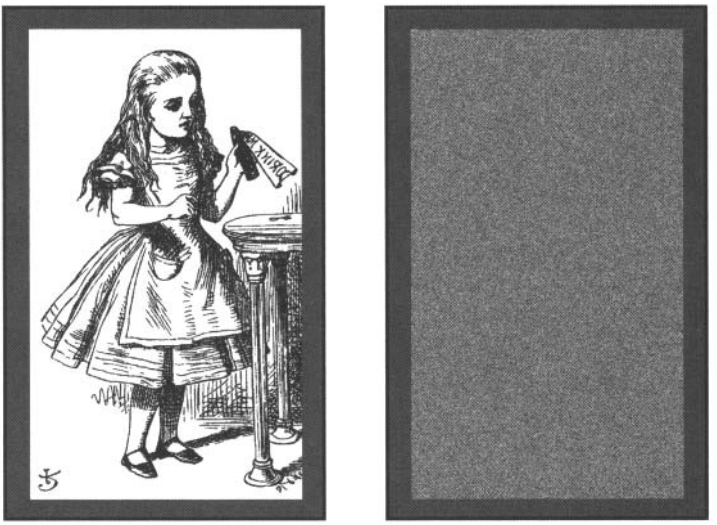

Figure 3.4: Alice Prefers CBC Mode

Due to the chaining, a possible concern with CBC mode is error propagation. When the ciphertext is transmitted, garbles can occur—a 0 bit could become a 1 bit or vice versa. If a single transmission error made the plaintext

unrecoverable, then CBC would be useless in practice. Fortunately, this is not the case.

Suppose the ciphertext block C_i is garbled to, say, $G \neq C_i$. Then

$$P_i \neq D(G, K) \oplus C_{i-1} \text{ and } P_{i+1} \neq D(C_{i+1}, K) \oplus G$$

but

$$P_{i+2} = D(C_{i+2}, K) \oplus C_{i+1}$$

and all subsequent blocks are decrypted correctly. That is, each plaintext block only depends on two consecutive ciphertext blocks, so errors do not propagate beyond two blocks. However, the fact that a single-bit error can cause two entire blocks to be garbled is a serious concern in high error-rate environments such as wireless. Stream ciphers do not have this problem—a single garbled ciphertext bit results in a single garbled plaintext bit—and that is one reason why stream ciphers are often preferred in wireless applications.

Another concern with a block cipher is a *cut-and-paste attack*. Suppose the plaintext

$$\texttt{Money␣for␣Alice␣is␣\$1000}$$
$$\texttt{Money␣for␣Trudy␣is␣\$2␣␣␣}$$

where "␣" is a blank space, is to be encrypted with a block cipher that has a 64-bit block size. Assuming that each character requires 8 bits (e.g., 8-bit ASCII), the plaintext blocks are

$$P_0 = \texttt{Money␣fo}$$
$$P_1 = \texttt{r␣Alice␣}$$
$$P_2 = \texttt{is␣\$1000}$$
$$P_3 = \texttt{Money␣fo}$$
$$P_4 = \texttt{r␣Trudy␣}$$
$$P_5 = \texttt{is␣\$2␣␣␣}$$

Suppose this data is encrypted using ECB mode.[9] Then the ciphertext blocks are computed as $C_i = E(P_i, K)$ for $i = 0, 1, \ldots, 5$.

Now suppose that Trudy knows that ECB mode is used, she knows the general structure of the plaintext, and she knows that she will receive \$2. But Trudy doesn't know how much Alice will receive—though she suspects it's much more than \$2. If Trudy can rearrange the order of the ciphertext blocks to

$$C_0, C_1, C_5, C_3, C_4, C_2, \tag{3.8}$$

then Bob will decrypt this as

[9]Of course, you should never use ECB mode. However, this same problem arises with other modes (and types of ciphers), but it's easiest to illustrate using ECB mode.

```
Money␣for␣Alice␣is␣$2␣␣␣
Money␣for␣Trudy␣is␣$1000
```

which is clearly a preferable outcome from Trudy's perspective.

You might think that CBC mode would eliminate the cut-and-paste attack. If so, you'd be wrong. With CBC mode, a cut-and-paste attack is still possible, although it's slightly more difficult and some data will be corrupted. This is explored further in the problems at the end of the chapter.

It is also possible to use a block cipher to generate a keystream, which can then be used just like a stream cipher keystream. There are several acceptable ways to accomplish this feat, but we'll only mention the most popular, namely, *counter mode*, or CTR. As with CBC mode, CTR mode employs an initialization vector, or IV. The CTR encryption formula is

$$C_i = P_i \oplus E(\text{IV} + i, K)$$

and decryption is accomplished via[10]

$$P_i = C_i \oplus E(\text{IV} + i, K).$$

CTR mode is often used when random access is required. While random access is also fairly straightforward with CBC mode, in some cases CBC mode would not be desirable for random access—see Problem 27.

Beyond ECB, CBC, and CTR, there are many other block cipher modes; see [258] for descriptions of the more common ones. However, the three modes discussed here certainly account for the vast majority of block cipher usage.

Finally, it is worth noting that data confidentiality comes in two slightly different flavors. On the one hand, we encrypt data so that it can be transmitted over an insecure channel. On the other hand, we encrypt data that is stored on an insecure media, such as a computer hard drive. Symmetric ciphers can be used to solve either of these two closely related problems. In addition, symmetric key crypto can also be used to protect data integrity, as we see in the next section.

3.4 Integrity

Whereas confidentiality deals with preventing unauthorized reading, integrity is concerned with detecting unauthorized writing. For example, suppose that you electronically transfer funds from one one account to another. You may not want others to know about this transaction, in which case encryption will effectively provide the desired confidential. But, whether you are concerned about confidentiality or not, you certainly want the transaction to be accurately received. This is where integrity comes into the picture.

[10]The use of the encryption "E" for both the encryption and decryption formulas is *not* a typo.

In the previous section, we studied block ciphers and their use for confidentiality. Here we show that block ciphers can also provide data integrity.

It is important to realize that confidentiality and integrity are two very different concepts. Encryption with any cipher—from the one-time pad to modern block ciphers—does not protect the data from malicious or inadvertent changes. If Trudy changes the ciphertext or if garbles occur in transmission, the integrity of the data has been lost and we want to be able to automatically detect that a change has occurred. We've seen several examples—and you should be able to give several more—to show that encryption does not assure integrity.

A *message authentication code*, or MAC, uses a block cipher to ensure data integrity. The procedure is simply to encrypt the data in CBC mode, discarding all ciphertext blocks except the final one. This final ciphertext block, which is known as the CBC residue, serves as the MAC. Then the formula for the MAC, assuming N blocks of data, $P_0, P_1, P_2, \ldots, P_{N-1}$, is given by

$$C_0 = E(P_0 \oplus \text{IV}, K),\ C_1 = E(P_1 \oplus C_0, K), \ldots,$$
$$C_{N-1} = E(P_{N-1} \oplus C_{N-2}, K) = \text{MAC}.$$

Note that we use an initialization vector, and that a shared symmetric key is required.

For simplicity, suppose that Alice and Bob require integrity, but they are not concerned with confidentiality. Then using a key K that Alice and Bob share, Alice computes the MAC and send the plaintext, the IV, and the MAC to Bob. Upon receiving the message, Bob computes the MAC using the key and received IV and plaintext. If his computed "MAC" matches the received MAC, then he is satisfied with the integrity of the data. On the other hand, if Bob's computed MAC does not match the received MAC, then Bob knows that something is amiss. Again, as in CBC mode, the sender and receiver must share a symmetric key K in advance.

Why does this MAC computation work? Suppose Alice sends

$$\text{IV}, P_0, P_1, P_2, P_3, \text{MAC}$$

to Bob. Now, if Trudy changes plaintext block P_1 to, say, Q during transmission, then when Bob attempts to verify the MAC, he computes

$$C_0 = E(P_0 \oplus \text{IV}, K),\ \tilde{C}_1 = E(Q \oplus C_0, K),\ \tilde{C}_2 = E(P_2 \oplus \tilde{C}_1, K),$$
$$\tilde{C}_3 = E(P_3 \oplus \tilde{C}_2, K) = \text{"MAC"} \neq \text{MAC}.$$

The reason this works is because any change to a plaintext block propagates into subsequent blocks in the process of computing the MAC.

Recall that with CBC *decryption* a change in a ciphertext block only affects two of the recovered plaintext blocks. In contrast, the MAC takes advantage of the fact that for CBC *encryption*, any change in the plaintext almost certainly propagates through to the final block. This is the crucial property that enables a MAC to provide integrity.

Often confidentiality and integrity are both required. To accomplish this, we could compute a MAC with one key, then encrypt the data with another key. However, this is twice as much work as is needed for either confidentiality or integrity alone. For the sake of efficiency, it would be useful to obtain both confidentiality and integrity protection with a single CBC encryption of the data. So, suppose we CBC encrypt the data once and send the resulting ciphertext and the computed "MAC." Then we would send the entire ciphertext, along with the final ciphertext block (again). That is, the final ciphertext block would be duplicated and sent twice. Obviously, sending the same thing twice cannot provide any additional security. Unfortunately, there is no obvious way to obtain both confidentiality and integrity with a single encryption of the data. These topics are explored further in the problems at the end of the chapter.

Computing a MAC based on CBC encryption is not the only way to provide for data integrity. A hashed MAC, or HMAC, is another standard approach to integrity and a digital signature is yet another option. We'll discuss the HMAC in Chapter 5 and digital signatures in Chapters 4 and 5.

3.5 Summary

In this chapter we've covered a great deal of material on symmetric key cryptography. There are two distinct types of symmetric ciphers: stream ciphers and block ciphers. We briefly discussed two stream ciphers, A5/1 and RC4. Recall that stream ciphers generalize the one-time pad, where provable security is traded for practicality.

Block ciphers, on the other hand, can be viewed as the "electronic" equivalent of a classic codebook. We discussed the block cipher DES in considerable detail and briefly mentioned several other block ciphers. We then considered various modes of using block ciphers (specifically, ECB, CBC, and CTR modes). We also showed that block ciphers—using CBC mode—can provide data integrity.

In later chapters we'll see that symmetric ciphers are also useful in authentication protocols. As an aside, it's interesting to note that stream ciphers, block ciphers, and hash functions (covered in a later chapter) are all equivalent in the sense that anything you can do with one, you can accomplish with the other two, although in some cases it would be fairly unnatural to actually do so. For this reason, these three are equivalent cryptographic "primitives."

Symmetric key cryptography is a big topic and we've only scratched the surface here. But, armed with the background from this chapter, we'll be prepared to tackle any issues involving symmetric ciphers that arise in later chapters.

Finally, to really understand the reasoning behind block cipher design, it's necessary to delve more deeply into the field of cryptanalysis. Chapter 6, which deals with advanced cryptanalysis, is highly recommended for anyone who wants to gain a deeper understanding of block cipher design principles.

3.6 Problems

1. A stream cipher can be viewed as a generalization of a one-time pad. Recall that the one-time pad is provably secure. Why can't we prove that a stream cipher is secure using the same argument that was used for the one-time pad?

2. This problem deals with stream ciphers.

   a. If we generate a sufficiently long keystream, the keystream must eventually repeat. Why?

   b. Why is it a security concern if the keystream repeats?

3. Suppose that Alice uses a stream cipher to encrypt plaintext P, obtaining ciphertext C, and Alice then sends C to Bob. Suppose that Trudy happens to know the plaintext P, but Trudy does not know the key K that was used in the stream cipher.

   a. Show that Trudy can easily determine the keystream that was used to encrypt P.

   b. Show that Trudy can, in effect, replace P with plaintext of her choosing, say, P'. That is, show that Trudy can create a ciphertext message C' so that when Bob decrypts C' he will obtain P'.

4. This problem deals with the A5/1 cipher. For each part, justify your answer.

   a. On average, how often does the X register step?

   b. On average, how often does the Y register step?

   c. On average, how often does the Z register step?

   d. On average, how often do all three registers step?

   e. On average, how often do exactly two registers step?

   f. On average, how often does exactly one register step?

g. On average, how often does no register step?

5. Implement the A5/1 algorithm. Suppose that, after a particular step, the values in the registers are

$$X = (x_0, x_1, \ldots, x_{18}) = (1010101010101010101)$$
$$Y = (y_0, y_1, \ldots, y_{21}) = (1100110011001100110011)$$
$$Z = (z_0, z_1, \ldots, z_{22}) = (11100001111000011110000)$$

List the next 32 keystream bits and give the contents of X, Y, and Z after these 32 bits have been generated.

6. For bits x, y, and z, the function $\text{maj}(x, y, z)$ is defined to be the majority vote, that is, if two or more of the three bits are 0, then the function returns 0; otherwise, it returns 1. Write the truth table for this function and derive the boolean function that is equivalent to $\text{maj}(x, y, z)$.

7. The RC4 cipher consists of a lookup table S, which contains 256 byte values, and two indices, i and j.

   a. The lookup table S is initialized to contain the identity permutation $0, 1, 2, \ldots, 255$ and at each step of the algorithm, S contains a permutation. How is this achieved? That is, why does S always contain a permutation?

   b. Where is RC4 used in the real world?

8. This problem deals with the RC4 stream cipher.

   a. Find a reasonable upper bound on the size of the RC4 state space. That is, find an upper bound for the number of different states that are possible for the RC4 cipher. Hint: The RC4 cipher consists of a lookup table S, and two indices i and j. Count the number of possible distinct tables S and the number of distinct indices i and j, then compute the product of these numbers.

   b. Why is the size of the state space relevant when analyzing a stream cipher?

9. Implement the RC4 algorithm. Suppose the key consists of the following seven bytes: (`0x1A`, `0x2B`, `0x3C`, `0x4D`, `0x5E`, `0x6F`, `0x77`). For each of the following, give S in the form of a 16×16 array where each entry is in hex.

   a. List the permutation S and indices i and j after the initialization phase has completed.

b. List the permutation S and indices i and j after the first 100 bytes of keystream have been generated.

c. List the permutation S and indices i and j after the first 1000 bytes of keystream have been generated.

10. Suppose that Trudy has a ciphertext message that was encrypted with the RC4 cipher—see Tables 3.1 and 3.2. For RC4, the encryption formula is given by $c_i = p_i \oplus k_i$, where k_i is the ith byte of the keystream, p_i is the ith byte of the plaintext, and c_i is the ith byte of the ciphertext. Suppose that Trudy knows the first ciphertext byte, and the first plaintext byte, that is, Trudy knows c_0 and p_0.

a. Show that Trudy can determine the first byte of the keystream k_0.

b. Show that Trudy can replace c_0 with c_0', where c_0' decrypts to a byte of Trudy's choosing, say, p_0'.

c. Suppose that a CRC [326] is used to detect errors in transmission. Can Trudy's attack in part b still succeed? Explain.

d. Suppose that a cryptographic integrity check is used (either a MAC, HMAC, or digital signature). Can Trudy's attack in part b still succeed? Explain.

11. This problem deals with a Feistel Cipher.

a. Give the definition of a Feistel Cipher.

b. Is DES a Feistel Cipher?

c. Is AES a Feistel Cipher?

d. Why is the Tiny Encryption Algorithm, TEA, "almost" a Feistel Cipher?

12. Consider a Feistel cipher with four rounds. Then the plaintext is denoted as $P = (L_0, R_0)$ and the corresponding ciphertext is $C = (L_4, R_4)$. What is the ciphertext C, in terms of L_0, R_0, and the subkey, for each of the following round functions?

a. $F(R_{i-1}, K_i) = 0$

b. $F(R_{i-1}, K_i) = R_{i-1}$

c. $F(R_{i-1}, K_i) = K_i$

d. $F(R_{i-1}, K_i) = R_{i-1} \oplus K_i$

13. Within a single round, DES employs both confusion and diffusion.

a. Give one source of confusion within a DES round.

b. Give one source of diffusion within a DES round.

14. This problem deals with the DES cipher.
    a. How many bits in each plaintext block?
    b. How many bits in each ciphertext block?
    c. How many bits in the key?
    d. How many bits in each subkey?
    e. How many rounds?
    f. How many S-boxes?
    g. An S-box requires how many bits of input?
    h. An S-box generates how many bits of output?

15. DES swaps the output of the final round, that is, the ciphertext is not $C = (L_{16}, R_{16})$ but instead it is $C = (R_{16}, L_{16})$. What is the purpose of this swap?

16. Recall the attack on double DES discussed in the text. Suppose that we instead define double DES as $C = D(E(P, K_1), K_2)$. Describe a meet-in-the-middle attack on this cipher.

17. Recall that for a block cipher, a key schedule algorithm determines the subkey for each round, based on the key K. Let $K = (k_0 k_1 k_2 \ldots k_{55})$ be a 56-bit DES key.
    a. List the 48 bits for each of the 16 DES subkeys $K_1, K_2, \ldots, K_{16}$, in terms of the key bits k_i.
    b. Make a table that contains the number of subkeys in which each key bit k_i is used.
    c. Can you design a DES key schedule algorithm in which each key bit is used an equal number of times?

18. Recall the meet-in-the-middle attack on double DES discussed in this chapter. Assuming that chosen plaintext is available, this attack recovers a 112-bit key with about the same work needed for an exhaustive search to recover a 56-bit key, that is, about 2^{55}.
    a. If we only have known plaintext available, not chosen plaintext, what changes do we need to make to the double DES attack?
    b. What is the work factor for the known plaintext version of the meet-in-the-middle double DES attack?

19. AES consists of four functions in three layers.
    a. Which of the four functions are primarily for confusion and which are primarily for diffusion? Justify your answer.

b. Which of the three layers are for confusion and which are for diffusion? Justify your answer.

20. Implement the Tiny Encryption Algorithm (TEA).

a. Use your TEA algorithm to encrypt the 64-bit plaintext block

0x0123456789ABCDEF

using the 128-bit key

0xA56BABCD00000000FFFFFFFFABCDEF01.

Decrypt the resulting ciphertext and verify that you obtain the original plaintext.

b. Using the key in part a, encrypt and decrypt the following message using each of the three block cipher modes discussed in the text (ECB mode, CBC mode, and CTR mode).

> Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

21. Give a diagram analogous to that in Figure 3.2 for the TEA cipher.

22. Recall that an initialization vector (IV) need not be secret.

a. Does an IV need to be random?

b. Discuss possible security disadvantages (or advantages) if IVs are selected in sequence instead of being generated at random.

23. Draw diagrams to illustrate encryption and decryption in CBC mode. Note that these diagrams are independent of the particular block cipher that is used.

24. The formula for counter mode encryption is

$$C_i = P_i \oplus E(\mathrm{IV} + i, K).$$

Suppose instead we use the formula

$$C_i = P_i \oplus E(K, \mathrm{IV} + i).$$

Is this secure? If so, why? If not, why not?

25. Suppose that we use a block cipher to encrypt according to the rule

$$C_0 = \mathrm{IV} \oplus E(P_0, K),\ C_1 = C_0 \oplus E(P_1, K),\ C_2 = C_1 \oplus E(P_2, K),\ \ldots$$

a. What is the corresponding decryption rule?

b. Give two security disadvantages of this mode as compared to CBC mode.

26. Suppose that ten ciphertext blocks are encrypted in CBC mode. Show that a cut-and-paste attack is possible. That is, show that it is possible to rearrange the blocks so that some of the blocks decrypt correctly, in spite of the fact that the blocks are not in the correct order.

27. Explain how to do random access on data encrypted in CBC mode. Are there any significant disadvantages of using CBC mode for random access as compared to CTR mode?

28. CTR mode generates a keystream using a block cipher. Devise another method for using a block cipher as a stream cipher. Does your method support random access?

29. Suppose that the ciphertext in equation (3.8) had been encrypted in CBC mode instead of ECB mode. If Trudy believes ECB mode is used and tries the same cut-and-paste attack discussed in the text, which blocks decrypt correctly?

30. Obtain the files `Alice.bmp` and `Alice.jpg` from the textbook website.

a. Use the TEA cipher to encrypt `Alice.bmp` in ECB mode, leaving the first 10 blocks unencrypted. View the encrypted image. What do you see? Explain the result.

b. Use the TEA cipher to encrypt `Alice.jpg` in ECB mode, leaving the first 10 blocks unencrypted. View the encrypted image. What do you see? Explain the result.

31. Suppose that Alice and Bob decide to always use the same IV instead of choosing IVs at random.

a. Discuss a security problem this creates if CBC mode is used.

b. Discuss a security problem this creates if CTR mode is used.

c. If the same IV is always used, which is more secure, CBC or CTR mode?

32. Suppose that Alice and Bob use CBC mode encryption.

a. What security problems arise if they always use a fixed initialization vector (IV), as opposed to choosing IVs at random? Explain.

b. Suppose that Alice and Bob choose IVs in sequence, that is, they first use 0 as an IV, then they use 1 as their IV, then 2, and so on. Does this create any security problems as compared to choosing the IVs at random?

33. Give two ways to encrypt a partial block using a block cipher. Your first method should result in ciphertext that is the size of a complete block, while your second method should not expand the data. Discuss any possible security concerns for your two methods.

34. Recall that a MAC is given by the CBC residue, that is, the last ciphertext block when the data is encrypted in CBC mode. Given data X, key K, and an IV, define $F(X)$ to be the MAC of X.

    a. Is F one-way, that is, given $F(X)$ is it possible to determine X?

    b. Is F collision resistant, that is, given $F(X)$ is it possible to find a value Y such that $F(Y) = F(X)$?

35. Suppose Alice uses DES to compute a MAC. She then sends the plaintext, the IV, and the corresponding MAC to Bob. If Trudy alters one block of plaintext before Bob receives it, what is the probability that Bob will not detect the change?

36. Alice has four blocks of plaintext, P_0, P_1, P_2, P_3, which she encrypts using CBC mode to obtain C_0, C_1, C_2, C_3. She then sends the IV and ciphertext to Bob. Upon receiving the ciphertext, Bob plans to verify the integrity as follows. He'll first decrypt to obtain the putative plaintext, and then he'll re-encrypt this plaintext using CBC mode and the received IV. If he obtains the same C_3 as the final ciphertext block, he will trust the integrity of the plaintext.

    a. Suppose that Trudy changes C_1 to X, leaving all other blocks and the IV unchanged. Will Bob detect that the data lacks integrity?

    b. Suppose that Trudy changes C_3 to the value Y, leaving all other blocks and the IV unchanged. Will Bob detect that the data lacks integrity?

    c. Is Bob's integrity checking method secure?

37. Using CBC mode, Alice encrypts four blocks of plaintext, P_0, P_1, P_2, P_3 and she sends the resulting ciphertext blocks, C_0, C_1, C_2, C_3, and the IV to Bob. Suppose that Trudy is able to change any of the ciphertext blocks before they are received by Bob. If Trudy knows P_1, show that she can replace P_1 with X. Hint: Determine $\tilde{C}$ so that if Trudy replaces C_0 with $\tilde{C}$, when Bob decrypts C_1, he will obtain X instead of P_1.

38. Suppose we encrypt in CBC mode using the key K and we compute a MAC using the key $K \oplus X$, where X is a known constant. Assuming the ciphertext and the MAC are sent from Alice to Bob, show that Bob will detect a cut-and-paste attack.

39. Suppose Alice has four blocks of plaintext, P_0, P_1, P_2, P_3. She computes a MAC using key K_1, and then CBC encrypts the data using key K_2 to obtain C_0, C_1, C_2, C_3. Alice sends the IV, the ciphertext, and the MAC to Bob. Trudy intercepts the message and replaces C_1 with X so that Bob receives IV, C_0, X, C_2, C_3, and the MAC. Bob attempts to verify the integrity of the data by decrypting (using key K_2) and then computing a MAC (using key K_1) on the putative plaintext.

    a. Show that Bob will detect Trudy's tampering.

    b. Suppose that Alice and Bob only share a single symmetric key K. They agree to let $K_1 = K$ and $K_2 = K \oplus Y$, where Y is known to Alice, Bob, and Trudy. Assuming Alice and Bob use the same scheme as above, does this create any security problem?

40. Suppose that Alice and Bob have access to two secure block ciphers, say, Cipher A and Cipher B, where Cipher A uses a 64-bit key, while Cipher B uses a 128-bit key. Alice prefers Cipher A, while Bob wants the additional security provided by a 128-bit key, so he insists on Cipher B. As a compromise, Alice proposes that they use Cipher A, but they encrypt each message twice, using two independent 64-bit keys. Assume that no shortcut attack is available for either cipher. Is Alice's approach as secure as Bob's?

41. Suppose that Alice has a secure block cipher, but the cipher only uses an 8-bit key. To make this cipher "more secure," Alice generates a random 64-bit key K, and iterates the cipher eight times, that is, she encrypts the plaintext P according to the rule

$$C = E(E(E(E(E(E(E(E(P, K_0), K_1), K_2), K_3), K_4), K_5), K_6), K_7),$$

    where $K_0, K_1, \ldots, K_7$ are the bytes of the 64-bit key K.

    a. Assuming known plaintext is available, how much work is required to determine the key K?

    b. Assuming a ciphertext-only attack, how much work is required to break this encryption scheme?

42. Suppose that we define triple 3DES with a 168-bit key as

$$C = E(E(E(P, K_1), K_2), K_3).$$

Suppose that we can compute and store a table of size 2^{56}, and a chosen plaintext attack is possible. Show that this triple 3DES is no more secure than the usual 3DES, which only uses a 112-bit key. Hint: Mimic the meet-in-the-middle attack on double DES.

43. Suppose that you know a MAC value X and the key K that was used to compute the MAC, but you do not know the original message. (It may be instructive to compare this problem to Problem 16 in Chapter 5.)

    a. Show that you can construct a message M that also has its MAC equal to X. Note that we are assuming that you know the key K and the same key is used for both MAC computations.

    b. How much of the message M are you free to choose?