Chapter 3

Block Ciphers

Block ciphers are the central tool in the design of protocols for shared-key cryptography (aka. symmetric) 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 an excellent 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.

This chapter gets you acquainted with some typical block ciphers, and discusses attacks on them. In particular we'll look at two examples, DES and AES. DES is the "old standby." It is currently 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.

3.1 What is a block cipher?

A block cipher is a function $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. This notation means that E takes two inputs, one being a k-bit string and the other an n-bit string, and returns an n-bit string. The first input is the key. The second might be called the plaintext, and the output might be called a ciphertext. 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 $E_K\colon \{0,1\}^n \to \{0,1\}^n$ 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 $\{0,1\}^n$. (For every $C \in \{0,1\}^n$ there is exactly one $M \in \{0,1\}^n$ such that $E_K(M) = C$.) Accordingly E_K has an inverse, and we 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 $E^{-1}(K, C)$. By "readily compute" we mean that there are public and relatively efficient programs available for these tasks.

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, we will assume the adversary will be able to obtain some 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 the view that the adversary's goal is key-recovery, seeing that security against key-recovery is a necessary but not sufficient condition for the security of a block cipher.)

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.

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

3.2 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.

3.2.1 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.

3.2.2 Construction

The DES algorithm is depicted in Fig. 3.1. It takes input a 56-bit key K and a 64 bit plaintext M. The key-schedule *KeySchedule* produces from the 56-bit key K a sequence of 16 subkeys, one

```
function $\mathrm{DES}_K(M)$    // $|K| = 56$ and $|M| = 64$
    $(K_1, \ldots, K_{16}) \leftarrow \mathit{KeySchedule}(K)$    // $|K_i| = 48$ for $1 \leq i \leq 16$
    $M \leftarrow \mathit{IP}(M)$
    Parse $M$ as $L_0 \,\|\, R_0$    // $|L_0| = |R_0| = 32$
    for $r = 1$ to $16$ do
        $L_r \leftarrow R_{r-1}$ ; $R_r \leftarrow f(K_r, R_{r-1}) \oplus L_{r-1}$
    $C \leftarrow \mathit{IP}^{-1}(L_{16} \,\|\, R_{16})$
    return $C$
```

Figure 3.1: The DES block cipher. The text and other figures describe the subroutines $\mathit{KeySchedule}, f, \mathit{IP}, \mathit{IP}^{-1}$.

IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Figure 3.2: Tables describing the DES initial permutation *IP* and its inverse IP^{-1}.

for each of the rounds that follows. Each subkey is 48-bits long. We postpone the discussion of the *KeySchedule* algorithm.

The initial permutation *IP* simply permutes the bits of M, as described by the table of Fig. 3.2. The table says that bit 1 of the output is bit 58 of the input; bit 2 of the output is bit 50 of the input; ... ; bit 64 of the output is bit 7 of the input. Note that the key is not involved in this permutation. The initial permutation does not appear to affect the cryptographic strength of the algorithm, and its purpose remains a bit of a mystery.

The permuted plaintext is now input to a loop, which operates on it in 16 rounds. Each round takes a 64-bit input, viewed as consisting of a 32-bit left half and a 32-bit right half, and, under the influence of the sub-key K_r, produces a 64-bit output. The input to round r is $L_{r-1} \,\|\, R_{r-1}$, and the output of round r is $L_r \,\|\, R_r$. Each round is what is called a Feistel round, named after Horst Feistel, one the IBM designers of a precursor of DES. Fig. 3.1 shows how it works, meaning how $L_r \,\|\, R_r$ is computed as a function of $L_{r-1} \,\|\, R_{r-1}$, by way of the function f, the latter depending on the sub-key K_r associated to the r-th round.

One of the reasons to use this round structure is that it is reversible, important to ensure that DES_K is a permutation for each key K, as it should be to qualify as a block cipher. Indeed, given $L_r \,\|\, R_r$ (and K_r) we can recover $L_{r-1} \,\|\, R_{r-1}$ via $R_{r-1} \leftarrow L_r$ and $L_{r-1} \leftarrow f(K - r, L_r) \oplus R_r$.

Following the 16 rounds, the inverse of the permutation *IP*, also depicted in Fig. 3.2, is applied to the 64-bit output of the 16-th round, and the result of this is the output ciphertext.

A sequence of Feistel rounds is a common high-level design for a block cipher. For a closer look we need to see how the function $f(\cdot, \cdot)$ works. It is shown in Fig. 3.3. It takes a 48-bit subkey J

```
function $f(J, R)$    // $|J| = 48$ and $|R| = 32$
    $R \leftarrow E(R)$ ; $R \leftarrow R \oplus J$
    Parse $R$ as $R_1 \| R_2 \| R_3 \| R_4 \| R_5 \| R_6 \| R_7 \| R_8$    // $|R_i| = 6$ for $1 \leq i \leq 8$
    for $i = 1, \ldots, 8$ do
        $R_i \leftarrow \mathbf{S}_i(R_i)$    // Each S-box returns 4 bits
    $R \leftarrow R_1 \| R_2 \| R_3 \| R_4 \| R_5 \| R_6 \| R_7 \| R_8$    // $|R| = 32$ bits
    $R \leftarrow P(R)$
    return $R$
```

Figure 3.3: The f-function of DES. The text and other figures describe the subroutines used.

E

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Figure 3.4: Tables describing the expansion function E and final permutation P of the DES f-function.

and a 32-bit input R to return a 32-bit output. The 32-bit R is first expanded into a 48-bit via the function E described by the table of Fig. 3.4. This says that bit 1 of the output is bit 32 of the input; bit 2 of the output is bit 1 of the input; ... ; bit 48 of the output is bit 1 of the input.

Note the E function is quite structured. 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 the sub-key J is XORed with the output of the E function to yield a 48-bit result that we continue to denote by R. This is split into 8 blocks, each 6-bits long. To the i-th block we apply the function $\mathbf{S}_i$ called the i-th S-box. Each S-box is a function taking 6 bits and returning 4 bits. The result is that the 48-bit R is compressed to 32 bits. These 32 bits are permuted according to the P permutation described in the usual way by the table of Fig. 3.4, and the result is the output of the f function. Let us now discuss the S-boxes.

Each S-box is described by a table as shown in Fig. 3.5. Read these tables as follows. $\mathbf{S}_i$ takes a 6-bit input. Write it as $b_1 b_2 b_3 b_4 b_5 b_6$. Read $b_3 b_4 b_5 b_6$ as an integer in the range $0, \ldots, 15$, naming a column in the table describing $\mathbf{S}_i$. Let $b_1 b_2$ name a row in the table describing $\mathbf{S}_i$. Take the row $b_1 b_2$, column $b_3 b_4 b_5 b_6$ entry of the table of $\mathbf{S}_i$ to get an integer in the range $0, \ldots, 15$. The output of $\mathbf{S}_i$ on input $b_1 b_2 b_3 b_4 b_5 b_6$ is the 4-bit string corresponding to this table entry.

The S-boxes are the heart of the algorithm, and much effort was put into designing them to achieve various security goals and resistance to certain attacks.

Finally, we discuss the key schedule. It is shown in Fig. 3.6. Each round sub-key K_r is formed by taking some 48 bits of K. Specifically, a permutation called PC-1 is first applied to the 56-bit

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_1$:	0	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	1	0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	1	1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_2$:	0	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	0	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	1	0	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	1	1	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_3$:	0	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	0	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	1	0	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	1	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_4$:	0	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	0	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	1	0	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	1	1	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_5$:	0	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	0	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	1	0	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	1	1	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_6$:	0	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	0	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	1	0	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	1	1	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_7$:	0	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	0	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	0	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	1	1	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathbf{S}_8$:	0	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	0	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	1	0	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	1	1	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 3.5: The DES S-boxes.

key to yield a permuted version of it. This is then divided into two 28-bit halves and denoted $C_0 \parallel D_0$. The algorithm now goes through 16 rounds. The r-th round takes input $C_{r-1} \parallel D_{r-1}$, computes $C_r \parallel D_r$, and applies a function PC-2 that extracts 48 bits from this 56-bit quantity. This is the sub-key K_r for the r-th round. The computation of $C_r \parallel D_r$ is quite simple. The bits of C_{r-1} are rotated to the left j positions to get C_r, and D_r is obtained similarly from D_{r-1}, where j is either 1 or 2, depending on r.

```
Algorithm $KeySchedule(K)$    // $|K| = 56$
    $K \leftarrow PC\text{-}1(K)$
    Parse $K$ as $C_0 \parallel D_0$
    for $r = 1, \ldots, 16$ do
        if $r \in \{1, 2, 9, 16\}$ then $j \leftarrow 1$ else $j \leftarrow 2$ fi
        $C_r \leftarrow leftshift_j(C_{r-1})$ ; $D_r \leftarrow leftshift_j(D_{r-1})$
        $K_r \leftarrow PC\text{-}2(C_r \parallel D_r)$
    return$(K_1, \ldots, K_{16})$
```

Figure 3.6: The key schedule of DES. Here $leftshift_j$ denotes the function that rotates its input to the left by j positions.

PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Figure 3.7: Tables describing the *PC*-1 and *PC*-2 functions used by the DES key schedule of Fig. 3.6.

The functions *PC*-1 and *PC*-2 are tabulated in Fig. 3.7. The first table needs to be read in a strange way. It contains 56 integers, these being all integers in the range $1, \ldots, 64$ barring multiples of 8. Given a 56-bit string $K = K[1] \ldots K[56]$ as input, the corresponding function returns the 56-bit string $L = L[1] \ldots L[56]$ computed as follows. Suppose $1 \leq i \leq 56$, and let a be the i-th entry of the table. Write $a = 8q + r$ where $1 \leq r \leq 7$. Then let $L[i] = K[a - q]$. As an example, let us determine the first bit, $L[1]$, of the output of the function on input K. We look at the first entry in the table, which is 57. We divide it by 8 to get $57 = 8(7) + 1$. So $L[1]$ equals $K[57 - 7] = K[50]$, meaning the 1st bit of the output is the 50-th bit of the input. On the other hand *PC*-2 is read in the usual way as a map taking a 56-bit input to a 48 bit output: bit 1 of the output is bit 14 of the input; bit 2 of the output is bit 17 of the input; ... ; bit 56 of the output is bit 32 of the input.

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 provides some information.

3.2.3 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-box 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.

3.3 Key recovery attacks on block ciphers

Now that we know what a block cipher looks like, let us consider attacking one. This is called cryptanalysis of the block cipher.

We fix a block cipher $E\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ having key-size k and block size n. It is assumed that the attacker knows the description of E and can compute it. For concreteness, you can think of E as being DES.

Historically, cryptanalysis of block ciphers has focused on key-recovery. The cryptanalyst may think of the problem to be solved as something like this. A k-bit key T, called the target key, 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_T, say

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

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

Find: The adversary wants to find the target key T.

Let us say that a key K is *consistent with the input-output examples* $(M_1, C_1), \ldots, (M_q, C_q)$ if $E_K(M_i) = C_i$ for all $1 \leq i \leq q$. We let

$$\mathsf{Cons}_E((M_1, C_1), \ldots, (M_q, C_q))$$

be the set of all keys consistent with the input-output examples $(M_1, C_1), \ldots, (M_q, C_q)$. Of course the target key T is in this set. But the set might be larger, containing other keys. A key-recovery attack cannot hope to differentiate the target key from other members of the set $\mathsf{Cons}_E((M_1, C_1), \ldots, (M_q, C_q))$. Thus, the goal is sometimes viewed as simply being to find some key in this set, or even the entire set. For practical block ciphers though, if enough input-output examples are used, the size of this set is usually one, so that one can indeed find the target key. We will exemplify all this when we consider specific attacks.

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 strategy is exhaustive key search. The adversary goes through all possible keys $K' \in \{0,1\}^k$ until it finds one that explains the input-output pairs. Here is the attack in detail, using $q = 1$, meaning one input-output example. For $i = 1, \ldots, 2^k$ let T_i denote the i-th k-bit string (in lexicographic order).

```
$EKS_E(M_1, C_1)$
    for  $i = 1, \ldots, 2^k$ do
        if $E(T_i, M_1) = C_1$ then  return $T_i$ fi
```

This attack always returns a key consistent with the given input-output example (M_1, C_1). Whether or not it is the target key depends on the block cipher, and in particular on its key length and block length, and in some cases the probability of this is too small. The likelihood of the attack returning the target key can be increased by testing against more input-output examples:

```
$EKS_E((M_1, C_1), \ldots, (M_q, C_q))$
    for  $i = 1, \ldots, 2^k$ do
        if $E(T_i, M_1) = C_1$ then
            if ( $E(T_i, M_2) = C_2$ AND $\cdots$ AND $E(T_i, M_q) = C_q$ ) then  return $T_i$ fi
```

A fairly small vaue of q, say somewhat more than k/n, is enough that this attack will usually return the target key itself. For DES, $q = 2$ is enough.

Thus, no block cipher is perfectly secure. It is always possible for an attacker to recover the key. A good block cipher, however, is designed to make this task computationally prohibitive.

How long does exhaustive key-search take? Since q is small we can neglect the difference in running time between the two versions of the attack above, and focus for simplicity on the first attack. In the worst case, it uses 2^k computations of the block cipher. However it could be less since one could get lucky. For example if the target key is in the first half of the search space, only 2^{k-1} computations would be used. So a better measure is how long it takes on the average. This is

$$\sum_{i=1}^{2^k} i \cdot \Pr[K = T_i] \;=\; \sum_{i=1}^{2^k} \frac{i}{2^k} \;=\; \frac{1}{2^k} \cdot \sum_{i=1}^{2^k} i \;=\; \frac{1}{2^k} \cdot \frac{2^k(2^k+1)}{2} \;=\; \frac{2^k+1}{2} \approx 2^{k-1}$$

computations of the block cipher. This is because the target key is chosen at random, so with probability $1/2^k$ equals T_i, and in that case the attack uses i E-computations to find it.

Thus to make key-recovery by exhaustive search computationally prohibitive, one must make the key-length k of the block cipher large enough.

Let's look at DES. We noted above that there is VLSI chip that can compute it at the rate of 1.6 Gbits/sec. How long would key-recovery via exhaustive search take using this chip? Since a DES plaintext is 64 bits, the chip enables us to perform $(1.6 \cdot 10^9)/64 = 2.5 \cdot 10^7$ DES computations per second. To perform 2^{55} computations (here $k = 56$) we thus need $2^{55}/(2.5 \cdot 10^7) \approx 1.44 \cdot 10^9$ seconds, which is about 45.7 years. This is clearly prohibitive.

It turns out that that DES has a property called key-complementation that one can exploit to reduce the size of the search space by one-half, so that the time to find a key by exhaustive search comes down to 22.8 years. But this is still prohibitive.

Yet, the conclusion that DES is secure against exhaustive key search is actually too hasty. We will return to this later and see why.

Exhaustive key search is a generic attack in the sense that it works against any block cipher. It only involves computing the block cipher and makes no attempt to analyze the cipher and find and exploit weaknesses. Cryptanalysts also need to ask themselves if there is some weakness in the structure of the block cipher they can exploit to obtain an attack performing better than exhaustive key search.

For DES, the discovery of such attacks waited until 1990. Differential cryptanalysis is capable of finding a DES key using about 2^{47} input-output examples (that is, $q = 2^{47}$) in a chosen-message attack [1, 2]. Linear cryptanalysis [4] improved differential in two ways. The number of input-output examples required is reduced to 2^{44}, and only a known-message attack is required. (An alternative version uses 2^{42} chosen plaintexts [6].)

These were major breakthroughs in cryptanalysis that required careful analysis of the DES construction to find and exploit weaknesses. Yet, the practical impact of these attacks is small. Why? Ordinarily it would be impossible to obtain 2^{44} input-output examples. Furthermore, the storage requirement for these examples is prohibitive. A single input-output pair, consisting of a 64-bit plaintext and 64-bit ciphertext, takes 16 bytes of storage. When there are 2^{44} such pairs, we need $16 \cdot 2^{44} = 2.81 \cdot 10^{14}$ bits, or about 281 terabytes of storage, which is enormous.

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 that the DES computations in this attack can be performed in parallel. In 1993, Weiner argued that one can design a $1 million machine that does the exhaustive key search for DES in about 3.5 hours on the average [7]. His machine would have about 57,000 chips, each performing numerous DES computations. More recently, a DES key search machine was actually built by the Electronic Frontier Foundation, at a cost of $250,000 [5]. It finds the key in 56 hours, or about 2.5 days on the average. 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 a weak algorithm. 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.

3.4 Iterated-DES and DESX

The emergence of the above-discussed key-search engines lead to the view that in practice DES should be considered broken. Its shortcoming was its key-length of 56, not long enough to resist exhaustive key search.

People looked for cheap ways to strengthen DES, turning it, in some simple way, into a cipher with a larger key length. One paradigm towards this end is iteration.

3.4.1 Double-DES

Let K_1, K_2 be 56-bit DES keys and let M be a 64-bit plaintext. Let

$$\mathsf{2DES}(K_1 \,\|\, K_2, M) \;=\; \mathrm{DES}(K_2, \mathrm{DES}(K_1, M)) \;.$$

This defines a block cipher $\mathsf{2DES}\colon \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$ that we call *Double-DES*. It has a 112-bit key, viewed as consisting of two 56-bit DES keys. Note that it is reversible, as required to be a block cipher:

$$\mathsf{2DES}^{-1}(K_1 \,\|\, K_2, C) \;=\; \mathrm{DES}^{-1}(K_1, \mathrm{DES}^{-1}(K_2, C)) \;.$$

for any 64-bit C.

The key length of 112 is large enough that there seems little danger of $\mathsf{2DES}$ succumbing to an exhaustive key search attack, even while exploiting the potential for parallelism and special-purpose hardware. On the other hand, $\mathsf{2DES}$ also seems secure against the best known cryptanalytic techniques, namely differential and linear cryptanalysis, since the iteration effectively increases the number of Feistel rounds. This would indicate that $\mathsf{2DES}$ is a good way to obtain a DES-based cipher more secure than DES itself.

However, although $\mathsf{2DES}$ has a key-length of 112, it turns out that it can be broken using about 2^{57} DES and DES^{-1} computations by what is called a meet-in-the-middle attack, as we now illustrate. Let $K_1 \,\|\, K_2$ denote the target key and let $C_1 = \mathsf{2DES}(K_1 \,\|\, K_2, M_1)$. The attacker, given M_1, C_1, is attempting to find $K_1 \,\|\, K_2$. We observe that

$$C_1 = \mathrm{DES}(K_2, \mathrm{DES}(K_1, M_1)) \qquad \Rightarrow \qquad \mathrm{DES}^{-1}(K_2, C_1) = \mathrm{DES}(K_1, M_1) \;.$$

This leads to the following attack. Below, for $i = 1, \ldots, 2^{56}$ we let T_i denote the i-th 56-bit string (in lexicographic order):

```
$MinM_{\mathsf{2DES}}(M_1, C_1)$
    for $i = 1, \ldots, 2^{56}$ do $L[i] \leftarrow \mathrm{DES}(T_i, M_1)$
    for $j = 1, \ldots, 2^{56}$ do $R[j] \leftarrow \mathrm{DES}^{-1}(T_j, C_1)$
    $S \leftarrow \{\, (i, j) \;:\; L[i] = R[j] \,\}$
    Pick some $(l, r) \in S$ and return $T_l \,\|\, T_r$
```

For any $(i, j) \in S$ we have

$$\mathrm{DES}(T_i, M_1) = L[i] = R[j] = \mathrm{DES}^{-1}(T_j, C_1)$$

and as a consequence $\mathrm{DES}(T_j, \mathrm{DES}(T_i, M_1)) = C_1$. So the key $T_i \,\|\, T_j$ is consistent with the input-output example (M_1, C_1). Thus,

$$\{\, T_l \,\|\, T_r \;:\; (l, r) \in S \,\} \;=\; \mathsf{Cons}_E((M_1, C_1)) \;.$$

The attack picks some pair (l, r) from S and outputs $T_l \,\|\, T_r$, thus returning a key consistent with the input-output example (M_1, C_1).

The set S above is likely to be quite large, of size about $2^{56+56}/2^{64} = 2^{48}$, meaning the attack as written is not likely to return the target key itself. However, by using a few more input-output examples, it is easy to whittle down the choices in the set S until it is likely that only the target key remains.

The attack makes $2^{56} + 2^{56} = 2^{57}$ DES or DES^{-1} computations. The step of forming the set S can be implemented in linear time in the size of the arrays involved, say using hashing. (A naive strategy takes time quadratic in the size of the arrays.) Thus the running time is dominated by the $\mathrm{DES}, \mathrm{DES}^{-1}$ computations.

The meet-in-the-middle attack shows that $\mathsf{2DES}$ is quite far from the ideal of a cipher where the best attack is exhaustive key search. However, this attack is not particularly practical, even if special purpose machines are designed to implement it. The machines could do the $\mathrm{DES}, \mathrm{DES}^{-1}$ computations quickly in parallel, but to form the set S the attack needs to store the arrays L, R, each of which has 2^{56} entries, each entry being 64 bits. The amount of storage required is $8 \cdot 2^{57} \approx 1.15 \cdot 10^{18}$ bytes, or about $1.15 \cdot 10^6$ terabytes, which is so large that implementing the attack is impractical.

There are some strategies that modify the attack to reduce the storage overhead at the cost of some added time, but still the attack does not appear to be practical.

Since a 112-bit $\mathsf{2DES}$ key can be found using 2^{57} DES or DES^{-1} computations, we sometimes say that $\mathsf{2DES}$ has an *effective key length* of 57.

3.4.2 Triple-DES

The triple-DES ciphers use three iterations of DES or DES^{-1}. The three-key variant is defined by

$$\mathsf{3DES3}(K_1 \,\|\, K_2 \,\|\, K_3, M) \;=\; \mathrm{DES}(K_3, \mathrm{DES}^{-1}(K_2, \mathrm{DES}(K_1, M)))\;,$$

so that $\mathsf{3DES3}\colon\ \{0,1\}^{168} \times \{0,1\}^{64} \to \{0,1\}^{64}$. The two-key variant is defined by

$$\mathsf{3DES2}(K_1 \,\|\, K_2, M) \;=\; \mathrm{DES}(K_2, \mathrm{DES}^{-1}(K_1, \mathrm{DES}(K_2, M)))\;,$$

so that $\mathsf{3DES2}\colon\ \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$. You should check that these functions are reversible so that they do qualify as block ciphers. The term “triple” refers to there being three applications of DES or DES^{-1}. The rationale for the middle application being DES^{-1} rather than DES is that DES is easily recovered via

$$\mathrm{DES}(K, M) \;=\; \mathsf{3DES3}(K \,\|\, K \,\|\, K, M) \qquad (3.1)$$

$$\mathrm{DES}(K, M) \;=\; \mathsf{3DES2}(K \,\|\, K, M)\;. \qquad (3.2)$$

As with $\mathsf{2DES}$, the key length of these ciphers appears long enough to make exhaustive key search prohibitive, even with the best possible engines, and, additionally, differential and linear cryptanalysis are not particularly effective because iteration effectively increases the number of Feistel rounds.

$\mathsf{3DES3}$ is subject to a meet-in-the-middle attack that finds the 168-bit key using about 2^{112} computations of DES or DES^{-1}, so that it has an effective key length of 112. There does not appear to be a meet-in-the-middle attack on $\mathsf{3DES2}$ however, so that its key length of 112 is also its effective key length.

The $\mathsf{3DES2}$ cipher is popular in practice and functions as a canonical and standard replacement for DES. $\mathsf{2DES}$, although having the same effective key length as $\mathsf{3DES2}$ and offering what appears to be the same or at least adequate security, is not popular in practice. It is not entirely apparent why $\mathsf{3DES2}$ is preferred over $\mathsf{2DES}$, but the reason might be Equation (3.2).

3.4.3 DESX

Although $\mathsf{2DES}, \mathsf{3DES3}$ and $\mathsf{3DES2}$ appear to provide adequate security, they are slow. The first is twice as slow as DES and the other two are three times as slow. It would be nice to have a DES based block cipher that had a longer key than DES but was not significantly more costly. Interestingly, there is a simple design that does just this. Let K be a 56-bit DES key, let K_1, K_2 be 64-bit strings, and let M be a 64-bit plaintext. Let

$$\mathsf{DESX}(K \,\|\, K_1 \,\|\, K_2, M) \;=\; K_2 \oplus \mathrm{DES}(K, K_1 \oplus M)\;.$$

This defines a block cipher $\mathsf{DESX}\colon\ \{0,1\}^{184} \times \{0,1\}^{64} \to \{0,1\}^{64}$. It has a 184-bit key, viewed as consisting of a 56-bit DES key plus two auxiliary keys, each 64 bits long. Note that it is reversible, as required to be a block cipher:

$$\mathsf{DESX}^{-1}(K \parallel K_1 \parallel K_2, C) \;=\; K_1 \oplus \mathrm{DES}^{-1}(K, K_2 \oplus C)\;.$$

The key length of 184 is certainly enough to preclude exhaustive key search attacks. DESX is no more secure than DES against linear of differential cryptanalysis, but we already saw that these are not really practical attacks.

There is a meet-in-the-middle attack on DESX. It finds a 184-bit DESX key using 2^{120} DES and DES^{-1} computations. So the effective key length of DESX seems to be 120, which is large enough for security.

DESX is less secure than Double or Triple DES because the latter are more more resistant than DES to linear and differential cryptanalysis while DESX is only as good as DES itself in this regard. However, this is good enough; we saw that in practice the weakness of DES was not these attacks but rather the short key length leading to successful exhaustive search attacks. DESX fixes this, and very cheaply. In summary, DESX is popular because it is much cheaper than Double of Triple DES while providing adequate security.

3.4.4 Why a new cipher?

DESX is arguably a fine cipher. Nonetheless, there were important reasons to find and standardize a new cipher.

We will see later that the security provided by a block cipher depends not only on its key length and resistance to key-search attacks but on its block length. A block cipher with block length n can be "broken" in time around $2^{n/2}$. When $n = 64$, this is 2^{32}, which is quite small. Although $\mathsf{2DES}, \mathsf{3DES3}, \mathsf{3DES2}, \mathsf{DESX}$ have a higher (effective) key length than DES, they preserve its block size and thus are no more secure than DES from this point of view. It was seen as important to have a block cipher with a block length n large enough that a $2^{n/2}$ time attack was not practical. This was one motivation for a new cipher.

Perhaps the larger motivation was speed. Desired was a block cipher that ran faster than DES in software.

3.5 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.

```
**function** $\mathrm{AES}_K(M)$
    $(K_0, \ldots, K_{10}) \leftarrow \mathit{expand}(K)$
    $s \leftarrow M \oplus K_0$
    **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** $s$
```

Figure 3.8: The function AES128. See the accompanying text and figures for definitions of the maps *expand*, S, *shift-rows*, *mix-cols*.

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 Fig. 3.8.

Refer to Fig. 3.8. The value s is called the *state*. One initizlizes the state to M and the final state is the ciphertext C one gets by enciphering M. What happens in each iteration of the **for** loop is called a *round*. So AES 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, we 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 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 $\{\text{a1}\} \cdot \{03\}$ compute $\{\text{a1}\} \cdot (\{02\} \oplus \{01\}) = \{\text{a1}\} \cdot \{02\} \oplus \{\text{a1}\} \cdot \{01\} = \{42\} \oplus \{1\text{b}\} \oplus a1 = \{\text{f8}\}$. 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 Fig. 3.9, 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 $s =$

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 3.9: 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: $S(00) = 63$, $S(01) = 7\mathrm{c}$, …, $S(\mathrm{ff}) = 16$.

$s_0 s_1 \ldots s_{15}$ going top-to-bottom, then left-to-right, to make a 4 × 4 table:

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

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 *mix-cols*(s) on 4-byte words, and then extend this to a 16-byte quantity wordwise. The value of *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 $\mathtt{x}^4 + 1$.

```
function $expand(K)$
    $K_0 \leftarrow K$
    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 $(K_0, \ldots, K_{10})$
```

Figure 3.10: 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.

At this point we have described everything but the key-expansion map, *expand*. That map is given in Fig. 3.10.

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. 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 [3]).

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.

3.6 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 T is a random, unknown key and $C_i = E_T(M_i)$, how hard is it for an attacker to find T? 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.

As an example, consider that we typically want to think of $C = E_K(M)$ as an "encryption" of plaintext M under key K. An adversary in possession of C but not knowing K should find it computationally infeasible to recover M, or even some part of M such as its first half. Security against key-recovery is certainly necessary for this, because if the adversary could find K it could certainly compute M, via $M = E_K^{-1}(M)$. But security against key-recovery is not sufficient to ensure that M cannot be recovered given K alone. As an example, consider the block cipher E: $\{0,1\}^{128} \times \{0,1\}^{256} \to \{0,1\}^{256}$ defined by $E_K(M) = \mathrm{AES}_K(M[1]) \parallel M[2]$ where $M[1]$ is the first 128 bits of M and $M[2]$ is the last 128 bits of M. Key recovery is as hard as for AES, but a ciphertext reveals the second half of the plaintext.

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.

But then what does make a good block cipher? This questions turns out to not be so easy to answer. Certainly one can list various desirable properties. For example, the ciphertext should not reveal half the bits of the plaintext. But that is not enough either. 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.

Such a 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.

Such a property is that the block cipher be a pseudorandom permutation (PRF), a notion explored in another chapter.

3.7 Problems

Problem 3.1 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. ▌

Problem 3.2 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. ▌

Problem 3.3 Find a key K such that $\mathrm{DES}_K(\cdot) = \mathrm{DES}_K^{-1}(\cdot)$. Such a key is sometimes called a "weak" key. ▌

Problem 3.4 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:

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

▮

Problem 3.5 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. ▮

Problem 3.6 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. ▮

Problem 3.7 How many different permutations are there from 128 bits to 128 bits? How man different functions are then from 128 bits to 128 bits? ▮

Problem 3.8 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 3.9 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 3.10 Justify and then refute (both) the following proposition: enciphering under AES can be implemented faster than deciphering. ▮

Bibliography

[1] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *J. of Cryptology*, Vol. 4, No. 1, pp. 3–72, 1991.

[2] E. Biham and A. Shamir. Differential cryptanalysis of the Full 16-round DES. *Advances in Cryptology – CRYPTO '92*, Lecture Notes in Computer Science Vol. 740, E. Brickell ed., Springer-Verlag, 1992.

[3] J. Daemen and V. Rijmen. The Design of Rijndael. Springer, 2001.

[4] M. Matsui. Linear cryptanalysis method for DES cipher. *Advances in Cryptology – EUROCRYPT '93*, Lecture Notes in Computer Science Vol. 765, T. Helleseth ed., Springer-Verlag, 1993.

[5] EFF DES Cracker Project. `http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/`.

[6] L. Knudsen and J. E. Mathiassen. A Chosen-Plaintext Linear Attack on DES. *Fast Software Encryption '00*, Lecture Notes in Computer Science Vol. 1978, B. Schneier ed., Springer-Verlag, 2000.

[7] M. Wiener. Efficient DES Key Search. In Practical Cryptography for Data Internetworks, W. Stallings editor, IEEE Computer Society Press, 1996, pp. 31–79. `http://www3.sympatico.ca/wienerfamily/Michael/MichaelPapers/dessearch.pdf`.