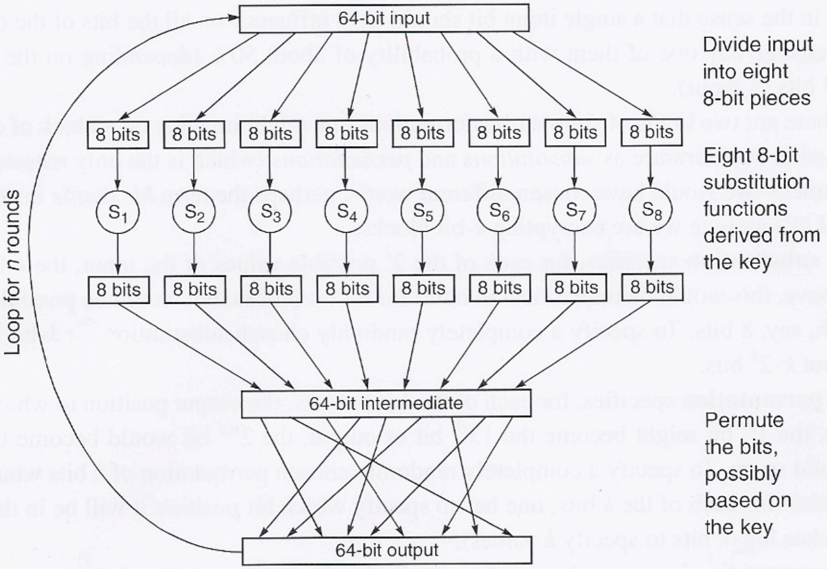

64-bit input
Divide input into eight 8-bit pieces
8 bits
Eight 8-bit substitution functions derived from the key
S_1
S_2
S_3
S_4
S_5
S_6
S_7
S_8
Loop for *n* rounds
64-bit intermediate
Permute the bits, possibly based on the key
64-bit output

Figure 3-1. Example of Block Encryption

3.3 DATA ENCRYPTION STANDARD (DES)

DES was published in 1977 by the National Bureau of Standards (since renamed to the National Institute of Standards and Technology) for use in commercial and unclassified (hmm…) U.S. Government applications. It was designed by IBM based on their own *Lucifer cipher* and input from NSA. DES uses a 56-bit key, and maps a 64-bit input block into a 64-bit output block. The key actually looks like a 64-bit quantity, but one bit in each of the 8 octets is used for odd parity on each octet. Therefore, only 7 of the bits in each octet are actually meaningful as a key.

DES is efficient to implement in hardware but relatively slow if implemented in software. Although making software implementations difficult was not a documented goal of DES, people have asserted that DES was specifically designed with this in mind, perhaps because this would limit its use to organizations that could afford hardware-based solutions, or perhaps because it made it easier to control access to the technology. At any rate, advances in CPUs have made it feasible to do DES in software. For instance, a 500-MIP CPU can encrypt at about 30 Koctets per second (and perhaps more depending on the details of the CPU design and the cleverness of the implementation). This is adequate for many applications.

Why 56 bits?

Use of a 56-bit key is one of the most controversial aspects of DES. Even before DES was adopted, people outside of the intelligence community complained that 56 bits provided inadequate security [DENN82, DIFF76a, DIFF77, HELL79]. So why were only 56 of the 64 bits of a DES key used in the algorithm? The disadvantage of using 8 bits of the key for parity checking is that it makes DES considerably less secure (256 times less secure against exhaustive search).

OK, so what is the advantage of using 8 bits of the key for parity? Well, uh, let's say you receive a key electronically, and you want to sanity-check it to see if it could actually be a key. If you check the parity of the quantity, and it winds up not having the correct parity, then you'll know something went wrong.

There are two problems with this reasoning. One is that there is a 1 in 256 chance (given the parity scheme) that even if you were to get 64 bits of garbage, that the result will happen to have the correct parity and therefore look like a key. That is way too large a possibility of error for it to afford any useful protection to any application. The other problem with the reasoning is that there is nothing terribly bad about getting a bad key. You'll discover the key is bad when you try to use it for encrypting or decrypting.

The key, at 56 bits, is pretty much universally acknowledged to be too small to be secure. Perhaps one might argue that a smaller key is an advantage because it saves storage—but that argument doesn't hold since nobody does data compression on the 64-bit keys in order to fit them into 56 bits. So what benefits are there to usurping 8 bits for parity that offset the loss in security?

People (not us, surely!) have suggested that our government consciously decided to weaken the security of DES just enough so that NSA would be able to break it. We would like to think there is an alternative explanation, but we have never heard a plausible one proposed.

Advances in semiconductor technology make the key-length issue more critical. Chip speeds have caught up so that DES keys can be broken with a bit of cleverness and exhaustive search. Perhaps a 64-bit key might have extended its useful lifetime by a few years. Given hardware price/performance improving about 40% per year, keys must grow by about 1 bit every 2 years. Assuming 56 bits was just sufficient in 1979 (when DES was standardized), 64 bits was about right in 1995, and 128 bits would suffice until 2123.

How secure is DES?

Suppose you have a single block of ⟨plaintext, ciphertext⟩. *Breaking* DES in this case would mean finding a key that maps that plaintext to that ciphertext. With DES implemented in software, it would take on the order of half a million MIP-years, through brute force, to find the key. (Is it possible to find the "wrong" key, given a particular pair? Might two different keys map the same plaintext to the same ciphertext? How many keys on the average map a particular pair? See Homework Problem 3.)

Often the attacker does not have a ⟨plaintext, ciphertext⟩ block. Instead the attacker has a reasonable amount of ciphertext only. It might be known, for example, that the encrypted data is likely to be 7-bit ASCII. In that case, it is still just about as efficient to do brute-force search. The ciphertext is decrypted with the guessed key, and if all the 8^{th} bits are zero (which will happen with an incorrect key with probability 1 in 256), then another block is decrypted. After several (say ten) blocks are decrypted, and the result always appears to be 7-bit ASCII, the key has a high probability of being correct.

Current commercial DES chips do not lend themselves to doing exhaustive key search—they allow encrypting lots of data with a particular key. The relative speed of key loading is much less than the speed of encrypting data. However, it is straightforward to design and manufacture a key-searching DES chip.

In 1977, Diffie and Hellman [DIFF77] did a detailed analysis of what it would cost to build a DES-breaking engine and concluded that for $20 million you could build a million-chip machine that could find a DES key in twelve hours (given a ⟨plaintext, ciphertext⟩ pair). In 1998, EFF (Electronic Frontier Foundation) [EFF98] built a special-purpose DES-breaking engine, called the EFF DES Cracker, for under $250K. It was designed to find a DES key in 4.5 days. With the design done, the cost of replicating the engine was under $150K.

There are published papers [BIHA93] claiming that less straightforward attacks can break DES faster than simply searching the key space. However, these attacks involve the premise, unlikely in real-life situations, that the attacker can choose lots of plaintext and obtain the corresponding ciphertext.

Still it is possible to encrypt multiple times with different keys (see §4.4 *Multiple Encryption DES*). It is generally believed that DES with triple encryption is 2^{56} times as difficult to crack and therefore will be secure for the foreseeable future.

3.3.1 DES Overview

DES is quite understandable, and has some very elegant tricks. Let's start with the basic structure of DES (Figure 3-2).

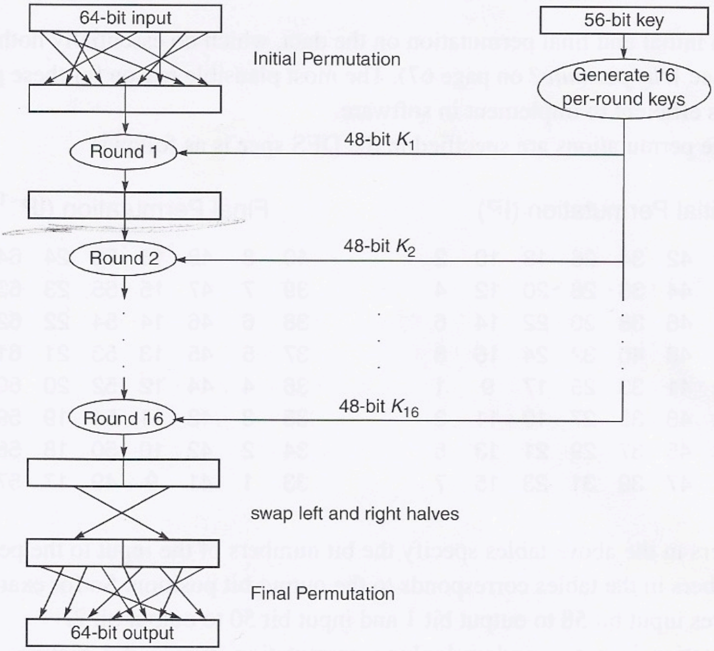


Figure 3-2. Basic Structure of DES

The 64-bit input is subjected to an initial permutation to obtain a 64-bit result (which is just the input with the bits shuffled). The 56-bit key is used to generate sixteen 48-bit per-round keys, by taking a different 48-bit subset of the 56 bits for each of the keys. Each round takes as input the 64-bit output of the previous round, and the 48-bit per-round key, and produces a 64-bit output. After the 16^{th} round, the 64-bit output has its halves swapped and is then subjected to another permutation, which happens to be the inverse of the initial permutation.

encrypt

That is the overview of how encryption works. Decryption works by essentially running DES backwards. To decrypt a block, you'd first run it through the initial permutation to undo the final permutation (the initial and final permutations are inverses of each other). You'd do the same key generation, though you'd use the keys in the opposite order (first use K_{16}, the key you generated last). Then you run 16 rounds just like for encryption. Why this works will be explained when we explain what happens during a round. After 16 rounds of decryption, the output has its halves swapped and is then subjected to the final permutation (to undo the initial permutation).

decrypt

To fully specify DES, we need to specify the initial and final permutations, how the per round keys are generated, and what happens during a round. Let's start with the initial and final permutations of the data.

3.3.2 The Permutations of the Data

DES performs an initial and final permutation on the data, which do essentially nothing to enhance DES's security (see *Why permute?* on page 67). The most plausible reason for these permutations is to make DES less efficient to implement in software.

The way the permutations are specified in the DES spec is as follows:

Initial Permutation (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

Final Permutation (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

The numbers in the above tables specify the bit numbers of the input to the permutation. The order of the numbers in the tables corresponds to the output bit position. So for example, the initial permutation moves input bit 58 to output bit 1 and input bit 50 to output bit 2.

The permutation is not a random-looking permutation. Figure 3-3 pictures it. The arrows indicate the initial permutation. Reverse the arrows to get the final permutation. We hope you appreciate the time we spent staring at the numbers and discovering this completely useless structure.

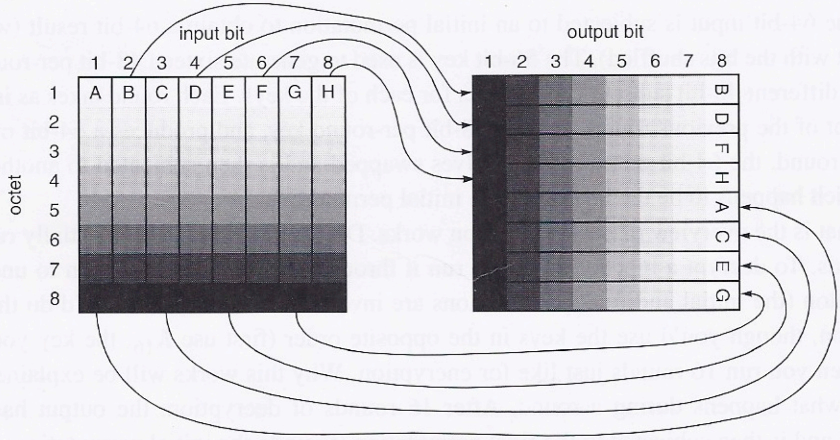


Figure 3-3. Initial Permutation of Data Block

The input is 8 octets. The output is 8 octets. The bits in the first octet of input get spread into the 8^{th} bits of each of the octets. The bits in the second octet of input get spread into the 7^{th} bits of all the octets. And in general, the bits of the i^{th} octet get spread into the $(9-i)^{th}$ bits of all the octets. The pattern of spreading of the 8 bits in octet i of the input among the output octets is that the even-numbered bits go into octets 1–4, and the odd-numbered bits go into octets 5–8. Note that if the data happens to be 7-bit ASCII, with the top bit set to zero, then after the permutation the entire 5^{th} octet will be zero. Since the permutation appears to have no security value, it seems nearly certain that there is no security significance to this particular permutation.

Why permute?

Why can't the initial and final permutations of the data be of security value? Well, suppose they were important, i.e., if DES did not have them it would be possible to break DES. Let's call a modified DES that does not have the initial and final permutation **EDS**. Let's say we can break EDS, i.e., given a $\langle$plaintext,ciphertext$\rangle$ EDS pair, we can easily calculate the EDS key that converts the plaintext into the ciphertext. In that case, we can easily break DES as well. Given a DES $\langle$plaintext,ciphertext$\rangle$ pair $\langle m,c \rangle$, we simply do the inverse of the initial permutation (i.e. the final permutation) on m to get m', and the inverse of the final permutation (i.e. the initial permutation) on c to get c', and feed $\langle m',c' \rangle$ to our EDS-breaking code. The resulting EDS key will work as the DES key for $\langle m,c \rangle$.

Note that when multiple encryptions of DES are being performed, the permutation might have some value. However, if encryption with key$_1$ is followed by encryption with key$_2$, then the final permutation following encryption with key$_1$ will cancel the initial permutation for key$_2$. That is one of the reasons people discuss alternating encrypt operations with decrypt operations (see §4.4 *Multiple Encryption DES*).

In §3.3.3 *Generating the Per-Round Keys*, we'll see there is also a permutation of the key. It also has no security value (by a similar argument).

3.3.3 Generating the Per-Round Keys

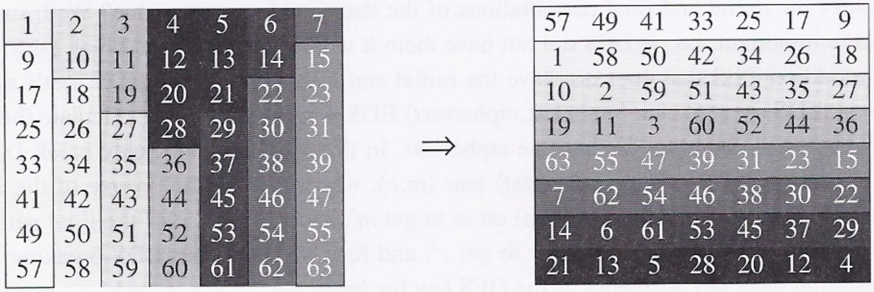
Next we'll specify how the sixteen 48-bit per-round keys are generated from the DES key. The DES key looks like it's 64 bits long, but 8 of the bits are parity. Let's number the bits of the DES key from left to right as 1, 2,...64. Bits 8, 16,...64 are the parity bits. DES performs a function, which we are about to specify, on these 64 bits to generate sixteen 48-bit keys, which are K_1, K_2,...K_{16}.

First it does an initial permutation on the 56 useful bits of the key, to generate a 56-bit output, which it divides into two 28-bit values, called C_0 and D_0. The permutation is specified as

C_0								D_0						
57	49	41	33	25	17	9		63	55	47	39	31	23	15
1	58	50	42	34	26	18		7	62	54	46	38	30	22
10	2	59	51	43	35	27		14	6	61	53	45	37	29
19	11	3	60	52	44	36		21	13	5	28	20	12	4

The way to read the table above is that the leftmost bit of the output is obtained by extracting bit 57 from the key. The next bit is bit 49 of the key, and so forth, with the final bit of D_0 being bit 4 of the key. Notice that none of the parity bits (8, 16,…64) is used in C_0 or D_0.

This permutation is not random. Figure 3-4 pictures it. Feel free to draw in any arrows or other graphic aids to make it clearer.



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

⇒

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

Figure 3-4. Initial Permutation of Key

The initial and final permutations of the bits in the key have no security value (just like the initial and final permutations of the data), so the permutations didn't have to be random—the identity permutation would have done nicely.

Now the generation of the K_i proceeds in 16 rounds (see Figure 3-5). The number of bits shifted is different in the different rounds. In rounds 1, 2, 9, and 16, it is a single-bit rotate left (with

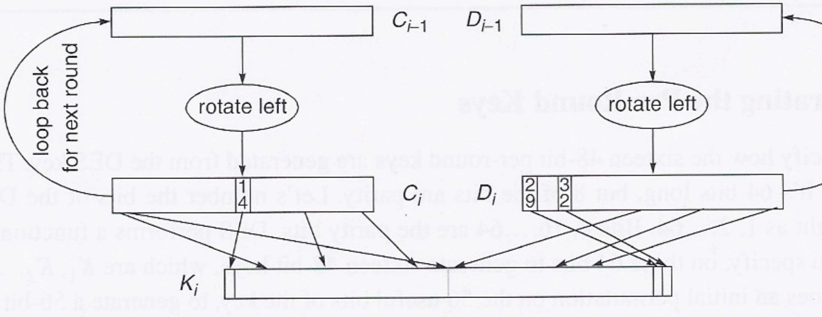


Figure 3-5. Round *i* for generating K_i

the bit shifted off the left end carried around and shifted into the right end). In the other rounds, it is a two-bit rotate left.

The permutations in this case are likely to be of some security value.

The permutation of C_i that produces the left half of K_i is the following. Note that bits 9, 18, 22, and 25 are discarded.

permutation to obtain the left half of K_i:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2

The permutation of the rotated D_{i-1} that produces the right half of K_i is as follows (where the bits of the rotated D_{i-1} are numbered 29, 30,...56, and bits 35, 38, 43, and 54 are discarded).

permutation to obtain the right half of K_i:

41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Each of the halves of K_i is 24 bits, so K_i is 48 bits long.

3.3.4 A DES Round

Now let's look at what a single round of DES does. Figure 3-6 shows both how encryption and decryption work.

In encryption, the 64-bit input is divided into two 32-bit halves called L_n and R_n. The round generates as output 32-bit quantities L_{n+1} and R_{n+1}. The concatenation of L_{n+1} and R_{n+1} is the 64-bit output of the round.

L_{n+1} is simply R_n. R_{n+1} is obtained as follows. First R_n and K_n are input to what we call a *mangler function*, which outputs a 32-bit quantity. That quantity is ⊕'d with L_n to obtain the new R_{n+1}. The mangler takes as input 32 bits of the data plus 48 bits of the key to produce a 32-bit output.

Given the above, suppose you want to run DES backward, i.e. to decrypt something. Suppose you know L_{n+1} and R_{n+1}. How do you get L_n and R_n?

Well, R_n is just L_{n+1}. Now you know R_n, L_{n+1}, R_{n+1} and K_n. You also know that R_{n+1} equals $L_n \oplus \text{mangler}(R_n, K_n)$. You can compute mangler(R_n, K_n), since you know R_n and K_n. Now ⊕ that with R_{n+1}. The result will be L_n. Note that the mangler is never run backwards. DES is elegantly designed to be reversible without constraining the mangler function to be reversible. This design is

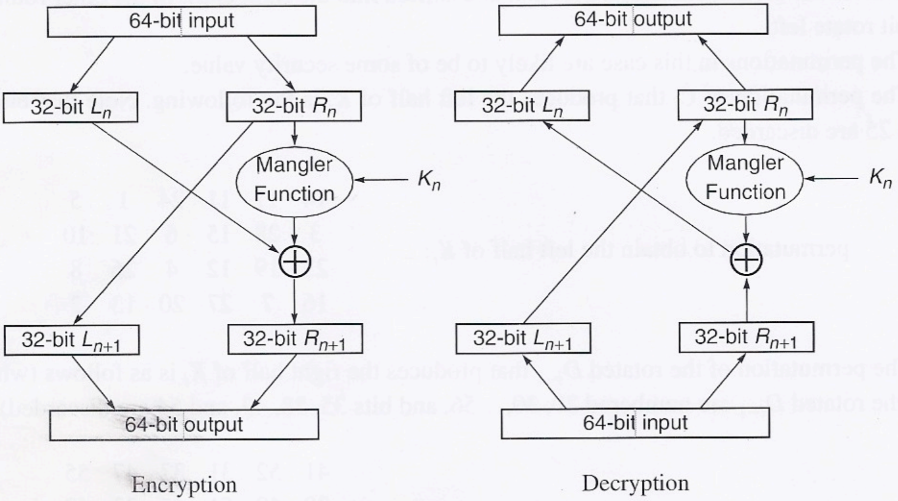


Figure 3-6. DES Round

due to Feistel [FEIS73]. Theoretically the mangler could map all values to zero, and it would still be possible to run DES backwards, but having the mangler function map all functions to zero would make DES pretty unsecure (see Homework Problem 5).

If you examine Figure 3-6 carefully, you will see that decryption is identical to encryption with the 32-bit halves swapped. In other words, feeding $R_{n+1}|L_{n+1}$ into round n produces $R_n|L_n$ as output.

3.3.5 The Mangler Function

The mangler function takes as input the 32-bit R_n, which we'll simply call R, and the 48-bit K_n, which we'll call K, and produces a 32-bit output which, when ⊕'d with L_n, produces R_{n+1} (the next R).

The mangler function first expands R from a 32-bit value to a 48-bit value. It does this by breaking R into eight 4-bit chunks and then expanding each of those chunks to 6 bits by taking the adjacent bits and concatenating them to the chunk. The leftmost and rightmost bits of *R* are considered adjacent.

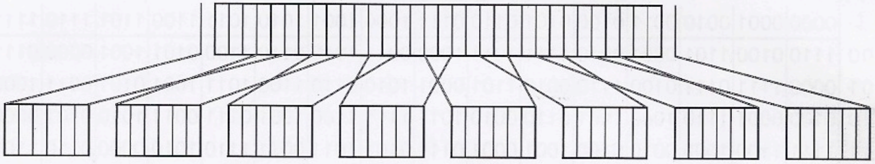


Figure 3-7. Expansion of *R* to 48 bits

The 48-bit *K* is broken into eight 6-bit chunks. Chunk *i* of the expanded *R* is $\oplus$'d with chunk *i* of *K* to yield a 6-bit output. That 6-bit output is fed into an **S-box**, a substitution which produces a 4-bit output for each possible 6-bit input. Since there are 64 possible input values (6 bits) and only 16 possible output values (4 bits), the S-box clearly maps several input values to the same output value. As it turns out, there are exactly four input values that map to each possible output value. There's even more pattern to it than that. Each S-box could be thought of as four separate 4-bit to 4-

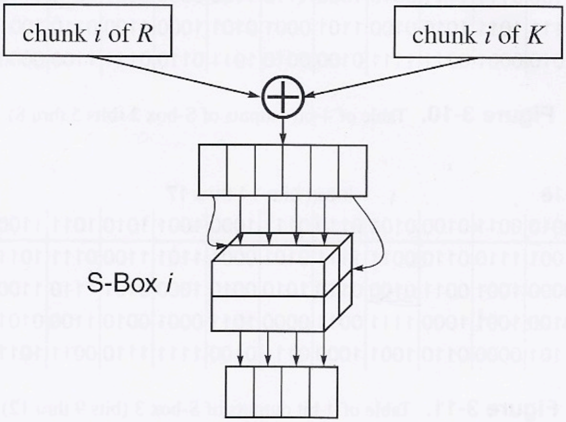


Figure 3-8. Chunk Transformation

bit S-boxes, with the inner 4 bits of the 6-bit chunk serving as input, and the outer 2 bits selecting which of the four 4-bit S-boxes to use. The S-boxes are specified as follows:

Input bits 1 and 6	Input bits 2 thru 5															
↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1110	0100	1101	0001	0010	1111	1011	1000	0011	1010	0110	1100	0101	1001	0000	0111
01	0000	1111	0111	0100	1110	0010	1101	0001	1010	0110	1100	1011	1001	0101	0011	1000
10	0100	0001	1110	1000	1101	0110	0010	1011	1111	1100	1001	0111	0011	1010	0101	0000
11	1111	1100	1000	0010	0100	1001	0001	0111	0101	1011	0011	1110	1010	0000	0110	1101

Figure 3-9. Table of 4-bit outputs of S-box 1 (bits 1 thru 4)

Input bits 7 and 12	Input bits 8 thru 11															
↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1111	0001	1000	1110	0110	1011	0011	0100	1001	0111	0010	1101	1100	0000	0101	1010
01	0011	1101	0100	0111	1111	0010	1000	1110	1100	0000	0001	1010	0110	1001	1011	0101
10	0000	1110	0111	1011	1010	0100	1101	0001	0101	1000	1100	0110	1001	0011	0010	1111
11	1101	1000	1010	0001	0011	1111	0100	0010	1011	0110	0111	1100	0000	0101	1110	1001

Figure 3-10. Table of 4-bit outputs of S-box 2 (bits 5 thru 8)

Input bits 13 and 18	Input bits 14 thru 17															
↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1010	0000	1001	1110	0110	0011	1111	0101	0001	1101	1100	0111	1011	0100	0010	1000
01	1101	0111	0000	1001	0011	0100	0110	1010	0010	1000	0101	1110	1100	1011	1111	0001
10	1101	0110	0100	1001	1000	1111	0011	0000	1011	0001	0010	1100	0101	1010	1110	0111
11	0001	1010	1101	0000	0110	1001	1000	0111	0100	1111	1110	0011	1011	0101	0010	1100

Figure 3-11. Table of 4-bit outputs of S-box 3 (bits 9 thru 12)

Input bits 19 and 24	Input bits 20 thru 23															
↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0111	1101	1110	0011	0000	0110	1001	1010	0001	0010	1000	0101	1011	1100	0100	1111
01	1101	1000	1011	0101	0110	1111	0000	0011	0100	0111	0010	1100	0001	1010	1110	1001
10	1010	0110	1001	0000	1100	1011	0111	1101	1111	0001	0011	1110	0101	0010	1000	0100
11	0011	1111	0000	0110	1010	0001	1101	1000	1001	0100	0101	1011	1100	0111	0010	1110

Figure 3-12. Table of 4-bit outputs of S-box 4 (bits 13 thru 16)

Input bits 25 and 30 Input bits 26 thru 29

↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

Figure 3-13. Table of 4-bit outputs of S-box 5 (bits 17 thru 20)

Input bits 31 and 36 Input bits 32 thru 35

↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1100	0001	1010	1111	1001	0010	0110	1000	0000	1101	0011	0100	1110	0111	0101	1011
01	1010	1111	0100	0010	0111	1100	1001	0101	0110	0001	1101	1110	0000	1011	0011	1000
10	1001	1110	1111	0101	0010	1000	1100	0011	0111	0000	0100	1010	0001	1101	1011	0110
11	0100	0011	0010	1100	1001	0101	1111	1010	1011	1110	0001	0111	0110	0000	1000	1101

Figure 3-14. Table of 4-bit outputs of S-box 6 (bits 21 thru 24)

Input bits 37 and 42 Input bits 38 thru 41

↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0100	1011	0010	1110	1111	0000	1000	1101	0011	1100	1001	0111	0101	1010	0110	0001
01	1101	0000	1011	0111	0100	1001	0001	1010	1110	0011	0101	1100	0010	1111	1000	0110
10	0001	0100	1011	1101	1100	0011	0111	1110	1010	1111	0110	1000	0000	0101	1001	0010
11	0110	1011	1101	1000	0001	0100	1010	0111	1001	0101	0000	1111	1110	0010	0011	1100

Figure 3-15. Table of 4-bit outputs of S-box 7 (bits 25 thru 28)

Input bits 43 and 48 Input bits 44 thru 47

↓	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1101	0010	1000	0100	0110	1111	1011	0001	1010	1001	0011	1110	0101	0000	1100	0111
01	0001	1111	1101	1000	1010	0011	0111	0100	1100	0101	0110	1011	0000	1110	1001	0010
10	0111	1011	0100	0001	1001	1100	1110	0010	0000	0110	1010	1101	1111	0011	0101	1000
11	0010	0001	1110	0111	0100	1010	1000	1101	1111	1100	1001	0000	0011	0101	0110	1011

Figure 3-16. Table of 4-bit outputs of S-box 8 (bits 29 thru 32)

The 4-bit output of each of the eight S-boxes is combined into a 32-bit quantity whose bits are then permuted. A permutation at this point is of security value to DES in order to ensure that the bits of the output of an S-box on one round of DES affects the input of multiple S-boxes on the next round. Without the permutation, an input bit on the left would mostly affect the output bits on the left.

The actual permutation used is very random looking (we can't find any nice patterns to make the permutation easy to visualize—it's possible a non-random looking permutation would not be as secure).

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 3-17. Permutation of the 32 bits from the S-boxes

The way to read this is that the 1^{st} bit of output of the permutation is the 16^{th} input bit, the 2^{nd} output bit is the 7^{th} input bit,...the 32^{nd} output bit is the 25^{th} input bit.

3.3.6 Weak and Semi-Weak Keys

We include this section mainly for completeness. There are sixteen DES keys that the security community warns people against using, because they have strange properties. But the probability of randomly generating one of these keys is only $16/2^{56}$, which in our opinion is nothing to worry about. It's probably equally insecure to use a key with a value less than a thousand, since an attacker might be likely to start searching for keys from the bottom.

Remember from §3.3.3 *Generating the Per-Round Keys* that the key is subjected to an initial permutation to generate two 28-bit quantities, C_0 and D_0. The sixteen suspect keys are ones for which C_0 and D_0 are one of the four values: all ones, all zeroes, alternating ones and zeroes, alternating zeroes and ones. Since there are four possible values for each half, there are sixteen possibilities in all. The four **weak keys** are the ones for which each of C_0 and D_0 are all ones or all zeroes. Weak keys are their own inverses.* The remaining twelve keys are the **semi-weak keys**. Each is the inverse of one of the others.

3.3.7 What's So Special About DES?

DES is actually quite simple, as is IDEA (which we'll explain next). One gets the impression that anyone could design a secret key encryption algorithm. Just take the bits, shuffle them, shuffle them some more, and you have an algorithm. In fact, however, these things are very mysterious. For example, the S-boxes seem totally arbitrary. Did anyone put any thought into exactly what substitutions each S-box should perform? Well, Biham and Shamir [BIHA91] have shown that with an incredibly trivial change to DES consisting of swapping S-box 3 with S-box 7, DES is about an order of magnitude less secure in the face of a specific (admittedly not very likely) attack.

*Two keys are inverses if encrypting with one is the same as decrypting with the other.

It is unfortunate that the design process for DES was not more public. We don't know if the particular details were well-chosen for strength, whether someone flipped coins, for instance, to construct the S-boxes, or even whether the particular details were well-chosen to have some sort of weakness that could only be exploited by someone involved in the design process. The claim for why the design process was kept secret, and it is a plausible claim, is that the DES designers knew about many kinds of cryptanalytic attacks, and that they specifically designed DES to be strong against all the ones they knew about. If they publicized the design process, they'd have to divulge all the cryptanalytic attacks they knew about, which would then further educate potential bad guys, which might make some cryptographic standards that were designed without this knowledge vulnerable.

In the hash algorithms designed by Ron Rivest (MD2, MD4, MD5), in order to eliminate the suspicion that they might be specifically chosen to have secret weaknesses, constants that should be reasonably random were chosen through some demonstrable manner, for instance by being the digits of an irrational number such as $\sqrt{2}$.