THIS HANDOUT IS DRAWN FROM CHAPTER 5 OF

"INTRODUCTION TO MODERN CRYPTOGRAPHY"

BY JONATHAN KATZ AND YEHUDA LINDELL

(CHAPMAN & HALL/CRC PUBLISHERS, 2008)

PLEASE USE THIS MATERIAL TO SUPPLEMENT OUR COVERAGE OF THE FOLLOWING TOPICS IN SYMMETRIC-KEY CRYPTO.

- SUBSTITUTION-PERMUTATION NETWORKS.
- FEISTEL NETWORKS
- DES & ATTACKS ON DES.

chosen-ciphertext attacks. In an asymptotic sense, this refers to attacks running in polynomial time. As mentioned just previously, though, block ciphers typically have some fixed block length and key length, and so asymptotic measures are useless. Instead, the goal is for a block cipher to be unbreakable in any reasonable amount of time. This is interpreted very strictly in the context of block ciphers, and a block cipher is generally only considered "good" if the best known attack has time complexity roughly equivalent to a brute-force search for the key. Thus, if a cipher with key length $n = 112$ can be broken in time 2^{56} (we will see such an example later), the cipher is (generally) considered insecure even though 2^{56} is still a relatively large number. Note that in an asymptotic setting, an attack of complexity $2^{n/2}$ is not feasible since it requires exponential time (and thus a cipher where such an attack is possible might still satisfy the definition of being a pseudorandom permutation). In a non-asymptotic setting, however, we must worry about the actual time complexity of the attack (rather than its asymptotic behavior). Furthermore, we are concerned that existence of such an attack may indicate some more fundamental weakness in the design of the cipher.

The Aim of this Chapter

To head off any confusion, we stress that the main aim of this chapter is to present some design principles used in the construction of modern block ciphers, with a secondary aim being to introduce the reader to the popular block ciphers DES and AES. We caution the reader that:

- It is *not* our intent to present the low-level details of DES or AES, and our description of these block ciphers should not be relied upon for implementation. To be clear: our descriptions of these ciphers are often (purposefully) inaccurate, as we omit certain technical details when they are not relevant to the broader point we are trying to emphasize.

- It is also *not* the aim of this chapter to teach how to construct secure block ciphers. On the contrary, we strongly believe that new (and proprietary) block ciphers should neither be constructed nor used since numerous excellent block ciphers are readily available.

  Those who are interested in developing expertise in constructing block ciphers are advised to start with the references at the end of this chapter.

5.1 Substitution-Permutation Networks

The main property required of a block cipher is that it should behave like a random permutation. Of course, a truly random permutation would be perfect. However, a random permutation having a *block length* (i.e., input and output length) of n bits would require $\log(2^n!) \approx n \cdot 2^n$ bits for its representation, something that is impractical for $n > 20$ and completely infeasible for $n > 50$. (Looking ahead, in practice a block length of $n \approx 64$ is already too small in some cases, and modern block ciphers thus have block lengths of $n \geq 128$.) Thus, we need to somehow construct a *concise* function that behaves like a random one.

The confusion-diffusion paradigm. In addition to his work on perfect secrecy, Shannon introduced a basic paradigm for constructing concise random-looking permutations. The basic idea is to construct a random-looking permutation F with a large block length from many smaller random (or random-looking) permutations $\{f_i\}$ having a small block length. Let us see how this works on the most basic level. Say we want F to have a block length of 128 bits. We can define F as follows: the key k for F will specify 16 *random* permutations $f_1, \ldots, f_{16}$ that each have an 8-bit block length.[3] Given an input $x \in \{0,1\}^{128}$, we parse it as 16 consecutive 8-bit blocks $x_1 \cdots x_{16}$ and then set

$$F_k(x) = f_1(x_1) \cdots f_{16}(x_{16}). \tag{5.1}$$

We say (informally) that these $\{f_i\}$ introduce *confusion* into F.

It should be immediately clear, however, that F as defined above will *not* be pseudorandom. Specifically, if x and x' differ only in their first bit then $F_k(x)$ and $F_k(x')$ will differ only in their first byte (regardless of the value of k). In contrast, if F were a truly random permutation then changing the first bit of the input would be expected to affect all bytes of the output.

For this reason, two additional changes are introduced. First, a *diffusion* step is introduced whereby the bits of the output are permuted[4] or "mixed". Second, the confusion/diffusion steps — together called a *round* — are repeated multiple times. As an example, a two-round block cipher would operate as follows: first, $x' := F_k(x)$ would be computed as in Equation (5.1). Then the bits of x' would be re-ordered to give x_1. Then $x'_1 := F_k(x_1)$ would be computed, and the bits of x'_1 would be re-ordered to give the output x_2. We remark that the functions $\{f_i\}$ as well as the permutations used in each round need not be the same. It is typical to assume that the *mixing permutations* used in each round are fixed (i.e., independent of the key), though a dependence on the key could also be introduced.

Repeated use of confusion and diffusion ensures that any small change in the input will be mixed throughout and propagated to all the bits of the output. The effect is that small changes to the input have a significant effect on the output, as one would expect of a random permutation.

[3] Since a random permutation on 8 bits can be represented using $\approx 8 \cdot 2^8$ bits, the length of the key for F is about $16 \cdot 8 \cdot 2^8$ bits, or 32 Kbits. This is much smaller than the $\approx 128 \cdot 2^{128}$ bits that would be required to specify a random permutation on 128 bits.

[4] In this context, "permuting" refers to re-ordering the bits.

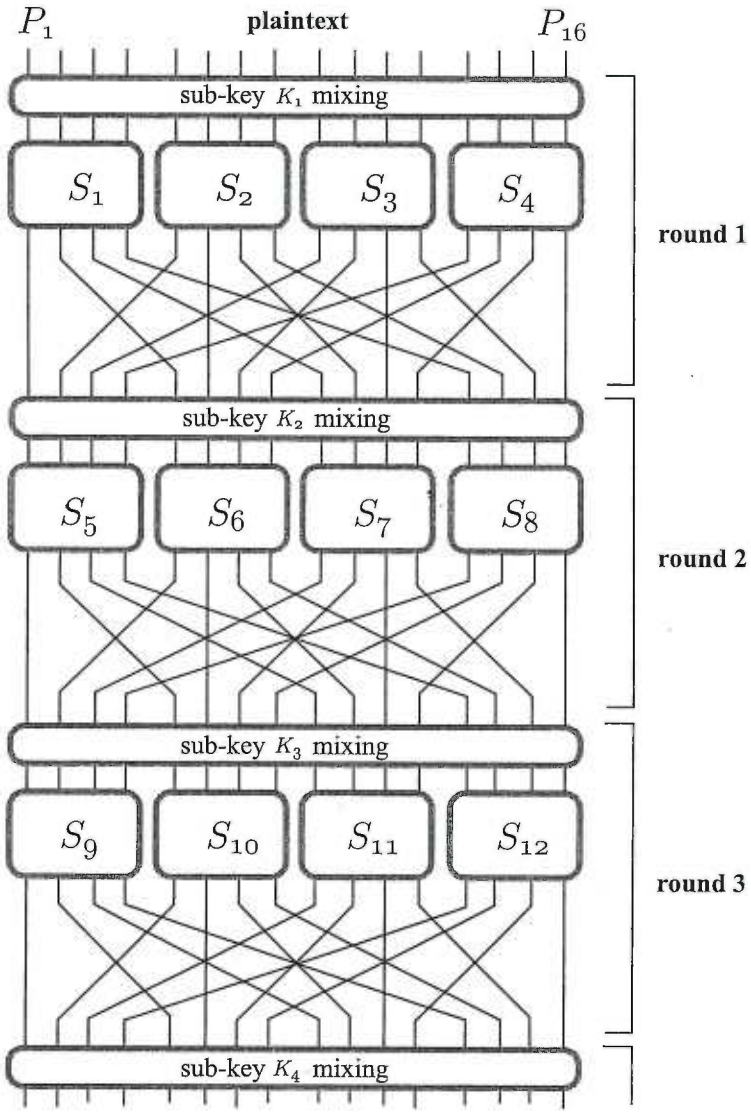

FIGURE 5.1: A substitution-permutation network.

Substitution-permutation networks. A substitution-permutation network can be viewed as a direct implementation of the confusion-diffusion paradigm. The main difference here is that we view the round functions $\{f_i\}$ as being *fixed* (rather than depending on the key k), and the key is used for a different purpose as we will shortly explain. We now refer to the $\{f_i\}$ as *S-boxes* since they act as fixed "substitution functions" (we continue to require that they be permutations).

A substitution-permutation network essentially follows the steps of the confusion-diffusion paradigm outlined earlier. However, since the S-boxes no longer depend on the key, we need to introduce dependence in some other way. (In accordance with Kerckhoffs' principle, we assume that the exact structure of the S-boxes and the mixing permutations are publicly-known, with the only secret being the key.) There are many ways this can be done, but we will focus here on the case where this is done by simply XORing some function of the key with the intermediate results that are fed as input to each round of the network. The key to the block cipher is sometimes referred to as the *master key*, and the sub-keys that are XORed with the intermediate results in each round are derived from the master key according to a *key schedule*. The key schedule is often very simple and may work by just taking subsets of the bits of the key (for example, a two-round network may use the first half of the master key in the first round and the second half of the master key in the second round), though more complex schedules can also be defined. See Figure 5.1 for the high-level structure of a substitution-permutation network, and Figure 5.2 for a closer look at a single round of such a network.

The exact choices of the S-boxes, mixing permutations, and key schedule are what ultimately determine whether a given block cipher is trivially breakable or highly secure. We will only discuss at a cursory level some basic principles behind their design. The first principle is simply a functional requirement, while the second is more specifically related to security.

Design principle 1 — invertibility of the S-boxes. In a substitution-permutation network, the S-boxes must be invertible; that is, they must be one-to-one and onto functions. The reason for this is that otherwise the block cipher will not be a permutation. To see that making the S-boxes one-to-one and onto suffices, we show that when this holds it is possible to fully determine the input given the output and the key. Specifically, we show that every round can be inverted (implying that the entire network can be inverted by working from the end back to the beginning). Recall that a round now consists of three stages: XORing the sub-key with the output of the previous round, passing the result through the S-boxes (as in Equation (5.1)), and finally re-ordering the bits of this result using a mixing permutation. The mixing permutation can easily be inverted since it is just a re-ordering of bits. If the S-boxes are one-to-one and onto, these too can be inverted. The result can then be XORed with the appropriate sub-key to obtain the original input. We therefore have:

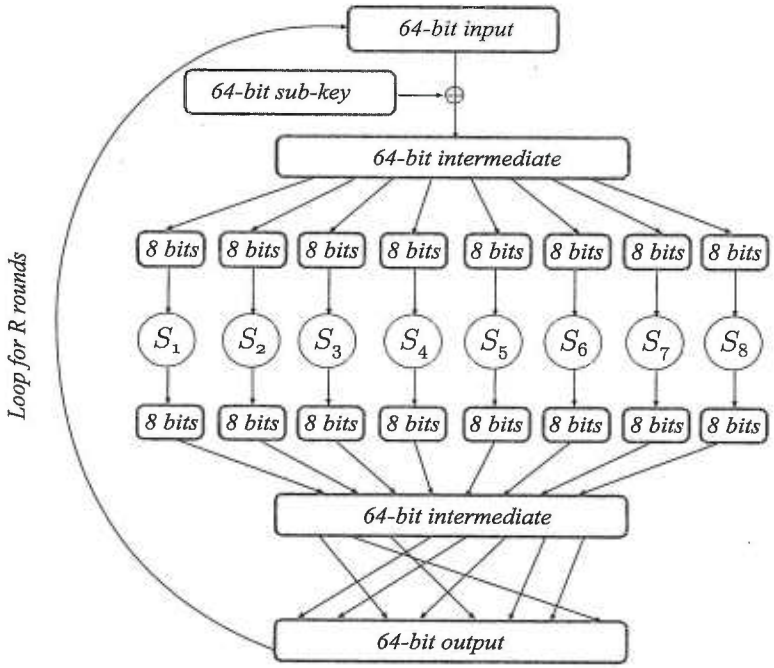

FIGURE 5.2: A single round of a substitution-permutation network.

PROPOSITION 5.1 *Let F be a keyed function defined by a substitution-permutation network in which the S-boxes are all one-to-one and onto. Then regardless of the key schedule and the number of rounds, F_k is a permutation for any choice of k.*

Design principle 2 — the avalanche effect. An important property in any block cipher is that small changes to the input must result in large changes to the output. Otherwise, the outputs of the block cipher on two similar inputs will not look independent (whereas in a random permutation, the outputs of any two unequal inputs are independently distributed). To ensure that this is the case, block ciphers are designed to exhibit the *avalanche effect*, meaning that changing a single bit of the input affects every bit of the output. (This does not mean that changing one bit of the input *changes* every bit of the output, only that it has some effect on every bit of the output. Note that even for a completely random function, changing one bit of the input is expected to change only half the bits of the output, on average.)

It is easy to demonstrate that the avalanche effect holds in a substitution-permutation network provided that the following two properties hold (and sufficiently-many rounds are used):

1. *The S-boxes are designed so that changing a single bit of the input to an S-box changes at least* two bits *in the output of the S-box.*
2. *The mixing permutations are designed so that the output bits of any given S-box are spread into* different *S-boxes in the next round.*

To see how this yields the avalanche effect, assume that the S-boxes are all such that changing a single bit of the input of the S-box results in a change in exactly two bits of the output of the S-box, and that the mixing permutations are chosen as required above. For concreteness, assume the S-boxes have input/output size of 4 bits, and that the block length of the cipher is 128 bits. Consider now what happens when the block cipher is applied to two inputs that differ by only a single bit:

1. After the first round, the intermediate values differ in exactly two bit-positions. This is because XORing the current sub-key maintains the 1-bit difference in the intermediate values, and so the inputs to all the S-boxes except one are identical. In the one S-box where the inputs differ, the output of the S-box causes a 2-bit difference. The mixing permutation applied to the results changes the positions of these differences, but maintains a 2-bit difference.

2. By the second property mentioned earlier, the mixing permutation applied at the end of the first round spreads the two bit-positions where the intermediate results differ into two *different* S-boxes in the second round. (This remains true even after the appropriate sub-key is XORed with the result of the previous round.) So, in the second round there are now *two* S-boxes that receive inputs differing by a single bit. Following the same argument as before, we see that at the end of the second round the intermediate values differ in 4 bits.

3. Continuing with the same argument, we expect 8 bits of the intermediate value to be affected after the 3rd round, 16 bits to be affected after the 4th round, and all 128 bits of the output to be affected at the end of the 7th round.

The last point is not quite precise and it is certainly possible that (depending on the exact inputs, as well as the exact choice of the S-boxes and mixing permutations) there will be fewer differences than expected at the end of some round. For this reason, it is typical to use more than 7 rounds. The importance of the last point is that it gives a *lower bound* on the number of rounds: if fewer than 7 rounds are used then there must be some set of output bits that are not affected, implying that it will be possible to distinguish the cipher from a random permutation.

One might expect that the "best" way to design S-boxes would be to choose them at random (subject to the restriction that they should be one-to-one

and onto). Interestingly, this turns out not to be the case,[5] at least if we want to satisfy the above criterion. For example, consider the case of an S-box operating on 4-bit inputs and let x and x' be two different inputs. Let $y = S(x)$, and now consider choosing $y' \neq y$ at random as the value of $S(x')$. There are 4 strings that differ from y in only 1 bit, and so with probability $4/15 > 1/4$ we will choose y' that does *not* differ from y in two or more bits. The problem is compounded when we consider all inputs, and becomes even worse when we consider that multiple S-boxes are needed. We conclude based on this example that, as a general rule, it is best to carefully design S-boxes with certain desired properties (in addition to the one discussed above) rather than choosing them blindly at random.

In addition to the above, we remark also that randomly-chosen S-boxes are not the best for defending against attacks like the ones we will show in Section 5.6.

Security of Substitution-Permutation Networks

Experience, along with many years of cryptanalytic effort, indicate that substitution-permutation networks are a good choice for constructing pseudorandom permutations as long as great care is taken in the choice of the S-boxes, the mixing permutations, and the key schedule. The Advanced Encryption Standard (AES), described in Section 5.5, is similar in structure to the substitution-permutation network described above, and is widely believed to be a very strong pseudorandom permutation.

It is important to understand, however, that the strength of a cipher constructed in this way depends heavily on the number of rounds used. In order to obtain more of an insight into substitution-permutation networks, we will demonstrate attacks on block ciphers of this type that have very few rounds. These attacks are straightforward, but are worth seeing as they demonstrate conclusively why a large number of rounds are needed.

Attacks on reduced-round substitution-permutation networks. According to the definition of a pseudorandom permutation (see Definition 3.23), the adversary is given an oracle that is either a random permutation or the given block cipher (with a randomly-chosen key). The aim of the adversary is to determine which is the case. Clearly, if an adversary can obtain the secret key of the block cipher, then it can distinguish it from a random permutation. Such an attack is called a *complete break* because once the secret key is learned, no security remains.

[5] The situation here is different from our earlier discussion of the confusion-diffusion paradigm. There, the round functions/S-boxes *depended on the key* and were therefore unknown to the adversary. Here, the round functions are *fixed* and publicly-known, and the question is what fixed functions are best.

Attack on a single-round substitution-permutation network: Let F be a single-round substitution-permutation network. We demonstrate an attack where the adversary is given only a *single* input/output pair (x, y) for a randomly-chosen input value x, and easily learns the secret key k for which $y = F_k(x)$. The adversary begins with the output value y and then inverts the mixing permutation and the S-boxes. It can do this because the specification of the permutation and the S-boxes is public. The intermediate value that the adversary computes from these inversions is exactly $x \oplus k$ (assuming, without loss of generality, that the master key is used as the sub-key in the only round of the network). Since the adversary also has the input x, it immediately derives the secret key k. This is therefore a complete break.

Attack on a two-round substitution-permutation network: We again show an attack that recovers the secret key, though the attack now takes more time. Consider the following concrete parameters. Let the block length of the cipher be 64 bits, and let each S-box have a 4-bit input/output length. Furthermore, let the key k be of length 128 bits where the first half $k^a \in \{0,1\}^{64}$ of the key is used in the first round and the second half $k^b \in \{0,1\}^{64}$ is used in the second round. We use independent keys here to simplify the description of the attack below, but this only makes the attack more difficult.

Say the adversary is given an input x and the output $y = F_k(x)$ of the cipher. The adversary begins by "working backward", inverting the mixing permutation and S-boxes in the second round of the cipher (as in the previous attack). Denote by w_1 the first 4 bits of the result. Letting α_1 denote the first 4 bits of the output of the first round, we have that $w_1 = \alpha_1 \oplus k_1^b$, where k_1^b denotes the first 4 bits of k^b. (The adversary does not know α_1 or k_1^b.) The important observation here is that when "working forward" starting with the input x, the value of α_1 is influenced by at most 4 different S-boxes (because, in the worst case, each bit of α_1 comes from a different S-box in the first round). Furthermore, since the mixing permutation of the first round is known, the adversary knows exactly which of the S-boxes influence α_1. This, in turn, means that at most 16 bits of the key k^a (in known positions) influence the computation of these four S-boxes. It follows that the adversary can guess the appropriate 16 bits of k^a and the 4-bit value k_1^b, and then *verify* possible correctness of this guess using the known input/output pair (x, y). This verification is carried out by XORing the relevant 16 bits of the input x with the relevant 16 bits of k^a, computing the resulting α_1, and then comparing w_1 to $\alpha_1 \oplus k_1^b$ (where k_1^b is also part of the guess). If equality is not obtained, then the guess is certainly incorrect. If equality is obtained, then the guess *may* be correct. Proceeding in this way, the adversary can exhaustively find all values of these 20 bits of the key that are consistent with the given (x, y). This takes time 2^{20} to try each possibility.

If we make the simplifying assumption that an incorrect guess (i.e., one which does not correspond to the bits of the key k that is actually being used) yields a random value for $\alpha_1 \oplus k_1^b$, then we expect an incorrect guess to

pass the above verification test with probability $1/2^{|w_1|} = 1/16$. This means that we expect roughly $2^{20}/16 = 2^{16}$ possibilities to pass the test (including the correct possibility). If we now repeat the above using an additional input/output pair (x', y'), we expect to again eliminate roughly 15/16 of the incorrect possibilities, and we see that if we carry this out repeatedly using many different input/output pairs, we expect to be left with only one guess of the 20 bits of the key that is consistent with all the given input/output pairs. This will, of course, have to be correct.

For concreteness, assume that 8 input/output pairs are used to narrow down to a single possibility. Then the adversary learns the 4 bits of k_1^b in time $8 \cdot 2^{20} = 2^{23}$. This can be repeated for all 16 portions of k^b, leading to an attack with total complexity of $16 \cdot 2^{23} = 2^{27}$ for learning the 64-bit value k^b. Along the way the adversary also learns all 64 bits of k^a. This in fact over-estimates the time complexity of the attack since certain portions of k^a will be re-used, and previously-determined portions of k^a do not need to be guessed again. In any case, an attack having time complexity 2^{27} is well within practical reach, and is much less than the 2^{128} complexity that "should" be required to perform an exhaustive search for a 128-bit key.

There is an important lesson to be learned from the above. The attack is possible since different parts of the key can be isolated from other parts (it is much quicker to carry out 16 attacks of time 2^{23} that reveal 4 bits of k^b each time, than a single attack of time $2^{16 \cdot 4} = 2^{64}$). Thus, further *diffusion* is needed to make sure that all the bits of the key affect all of the bits of the output. Two rounds are not enough for this to take place.

Attack on a three-round substitution-permutation network: In this case we present a weaker attack; instead of learning the key, we just show that it is easy to distinguish a three-round substitution-permutation network from a pseudorandom permutation. This attack is based on the observation, mentioned earlier, that the avalanche effect is not complete after only three rounds (of course, this depends on the block length of the cipher and the input/output length of the S-boxes, but with reasonable parameters this will be the case). Thus, the adversary just needs to ask for the function to be computed on two strings that differ on only a single bit. A three-round block cipher will have the property that many bits of the output will be the same in each case, making it easy to distinguish from a random permutation.

5.2 Feistel Networks

A Feistel network is an alternative approach for constructing a block cipher. The low-level building blocks (S-boxes, mixing permutations, and a key schedule) are the same: the difference is in the high-level design. The

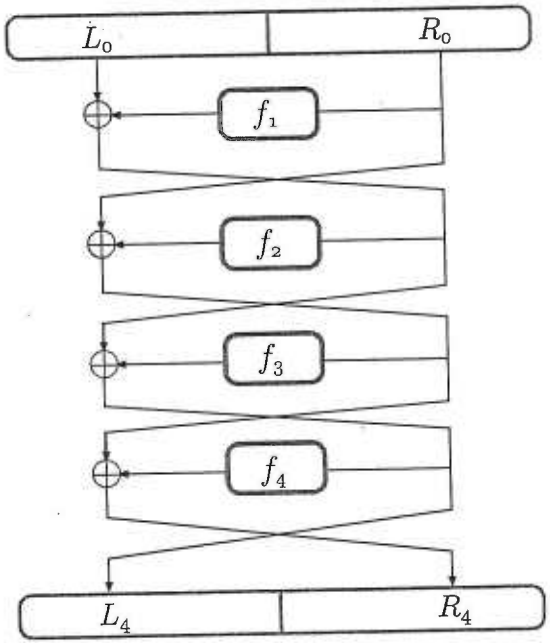

FIGURE 5.3: A 4-round Feistel network.

advantage of Feistel networks over substitution-permutation networks is that a Feistel network eliminates the requirement that S-boxes be invertible. This is important because a good block cipher should have "unstructured" behavior (so that it looks random); however, requiring that all the components of the construction be invertible inherently introduces structure. *A Feistel network is thus a way of constructing an invertible function from non-invertible components.* This seems like a contradiction in terms — if you cannot invert the components, it seems impossible to invert the overall structure — but the Feistel design ingeniously achieves this.

A Feistel network, as in the case of a substitution-permutation network, operates in a series of rounds. In each round, a *round function* is applied in a specific manner that will be described below. In a Feistel network, round functions need not be invertible. Round functions typically contain components like S-boxes and mixing permutations, but a Feistel network can deal with *any* round functions irrespective of their design. When the round functions *are* constructed from S-boxes, the designer has more freedom since the S-boxes need not be invertible.

The ith round of a Feistel network operates as follows. The input to the round is divided into two halves denoted L_{i-1} and R_{i-1} (with L and R denoting the "left half" and "right half" of the input, respectively). If the block length of the cipher is n bits, then L_{i-1} and R_{i-1} each have length $n/2$, and the ith round function f_i will take an $n/2$-bit input and produce an $n/2$-bit

output. (We stress again that although the input and output lengths of f_i are the same, it is not necessarily one-to-one and onto.) The output (L_i, R_i) of the round, where L_i and R_i again denote the left and right halves, is given by

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1}). \tag{5.2}$$

In a t-round Feistel network, the n-bit input to the network is parsed as (L_0, R_0), and the output is the n-bit value (L_t, R_t) obtained after applying all t rounds. A 4-round Feistel network is shown in Figure 5.3.

We have not yet discussed how dependence on the key is introduced. As in the case of substitution-permutation networks, the master key k is used to derive sub-keys that are used in each round; the ith round function f_i depends on the ith sub-key, denoted k_i. Formally, the design of a Feistel network specifies a publicly-known *mangler function* $\hat{f}_i$ associated with each round i. This function $\hat{f}_i$ takes as input a sub-key k_i and an $n/2$-bit string and outputs an $n/2$-bit string. When the master key is fixed — thereby fixing each sub-key k_i — the ith round function f_i is defined via $f_i(R) \stackrel{\text{def}}{=} \hat{f}_i(k_i, R)$.

Inverting a Feistel network. A Feistel network is invertible *regardless of the round functions* $\{f_i\}$ (and thus regardless of the mangler functions $\{\hat{f}_i\}$). To show this we need only show that any given round of the network can be inverted. Given the output (L_i, R_i) of the ith round, we can compute (L_{i-1}, R_{i-1}) as follows: first set $R_{i-1} := L_i$. Then compute

$$L_{i-1} := R_i \oplus f_i(R_{i-1}).$$

(The function f_i can be derived from $\hat{f}_i$ if the master key is known.) It can be verified that this gives the correct value (L_{i-1}, R_{i-1}) that was the input of this round (i.e., it computes the inverse of Equation (5.2)). Notice that f_i is evaluated only in the forward direction, as required.

We thus have:

PROPOSITION 5.2 *Let F be a keyed function defined by a Feistel network. Then regardless of the mangler functions $\{\hat{f}_i\}$ and the number of rounds, F_k is a permutation for any choice of k.*

As in the case of substitution-permutation networks, attacks on Feistel networks are possible when the number of rounds is too low. We will see such attacks when we discuss DES in the next section. Theoretical results concerning the security of Feistel networks are discussed in Section 6.6.

5.3 DES – The Data Encryption Standard

The Data Encryption Standard, or DES, was developed in the 1970s at IBM (with help from the National Security Agency), and adopted in 1977 as a Federal Information Processing Standard (FIPS) for the US. In its basic form, DES is no longer considered secure due to its short key length of 56 bits. Nevertheless, it remains in wide use today in its strengthened form of triple-DES (as described in Section 5.4).

DES is of great historical significance, and has undergone intensive scrutiny within the cryptographic community, arguably more than any other cryptographic algorithm in history. The common consensus is that, relative to its key length, DES is extremely secure. Indeed, even after so many years, the best known attack on DES *in practice* is a brute-force search over all 2^{56} possible keys. As we will see later, there are important theoretical attacks on DES that require less computation than such a brute force attack; however, these attacks assume certain conditions that seem unlikely to hold in practice.

In this section, we provide a high-level overview of the main components of DES. We stress that we will not provide a full specification that is correct in every detail, and some parts of the design will be omitted from our description. Our aim is to present the basic ideas underlying the construction of DES, and not all the low-level details; the reader interested in such details can consult the references at the end of this chapter.

5.3.1 The Design of DES

The DES block cipher is a 16-round Feistel network with a block length of 64 bits and a key length of 56 bits. Recall that in a Feistel network the internal f-functions that are used in each round operate on half a block at a time. Thus, the input/output length of a DES round function is 32 bits. The round functions used in each of the 16 rounds of DES are all derived from the same mangler function $\hat{f}_i = \hat{f}$. The *key schedule* of DES is used to derive a 48-bit sub-key k_i for each round from the 56-bit master key k. As discussed in the previous section, the ith round function f_i is then defined as $f_i(R) \stackrel{\text{def}}{=} \hat{f}(k_i, R)$. As is to be expected from the fact that DES uses a Feistel structure, the round functions are *non-invertible*.

The key schedule of DES is relatively simple, with each sub-key k_i being a permuted subset of 48 bits from the master key. We will not describe the key schedule exactly. It suffices for us to note that the 56 bits of the master key are divided into two halves — a "left half" and a "right half" — each containing 28 bits (actually, this division occurs after an initial permutation is applied to the key, but we ignore this in our description). In each round, the left-most 24 bits of the sub-key are taken as some subset of the 28 bits in the left half of the master key, and the right-most 24 bits of the sub-key

are taken as some subset of the 28 bits in the right half of the master key. We stress that the entire key schedule (including the manner in which bits are divided into the left and right halves, and which bits are used in forming sub-key k_i) is fixed and public, and the only secret is the master key itself.

The DES mangler function $\hat{f}$. Recall that the mangler function $\hat{f}$ and the ith sub-key k_i jointly determine the ith round function f_i. The mangler function in DES is constructed using a paradigm we have previously analyzed: it is (essentially) just a 1-round substitution-permutation network! In more detail, computation of $\hat{f}(k_i, R)$ with $k_i \in \{0,1\}^{48}$ and $R \in \{0,1\}^{32}$ proceeds as follows: first, R is *expanded* to a 48-bit value R'. This is done by simply duplicating half the bits of R; we denote this by $R' := E(R)$ where E represents the *expansion function*. Following this step, computation proceeds exactly as in our earlier discussion of substitution-permutation networks: The expanded value R' is XORed with k_i, and the resulting value is divided into 8 blocks, each of which is 6 bits long. Each block is passed through a (different) S-box that takes a 6-bit input and yields a 4-bit output; concatenating the output from the 8 S-boxes gives a 32-bit result. As the final step, a mixing permutation is applied to the bits of this result to obtain the final output of $\hat{f}$. See Figure 5.4 for a diagram of the construction.

One difference here (as compared to our original discussion of substitution-permutation networks) is that the S-boxes referred to above are *not* invertible; indeed, they cannot possibly be invertible since their inputs are longer than their outputs. Further discussion regarding the structural details of the S-boxes is given below.

We stress once again that everything in the above description (including the S-boxes themselves as well as the mixing permutation) is *publicly-known*. The only secret is the master key which is used to derive all the sub-keys.

The S-boxes. The eight S-boxes that form the "core" of $\hat{f}$ are a crucial element of the DES construction, and were very carefully designed (reportedly, with the help of the National Security Agency). Studies of DES have shown that if small changes to the S-boxes had been introduced, or if the S-boxes had been chosen at random, DES would have been much more vulnerable to attack. This should serve as a warning to anyone who wishes to design a block cipher: seemingly arbitrary choices are not arbitrary at all, and if not made correctly may render the entire construction insecure.

Recall that each S-box maps 6-bit strings to 4-bit strings. Each S-box can be viewed as a table with 4 rows and 16 columns, where each cell of the table contains a 4-bit entry. A 6-bit input can be viewed as indexing one of the $2^6 = 64$ cells of the table in the following way: The first and last input bits are used to choose the table row, and bits 2–5 are used to choose the table column. The 4-bit entry at a particular cell represents the output value for the input associated with that position.

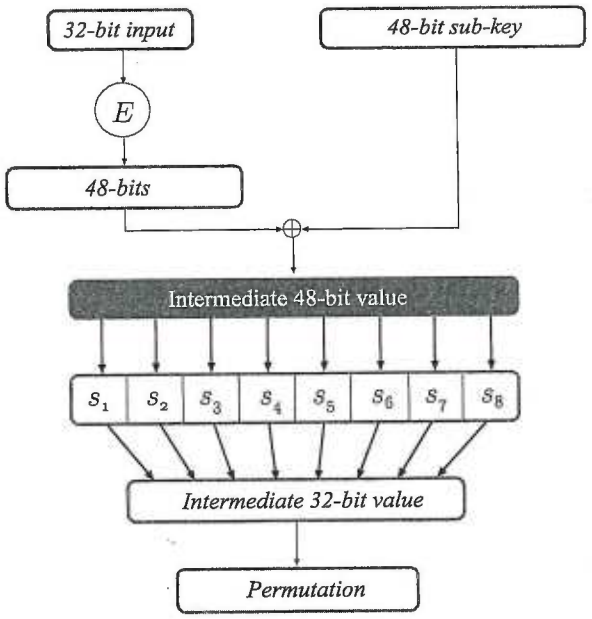

FIGURE 5.4: The DES mangler function.

Due to their importance, we will describe some basic properties of the DES S-boxes:

1. Each S-box is a 4-to-1 function. (That is, exactly 4 inputs are mapped to each possible output.) This follows from the properties below.
2. Each row in the table contains each of the 16 possible 4-bit strings exactly once. (That is, each row is a *permutation* of the 16 possible 4-bit strings.)
3. Changing *one bit* of the input always changes at least *two bits* of the output.

We will use the above properties in our analysis of reduced-round DES below.

The DES avalanche effect. As discussed earlier, the avalanche effect is a crucial property of any secure block cipher. The third property of the DES S-boxes described above, along with the mixing permutation that is used in the mangler function, ensure that DES exhibits a strong avalanche effect. In order to see this, we will trace the difference between the intermediate values in a DES computation of two inputs that differ by just a single bit. Let us denote the two inputs by (L_0, R_0) and (L'_0, R'_0), where we assume that $R_0 = R'_0$ and so the single-bit difference occurs in the left half of the inputs (it may help to refer to Equation (5.2) in what follows). After the first round the intermediate values (L_1, R_1) and (L'_1, R'_1) still differ by only a single bit, though now this

difference is in the right half. In the second round of DES, the right half of each input is run through $\hat{f}$. Assuming that the bit where R_1 and R'_1 differ is not duplicated in the expansion step, the intermediate values before applying the S-boxes still differ by only a single bit. By property 3, the intermediate values *after* the S-box computation differ in at least *two* bits. The result is that the intermediate values (L_2, R_2) and (L'_2, R'_2) differ in *three* bits: there is a 1-bit difference between L_2 and L'_2 (carried over from the difference between R_1 and R'_1) and a 2-bit difference between R_2 and R'_2.

The mixing permutation spreads the two-bit difference between R_2 and R'_2 into different regions of these strings. The effect is that, in the following round, each of the two different bits is used as input for a *different* S-box, resulting in a difference of 4 bits in the right halves of the intermediate values. (There is also now a 2-bit difference in the left halves). As with a substitution-permutation network, we have an exponential effect and so after 7 rounds we expect all 32 bits in the right half to be affected (and after 8 rounds all 32 bits in the left half will be affected as well).

DES has 16 rounds, and so the avalanche effect is completed very early in the computation. This ensures that the computation of DES on similar inputs yields completely different and independent-looking outputs. We remark that the avalanche effect in DES is also due to a careful choice of the mixing permutation, and in fact it has been shown that a *random* mixing permutation would yield a far weaker avalanche effect.

5.3.2 Attacks on Reduced-Round Variants of DES

A useful exercise for understanding more about the DES construction and its security is to look at the behavior of DES with only a few rounds. We will show attacks on one-, two-, and three-round variants of DES (recall that the real DES has 16 rounds). Clearly DES with three rounds or fewer cannot be a pseudorandom function because the avalanche effect is not yet complete after only three rounds. Thus, we will be interested in demonstrating more difficult (and more damaging) *key-recovery attacks* which compute the key k using only a relatively small number of input/output pairs computed using that key. Some of the attacks are similar to those we have seen in the context of substitution-permutation networks; here, however, we will see how they are applied to a concrete block cipher rather than to an abstract design.

All the attacks below will be known-plaintext attacks whereby the adversary has plaintext/ciphertext pairs $\{(x_i, y_i)\}$ with $y_i = DES_k(x_i)$ for some secret key k. When we describe the attacks, we will focus on a particular input/output pair (x, y) and will describe the information about the key that an adversary can derive from this pair. Continuing to use the notation developed earlier, we denote the left and right halves of the input x as L_0 and R_0, respectively, and let L_i, R_i denote the left and right halves after the ith round. We continue to let E denote the DES expansion function, f_i denote the round function applied in round i, and k_i denote the sub-key used in round i.

Single-round DES. In single-round DES, we have that $y = (L_1, R_1)$ where $L_1 = R_0$ and $R_1 = L_0 \oplus f_1(R_0)$. We therefore know an input/output pair for f_1; specifically, we know that $f_1(R_0) = R_1 \oplus L_0$ (note that all these values are known). By applying the inverse of the mixing permutation to the output $R_1 \oplus L_0$, we obtain the intermediate value that contains the outputs from all the S-boxes, where the first 4 bits are the output from the first S-box, the next 4 bits are the output from the second S-box, and so on. This means that we have the exact output of each S-box.

Consider the (known) 4-bit output of the first S-box. Recalling that each S-box is a 4-to-1 function, this means that there are exactly four possible inputs to this S-box that would result in the given output, and similarly for all the other S-boxes; each such input is 6 bits long. The input to the S-boxes is simply the XOR of $E(R_0)$ with the key k_1 used in this round. (Actually, for single-round DES, k_1 is the only key.) Since R_0 is known, we conclude that for each 6-bit portion of k_1 there are four possible values (and we can compute them). This means we have reduced the number of possible keys k_1 from 2^{48} to $4^{48/6} = 4^8 = 2^{16}$ (since there are four possibilities for each of the eight 6-bit portions of k_1). This is already a small number and so we can just try all the possibilities on a different input/output pair (x', y') to find the right one. We thus obtain the full key using only two known plaintexts in time roughly 2^{16}.

Two-round DES. In two-round DES, the output y is equal to (L_2, R_2) where

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus f_1(R_0) \\ L_2 &= R_1 = L_0 \oplus f_1(R_0) \\ R_2 &= L_1 \oplus f_2(R_1). \end{aligned}$$

Note that L_0, R_0, L_2, R_2 are known from the given input/output pair (x, y), and thus we also know $L_1 = R_0$ and $R_1 = L_2$. This means that we know the input/output of both f_1 and f_2, and so the same method used in the attack on single-round DES can be used here to determine both k_1 and k_2 in time roughly $2 \cdot 2^{16}$. This attack works even if k_1 and k_2 are completely independent keys. In fact, the key schedule of DES ensures that many of the bits of k_1 and k_2 are equal, which can be used to further speed up the attack.

Three-round DES. See Figure 5.5 for a diagram of three-round DES. The output value y is equal to (L_3, R_3). Since $L_1 = R_0$ and $R_2 = L_3$, the only unknown values in the figure are R_1 and L_2 (which are equal).

Now we no longer have the input/output to any round function f_i. For example, consider f_2. In this case, the output value is equal to $L_1 \oplus R_2$ where both of these values are known. However, we do *not* know the value R_1 that is input to f_2. We do know that $R_1 = L_0 \oplus f_1(R_0)$, and that $R_1 = R_3 \oplus f_3(R_2)$, but the outputs of f_1 and f_3 are unknown. A similar exercise shows that for

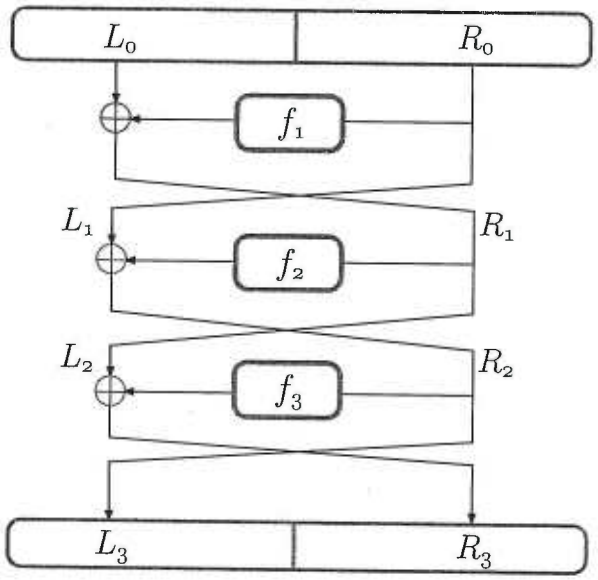

FIGURE 5.5: 3-round DES.

f_1 and f_3 we can determine the inputs but not the outputs. Thus, the attack we used to break one-round and two-round DES will not work here.

Instead of relying on full knowledge of the input and output of one of the round functions, we will use knowledge of a certain relation between the inputs and outputs of f_1 and f_3. Observe that the output of f_1 is equal to $L_0 \oplus R_1 = L_0 \oplus L_2$. The output of f_3 is equal to $L_2 \oplus R_3$. This means that the XOR of the output of f_1 with the output of f_3 is equal to

$$(L_0 \oplus L_2) \oplus (L_2 \oplus R_3) = L_0 \oplus R_3,$$

which is known. That is, *the XOR of the outputs of* f_1 *and* f_3 is known. Furthermore, the input to f_1 is R_0 and the input to f_3 is L_3, both of which are known. We conclude that we can determine the inputs to f_1 and f_3, and the XOR of their outputs. We now describe an attack that finds the secret key based on this information.

Recall that the key schedule of DES has the property that the master key is divided into a "left half", which we denote by k_L, and a "right half" k_R, each containing 28 bits. Furthermore, the left-most bits of the sub-key used in each round are taken only from k_L and the right-most bits of each sub-key are taken only from k_R. This means that the left half of the master key affects the inputs only to the first four S-boxes in any round, while the right half of the master key affects the inputs only to the last four S-boxes. Since the mixing permutation is known, we also know which bits of the output of each round function come out of each S-box.

The idea behind the attack is to separately traverse the key-space for each half of the master key, giving an attack with complexity (roughly) $2 \cdot 2^{28}$ rather than complexity 2^{56}. Such an attack will be possible if we can verify a guess of half the master key, and we now show how this can be done. Let k_L be a guess for the left half of the master key. We know the input R_0 of f_1, and so using our guess of k_L we can compute the input to the first four S-boxes. This means that we can compute half the output bits of f_1 (the mixing permutation spreads out the bits we know, but since the mixing permutation is known we know exactly which bits these are). Likewise, we can compute the same locations in the output of f_3 by using the known input L_3 to f_3 and the same guess k_L. Finally, we can compute the XOR of these output values and check whether they match the appropriate bits in the known value of the XOR of the outputs of f_1 and f_3. If they are not equal, then our guess k_L is incorrect. The correct half-key k_L will always pass this test, and an incorrect half-key is expected to pass this test only with probability roughly 2^{-16} (since we check equality of 16 bits in two computed values). There are 2^{28} possible half-keys to try and so we expect to be left with $2^{28}/2^{16} = 2^{12}$ possibilities for k_L after the above.

By performing the above for each half of the master key, we obtain in time $2 \cdot 2^{28}$ approximately 2^{12} candidates for the left half and 2^{12} candidates for the right half. Since each combination of the left half and right half is possible, we have 2^{24} candidate keys overall and can run a brute-force search over this set using an additional input/output pair (x', y'). The time complexity of the attack is roughly $2 \cdot 2^{28} + 2^{24} < 2^{30}$, and its space complexity is $2 \cdot 2^{12}$. An attack of this complexity could be carried out on a standard personal computer.

5.3.3 The Security of DES

After almost 30 years of intensive study, the best known practical attack on DES is still just an exhaustive search through its key space. (We discuss some important theoretical attacks below. These attacks require a large number of input/output pairs which would be difficult to obtain in an attack on any real-world system using DES.) Unfortunately, the 56-bit key length of DES is short enough that an exhaustive search through all 2^{56} possible keys is now feasible (though still non-trivial). Already in the late '70s there were strong objections to the choice of such a short key for DES. Back then, the objection was theoretical as the computational power needed to search through that many keys was generally unavailable.[6] The practicality of a brute force attack on DES nowadays, however, was demonstrated in 1997 when a number of DES challenges set up by RSA Security were solved (these

[6] In 1977, it was estimated that a computer that could crack DES in one day would cost $20 million to build.

challenges were in the form of input/output pairs and a reward was given to the first person or organization to find the secret key that was used). The first challenge was broken in 1997 by the DESCHALL project using thousands of computers coordinated across the Internet; the computation took 96 days. A second challenge was broken the following year in just 41 days by the `distributed.net` project. A significant breakthrough came later in 1998 when the third challenge was solved in just *56 hours*. This impressive feat was achieved via a special-purpose DES-breaking machine called *Deep Crack* that was built by the Electronic Frontier Foundation at a cost of $250,000. The latest challenge was solved in just over 22 hours (as a combined effort of Deep Crack and distributed.net). The bottom line is that DES has a key that is far too short and cannot be considered secure for any application today.

Less important than the short key length of DES, but still a concern, is its relatively short *block length*. The reason that a small block length is problematic is that the security of many constructions based on block ciphers depends on the block length (*even if the cipher used is "perfect"* and thus regardless of the key length). For example, the proof of security for counter mode encryption (cf. Theorem 3.29) shows that even when a completely random function is used an attacker can break the security of this encryption scheme with probability $\Theta(q^2/2^n)$ if it obtains q plaintext/ciphertext pairs, where n here represents the block length. In the case of DES where $n = 64$, this means that if an attacker obtains only $q = 2^{27}$ plaintext/ciphertext pairs, security is compromised with high probability. Obtaining plaintext/ciphertext pairs is relatively easy if an adversary eavesdrops on the encryption of messages containing known headers, redundancies, etc. (though obtaining 2^{27} such pairs may be out of reach).

We stress that the insecurity of DES has nothing to do with its internal structure and design, but rather is due only to its short key length (and, to a lesser extent, its short block length). This is a great tribute to the designers of DES who seem to have succeeded in constructing an almost "perfect" block cipher (with the glaring exception of its too-short key[7]). Since DES itself seems not to have significant structural weaknesses, it makes sense to use DES as a building block in order to construct a block cipher with a longer key. We discuss such an approach in Section 5.4.

Looking ahead a bit, we note that the Advanced Encryption Standard (AES) — the replacement for DES — was explicitly designed to address concerns regarding the short key length and block length of DES. AES supports keys of length 128 bits (and more), and a block length of 128 bits.

[7] Actually, the designers of DES almost certainly recognized that the key was too short; it has been suggested that the NSA requested that the key be short enough for them to crack.

Advanced Cryptanalytic Attacks on DES

The successful brute-force attacks described above do not utilize any internal weaknesses of DES. Indeed, for many years no such weaknesses were known to exist. The first breakthrough on this front was by Biham and Shamir in the late '80s who developed a technique called *differential cryptanalysis* and used it to design an attack on DES using less time than a brute-force search. Their specific attack takes time 2^{37} (and uses negligible memory) but requires the attacker to analyze 2^{36} ciphertexts obtained from a pool of 2^{47} chosen plaintexts. While the existence of this attack was a breakthrough result from a theoretical standpoint, it does not appear to be of much practical concern since it is hard to imagine any realistic scenario where an adversary can obtain this many values in a chosen-plaintext attack.

Interestingly, the work of Biham and Shamir indicated that the DES S-boxes had been specifically designed to be resistant to differential cryptanalysis (to some extent), suggesting that the technique of differential cryptanalysis was known (but not publicly revealed) by the designers of DES. After Biham and Shamir announced their result, the designers of DES claimed that they were indeed aware of differential cryptanalysis and had designed DES to thwart this type of attack (but were asked by the NSA to keep it quiet in the interests of national security).

Following Biham and Shamir's breakthrough, *linear cryptanalysis* was developed by Matsui in the early '90s and was also applied successfully to DES. The advantage of Matsui's attack is that although it still requires a large number of outputs (2^{43} to be exact), they may be arbitrary and need not be chosen by the attacker. (That is, it utilizes a known-plaintext attack rather than a chosen-plaintext attack.) Nevertheless, it is still hard to conceive of any real scenario where it would be possible to obtain such a large number of input/output (or plaintext/ciphertext) pairs.

We briefly describe the basic ideas behind differential and linear cryptanalysis in Section 5.6. In conclusion, however, we emphasize that although it is possible to break DES in less time than that required by a brute-force search using sophisticated cryptanalytic techniques, an exhaustive key search is still the most effective attack in practice.

5.4 Increasing the Key Length of a Block Cipher

The only known practical weakness of DES is its relatively short key. It thus makes sense to try to design a block cipher with a larger key length using "basic" DES as a building block. Some approaches to doing so are discussed in this section. Although we refer to DES throughout the discussion, and DES is the most prominent instance where these techniques have been applied, everything we say here applies generically to any block cipher.