Block Cipher:
The DEADS Encryption Scheme

Block ciphers encrypt plain-text in blocks and decrypt cipher-text in blocks. DEADS, the encryption algorithm that encrypts the secret instructions that deactivate the bomb, is a block cipher.

In the rest of this document, we first give a brief description of Feistel networks (the building block of DEADS); we then give a description of the DEADS encoding procedure; followed by descriptions of individual components in DEADS.

Although the description may seem complicated, all you really have to do is implement a handful of well-defined functions (e.g., `compress_permutation`, `expand_permutation`, `parity_checker`, ...).

Feistel networks

Many block ciphers take the form of *Feistel networks*. The structure of a Feistel network is illustrated in Figure 1.

In a Feistel network, the input is divided evenly into a left and a right block, and the output of the i^{th} round of the cipher is calculated as

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

The nice property of Feistel networks is that f can be made arbitrarily complicated, yet the input can always be recovered from the output as

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

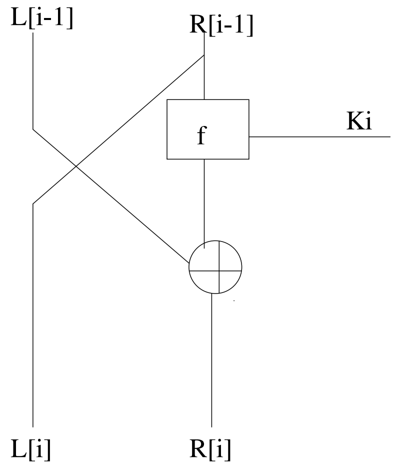

Figure 1: Feistel Network

Therefore even if f is not invertible, the algorithm itself will be invertible, allowing for decryption (if you know the K_i's, of course).

Feistel networks can cascade and form more powerful encryption algorithms. Each Feistel network in the cascade in referred as one **round** of the encryption. Many well known encryption algorithms such as: DES, Lucipher, FEAL, Khufu, LOKI, GOST, Blowfish use Feistel networks as the building block. The Feistel network is also the building block in the DEADS encryption scheme that you are about to use.

Description of DEADS

Outline of the Algorithm

DEADS is a block cipher; it encrypts the messages in 64-bit blocks. A 64-bit block of plain-text goes in one end of the algorithm and a 64-bit block of cipher-text comes out of the other end.

First the input, string of characters (8 bit per character) are packed into 32 bit integers. This transformation is described in Figure 2. The result is stored in an array of integers. This is the `plain-text`, that we will encrypt.

DEADS operates on a 64-bit block of plain-text (that is two integers) at a time. In the rest of our description, we use two integers `plaintext[0]` and `plaintext[1]` to represent the 64 bit block.

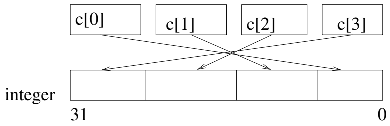

Figure 2: Pack character string to integer array

DEADS has 4 rounds, each with a different f_i function. The overall algorithm is illustrated in Figure 3.

There are two parts in each f_i function, p_i and q_i, as shown in Figure 4. The first part p_i takes a 32-bit integer and generates a 48-bit value To represent a 48-bit value we use two integers—the full 32 bits of one of the integers, and the lower 16 bits of the other. A `xor` operation is performed on the result of p_i and the key for that round. The output of this `xor` operation is the input of q_i, which is the second part of the function f_i. The q_i function takes a 48-bit value and generates a 32-bit integer.

- The operations in round 1 are the same, and are shown in Figure 5a.
- The operations in round 2 are shown in Figure 5b.
- The operations in round 3 are shown in Figure 5c.
- The operations in round 4 are shown in Figure 5d.

The Key Transformation

The initial key is given as two 32-bit integers, for a total of 64 bits. Every eighth bit of the initial key is dropped to form the 56 bit key[1]. After the 56-bit key is extracted, 48-bit sub-keys are generated for each of the 5 rounds of DEADS. Figure 6 shows how the sub-key generation works for a single round. First, the 56-bit key is divided into two 28-bit halves. Then the halves are circularly shifted to the left by several bits depending on the round. The number of bits shifted is 1, 1, 2, 2 for each round in that order. Note that each round uses the shifted keys from the previous round so the total shift for each round is 1, 2, 4, 6.

[1]Using 56 bits for DES keys was carefully chosen by the government so they could break it but no one else could. With modern technology the 56-bit keys in DES are no longer secure.

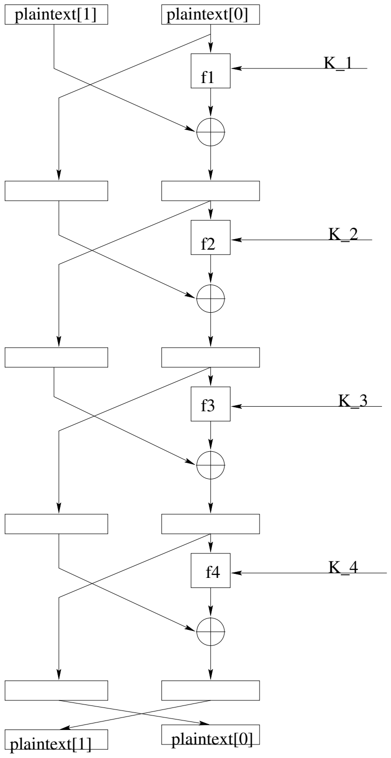

Figure 3: An overview of the encoding algorithm

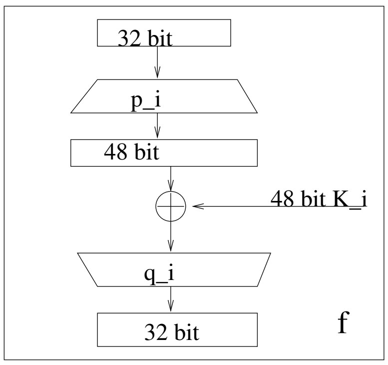

Figure 4: A decomposition of the f function

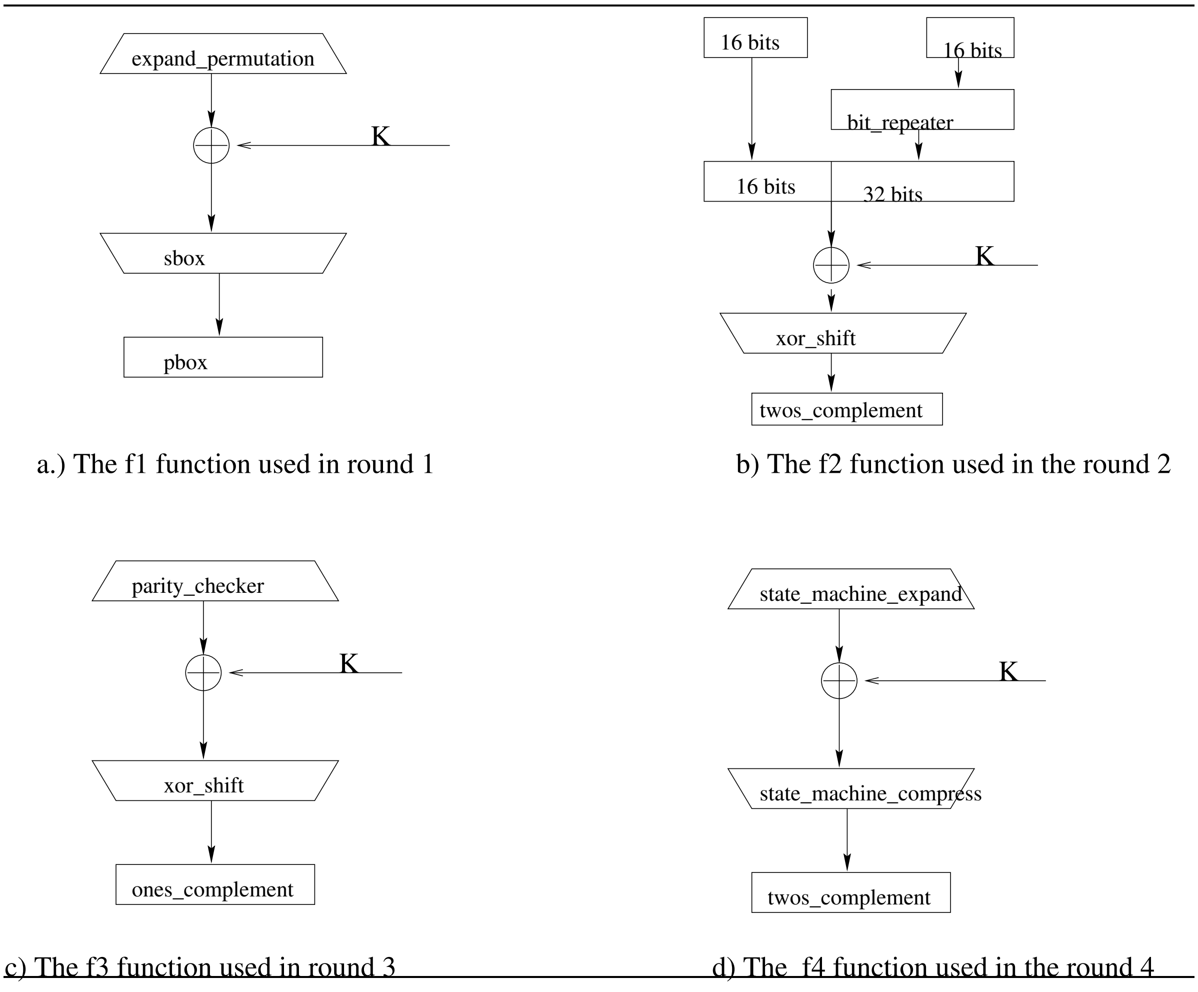

Figure 5: A graphical explanation of each round of the algorithm

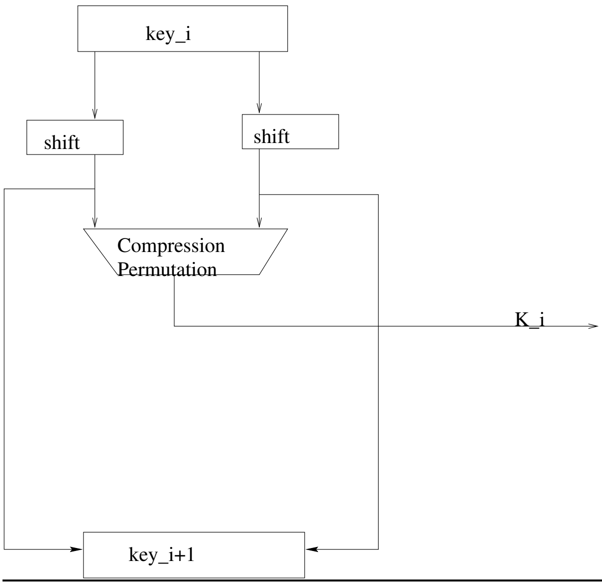

Figure 6: Key transformation in each round

After being shifted, 48 out of 56 bits are selected. The selected bits are permuted to form the key for that particular round. In our program, the two procedures (bit selection and permutation) are combined into one procedure, called `compress_permutation`. A table, called the `c_perm`, is defined in `bits.c`. For each bit on the new 48-bit key, `c_perm` gives the position of this bit in the old 56-bit key.

Expand_permutation

The DEADS `expand_permutation` function used in round 1 takes 32 bits as input and produces 48 bits as output. The 48 output bits are the 32 input bits with some of the input bits repeated. In particular, for each block of 4 input bits, the first and fourth bits appear twice in the output while the two middle bits appear once. A table, called the `e_perm`, is defined in `bits.c`. For every bit in the output of this function, this table gives the position of this bit in the 32-bit input.

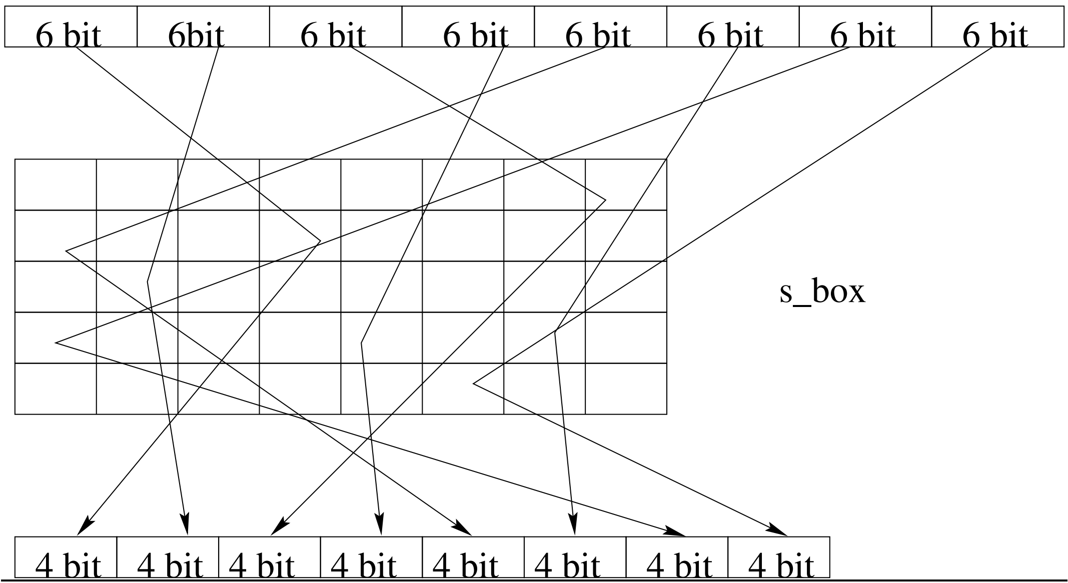

Figure 7: The S-box

P-box

The DEADS `P-Box` in round 1 is a straight 32-bit permutation (every input bit is used exactly once in the output).

S-box

S-box is used in round 1 to convert a 48 bit input to a 32 bit output. The s-box divide the 48 bit input into 8 6-bit blocks. Every 6-bit block is treated as single number, which is used to index into the `s_box` table, the content of the table entry is an integer (range from 0 - 15). This integer is represented using 4 bits. These 4 bits will be used to form the 32-bit output. See Figure 7 for details.

bit_repeater

Put the lower 16 bits of the 32 bit input at the lowest 16 bit of the 48 bit output. Repeat the most significant bit of this 16-bit number for the next 16 bits in the output. The higher 16 bits of the input is the highest 16 bits of the output.

parity_checker

If the number of bits that are 1 in the 32-bit input is odd, then the highest 16 bits are set to 1, otherwise they are set to 0. The lower 32 bits of the output are identical to the input.

xor_shift

Divide the lower 32 bits of the input into two halves, 16 bits each. Perform an `xor` operation between the highest 16 bits of the 48-bit input and each of the halves. The output is the result of the two `xor` operations.

State machine

Round 4 of the algorithm involves a 4-state state machine. The state-machine is used in two functions. In `state_machine_expand`, divide the input into 4-bit blocks. Simulate the state-machine using these 4 bit blocks as inputs starting with the lowest block and the lowest bit within each block. After each block insert the 2-bit state number that the machine is in immediately before the 4-bit block in the output for a 6-bit output. `state_machine_expand` generates a 48-bit number from the 32-bit input.

In `state_machine_compress`, the input is divided into 3-bit blocks. Simulate the state-machine using these 3-bit blocks as inputs. Output the state number of the resulting state as the output. `state_machine_compress` generates a 32 bit number from the 48 bit input. See Figure 8 for details. Reset the state of the state-machine to `(0, 0)` every time you call `state_machine_expand`, or `state_machine_compress`. Do NOT reset the state machine after every 4-bit or 3-bit input. The state diagram is in Figure 9.

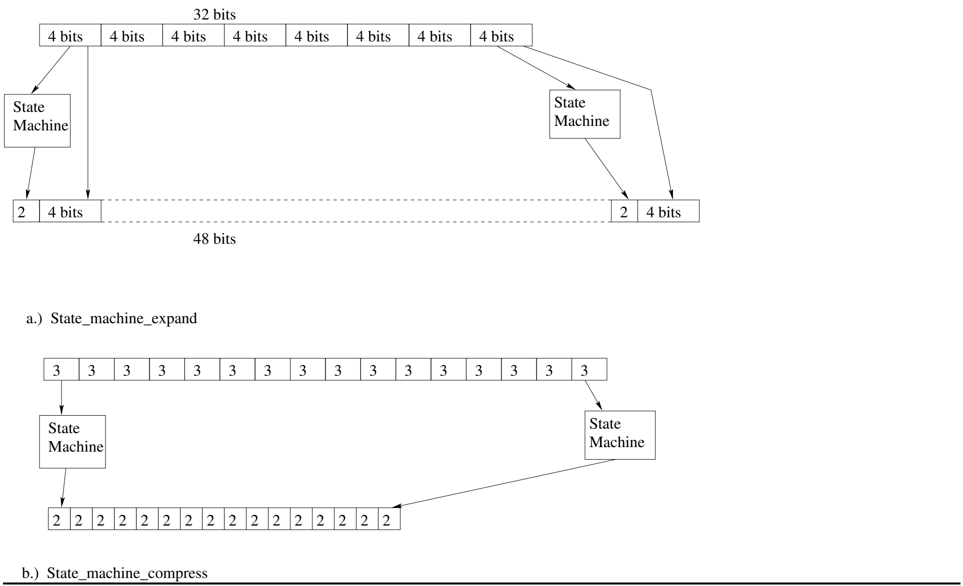

Figure 8: Bit expansion and compression with a state-machine

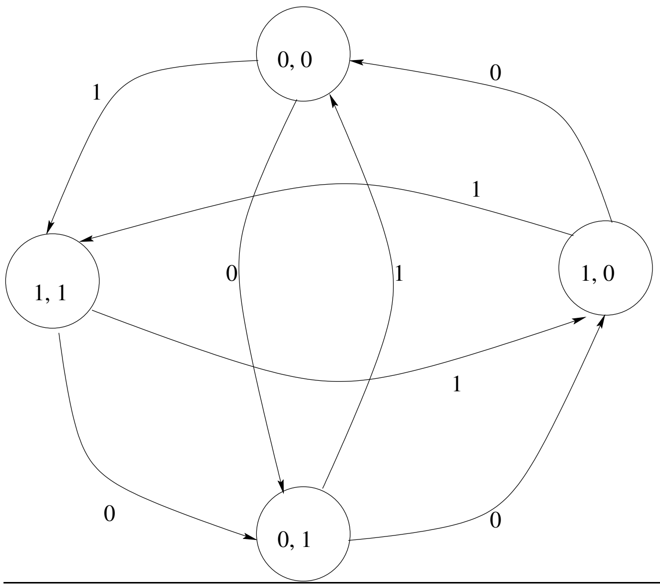

Figure 9: State Diagram