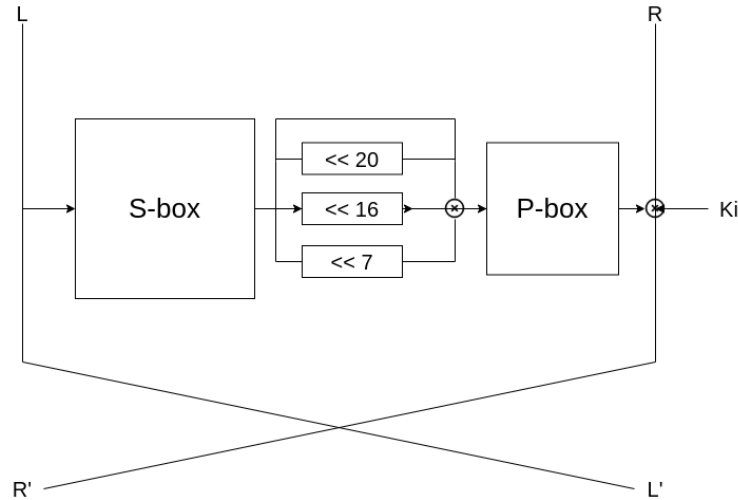


Final project

Ranin Nassra

1 Cipher

The block cipher has a 64-bit block size and a 64-bit key, computes the cipher in 12 rounds. Based on a feistel network.



1.1 Round Function

The round function is defined as:

$$F(l,r,k_i) = (\sigma(L(S(l))) \oplus r \oplus k_i, l)$$

Where l is the 16 most significant bits and r is the least 16 least significant bits of the plain text ($x=l|r$). k_i is the round key.

S is the parallel application of the 4-bit sbox:

sbox = (1, 0, 5, 3, e, 2, f, 7, d, a, 9, b, c, 8, 4, 6)

The diffusion layer consist of L:

$$L(x) = (x \ll 7) \oplus (x \ll 16) \oplus x$$

and σ is the following bit permutation:

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

1.2 Key schedule

Given master key $K=k_1|k_2$ where k_1 is the 16 most significant bits and k_2 is the least 16 least significant bits, the key for the i-th round is given by:

$$k_i = \begin{cases} \partial(k_i) \oplus 0x7 & \text{if } i=0,1 \\ \partial(k_0 \oplus k_1 \oplus \dots \oplus k_{i-1}) \oplus 0x7 & \text{else} \end{cases}$$

1.3 Test vectors

Plain text	Cipher text	Key
1234567890ABCDEF	BC62E3DCFF7AA374	0000000000000000
9D56F59C6E35487F	5C3933ECB664AF04	25A8B5EE0241D63E

1.4 Explanation

For the sbox I used the midori cipher s-box, checked the DDT of the sbox and found out that it has low enteries, so I decided to use it. The pbox is also one of midoris sboxes, slightly changed for more diffusion.

Each round key dependes on all the previous round keys, that is in case of when the attacker tries to guess the master key she would have to guess all the round keys.

The block cipher computes the cipher in 12 rounds, for a fully diffussion it takes 2 rounds. Doubling it we would have 4 rounds. In case of differential cryptanalisys attack best probability of first rounds is 2^{-8} that is because of the L function that difusses the input, for example when giving her 0000-000X differential the differential output would be 00XX-0X0X. With each round the probability becomes lower. I took an average of 2^{-10} for each round for probability of 2^{-64} it would take 6.4 rounds, by doubeling it the total rounds are 12.

1.5 Implementation

```
#!/usr/bin/env python3
```

```
def sigma(word):
    new_word = 0

    new_word |= (word & 0b00010000000000000000100000000000) # 3
    #permutation of the first half of the word
    new_word |= (word & 0x80000000) >> 12 # 0
    new_word |= (word & 0x40000000) >> 8 # 1
    new_word |= (word & 0x20000000) >> 11 # 2
    new_word |= (word & 0x08000000) >> 10 # 4
    new_word |= (word & 0x04000000) >> 6 # 5
    new_word |= (word & 0x02000000) >> 9 # 6
    new_word |= (word & 0x01000000) >> 1 # 7
    new_word |= (word & 0x00800000) >> 2 # 8
    new_word |= (word & 0x00400000) << 2 # 9
    new_word |= (word & 0x00200000) << 9 # A
    new_word |= (word & 0x00100000) << 6 # B
    new_word |= (word & 0x00080000) << 12 # C
    new_word |= (word & 0x00040000) << 11 # D
    new_word |= (word & 0x00020000) << 10 # E
    new_word |= (word & 0x00010000) << 9 # F

    #permutation of the second half of the word
    new_word |= (word & 0x00008000) >> 12 # 0
    new_word |= (word & 0x00004000) >> 8 # 1
    new_word |= (word & 0x00002000) >> 11 # 2
    new_word |= (word & 0x00000800) >> 10 # 4
    new_word |= (word & 0x00000400) >> 6 # 5
```

```

new_word |= (word & 0x00000200) >> 9 # 6
new_word |= (word & 0x00000100) >> 1 # 7
new_word |= (word & 0x00000080) >> 2 # 8
new_word |= (word & 0x00000040) << 2 # 9
new_word |= (word & 0x00000020) << 9 # A
new_word |= (word & 0x00000010) << 6 # B
new_word |= (word & 0x00000008) << 12 # C
new_word |= (word & 0x00000004) << 11 # D
new_word |= (word & 0x00000002) << 10 # E
new_word |= (word & 0x00000001) << 9 # F

return new_word

def rotate_left(word, n, word_size=32):
    mask = 2*word_size - 1

    return ((word << n) & mask) | ((word >> (word_size - n) & mask))

def L(word):
    return (rotate_left(word, 8) ^ rotate_left(word, 16) ^ rotate_left(word, 20) ^ word)

def apply_sbox(word, nibbles=8):
    word_new = 0
    sbox = (0x1, 0x0, 0x5, 0x3, 0xe, 0x2, 0xf, 0x7, 0xd, 0xa, 0x9, 0xb, 0xc, 0x8, 0x4, 0x6)

    for i in range(nibbles): # 8 nibbles
        nibble = (word >> (i*4)) & 0xF # retrieve the ith nibble
        # insert the permuted nibble in the correct position
        word_new |= sbox[nibble] << i*4
    return word_new

def F(word):
    word = apply_sbox(word)
    word = L(word)
    return sigma(word)

def round_function(left, right, key):
    return ((F(left) ^ right ^ key), left)

def compute_roundkeys(key, rounds):
    key_parts = []
    for i in range(2): # compute first two roundkeys each one 32 bits
        key_parts.append(sigma((key & 0xFFFFFFFF) ^ 0x7))
        key >>= 32

    key_parts.reverse()

    for i in range(2, rounds): # compute the rest of the roundkeys
        rk = key_parts[0]
        for j in range(1, i): # xor with all the previous round keys
            rk = rk ^ key_parts[j]
        rk = sigma(rk) ^ 0x7
        key_parts.append(rk)

    return key_parts

def encrypt(word, key, rounds = 12):
    left = (word >> 32) & 0xFFFFFFFF
    right = word & 0xFFFFFFFF

    round_keys = compute_roundkeys(key, rounds)

    for i in range(rounds):
        left, right = round_function(left, right, round_keys[i])

    return (left << 32) | right

def decrypt(word, key, rounds = 12):
    left = word & 0xFFFFFFFF
    right = (word >> 32) & 0xFFFFFFFF

    round_keys = compute_roundkeys(key, rounds)
    round_keys.reverse()

```

```
    for i in range(rounds):
        left, right = round_function(left, right, round_keys[i])

    return (right << 32) | left

if __name__ == "__main__":
    import sys
    import random

    word = 0x9D56F59C6E35487F
    key = 0x25A8B5EE0241D63E
    cipher = encrypt(word, key, rounds = 12)

    print("%016X %016X"%(word, cipher))
```