Specification of CHOCOLATE

April 11, 2019

1 Cipher

The block cipher CHOCOLATE has a 64-bit blocksize and a 64 bit key. CHOCOLATE is a 12 rounds cipher based on a feistel network with 4-bit sboxes and a bit permutation as the linear layer, and also a function based on rotation and XOR a see Figure 1.1 for a visual representation.

1.1 Round function

Given a word $x = l|r$ where l consists of the 32 most significant bits and r consists of the 32 least significant bits. We divide to $l = l_{up}|l_{low}$ (each 16 bits). We define two functions:

$F_1(x) = \sigma(S(x))$, $F_2(x) = (x \lll 3) \oplus (x \lll 8) \oplus (x \lll 14)$

And we can define the round function F as follows:

$$F(l, r, k) = \left(\left(F_2\left(l_{low} \oplus F_1(l_{up})\right) \middle| F_1(l_{up}) \right) \oplus r \oplus k \middle| l \right)$$

Where S^0 is the parallel application of the 4-bit sbox S to the state and σ is a bit permutation.

see Figure 1.1 for a visual representation.

The sbox S is defined as follows (taken from block cipher PRESENT):

S = (C, 5, 6, B, 9, 0, A, D, 3, 0, F, 8, 4, 7, 1, 2)

and the bit permutation σ is defined as follows (taken from TC05):

$$\sigma = \left(\begin{array}{cccc|cccc|cccc|cccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ 6 & 0 & 1 & 7 & E & 8 & 9 & F & 2 & 4 & 5 & 3 & A & C & D & B \end{array} \right)$$

1.2 Key schedule

Given master key $K = k_1|k_0$ the round key k_i is defined as follows:

$$k_{i+1} = k_i \oplus \sigma(k_{i-1}) \oplus 0x3$$

1.3 Test vectors

plaintext	ciphertext	key
0000000000000000	75ede5d924e500e1	0000000000000000
0123456789abcdef	01a2dd0a93069e20	0123456789abcdef
abababab cdcdcdcd	dd163fce7667a031	0123012301230123

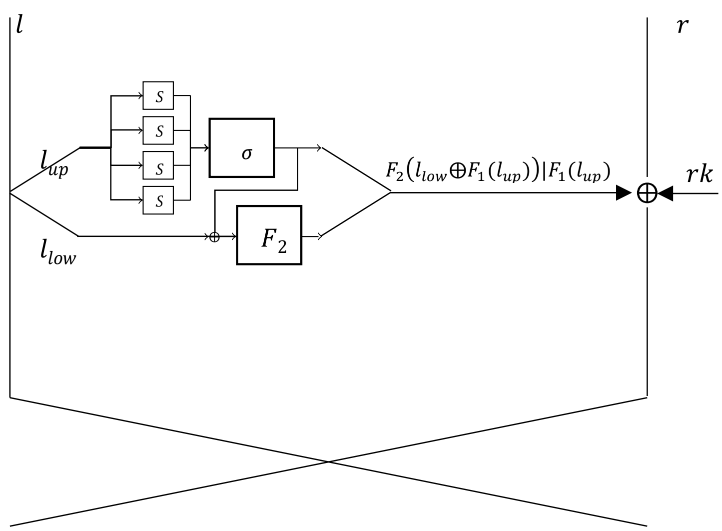

Figure 1: Round function of CHOCOLATE

1.4 Reference Implementation

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

def apply_sbox(word, nibbles=4):
    """ apply the sbox to every nibble """
    word_new = 0
    sbox = (0xC, 5, 6, 0xB, 9, 0, 0xA, 0xD, 3, 0xE, 0xF, 8, 4, 7, 1, 2)
    for i in range(nibbles):  # 16 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 sigma(word):
    """
    Implementing the sigma permutation on the 8 bit word.
    """
    new_word = 0
    # first move the two most significant bits of nibble 0 and 3
    new_word |= (word & 0b1100000000001100) >> 1  # 0, 1, C, D

    # now move the rest of the bits
    new_word |= (word & 0x2000) >> 6  # 2
    new_word |= (word & 0x1000) >> 8  # 3
    new_word |= (word & 0x0C00) >> 5  # 4, 5
    new_word |= (word & 0x0200) << 6  # 6
    new_word |= (word & 0x0100) << 4  # 7
    new_word |= (word & 0x00C0) << 3  # 8, 9
    new_word |= (word & 0x0020) >> 2  # A
    new_word |= (word & 0x0010) >> 4  # B
    new_word |= (word & 0x0002) << 10  # E
    new_word |= (word & 0x0001) << 8  # F

    return new_word


def F1(word):
    return sigma(apply_sbox(word))

def rotate_left(word, n, word_size=16):
    mask = 2**word_size - 1
    return ((word << n) & mask) | ((word >> (word_size - n) & mask))

def F2(word):
     return rotate_left(word, 3, 16) ^ rotate_left(word, 8, 16) ^ rotate_left(word, 14, 16)

def F(word):
    upper = (word >> 16) & 0xFFFF
    upper = F1(upper)
    lower = word & 0xFFFF
    lower = F2(lower^upper)
    return (lower << 16) | upper


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


def compute_roundkeys(key, rounds):
    key_parts = []
    key_parts.append(key & 0xFFFFFFFF)
    key >>= 32
    key_parts.append(key & 0xFFFFFFFF)

    for i in range(2, rounds):
        rk = key_parts[i - 1] ^ sigma(key_parts[i - 2]) ^ 0x3
        key_parts.append(rk)


    return key_parts
```

```python
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
```