Specification of TC05

April 4, 2019

1 Cipher

The block cipher TC05 has a 32-bit blocksize and a 64 bit key. TC05 is based on a feistel network with 4-bit sboxes and a bit permutation as the linear layer, see Figure 1.1 for a visual representation.

1.1 Round function

Given a word $x = l|r$ where l consists of the 16 most significant bits and r consists of the 16 least significant bits we can define the round function F as follows:

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

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

The sbox S is defined as follows:

$$S = (\texttt{E, B, 4, 6, A, D, 7, 0, 3, 8, F, C, 5, 9, 1, 2})$$

and the bit permutation σ is defined as follows:

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

1.2 Key schedule

Given master key $K = k_0|k_1|k_2|k_3$ the round key k_i is defined as follows:

$$k_{i+1} = k_{i-3} \oplus k_i \oplus \sigma(k_{i-1}) \oplus 0xC$$

1.3 Test vectors

Plaintext	Ciphertext	Key
00000000	9551EDDA	0000000000000000
12345678	C81335FD	1234567890ABCDEF

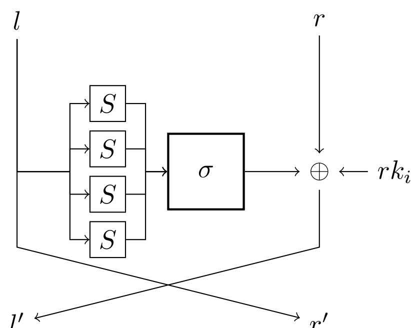

Figure 1: Round function of TC05

1.4 Reference Implementation

```
sbox = (0xE, 0xB, 4, 6, 0xA, 0xD, 7, 0, 3, 8, 0xF, 0xC, 5, 9, 1, 2)

def apply_sbox(word, sbox, nibbles=4):
    """ apply the sbox to every nibble """
    word_new = 0
    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 # E

    return new_word
```

```python
def F(word):
    return sigma(apply_sbox(word, sbox))

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

def compute_roundkeys(key, rounds):
    key_parts = []
    for i in range(4):
        key_parts.append(key & 0xFFFF)
        key >>= 16
    # Most significant part should be on index 0
    key_parts.reverse()

    for i in range(4, rounds):
        rk = key_parts[i-4] ^ key_parts[i-1] ^ sigma(key_parts[i-2]) ^ 0xC
        key_parts.append(rk)

    return key_parts

def encrypt(word, key, rounds=16):
    left = (word >> 16) & 0xFFFF
    right = word & 0xFFFF

    round_keys = compute_roundkeys(key, rounds)

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

    return (left << 16) | right

def decrypt(word, key, rounds=16):
    left = word & 0xFFFF
    right = (word >> 16) & 0xFFFF

    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 << 16) | left

def test():
    # first test the sigma function
    assert sigma(0x0000) == 0x0000
    assert sigma(0x8000) == 0x4000
    assert sigma(0xF000) == 0x6090
    assert sigma(0x0F00) == 0x9060
```

```
    assert sigma(0x00F0) == 0x0609
    assert sigma(0x000F) == 0x0906
    assert sigma(0xFFFF) == 0xFFFF
    # test the sbox
    assert apply_sbox(0xFF13, sbox) == 0x22B6
    assert apply_sbox(0x02DE, sbox) == 0xE491

    # test the invertibility
    assert decrypt(encrypt(0xFFFFFFFF, 0), 0) == 0xFFFFFFFF
    assert decrypt(encrypt(0xF4F31255, 0x1234567890ABCDEF), 0x1234567890ABCDEF) == 0x

def generate_testvectors():
    p, k = 0, 0
    c = encrypt(p, k)
    print("%08X %08X %016X"%(p, c, k))
    p, k = 0x12345678, 0x1234567890ABCDEF
    c = encrypt(p, k)
    print("%08X %08X %016X"%(p, c, k))


if __name__ == "__main__":
    test()
    generate_testvectors()
```