Brute forcing TC01

November 21, 2018

As a first exercise we focus on brute forcing a `Toy Cipher` [1], which we will name `TC01`. It is on purpose a very weak, but very easy to implement `Toy Cipher`. To get you started a Python implementation is given. My advice is to rewrite it in a language with less overhead (such as C, C++, Go, etc.).

1 Cipher

The cipher takes a 64-bit key and 64-bit words and computes the ciphertext in 20 rounds. The words are divided into 16 4-bit nibbles. We index s.t. the 0-th bit is the LSB and the 0-th nibble is bit 0 to bit 3.

1.1 Round function

The round function consists of a substitution and a permutation layer. The permutation layer consists of 16 parallel applications of a 4-bit s-box given by:

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

The diffusion layer operates on the full word and is defined by the following function:

$$L(x) = (x \lll 15) \oplus (x \lll 32) \oplus x$$

This leads to the following round function (where k_i is the i-th round key):

$$F(x, k_i) = L(S(x \oplus k_i))$$

[1] By using the term `Toy Cipher` I hope that no one comes up with the great idea to use this in any form, maybe I should use a very restricitive license to be able to enforce this.

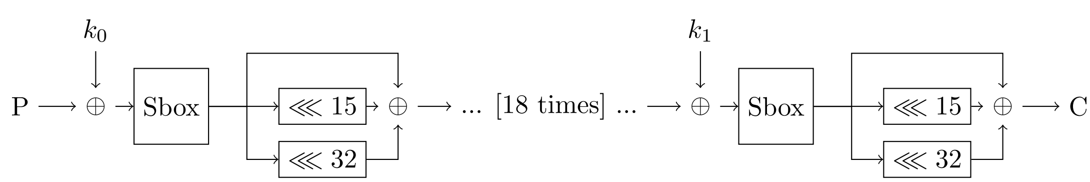

Figure 1: `TC01`

1.2 Key schedule

Given a master key K, the key for the i-th round is given by:

$$k_i = \begin{cases} L(k_{i-1}) \oplus \mathtt{0x3} & \text{if } i > 0 \\ K & \text{if } i = 0 \end{cases}$$

1.3 Reference Implementation

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

def rotate_left(word, n, word_size=64):
    mask = 2**word_size - 1
    return ((word << n) & mask) | ((word >> (word_size - n) & mask))

def L(word):
    return (rotate_left(word, 15) ^ rotate_left(word, 32) ^ word)

def apply_sbox(word, sbox):
    # apply the sbox to every nibble
    word_new = 0
    for i in range(16): # 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 round_function(word, key):
    # we first define the S-box, now sbox[0] = 2, sbox[1] = 4, etc.
    sbox = [0x2, 0x4, 0x5, 0x6, 0x1, 0xA, 0xF, 0x3,
            0xB, 0xE, 0x0, 0x7, 0x9, 0x8, 0xC, 0xD]

    # xor the key into the state
    word ^= key
    # apply the sbox to every nibble of the word
    word = apply_sbox(word, sbox)
    # apply the linear layer to the state
    word = L(word)
    # return the new word and the key for the next round
    return word, L(key)^0x3

def encrypt(word, key, rounds=20):
    # Apply the round function <rounds> times
    for r in range(rounds):
        word, key = round_function(word, key)

    return word
```