# Specification of SINGE

Yonatan Sigal

May 2019

## 1 Cipher

The cipher takes a 64-bit key and 64-bit words and computes the ciphertext in 16 rounds. The words are divided into 16 4-bit nibbles, called the state of the cipher.

### 1.1 Round function

The round function of SINGE is similar to the round function of AES. As in AES it consists of a sequential application of 4 layers. Add Roundkey (AR), Sub Cells (SC), Shift Rows (SR) and Mix Columns (MC). Note that my SR layer is round dependent

#### 1.1.1 Add Roundkey

In the Add Roundkey layer we XOR the key state with the cipher state.

#### 1.1.2 Sub Cells

In the Sub Cells (SC) layer we apply following substitution box (taken from [1]) (sbox) from to every nibble of the iternal state.

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

#### 1.1.3 Shift Rows

In the shift rows layer (SR) we rotate the nibbles in the rows depending on the round. we shift the rows by 0, 1, 2 and 3 to the left, and every round we start with another row i.e. on the first round we perform a regular SR, then on the next round start with the 4th row (starting one row upper on each round), and so on. note this illustration where $i$ is the round index and $<<<$ is nibble-wise rotate left

| |
|---|
| $row0 <<< i\ (mod\ 4)$ |
| $row1 <<< 1 + i\ (mod\ 4)$ |
| $row2 <<< 2 + i\ (mod\ 4)$ |
| $row3 <<< 3 + i\ (mod\ 4)$ |

### 1.1.4 Mix Columns

In the Mix Columns (MC) layer we mix the nibbles in every column according to a matrix. The matrix is given by:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

We multiply each column with the matrix.

## 1.2 Key Schedule

Given a master key $K = k0$, the keystate for the i-th round is given by applying the following recurrence relationship:

$$k_i = SR((k_{i-1} \bigoplus 0x00000000FFFFFFFF) <<< 16)$$

where $SR$ is the regular Shift Rows, not the round dependent

# 2 Test Vectors

Using $k = 0x0123456789ABCDEF$ as the key, here are some test examples:

| Plaintext | Ciphertext |
|---|---|
| 0x0000000000000000 | 0xB2AD8767AA0F1DDB |
| 0xDEADBEEFBE57F00D | 0x2507E9B425E90F9F |
| 0xDEADBEEFBAADF00D | 0xAAB0D2B332DF456F |
| 0x01236989EF16597A | 0x34A27566CE6CB740 |
| 0xAFDCD7290FAF64BA | 0x4716AEF7024E87B8 |
| 0xFBF074C91C4AD5EF | 0xADD4BFC0033D4F7E |

# 3 Design Rationale

I tried to create an AES like cipher, but wanted to be a little bit creative, especially with the SR function and Key Scheduling. I used TC0X Ciphers we broke as the shell to my cipher, but with tweaks and changes that (I think) made it stronger.

# 4 Python Reference Code

most of the code is reusing [2] reference code, with of course the changes needed for my own cipher

```
def get_rows(word):
    row_0 = (word >> 48) & 0xFFFF
    row_1 = (word >> 32) & 0xFFFF
    row_2 = (word >> 16) & 0xFFFF
    row_3 = (word >>  0) & 0xFFFF
    return row_0, row_1, row_2, row_3


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


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


def next_keystate(keystate):
    return shift_rows(rotate_left(keystate ^ 0x00000000FFFFFFFF, 16, 64))


def add_roundkey(word, keystate):
    return word ^ keystate


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 shift_rows(word, i = 0):
    row_0, row_1, row_2, row_3 = get_rows(word)

    # apply the shiftrows transformation
    row_0 = rotate_left(row_0, (0  + 4*i) % 16, 16)
    row_1 = rotate_left(row_1, (4  + 4*i) % 16, 16)
    row_2 = rotate_left(row_2, (8  + 4*i) % 16, 16)
    row_3 = rotate_left(row_3, (12 + 4*i) % 16, 16)
```

```
    # reconstruct the word
    new_word = row_0 << 48  # a |= b <==> a = a | b
    new_word |= row_1 << 32
    new_word |= row_2 << 16
    new_word |= row_3 << 0

    return new_word


def mix_columns(word):
    row_0, row_1, row_2, row_3 = get_rows(word)  # split up the word into rows
    # Apply the mix culomns transformation and reconstruct the word
    new_word = (row_0 ^ row_2) << 48
    new_word |= (row_1 ^ row_2) << 32
    new_word |= (row_0 ^ row_2 ^ row_3) << 16
    new_word |= (row_1 ^ row_3) << 0

    return new_word


def round_function(word, keystate, round):
    sbox = [0xE, 0x4, 0xD, 0x1, 0x2, 0xF, 0xB, 0x8,
            0x3, 0xA, 0x6, 0xC, 0x5, 0x9, 0x0, 0x7] # sbox taken from cited article

    word = add_roundkey(word, keystate)
    word = apply_sbox(word, sbox)
    word = shift_rows(word, round)
    word = mix_columns(word)

    return word


def encrypt(word, key, rounds):
    keystate = key
    for i in range(rounds):
        # apply the roundfunction to word
        word = round_function(word, keystate, i)
        # go to the next key state
        keystate = next_keystate(keystate)
    return word
```

# References

[1] Sankhanil Dey and Ghosh Ranjan. 4, 8, 32, 64 bit substitution box generation using irreducible or reducible polynomials over galois field $gf(p^q)$ for smart applications. *Arxiv*, page 5.

[2] Eran Lambooi. Tc0x cipher series. *https://cryptanex.hideinplainsight.io*, 2018.

LaTeX