Handle: BruteEncryptor

Specification
TC05_PRESENT

May 14, 2019

Changes from TC05:

- Cipher is no longer of feistel network structure
- Plain texts, and round keys, are now 64 bits long
- 8-bit sbox – AES sbox
  - The rationale: Aes DDT table is pretty good, and 8-bit sboxs seem to be more optimal
- New 64 bit premutation function (sigma) – the PRESENT cipher permutation layer
  - The rationale: Present permutation is a Light, Simple 64-bit permutation function
- New keyschedule
- 12 rounds

1 Cipher

The block cipher TC05_PRESENT has a 64-bit blocksize and a 64 bit key. TC05_PRESENT is based on a 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 of 64 bits, we can define the round function F as follows:

$$F(word, k_i) = (\sigma(S^0(word)) \oplus k_i)$$

Where S^0 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 = (63, 7C, 77, 7B, F2, 6B, 6F, C5, 30, 01, 67, 2B, FE, D7, AB, 76, CA, 82,
C9, 7D, FA, 59, 47, F0, AD, D4, A2, AF, 9C, A4, 72, C0, B7, FD, 93, 26, 36, 3F,
F7, CC, 34, A5, E5, F1, 71, D8, 31, 15, 04, C7, 23, C3, 18, 96, 05, 9A, 07, 12,
80, E2, EB, 27, B2, 75, 09, 83, 2C, 1A, 1B, 6E, 5A, A0, 52, 3B, D6, B3, 29, E3,
2F, 84, 53, D1, 00, ED, 20, FC, B1, 5B, 6A, CB, BE, 39, 4A, 4C, 58, CF, D0, EF,
AA, FB, 43, 4D, 33, 85, 45, F9, 02, 7F, 50, 3C, 9F, A8, 51, A3, 40, 8F, 92, 9D,
38, F5, BC, B6, DA, 21, 10, FF, F3, D2, CD, 0C, 13, EC, 5F, 97, 44, 17, C4, A7,
7E, 3D, 64, 5D, 19, 73, 60, 81, 4F, DC, 22, 2A, 90, 88, 46, EE, B8, 14, DE, 5E,
0B, DB, E0, 32, 3A, 0A, 49, 06, 24, 5C, C2, D3, AC, 62, 91, 95, E4, 79, E7, C8,
37, 6D, 8D, D5, 4E, A9, 6C, 56, F4, EA, 65, 7A, AE, 08, BA, 78, 25, 2E, 1C, A6,
B4, C6, E8, DD, 74, 1F, 4B, BD, 8B, 8A, 70, 3E, B5, 66, 48, 03, F6, 0E, 61, 35,
57, B9, 86, C1, 1D, 9E, E1, F8, 98, 11, 69, D9, 8E, 94, 9B, 1E, 87, E9, CE, 55,
28, DF, 8C, A1, 89, 0D, BF, E6, 42, 68, 41, 99, 2D, 0F, B0, 54, BB, 16)
```

and the bit permutation σ is defined as follows (now it is for 64 bit words):

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\sigma(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$\sigma(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$\sigma(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

1.2 Key schedule

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

$$k_{i+1} = (k_i >>> 15) \oplus 0x3$$

1.3 Test vectors

Plaintext	Ciphertext	Key
0000000000000000	A9B5129AE6A1640C	0000000000000000
123456789ABCDEF0	4DADBC2E8E229030	789A147132BCFDFA

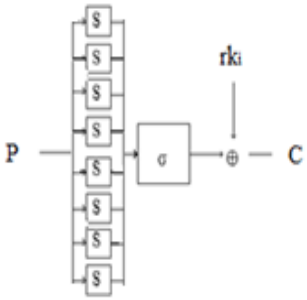

Figure 1: 1 Round of TC05_PRESENT

1.4 Pyhton reference Implementation

```
sbox = (0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76
        , 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0
        , 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15
        , 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75
        , 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84
        , 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf
        , 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8
        , 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2
        , 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73
        , 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb
        , 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79
        , 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08
        , 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a
        , 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e
        , 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf
        , 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb,0x16)

sigma_table = [0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, 35, 51,
               4, 20, 36, 52, 5, 21, 37, 53, 6, 22, 38, 54, 7, 23, 39, 55,
               8, 24, 40, 56, 9, 25, 41, 57, 10, 26, 42, 58, 11, 27, 43, 59,
               12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62, 15, 31, 47, 63]


def apply_sbox(word, bytes=8):
    """ apply the sbox to every nibble """
    word_new = 0
    for i in range(bytes):  # 16 nibbles
        byte = (word >> (i * 8)) & 0xFF  # retrieve the ith nibble
        # insert the permuted nibble in the correct position
        word_new |= sbox[byte] << i * 8
    return word_new


def sigma(word):
    """
    Implementing the sigma permutation on the 64 bit word .
    """
    new_word = 0
    for i in range(64):
        # bit position i moves to the position of sigma_table[i]
        new_word |= ((word >> i) & 0b1) << sigma_table[i]
    return new_word


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


def round_function(word, key):
    return f(word) ^ key


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


def compute_roundkeys(key, rounds):
    key_parts = [key]
    for i in range(1, rounds):
```

```
        # cyclic shift by 15 bits, and xor by 3
        rk = rotate_left(key_parts[i - 1], 15)
        key_parts.append(rk ^ 0b11)

    return key_parts



def encrypt(word, key, rounds=12):
    round_keys = compute_roundkeys(key, rounds)
    for i in range(rounds):
        word = round_function(word, round_keys[i])
    return word



''' ---- decrypt part ---- '''

reverse_sbox = [sbox.index(i) for i in range(256)]

reverse_sigma_table = [sigma_table.index(i) for i in range(64)]


def apply_reverse_sbox(word, bytes=8):
    word_new = 0
    for i in range(bytes):  # 16 nibbles
        byte = (word >> (i * 8)) & 0xFF  # retrieve the ith permuted nibble
        # retrieve back the unpermuted nibble
        word_new |= reverse_sbox[byte] << i * 8
    return word_new


def reverse_sigma(word):
    new_word = 0
    for i in range(64):
        # bit position i (sigma(j)) moves back to the position of reverse_sigma_table[i] (j)
        new_word |= ((word >> i) & 0b1) << reverse_sigma_table[i]
    return new_word


def reverse_f(word):
    return apply_reverse_sbox(reverse_sigma(word))


def reverse_round_function(word, key):
    return reverse_f(word ^ key)


def decrypt(word, key, rounds=12):
    round_keys = compute_roundkeys(key, rounds)
    round_keys.reverse()
    for i in range(rounds):
        word = reverse_round_function(word, round_keys[i])
    return word



''' test vectors '''

def test_vectors():
    # first test the sigma function
    assert sigma(0x0000000000000000) == 0x0000000000000000
    assert sigma(0x8000) == 0x0008000000000000
    assert sigma(0xF000) == 0x0008000800080008
    assert sigma(0xFFFFFFFFFFFFFFFF) == 0xFFFFFFFFFFFFFFFF

    # test the sbox
    assert apply_sbox(0x0000000000000000) & 0xFFFFFFFFFFFFFFFF == 0x6363636363636363
    assert apply_sbox(0x0123456789ABCDEF) & 0xFFFFFFFFFFFFFFFF == 0x7C266E85A762BDDF
    assert apply_sbox(0x789A147132BCFDFA) & 0xFFFFFFFFFFFFFFFF == 0xBCB8FAA32365542D

    # test the invertibility
    assert decrypt(encrypt(0xFFFFFFFFFFFFFFFF, 0), 0) & 0xFFFFFFFFFFFFFFFF == 0xFFFFFFFFFFFFFFFF
    assert decrypt(encrypt(0xF4F31255F4F31255, 0x123), 0x123) & 0xFFFFFFFFFFFFFFFF == 0xF4F31255F4F31255

    return


def generate_testvectors():
    p, k = 0, 0
    c = encrypt(p, k)
    print("E(p=%016X,k=%016X)=%016X" % (p, k, c))
    p, k = 0x123456789ABCDEF0, 0x1234567890ABCDEF
    c = encrypt(p, k)
    print("E(p=%016X,k=%016X)=%016X" % (p, k, c))

    return
```

1.5 C++ reference Implementation (optimized)

```
#include <iostream>
#include <ctime>

class TC05_PRESENT_Cipher {
public:
	TC05_PRESENT_Cipher() {
		compute_SBOX8_sigma();
		compute_reverse_sbox();
		compute_reverse_sigma();
		compute_reverse_sigma8();
		generate_testvectors();
		test_vectors();

	}

	/* ----------encrypt---------- */
	uint8_t sbox[256] = { 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76
		, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0
		, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15
		, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75
		, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84
		, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf
		, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8
		, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2
		, 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73
		, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb
		, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79
		, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08
		, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a
		, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e
		, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf
		, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

	int sigma_table[64] = { 0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, 35, 51,
		4, 20, 36, 52, 5, 21, 37, 53, 6, 22, 38, 54, 7, 23, 39, 55,
		8, 24, 40, 56, 9, 25, 41, 57, 10, 26, 42, 58, 11, 27, 43, 59,
		12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62, 15, 31, 47, 63 };

	//combination of sbox and sigma (as seen in hw5)
	uint64_t SBOX8_sigma[256][8]; // sbox8 (256*8) instead of sbox16 (65536*4) to prevent memory troubles

	uint64_t sigma(uint64_t word) {
		/*
		Implementing the sigma permutation on the 64 bit word .
		*/
		uint64_t new_word = 0;
		for (int i = 0; i < 64; i++) {
			// bit position i moves to the position of sigma_table[i]
			new_word |= ((word >> i) & 0b1) << sigma_table[i];
		}
		return new_word;
	}

	void compute_SBOX8_sigma() { // as seen in hw5
		uint64_t s, w;
		for (uint32_t x = 0; x < 256; x++) {
			s = apply_sbox(x);
			for (int i = 0; i < 8; i++) {
				w = ((uint64_t)s & 0xFF) << (i * 8);
				SBOX8_sigma[x][i] = sigma(w);
			}
		}
	}

	uint64_t apply_sbox8_sigma(uint64_t word) {
		uint64_t word_new = 0;
		for (int i = 0; i < 8; i++) {
			word_new |= SBOX8_sigma[word & 0xFF][i]; // SBOX8_sigma already contains the corresponding byte-
shifts
			word >>= 8;
		}
		return word_new;
	}

	uint64_t apply_sbox(uint64_t word, int bytes = 8) {
		uint64_t word_new = 0;
		int bits = bytes * 8;
		for (uint8_t i = 0; i < bits; i += 8) {
			uint8_t byte = (word >> i) & 0xFF;
			word_new |= ((uint64_t)sbox[byte] << i);
		}
		return word_new;
	}

	uint64_t rotate_left(uint64_t word, int n, int word_size = 64) {
```

```
        uint64_t mask = 0xFFFFFFFFFFFFFFFF;
        return ((word << n) & mask) | (word >> (word_size - n) & mask);
}

void compute_roundkeys(uint64_t* key_parts, uint64_t key, int rounds) {
        uint64_t rk;
        key_parts[0] = key;
        for (int i = 1; i < 12; i++) {
                // cyclic shift by 15 bits, and xor by 3
                rk = rotate_left(key_parts[i - 1], 15);
                key_parts[i] = rk ^ 0b11;
        }
}

uint64_t round_function(uint64_t word, uint64_t key) {
        return apply_sbox8_sigma(word) ^ key;
}

uint64_t encrypt(uint64_t word, uint64_t key, int rounds = 12) {
        uint64_t *round_keys = new uint64_t[rounds];
        compute_roundkeys(round_keys, key, rounds);
        for (uint8_t i = 0; i < rounds; i++) {
                word = round_function(word, round_keys[i]);
        }
        delete(round_keys);
        return word;
}


/* ----------decrypt---------- */
int reverse_sigma_table[64];
uint64_t reverse_sigma8_table[256][8];
// reverse_sbox table - contains the byte shifts as well.
uint64_t reverse_sbox[256][8]; // sbox8 (256*8) instead of sbox16 (65536*4) to prevent memory troubles

void compute_reverse_sbox() {
        for (uint64_t x = 0; x < 256; x++) {
                for (int i = 0; i < 8; i++) {
                        reverse_sbox[sbox[x]][i] = (uint64_t)x << (i * 8); // to save future shitfs as well
                }
        }
}

void compute_reverse_sigma() {
        for (int i = 0; i < 64; i++) {
                reverse_sigma_table[sigma_table[i]] = i;
        }
}

void compute_reverse_sigma8() {
        for (int x = 0; x < 256; x++) {
                uint64_t part = (uint64_t)x;
                for (int i = 0; i < 8; i++) {
                        reverse_sigma8_table[x][i] = reverse_sigma(part);
                        part <<= 8;
                }
        }
}

uint64_t reverse_sigma(uint64_t word) {
        uint64_t new_word = 0;
        for (int i = 0; i < 64; i++) {
                // bit position i (=sigma(j)) returns back to the position of reverse_sigma_table[i] (=j)
                new_word |= ((word >> i) & 0b1) << reverse_sigma_table[i];
        }
        return new_word;
}

uint64_t reverse_sigma8(uint64_t word) {
        uint64_t new_word = 0;
        for (int i = 0; i < 8; i++) {
                new_word |= reverse_sigma8_table[word & 0xFF][i];
                word >>= 8;
        }
        return new_word;
}

uint64_t apply_reverse_sbox(uint64_t word, uint8_t bytes = 8) {
        uint64_t word_new = 0;
        for (uint8_t i = 0; i < 8; i++) {
                word_new |= reverse_sbox[word & 0xFF][i];
                word >>= 8;
        }
        return word_new;
}

uint64_t reverse_round_function(uint64_t word, uint64_t key) {
        return apply_reverse_sbox(reverse_sigma8(word^key));
```

```
        }

        uint64_t decrypt(uint64_t word, uint64_t key, int rounds = 12) {
                uint64_t *round_keys = new uint64_t[rounds];
                compute_roundkeys(round_keys, key, rounds);
                for (int i = rounds - 1; i >= 0; i--) {
                        word = reverse_round_function(word, round_keys[i]);
                }
                delete(round_keys);
                return word;
        }

        /* -------- test vectors ---------*/
        void test_vectors() {
                if (sigma(0x0000000000000000) != 0x0000000000000000) { exit(1); }
                if (sigma(0x8000) != 0x0008000000000000) { exit(1); }
                if (sigma(0xF000) != 0x0008000800080008) { exit(1); }
                if (sigma(0xFFFFFFFFFFFFFFFF) != 0xFFFFFFFFFFFFFFFF) { exit(1); }
                if (reverse_sigma8(sigma(0x0000ABCADAAD0000)) != 0x0000ABCADAAD0000) { exit(1); }
                if (apply_sbox(0x0000000000000000) != 0x6363636363636363) { exit(1); }
                if (apply_sbox(0x0123456789ABCDEF) != 0x7C266E85A762BDDF) { exit(1); }
                if (apply_sbox(0x789A147132BCFDFA) != 0xBCB8FAA32365542D) { exit(1); }
                if (decrypt(encrypt(0xFFFFFFFFFFFFFFFF, 0), 0) != 0xFFFFFFFFFFFFFFFF) { exit(1); }
                if (decrypt(encrypt(0xF4F31255F4F31255, 0x123), 0x123) != 0xF4F31255F4F31255) { exit(1); }

                return;
        }

        void generate_testvectors() {
                int64_t p = 0, k = 0;
                int64_t c = encrypt(p, k);
                printf("E(p=%016llX,k=%016llX)=%016llX\n", p, k, c);
                p = 0x123456789ABCDEF0;
                k = 0x1234567890ABCDEF;
                c = encrypt(p, k);
                printf("E(p=%016llX,k=%016llX)=%016llX\n", p, k, c);
                return;
        }
};

int main() {
        TC05_PRESENT_Cipher cipher; // constructor is invoked

        clock_t begin = clock();
        // attackCipher(cipher)
        clock_t end = clock();
        double time_passed_in_sec = double(end - begin) / CLOCKS_PER_SEC;
        std::cout << "time passed: " << time_passed_in_sec << " seconds" << std::endl;

        return 0;
}
```