

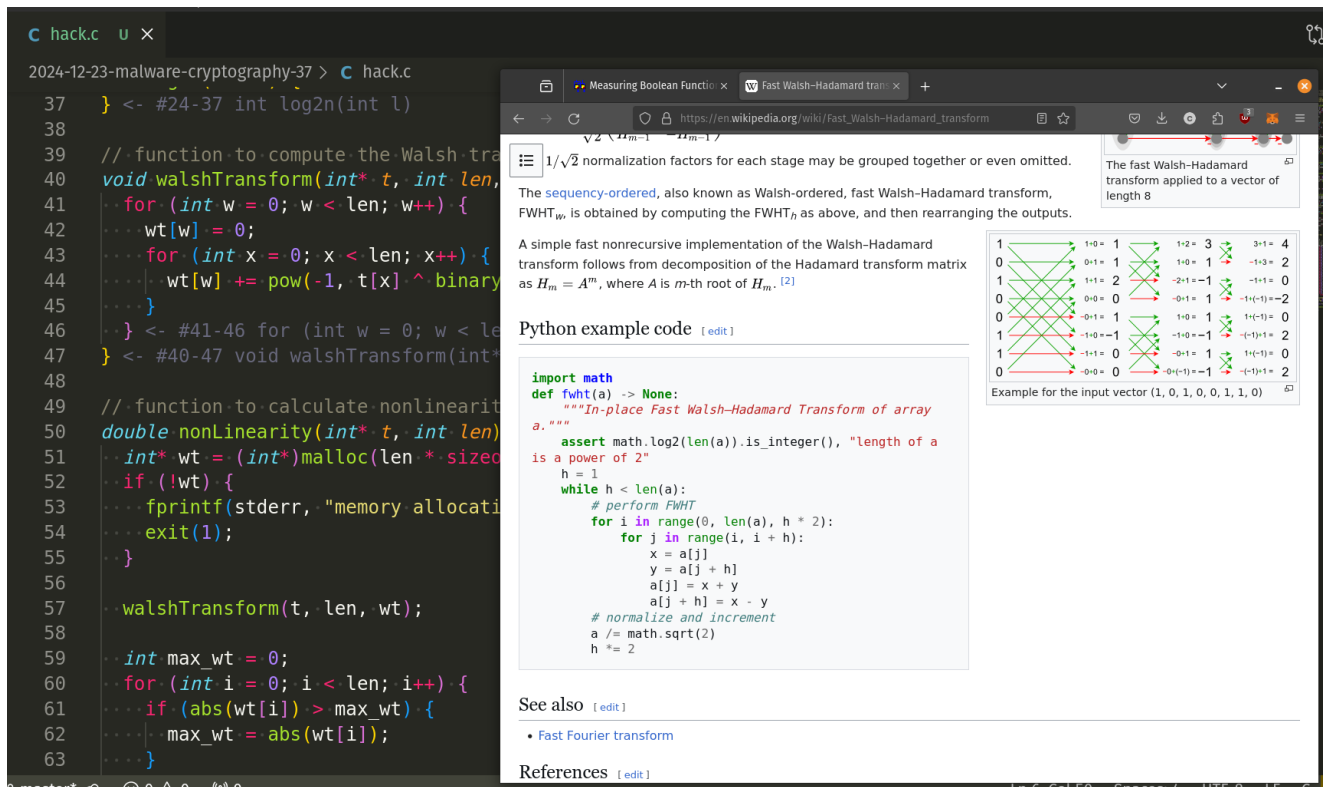
Malware and cryptography 37 - Nonlinearity. Walsh Transform. Simple C example.

 cocomelonc.github.io/malware/2024/12/16/malware-cryptography-37.html

December 16, 2024

17 minute read

Hello, cybersecurity enthusiasts and white hackers!



The screenshot displays a development environment with a C code editor on the left and a web browser on the right. The C code, named `walshTransform`, implements a fast Walsh-Hadamard transform using a recursive approach with bit-reversed indexing. The Python code, named `fwht`, implements an in-place fast Walsh-Hadamard transform using a butterfly network. The web browser shows a Wikipedia article titled "Fast Walsh-Hadamard transform" with a diagram illustrating the butterfly network for an 8-point transform.

```
2024-12-23-malware-cryptography-37 > C hack.c
37 } <- #24-37 int log2n(int l)
38
39 //function to compute the Walsh tra
40 void walshTransform(int* t, int len,
41 for (int w = 0; w < len; w++) {
42 wt[w] = 0;
43 for (int x = 0; x < len; x++) {
44 wt[w] += pow(-1, t[x] ^ binary
45 }
46 } <- #41-46 for (int w = 0; w < le
47 } <- #40-47 void walshTransform(int*
48
49 //function to calculate nonlinearity
50 double nonLinearity(int* t, int len)
51 int* wt = (int*) malloc(len * sizeof
52 if (!wt) {
53 fprintf(stderr, "memory allocati
54 exit(1);
55 }
56
57 walshTransform(t, len, wt);
58
59 int max_wt = 0;
60 for (int i = 0; i < len; i++) {
61 if (abs(wt[i]) > max_wt) {
62 max_wt = abs(wt[i]);
63 }
```

Measuring Boolean Function x Fast Walsh-Hadamard tran x +

1/√2 normalization factors for each stage may be grouped together or even omitted.

The sequence-ordered, also known as Walsh-ordered, fast Walsh-Hadamard transform, FWHT_{seq}, is obtained by computing the FWHT_h as above, and then rearranging the outputs.

A simple fast nonrecursive implementation of the Walsh-Hadamard transform follows from decomposition of the Hadamard transform matrix as $H_m = A^m$, where A is m -th root of H_m . [2]

Python example code [edit]

```
import math
def fwht(a) -> None:
    """In-place Fast Walsh-Hadamard Transform of array
    a."""
    assert math.log2(len(a)).is_integer(), "length of a
    is a power of 2"
    h = 1
    while h < len(a):
        # perform FWHT
        for i in range(0, len(a), h * 2):
            for j in range(i, i + h):
                x = a[j]
                y = a[j + h]
                a[j] = x + y
                a[j + h] = x - y
            # normalize and increment
            a /= math.sqrt(2)
            h *= 2
```

See also [edit]

- Fast Fourier transform

References [edit]

Example for the input vector (1, 0, 1, 0, 0, 1, 1, 0)

In the [previous post](#) I wrote about such a property of S blocks as nonlinearity. But how can this nonlinearity be analyzed and calculated? One of such methods is the Fourier transformation, or to be more precise, its subtype: Walsh Transformation. This post is the result of my own research on implementing Walsh Transformation.

the Walsh transform

The Walsh transform is a mathematical transformation that measures how “correlated” a binary sequence is with all possible linear functions. It is widely used in cryptography to analyze the nonlinearity of cryptographic functions, such as S-boxes, which are critical for resisting linear cryptanalysis.

The Walsh transform maps a binary sequence t into another sequence W , where each value in W indicates the “bias” of t towards a specific linear function. The formula for the Walsh transform is:

$$W(f)(\omega) = \sum_x (-1)^{f(x) \oplus (\text{inner product}(\omega, x))}$$

- $f(x)$ - is the binary function or sequence being transformed.
- $\text{inner product}(\omega, x)$ - is the dot product of (ω) and (x) in binary (bitwise AND followed by XOR).

The Walsh transform evaluates how much a binary sequence deviates from being linear. High nonlinearity is crucial for cryptographic primitives because it ensures resistance to linear cryptanalysis, a method that exploits linear relationships between input and output bits.

measuring nonlinearity of S-boxes

To measure the nonlinearity of an **S-box** using the Walsh transform:

- Transform the **S-box** into binary sequences: for each possible output bit, create binary sequences based on whether each **S-box** output bit is 0 or 1.
- Apply Walsh transform: for each binary sequence, compute the Walsh transform.
- Evaluate Nonlinearity: The nonlinearity is the minimum Hamming distance between the sequence and all linear functions.

This can be calculated using the Walsh transform as:

$$\text{Nonlinearity} = \frac{N}{2} - \frac{\max(\text{abs}(W(f)))}{2}$$

Where, (N) is the size of the **S-box** (e.g., $N=256$ for an 8-bit **S-box**). $(\max(|W(f)|))$ is the maximum absolute value of the Walsh transform output.

In other words, nonlinearity measures how far a binary sequence is from the closest linear function.

practical example

Let's create a simple program to calculate nonlinearity of custom S-boxes.

First of all we need function to compute binary inner product:

```

int binaryInnerProduct(int a, int b) {
    int ip = 0;
    int ab = a & b;
    while (ab > 0) {
        ip ^= (ab & 1);
        ab >>= 1;
    }
    return ip;
}

```

This is used to evaluate linear combinations.

Then, we need Walsh transformation logic:

```

// function to compute the Walsh transform of a binary sequence
void walshTransform(int* t, int len, int* wt) {
    for (int w = 0; w < len; w++) {
        wt[w] = 0;
        for (int x = 0; x < len; x++) {
            wt[w] += pow(-1, t[x] ^ binaryInnerProduct(w, x));
        }
    }
}

```

This function outputs a list of transformed values, representing the correlation with all possible linear functions.

Next one calculates the nonlinearity of the binary sequence:

```

// function to calculate nonlinearity of a binary sequence
double nonLinearity(int* t, int len) {
    int* wt = (int*)malloc(len * sizeof(int));
    if (!wt) {
        fprintf(stderr, "memory allocation failed.\n");
        exit(1);
    }

    walshTransform(t, len, wt);

    int max_wt = 0;
    for (int i = 0; i < len; i++) {
        if (abs(wt[i]) > max_wt) {
            max_wt = abs(wt[i]);
        }
    }

    free(wt);
    return len / 2.0 - 0.5 * max_wt;
}

```

At the final, just create function to calculate s-box nonlinearity:

```

// function to compute S-box nonlinearity
void sbxNonlinearity(int* sbx, int len) {
    double nlv_min = len / 2; // Minimum nonlinearity
    double nlv_max = 0;      // Maximum nonlinearity

    for (int c = 1; c < len; c++) { // Skip zero combination
        int* t = (int*)malloc(len * sizeof(int));
        if (!t) {
            fprintf(stderr, "memory allocation failed.\n");
            exit(1);
        }

        for (int i = 0; i < len; i++) {
            t[i] = binaryInnerProduct(c, sbx[i]);
        }

        double nl = nonLinearity(t, len);
        if (nl < nlv_min) {
            nlv_min = nl;
        }
        if (nl > nlv_max) {
            nlv_max = nl;
        }

        free(t);
    }

    printf("S-box nonlinearity (min, max): (%f, %f)\n", nlv_min, nlv_max);
}

```

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

```

/*
 * hack.c
 * S-box nonlinearity
 * via Walsh Transformation
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/12/23/malware-cryptography-37.html
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// function to compute binary inner product
int binaryInnerProduct(int a, int b) {
    int ip = 0;
    int ab = a & b;
    while (ab > 0) {
        ip ^= (ab & 1);
        ab >>= 1;
    }
    return ip;
}

// function to compute log2(n), ensuring n is a power of 2
int log2n(int l) {
    int n = 0;
    int x = l;
    while (x > 0) {
        x >>= 1;
        n++;
    }
    n--;
    if ((1 << n) != l) {
        fprintf(stderr, "error: length is not a power of 2.\n");
        exit(1);
    }
    return n;
}

// function to compute the Walsh transform of a binary sequence
void walshTransform(int* t, int len, int* wt) {
    for (int w = 0; w < len; w++) {
        wt[w] = 0;
        for (int x = 0; x < len; x++) {
            wt[w] += pow(-1, t[x] ^ binaryInnerProduct(w, x));
        }
    }
}

// function to calculate nonlinearity of a binary sequence
double nonLinearity(int* t, int len) {
    int* wt = (int*)malloc(len * sizeof(int));
    if (!wt) {

```

```

    fprintf(stderr, "memory allocation failed.\n");
    exit(1);
}

walshTransform(t, len, wt);

int max_wt = 0;
for (int i = 0; i < len; i++) {
    if (abs(wt[i]) > max_wt) {
        max_wt = abs(wt[i]);
    }
}

free(wt);
return len / 2.0 - 0.5 * max_wt;
}

// function to compute S-box nonlinearity
void sbxNonlinearity(int* sbx, int len) {
    double nlv_min = len / 2; // Minimum nonlinearity
    double nlv_max = 0;      // Maximum nonlinearity

    for (int c = 1; c < len; c++) { // Skip zero combination
        int* t = (int*)malloc(len * sizeof(int));
        if (!t) {
            fprintf(stderr, "memory allocation failed.\n");
            exit(1);
        }

        for (int i = 0; i < len; i++) {
            t[i] = binaryInnerProduct(c, sbx[i]);
        }

        double nl = nonLinearity(t, len);
        if (nl < nlv_min) {
            nlv_min = nl;
        }
        if (nl > nlv_max) {
            nlv_max = nl;
        }

        free(t);
    }

    printf("S-box nonlinearity (min, max): (%f, %f)\n", nlv_min, nlv_max);
}

int main() {
    // example custom S-box
    // int custom_sbox[] = {0x3, 0x6, 0x5, 0xC, 0xA, 0x0, 0xE, 0x9, 0xD, 0x8, 0xF, 0x4,
    // 0x7, 0x1, 0xB, 0x2};
    // int custom_sbox[] = {0x4a, 0x49, 0xa4, 0x2c, 0xc2, 0x6e, 0xb5, 0x4f, 0x00, 0xbb,

```

```

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

```

```

// AES encryption S-box

```

```

int custom_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};

```

```

int len = sizeof(custom_sbox) / sizeof(custom_sbox[0]);

```

```

printf("custom S-box:\n");
for (int i = 0; i < len; i++) {
    printf("0x%x ", custom_sbox[i]);
}
printf("\n\n");

sboxNonlinearity(custom_sbox, len);

return 0;
}

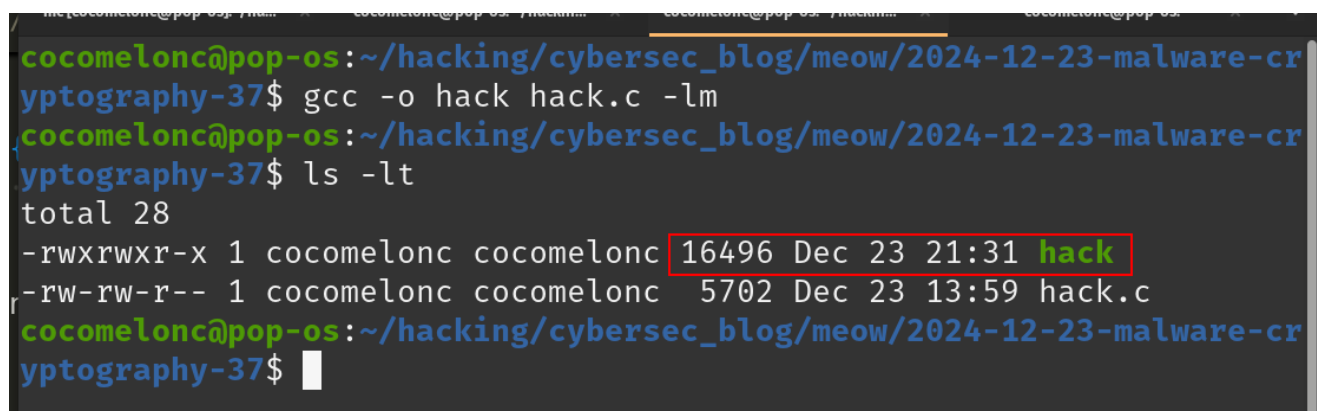
```

As you can see, the program prints the results for few example sequences. You can replace the sequences with your own to analyze other binary sequences.

demo

Let's go to see everything in action. Compile it:

```
gcc -o hack hack.c -lm
```



```

cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ gcc -o hack hack.c -lm
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ ls -lt
total 28
-rwxrwxr-x 1 cocomelonc cocomelonc 16496 Dec 23 21:31 hack
-rw-rw-r-- 1 cocomelonc cocomelonc 5702 Dec 23 13:59 hack.c
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$

```

Run it (AES S-box example):

```
./hack
```

```
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-23-malware-cryptography-37
0xab 0x76 0xca 0x82 0xc9 0x7d 0xfa 0x59 0x47 0xf0 0xad 0xd4 0xa2 0x
af 0x9c 0xa4 0x72 0xc0 0xb7 0xfd 0x93 0x26 0x36 0x3f 0xf7 0xcc 0x34
0xa5 0xe5 0xf1 0x71 0xd8 0x31 0x15 0x4 0xc7 0x23 0xc3 0x18 0x96 0x5
0x9a 0x7 0x12 0x80 0xe2 0xeb 0x27 0xb2 0x75 0x9 0x83 0x2c 0x1a 0x1b
0x6e 0x5a 0xa0 0x52 0x3b 0xd6 0xb3 0x29 0xe3 0x2f 0x84 0x53 0xd1 0x0
0xed 0x20 0xfc 0xb1 0x5b 0x6a 0xcb 0xbe 0x39 0x4a 0x4c 0x58 0xcf 0x
d0 0xef 0xaa 0xfb 0x43 0x4d 0x33 0x85 0x45 0xf9 0x2 0x7f 0x50 0x3c 0
x9f 0xa8 0x51 0xa3 0x40 0x8f 0x92 0x9d 0x38 0xf5 0xbc 0xb6 0xda 0x21
0x10 0xff 0xf3 0xd2 0xcd 0xc 0x13 0xec 0x5f 0x97 0x44 0x17 0xc4 0xa
7 0x7e 0x3d 0x64 0x5d 0x19 0x73 0x60 0x81 0x4f 0xdc 0x22 0x2a 0x90 0
x88 0x46 0xee 0xb8 0x14 0xde 0x5e 0xb 0xdb 0xe0 0x32 0x3a 0xa 0x49 0
x6 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 0x8 0xba
0x78 0x25 0x2e 0x1c 0xa6 0xb4 0xc6 0xe8 0xdd 0x74 0x1f 0x4b 0xbd 0x
8b 0x8a 0x70 0x3e 0xb5 0x66 0x48 0x3 0xf6 0xe 0x61 0x35 0x57 0xb9 0x
86 0xc1 0x1d 0x9e 0xe1 0xf8 0x98 0x11 0x69 0xd9 0x8e 0x94 0x9b 0x1e
0x87 0xe9 0xce 0x55 0x28 0xdf 0x8c 0xa1 0x89 0xd 0xbf 0xe6 0x42 0x68
0x41 0x99 0x2d 0xf 0xb0 0x54 0xbb 0x16

S-box nonlinearity (min, max): (112.000000, 112.000000)
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$
```

The AES S-box is a substitution-permutation matrix used to introduce nonlinearity. It is based on a finite field inversion followed by an affine transformation.

For S-box generated by Fisher-Yates randomization from previous post:

```
int custom_sbox[] = {0x4a, 0x49, 0xa4, 0x2c, 0xc2, 0x6e, 0xb5, 0x4f, 0x00, 0xbb,
0x8f, 0x75, 0x19, 0x56, 0x86, 0x77, 0x8e, 0xaa, 0x51, 0x01, 0xff, 0x6d, 0xcf, 0x14,
0x60, 0xc0, 0x7c, 0x22, 0xf4, 0xe0, 0xf5, 0xa, 0x44, 0x1c, 0x6a, 0x32, 0xe4, 0xad,
0xb4, 0xa8, 0xa3, 0xb2, 0x7f, 0xa0, 0x78, 0x80, 0xa2, 0x50, 0x8, 0x95, 0x8d, 0xaf,
0xd1, 0x10, 0xf8, 0xea, 0xd7, 0x0e, 0x5d, 0xd5, 0x43, 0x68, 0x7d, 0x29, 0xfd, 0x69,
0xd8, 0x59, 0x55, 0x97, 0x94, 0x73, 0xd9, 0xf9, 0xfb, 0x82, 0x31, 0xb9, 0x2f, 0x89,
0x76, 0xe8, 0x99, 0x88, 0x21, 0x7e, 0x2d, 0x92, 0xee, 0xa1, 0xae, 0xd0, 0x36, 0x42,
0x1a, 0x20, 0x03, 0x34, 0x2b, 0xcd, 0x66, 0x71, 0xcc, 0x3d, 0xd4, 0x2e, 0x40, 0xe9,
0x23, 0x70, 0x65, 0xcb, 0xb7, 0x33, 0xce, 0x9d, 0x18, 0xeb, 0x61, 0xa6, 0xfa, 0x3c,
0xbe, 0x53, 0xda, 0x9f, 0xd2, 0x90, 0xfc, 0xf1, 0xd3, 0xf0, 0x24, 0x12, 0xac, 0xbd,
0x5e, 0x4c, 0xc4, 0x38, 0xbf, 0x4e, 0xd6, 0xef, 0x3b, 0x8c, 0xc1, 0x2a, 0xb0, 0x4b,
0x06, 0x5f, 0x1e, 0x0c, 0x46, 0x13, 0xdf, 0xdd, 0x85, 0x57, 0xc3, 0xde, 0xe1, 0x8a,
0xbc, 0xc9, 0x9e, 0xca, 0x83, 0x72, 0x6f, 0xa5, 0x79, 0x54, 0x9b, 0x96, 0x93, 0x30,
0xe3, 0x1b, 0x39, 0xe5, 0x1f, 0x47, 0x3a, 0x64, 0xab, 0x9c, 0x63, 0xf2, 0x16, 0xe2,
0xfe, 0x87, 0xf7, 0x04, 0xdc, 0x5c, 0x7b, 0x15, 0x5a, 0xf6, 0x58, 0x6c, 0xc6, 0xb6,
0x5b, 0x1d, 0xb3, 0xc8, 0x25, 0x7a, 0xf3, 0x02, 0x3f, 0x0b, 0x37, 0x41, 0x3e, 0x4d,
0xe7, 0xc5, 0xc7, 0x67, 0x9a, 0x84, 0xb8, 0xd, 0x6b, 0x91, 0x11, 0x98, 0x8b, 0x35,
0xb1, 0x09, 0xdb, 0x28, 0x81, 0x17, 0x74, 0xed, 0x62, 0xa7, 0xa9, 0x07, 0x0f, 0xba,
0x27, 0x52, 0xe6, 0x05, 0x48, 0x45, 0x26, 0xec};
```

```
0x86 0x77 0x8e 0xaa 0x51 0x1 0xff 0x6d 0xcf 0x14 0x60 0xc0 0x7c 0x2
2 0xf4 0xe0 0xf5 0xa 0x44 0x1c 0x6a 0x32 0xe4 0xad 0xb4 0xa8 0xa3 0x
b2 0x7f 0xa0 0x78 0x80 0xa2 0x50 0x8 0x95 0x8d 0xaf 0xd1 0x10 0xf8 0
xea 0xd7 0xe 0x5d 0xd5 0x43 0x68 0x7d 0x29 0xfd 0x69 0xd8 0x59 0x55
0x97 0x94 0x73 0xd9 0xf9 0xfb 0x82 0x31 0xb9 0x2f 0x89 0x76 0xe8 0x9
9 0x88 0x21 0x7e 0x2d 0x92 0xee 0xa1 0xae 0xd0 0x36 0x42 0x1a 0x20 0
x3 0x34 0x2b 0xcd 0x66 0x71 0xcc 0x3d 0xd4 0x2e 0x40 0xe9 0x23 0x70
0x65 0xcb 0xb7 0x33 0xce 0x9d 0x18 0xeb 0x61 0xa6 0xfa 0x3c 0xbe 0x5
3 0xda 0x9f 0xd2 0x90 0xfc 0xf1 0xd3 0xf0 0x24 0x12 0xac 0xbd 0x5e 0
x4c 0xc4 0x38 0xbf 0x4e 0xd6 0xef 0x3b 0x8c 0xc1 0x2a 0xb0 0x4b 0x6
0x5f 0x1e 0xc 0x46 0x13 0xdf 0xdd 0x85 0x57 0xc3 0xde 0xe1 0x8a 0xbc
0xc9 0x9e 0xca 0x83 0x72 0x6f 0xa5 0x79 0x54 0x9b 0x96 0x93 0x30 0x
e3 0x1b 0x39 0xe5 0x1f 0x47 0x3a 0x64 0xab 0x9c 0x63 0xf2 0x16 0xe2
0xfe 0x87 0xf7 0x4 0xdc 0x5c 0x7b 0x15 0x5a 0xf6 0x58 0x6c 0xc6 0xb6
0x5b 0x1d 0xb3 0xc8 0x25 0x7a 0xf3 0x2 0x3f 0xb 0x37 0x41 0x3e 0x4d
0xe7 0xc5 0xc7 0x67 0x9a 0x84 0xb8 0xd 0x6b 0x91 0x11 0x98 0x8b 0x3
5 0xb1 0x9 0xdb 0x28 0x81 0x17 0x74 0xed 0x62 0xa7 0xa9 0x7 0xf 0xba
0x27 0x52 0xe6 0x5 0x48 0x45 0x26 0xec

S-box nonlinearity (min, max): (94.000000, 108.000000)
cocomelon@pop-os: ~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$
```

You can define your custom **S-box** as an array of integers in the main function, and the **sboxNonlinearity** function will calculate and print its minimum and maximum nonlinearity.

Python implementation:

```

def walshTransform(t):
    n = len(t) # Number of inputs
    wt = [0] * n # Initialize Walsh Transform output
    for w in range(n):
        for x in range(n):
            wt[w] += (-1) ** (t[x] ^ binaryInnerProduct(w, x))
    return wt

def binaryInnerProduct(a, b):
    """binary inner product of a and b (XOR of ANDed bits)."""
    ip = 0
    ab = a & b
    while ab > 0:
        ip ^= ab & 1
        ab >>= 1
    return ip

def nonLinearity(t):
    """compute nonlinearity of a binary sequence."""
    wt = walshTransform(t)
    max_wt = max(abs(w) for w in wt)
    return len(t) // 2 - max_wt / 2

def sbboxNonlinearity(sbox):
    """calculate min and max nonlinearity for an S-box."""
    n = len(sbox)
    nlv = [0] * (n - 1)
    for c in range(1, n): # Skip zero combination
        t = [binaryInnerProduct(c, sbox[i]) for i in range(n)]
        nlv[c - 1] = nonLinearity(t)
    return min(nlv), max(nlv)

# Example with a custom S-box
# custom_sbox = [0x3, 0x6, 0x5, 0xC, 0xA, 0x0, 0xE, 0x9, 0xD, 0x8, 0xF, 0x4, 0x7,
# 0x1, 0xB, 0x2]
# custom_sbox = [0x4a, 0x49, 0xa4, 0x2c, 0xc2, 0x6e, 0xb5, 0x4f, 0x00, 0xbb, 0x8f,
# 0x75, 0x19, 0x56, 0x86, 0x77, 0x8e, 0xaa, 0x51, 0x01, 0xff, 0x6d, 0xcf, 0x14, 0x60,
# 0xc0, 0x7c, 0x22, 0xf4, 0xe0, 0xf5, 0x0a, 0x44, 0x1c, 0x6a, 0x32, 0xe4, 0xad, 0xb4,
# 0xa8, 0xa3, 0xb2, 0x7f, 0xa0, 0x78, 0x80, 0xa2, 0x50, 0x08, 0x95, 0x8d, 0xaf, 0xd1,
# 0x10, 0xf8, 0xea, 0xd7, 0x0e, 0x5d, 0xd5, 0x43, 0x68, 0x7d, 0x29, 0xfd, 0x69, 0xd8,
# 0x59, 0x55, 0x97, 0x94, 0x73, 0xd9, 0xf9, 0xfb, 0x82, 0x31, 0xb9, 0x2f, 0x89, 0x76,
# 0xe8, 0x99, 0x88, 0x21, 0x7e, 0x2d, 0x92, 0xee, 0xa1, 0xae, 0xd0, 0x36, 0x42, 0x1a,
# 0x20, 0x03, 0x34, 0x2b, 0xcd, 0x66, 0x71, 0xcc, 0x3d, 0xd4, 0x2e, 0x40, 0xe9, 0x23,
# 0x70, 0x65, 0xcb, 0xb7, 0x33, 0xce, 0x9d, 0x18, 0xeb, 0x61, 0xa6, 0xfa, 0x3c, 0xbe,
# 0x53, 0xda, 0x9f, 0xd2, 0x90, 0xfc, 0xf1, 0xd3, 0xf0, 0x24, 0x12, 0xac, 0xbd, 0x5e,
# 0x4c, 0xc4, 0x38, 0xbf, 0x4e, 0xd6, 0xef, 0x3b, 0x8c, 0xc1, 0x2a, 0xb0, 0x4b, 0x06,
# 0x5f, 0x1e, 0x0c, 0x46, 0x13, 0xdf, 0xdd, 0x85, 0x57, 0xc3, 0xde, 0xe1, 0x8a, 0xbc,
# 0xc9, 0x9e, 0xca, 0x83, 0x72, 0x6f, 0xa5, 0x79, 0x54, 0x9b, 0x96, 0x93, 0x30, 0xe3,
# 0x1b, 0x39, 0xe5, 0x1f, 0x47, 0x3a, 0x64, 0xab, 0x9c, 0x63, 0xf2, 0x16, 0xe2, 0xfe,
# 0x87, 0xf7, 0x04, 0xdc, 0x5c, 0x7b, 0x15, 0x5a, 0xf6, 0x58, 0x6c, 0xc6, 0xb6, 0x5b,
# 0x1d, 0xb3, 0xc8, 0x25, 0x7a, 0xf3, 0x02, 0x3f, 0x0b, 0x37, 0x41, 0x3e, 0x4d, 0xe7,
# 0xc5, 0xc7, 0x67, 0x9a, 0x84, 0xb8, 0x0d, 0x6b, 0x91, 0x11, 0x98, 0x8b, 0x35, 0xb1,

```

```
0x09, 0xdb, 0x28, 0x81, 0x17, 0x74, 0xed, 0x62, 0xa7, 0xa9, 0x07, 0x0f, 0xba, 0x27,
0x52, 0xe6, 0x05, 0x48, 0x45, 0x26, 0xec]
```

```
# custom_sbox = [
```

```
#      14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7,
#      0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
#      4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
#      15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13
# ];
```

```
custom_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]
```

```
print("custom S-box nonlinearity (min, max) =", sboxNonlinearity(custom_sbox))
```

Run it via command:

```
python3 sbox_nonlinearity.py
```

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ python3 sbbox_nonlinearity.py
custom S-box nonlinearity (min, max) = (112.0, 112.0)
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ █
```

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

examples

Twofish uses precomputed S-boxes based on key-dependent transformations. Below is one of its default **S-boxes** (from the specification):

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

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ python3 sbbox_nonlinearity.py
custom S-box nonlinearity (min, max) = (96.0, 108.0)
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$
```

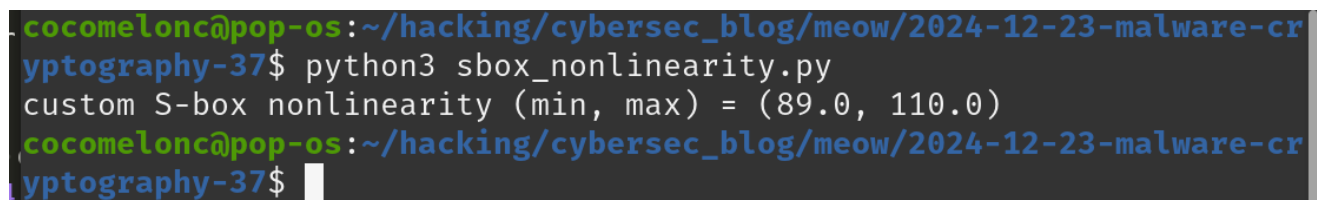
SM4 (a Chinese block cipher standard) has a 256-entry S-box, similar to AES. Here is the SM4 S-box:

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

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ python3 sbbox_nonlinearity.py
custom S-box nonlinearity (min, max) = (112.0, 112.0)
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$
```

Camelia cipher:

```
unsigned char custom_sbox[256] = {
    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, 200, 248, 87, 120, 7, 46, 182, 65, 131, 30, 94, 184, 212,
    140, 67, 128, 157, 158, 63, 174, 143, 122, 15, 2, 49, 160, 83, 153, 79,
    37, 150, 240, 117, 169, 74, 76, 219, 91, 208, 119, 207, 103, 178, 47, 138,
    32, 160, 203, 125, 20, 232, 98, 8, 181, 173, 102, 210, 178, 252, 50, 233,
    249, 156, 109, 159, 80, 19, 45, 147, 243, 16, 173, 17, 117, 54, 215, 132,
    96, 230, 218, 166, 209, 57, 59, 141, 88, 84, 152, 41, 195, 13, 90, 4,
    42, 70, 252, 38, 237, 72, 118, 24, 76, 129, 181, 211, 243, 33, 91, 170,
    51, 164, 87, 178, 194, 26, 109, 214, 224, 192, 149, 172, 230, 222, 148, 110,
    177, 45, 75, 191, 107, 226, 145, 58, 198, 108, 111, 64, 239, 221, 55, 18,
    201, 231, 134, 236, 190, 40, 185, 42, 158, 189, 146, 0, 48, 93, 228, 200,
    99, 25, 120, 170, 199, 100, 23, 98, 233, 153, 68, 161, 246, 72, 182, 167,
    122, 236, 135, 104, 158, 97, 27, 112, 169, 29, 20, 38, 11, 101, 89, 141
};
```



```
cocomelon@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$ python3 sbox_nonlinearity.py
custom S-box nonlinearity (min, max) = (89.0, 110.0)
cocomelon@pop-os:~/hacking/cybersec_blog/meow/2024-12-23-malware-cr
yptography-37$
```

minimum and maximum of non-linearity

Minimum nonlinearity: Represents the weakest point of the **S-box**. If an **S-box** has a low minimum nonlinearity, it means *at least one linear function approximates it more closely than others*. For example, if the minimum nonlinearity is low (e.g., 4), the **S-box** is *weaker* in resisting attacks in some specific linear approximations.

Maximum nonlinearity: Represents the strongest deviation from any linear function among all evaluated inputs. This value is typically close to the theoretical maximum for a strong **S-box**. A high maximum nonlinearity indicates that the **S-box** has *strong points that are highly resistant to linear approximations*. If the maximum nonlinearity is high (e.g., 112 for a typical **S-box** of size 256, (AES,)), the **S-box** has good resistance overall.

For an **8-bit S-box** (e.g., **AES**) theoretical maximum nonlinearity is 112. Good **S-box** designs often achieve nonlinearity values close to 112 across most inputs, with the minimum nonlinearity not falling significantly below that.

For example, if your **S-box** has minimum nonlinearity = 104, and maximum nonlinearity = 112, it means that the weakest linear combination of outputs can be approximated with a deviation of 104, which is still quite strong. The strongest deviation is 112, showing the **S-box** performs well in resisting linear approximations.

conclusion

Minimum nonlinearity shows the S-box's worst-case resistance to linear approximations. Maximum nonlinearity shows the best-case deviation from linearity. An ideal cryptographic S-box has a high minimum and maximum nonlinearity, both close to the theoretical maximum. This ensures consistent resistance across all possible attacks.

I wonder if there is a nonlinearity impact to, for example, entropy and, accordingly, with the level of AV/VirusTotal detection?

In theory: S-boxes with higher nonlinearity typically produce more uniform ciphertext distributions, increasing entropy. If a weak S-box is generated, the ciphertext may exhibit patterns that reduce entropy, potentially exposing the payload to statistical attacks or aiding detection by security tools.

Adds variability to each encryption round, making reverse engineering more difficult.

But all this is in theory, and I need to test it in practice.

In the following posts we will consider how this nonlinearity affects AV/VirusTotal detection on payload, traffic encryption or ransomware simulation.

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

[Walsh Transform: wikipedia](#)

[Measuring Boolean Function Nonlinearity by Walsh Transform](#)

[Marcin Kontak, Janusz Szmidt: Nonlinearity of the Round Function](#)

[Linear cryptanalysis](#)

[SM4 cipher](#)

[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