Cryptography – Summer 2023

by Prof. Dr. Alexander Wiesmaier

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Praktikum 02

Date:	09.05.2023		
Supervisor:	Prof. Wiesmaier		
Students:	Réault	Corentin	Matr.Nr.1123639
	Bazan	Pablo	Matr.Nr.1120012

Task 1 (Simple AES programming)

a) Writing a small program that computes the inverse S-box function

A simple function has been created to return the S-Box array: the values for each given index become indexes of the inverse S-Box and their indexes, the new values.

```
void init_inv_sbox() {
    if (INV_SBOX == NULL) {
        INV_SBOX = (unsigned char*)malloc(256 * sizeof(unsigned char));
        unsigned char* tmp_inv_sbox = calculate_inv_sbox();
        memcpy(INV_SBOX, tmp_inv_sbox, 256 * sizeof(unsigned char));
        delete[] tmp_inv_sbox;
    }
}
unsigned char* decrypt(const unsigned char* in,
                       const unsigned char* roundKey) {
    unsigned char* out = new unsigned char[32];

    init_inv_sbox();

    /*std::cout << std::endl << "INV_SBOX:" << std::endl;
    for (int i = 0; i < 256; i++) {
        std::cout << std::hex << std::setw(2) << std::setfill('0')
        << static_cast<int>(INV_SBOX[i]) << " ";
        if ((i + 1) % 16 == 0) {
```

```
            std::cout << std::endl;
        }
    }
    */

    decipher(in, roundKey, out);

    std::cout << std::endl << "Text after decryption:" << std::hex
    << std::endl;
    for (unsigned int i = 0; i < 16; ++i)
      std::cout << "0x" << std::setw(2) << std::setfill('0')
      << (unsigned int)out[i] << ", ";
    std::cout << std::endl;

    return out;
}
int main(int argc, char* argv[])
{
  unsigned char roundKey[240];

  // Sample
  {
    // Part1
    const unsigned char in[16] =
    {
      'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
      'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'
    };
    const unsigned char key[16] =
    {
      0xa3, 0x28, 0x4e, 0x09, 0xc6, 0xfe, 0x53, 0x29,
      0x97, 0xef, 0x6d, 0x10, 0x74, 0xc3, 0xde, 0xad
    };
    keyExpansion(key, roundKey);
    decrypt(encrypt(in, roundKey), roundKey);
  }
  return 0;
}
```

b) Validating our inverse S-box

```
INV_SBOX:
52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

Abbildung 1: Presentation of the inverse S-Box

```
[cocopops@cocopops-laptop Practical2]$ ./simple-aes

SBOX:
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

Text before encryption:
0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,

Text after encryption:
0x95, 0x19, 0x42, 0x68, 0xff, 0x11, 0x7f, 0xc8, 0x9f, 0x75, 0x5b, 0x96, 0x97, 0x9f, 0x24, 0x0b,

Text after decryption:
0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
[cocopops@cocopops-laptop Practical2]$ ./simple-aes

SBOX:
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

Text before encryption:
0x7a, 0x79, 0x78, 0x77, 0x76, 0x75, 0x74, 0x73, 0x72, 0x71, 0x70, 0x6f, 0x6e, 0x6d, 0x6c, 0x6b,

Text after encryption:
0x02, 0x50, 0x7f, 0x4b, 0xdb, 0x71, 0x9c, 0xb4, 0x87, 0x27, 0x31, 0xa0, 0xf5, 0x48, 0x94, 0x04,

Text after decryption:
0x7a, 0x79, 0x78, 0x77, 0x76, 0x75, 0x74, 0x73, 0x72, 0x71, 0x70, 0x6f, 0x6e, 0x6d, 0x6c, 0x6b,
[cocopops@cocopops-laptop Practical2]$
```

Abbildung 2: Demonstration of ciphering and deciphering with the inverse S-Box

Task 2 (Simple AES cracking)

a) Computing the number of possible keys and estimating the expected amount of time to retrieve the password

There are only 128 bits in the key and we know the first bytes of the key: 81596bfb39c62b716e52db91 that is the first 26 hexadecimal characters of the key. Knowing that a hexadecimal character corresponds to 16 possibilities, that is to say 2^4 (0 to 15) so also 4 bits. So we have $128 - (26 * 4) = 24$ bits left to find on the original 128. These 24 bits represent 2^{24} = 16777216 possibilities.

To estimate the time it would take for a brute-force attack to recover the password, we need to know the processing speed of our computer and the efficiency of our brute-force attack algorithm. We have written a program to test the speed of execution of a billion instructions and deduce the average speed of execution of an instruction. Thus we deduced that an instruction took on average 0.002275 microseconds.

Here is the code used to perform this test:

```cpp
#include <iostream>
#include <chrono>

int main() {
  int num_instructions = 1000000000;
  int a = 0;

  auto start = std::chrono::high_resolution_clock::now();

  for (int i = 0; i < num_instructions; i++) {
    a = a + 1;
  }

  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast
  <std::chrono::microseconds>(end - start).count();

  std::cout << "Total execution time: " << duration
  << " microseconds\n";
  std::cout << "Average execution time per instruction: "
  << duration / static_cast<double>(num_instructions)
  << " microseconds\n";

  return 0;
}
```

```
[cocopops@cocopops-laptop Practical2]$ ./test-execution-time
Total execution time: 2274630 microseconds
Average execution time per instruction: 0.00227463 microseconds
[cocopops@cocopops-laptop Practical2]$
```

Abbildung 3: Using a script to determine the average execution time

Then we calculated the complexity of the program to decrypt a ciphertext, the global complexity being equal to :

$$total_{searchcomplexity} = 2^k * (complexity_{keyhexpansion} + complexity_{decipher}) \tag{1}$$

$$complexity_{keyhexpansion} = O(n) + O(nm) \tag{2}$$

$$complexity_{addroundkey} = O(n^2) \tag{3}$$

$$complexity_{invsubbytes} = O(n^2) \tag{4}$$

$$complexity_{shiftrows} = O(1) \tag{5}$$

$$complexity_{invmixcolumns} = O(n) \tag{6}$$

$$\begin{gathered}
complexity_{decipher} = O(n^2) + complexity_{addroudkey} + \\
O(l) * (complexity_{invshiftrows} + complexity_{invsubbytes} + \\
complexity_{addroundkey} + complexity_{invmixcolumns}) + \\
complexity_{invshiftrows} + complexity_{invsubbytes} + \\
complexity_{addroundkey} + O(n^2) \\
complexity_{decipher} = O(n^2) + O(n^2) + O(l) * (O(n^2) + O(n^2) + O(n)) \\
+ O(1) + O(n^2) + O(n^2) \\
complexity_{decipher} = O(n^2 l) + O(5n^2)
\end{gathered} \tag{7}$$

So we have:

$$total_{searchcomplexity} = 2^k * (O(n) + O(nm) + O(n^2 l) + O(5n^2)) \tag{8}$$

with: k=24, n=4, m=40 and l=10

$$\begin{gathered}
total_{searchcomplexity} = 2^{24} * (4 + 4 * 40 + 4^2 * 10 + 6 * 4^2) \\
total_{searchcomplexity} = 6.778 * 10^9 instructions
\end{gathered} \tag{9}$$

If we use this result with the average execution time of an instruction, we obtain the following result:

$$\begin{aligned} total_{executiontime} &= total_{searchcomplexity} * average_{executiontime} \\ total_{executiontime} &= 6.778 * 10^9 * 0.002275 * 10^{-3} \\ total_{executiontime} &= 15420s = 257mn \ = 4hours20mn \end{aligned} \tag{10}$$

But this result is questionable because the instructions will not all be processed one after the other, there will most likely be instructions executed in parallel. If we take into account that the machine has 6 cores, and 12 threads, then we can assume that the result will be more like:

$$\begin{aligned} total_{executiontime,threaded} &= total_{executiontime}/12 \\ total_{executiontime,threaded} &= 1285s \ = 21mn \end{aligned} \tag{11}$$

b) Password retrieval

Here is the code realized to find the last 6 hexadecimal values of the secret key :

```
bool isAlphaNumHex(unsigned char* in, int size) {
  for (int i = 0; i < size; i++) {
    if (!((in[i] >= 0x61 && in[i] <= 0x7a) || (in[i] == 0x2e))) {
      std::cout << std::endl << "Character not in the list: "
      << "0x" << std::setw(2) << std::setfill('0')
      << (unsigned int)in[i] << ", text: " << in[i] << std::endl;
      return false;
    }
  }
  return true;
}

int countNonNull(unsigned char* arr, unsigned int size) {
    int count = 0;
    for(unsigned int i = 0; i < size; i++) {
        if(arr[i] != 0) {
```

```
            count++;
        }
    }
    return count;
}

int main(int argc, char* argv[])
{
  unsigned char roundKey[240];

  // Sample
  {
        // Part1
        // .....

    // Part 2

    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    auto t1 = high_resolution_clock::now();
    const unsigned char ciphertext[16] =
    {
      0xbf, 0x3f, 0xb7, 0x7d, 0x93, 0xdd, 0x6c, 0xfd,
      0xef, 0xb8, 0x82, 0x2b, 0x82, 0xd0, 0x35, 0x8a
    };

    unsigned char key[16] =
    {
      0x81, 0x59, 0x6b, 0xfb, 0x39, 0xc6, 0x2b, 0x71,
      0x6e, 0x52, 0xdb, 0x91, 0x81, '\0', '\0', '\0'
    };
    std::cout << std::endl << "The original key:"
    << std::hex << std::endl;
    for (unsigned int i = 0; i < 16; ++i) {
      std::cout << "0x" << (unsigned int)key[i] << ", ";
    }
    std::cout << std::endl;
```

```
unsigned char** plaintext_list = new unsigned char*[1000];
for (int i = 0; i < 1000; i++) {
  plaintext_list[i] = new unsigned char[32];
}
for (int i = 0; i < 1000; i++) {
  for (int j = 0; j < 16; j++) {
    plaintext_list[i][j] = '0';
  }
}
unsigned char** key_list = new unsigned char*[1000];
    for (int i = 0; i < 1000; i++) {
    key_list[i] = new unsigned char[32];
}
for (int i = 0; i < 1000; i++) {
            for (int j = 0; j < 16; j++) {
        key_list[i][j] = '\0';
    }
}

for(int i = 13; i < 16; i++) {
    key[i] = 0x00;
}

int index = 0;
int nb_try = 0;
bool done = false; //to stop the for loops

for(unsigned int a = 0x01; a < 0xff; a++) {
  if (done) break;
  for(unsigned int b = 0x01; b < 0xff; b++) {
        if (done) break;
    for(unsigned int c = 0x00; c < 0xff; c++) {
      nb_try++;
      std::cout << std::endl << "The␣tested␣key:" << std::hex
      << std::endl;
      for (unsigned int i = 0; i < 16; ++i) {
        std::cout << "0x" << std::setw(2) << std::setfill('0')
        << (unsigned int)key[i] << ",␣";
      }
      std::cout << std::endl;
```

```
      keyExpansion(key, roundKey);
      unsigned char* decrypted = decrypt(ciphertext, roundKey);
      if (isAlphaNumHex(decrypted, 16)) {
        std::cout << std::endl
        << "##################################################"
        << std::endl << "Matched␣text" << std::endl
        << "##################################################"
        << std::endl ;
        plaintext_list[index] = decrypted;
        key_list[index] = key;
        index++;
        // stop the for loop
        done = true;
        break;
      }
      std::cout << std::endl
      << "------------------------------------------------"
      << std::endl;
      key[15] = c;
    }
    key[14] = b;
  }
  key[13] = a;
}
auto t2 = high_resolution_clock::now();

for(int i=0; i<1000; i++) {
  if((key_list[i][0]=='\0')||(plaintext_list[i][0]=='\0')){
    break;
  }

  std::cout << std::endl << "Key␣value:" << std::hex << std::endl;
  for(int j=0; j<16; j++) {
    std::cout << "0x" << std::setw(2) << std::setfill('0')
    << (unsigned int)key_list[i][j] << "␣";
  }

  std::cout << std::endl << "Plaintext:" << std::hex << std::endl;
  for(int j=0; j<16; j++) {
    std::cout << plaintext_list[i][j] << "␣";
  }
```

```
      std::cout << std::endl;
    }

    duration<double, std::milli> ms_double = t2 - t1;
    std::cout << "Execution time: " << ms_double.count() << "ms\n"
    << std::endl;
    std::cout << "Number of tries: " << std::dec << nb_try << std::endl;
  }

  return 0;
}
```

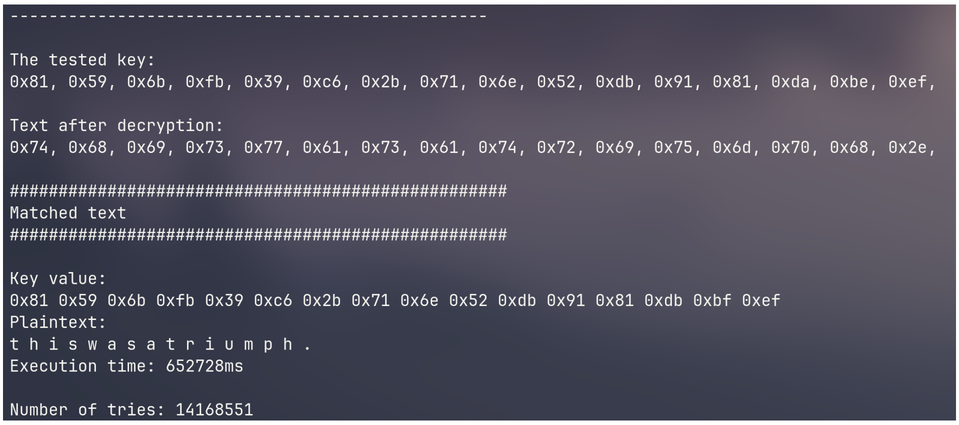

Abbildung 4: Use of the modified program to obtain the password

The key has been found thanks to the improvements made to the program.

Finding the key will have required 14 168 551 key attempts (out of 16 777 216 possible) and 652.728 seconds, that is 11 minutes.
This represents an average of 21,706 keys per second, and by simply using the rule of three, we get a theoretical time of almost 13 minutes to test all the keys. However, we have previously calculated a theoretical duration of 21 minutes, which corresponds to an error of about 38 percent:

$$error_{theoritical,experience} = (13 - 21)/21 = -0.3810 \tag{12}$$

This difference could be due to the maximum clock boost that brings the cpu clock frequency from 3.3 to 4.2 GHz in our case (i.e. a difference of about 27 percent) that may have been used more in the experiment compared to the test due to the lower iteration number in the average execution test software.
Moreover, our approximations in our calculations are also at the origin of errors:
- The code realized to calculate the average execution time of an instruction only

iterates on ä = a + 1;", which does not really reflect our use case
- The computed complexity is by definition an approximation, it is not equal to the real number of realized instructions
- We assumed that the execution time would be divided by 12 because there are 12 cores but this may not reflect the real issues of parallelization in our experiment