Cybersecurity MSc: Applied Cryptography 2024-25

1st Project: Shuffled AES (S-AES)

October 25, 2024 Due date: November 10, 2024

Changelog

- v1.0 - Initial version.
- v1.1 - The deadline was corrected to November 10.

1 Introduction

Rijndael is the algorithm that won the context to design AES. But the former is much flexible than AES, namely it could use more data block sizes and key sizes (from 128 up to 256, with 16 bit increments, for both). Also, the authors of Rijndael stated that its S-Box, which has a special construction, known as a Nyberg S-Box, is not critical for the security of the algorithm, and it can be replaced by other S-Boxes. This is a disclaimer that is often given to assure people that there is no special trapdoors hidden within S-Boxes.

2 Homework

The work consists on implementing a shuffled version of AES (S-AES), which will be similar to AES but with an extra 128-bit key (shuffling key, SK). For that, you can consider AES-128. Thus, the resulting algorithm will operate with an overall 256-bit key.

S-AES will operate as AES, but with two differences:

- shuffled round keys;
- One modified round.

All $N+1$ round keys should be shuffled deterministically based on 64 bits of the SK. You must use all those bits to create a reasonably pseudo-random permutation of the round keys. You can shuffle both the order of the keys and the bytes within each round key.

Regarding the round to be modified, consider that you modify an encryption round; then, modify the decryption sequence accordingly. That round will be selected with SK, and exclusively among the first 9 (encryption) rounds. The last round (10th) cannot be selected since it is different from the others (otherwise, it would be feasible to get some SK bits from timing measurements). The round selection should be as equally likely as possible[1].

The modified round must use the other 64 bits of SK to do the following:

- In the **AddRoundKey** step, the SK should be XORded with the entire round key. Note that the round key has 128 bits, while you use only 64 bits from SK. Thus, each of these last bits must be used twice.

[1] Since 9 is not a power of 2, we cannot select any round from SK with an exactly equal probability.

- The S-Box used in the **SubBytes** step should be a shuffled variant of the original S-Box. That shuffling should normally change the relative order of the S-Box bytes with SK. At least one-half (50%) of the S-Box bytes should change their position.

The new values used in the modified round should be computed only once, during the key set-up, in order to speed-up encryptions and decryptions. Do not forget that you need to pre-compute as well the inverse of the modified S-Box (but not from SK, but instead from the modified S-Box).

Besides implementing S-AES in a module (or library), you should implement two applications, `encrypt` and `decrypt`. These should receive two textual passwords as arguments, which will be separately used to generate the two 128-bit keys of S-AES – the normal AES key and SK, in this order. In the absence of the second parameter (SK), the program should operate with the normal AES.

The applications should process the input from `stdin` and produce a result to `stdout`. S-AES (or AES) should be used to process the input in ECB mode with a PKCS#7 padding.

Create a third application, `speed`, to evaluate the relative performance of your S-AES implementation and one or more library implementation of AES. For that, allocate a 4 kB buffer (a memory page), fill it with random values (you can use `/dev/urandom` for that), and evaluate the time it takes to encrypt and decrypt the buffer with AES (from the library) and S-AES (both in ECB mode). Perform at least 100 000 measurements of each operation, and present the lowest values observed for each (i.e. the maximum achievable speed). For each measurement, use new, random keys. For accurate timing you can use the Linux system call `clock_gettime` function, which provides a nanosecond precision. Note that the measurements must encompass only encryptions and decryptions, and not AES or S-AES key-related set-up operations.

2.1 Library implementations of AES in Linux

In C, you can use these library implementations:

- The OpenSSL crypto library;
- The Nettle library.

In C++, you can use this library implementations:

- The Crypto++ library.

In Java, you can use these library implementations:

- The Java Runtime Environment;
- IAIK JCE;
- Bouncy Castle Crypto Library.

In Python, you can use these library implementations:

- Cryptography;
- PyCrypto.

2.2 S-AES implementation with AES-NI

In order to have the minimum possible performance penalty when switching from an AES library version using AES-NI to S-AES, you can implement the latter using also AES-NI Intel assembly instructions, which are the following:

`AESKEYGENASSIST xmm1, xmm2/m128u, imm8`	Assist in round key generation using an 8-bit constant (`imm8`) with a 128-bit key specified in `xmm2/m128` and stores the result in `xmm1`. This instruction computes a value that needs to be combined with the key to generate the round key (presented below).
`AESENC xmm1, xmm2/m128`	Perform one encryption round (except the last one) over `xmm1` with the round key `xmm2/m128`.
`AESENCLAST xmm1, xmm2/m128`	Perform the last encryption round over `xmm1` with the round key `xmm2/m128`.
`AESDEC xmm1, xmm2/m128`	Perform one decryption round (except the last one) over `xmm1` with the round key `xmm2/m128`.
`AESDECLAST xmm1, xmm2/m128`	Perform the last decryption round over `xmm1` with the round key `xmm2/m128`.
`AESIMC xmm1, xmm2/m128`	Perform a transformation of a round key (1 to 9) to prepare it to be used in a decryption flow.

`xmm1` and `xmm2` are 128-bit registers, `m128` is the address of a 128-bit value and `imm8` is an 8-bit constant integer.

The GCC compiler has inline C functions for wrapping these AES-NI instructions. Those functions become available with the `-maes` compilation flag, and their prototype is the following:

```
// __m128i is a 128-bit integer type.

__m128i _mm_aeskeygenassist_si128 ( __m128i key, uint8_t const index );
__m128i _mm_aesenc_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesenclast_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesdec_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesdeclast_si128 ( __m128i input, __m128i rk );
__m128i _mm_aesimc_si128 ( __m128i input );
```

The prototype of these functions is provided by the `wmmintrinc.h` file which exists under the GCC installation directory (at `/usr/lib/gcc/x86_64-linux-gnu/[GCC major version]/include`).

The AES encryption flow with AES-NI is as follows:

```
x = x ^ RK[0];
x = aesenc ( x, RK[1] );
x = aesenc ( x, RK[2] );
...
x = aesenc ( x, RK[9] );
x = aesenclast ( x, RK[10] );

// aesenc = AddRoundKey( MixColumns( ShiftRows ( SubBytes( x ) ) ), RK )
```

and the inverse, decryption flow is as follows:

```
// RK'[i] = aesimc( RK[i] ) for i in [1-9]

x = x ^ RK[10];
x = aesdec ( x, RK'[9] );
x = aesdec ( x, RK'[8] );
...
x = aesdec ( x, RK'1] );
x = aesdeclast ( x, RK[0] );

// aesdec = AddRoundKey( InvMixColumns( InvSubBytes( InvShiftRows( x ) ) ), RK' )
```

The setup of round keys is performed from the original key and the output of `AESKEYGENASSIST`, and can be implemented in C with intrinsic functions as follows:

```
// key is the main key
// index is the output of AESKEYGENASSIST

// _mm_shuffle_epi32, _mm_slli_si128 and _mm_xor_si128 are GCC internal, inline functions

__m128i get_round_key (__m128i key, __m128i index )
{
    __m128i tmp;

    index = _mm_shuffle_epi32( index, 0xff );
    tmp   = _mm_slli_si128( key, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    tmp   = _mm_slli_si128( tmp, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    tmp   = _mm_slli_si128( tmp, 0x4 );
    key   = _mm_xor_si128( key, tmp );
    key   = _mm_xor_si128( key, index );

    return key; // Round key
}
```

3 Evaluation

The project will be evaluated as follows:

- Implementation of S-AES (in any language): 35%;
- Implementation of the applications: 5% for `encrypt`, 5% for `decrypt`, 10% for `speed`;
- Implementation of S-AES with AES-NI assembly instructions: 15%;
- Written report, with a complete explanations of the strategies followed and the results achieved: 30%

4 Homework delivery

Send your code to the course teachers through Elearning (a submission link will be provided). Include a small report, with no more than 10 pages, describing the decisions and strategies taken and implementation examples (not a complete copy of the code developed!) to illustrate your text.

Every piece of code imported from anywhere must be stated in the report and in the code itself. Failure to do so will be penalised.

5 References

- *Intel® Advanced Encryption Standard (AES) New Instructions Set*, Shay Gueron, White Paper, 2012, `https://www.intel.com/content/dam/develop/external/us/en/documents/`

aes-wp-2012-09-22-v01-165683.pdf

- *Advanced Encryption Standard (AES)*, Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, James F. Dray Jr., NIST FIPS 197, 2001, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf