Shuffle AES (S-AES)

tomas.matos@ua.pt, 108624
goncalomf@ua.pt, 107853

November 10, 2024

1 Introduction

This project is the first challenge given in the Applied Cryptography course from the Cybersecurity MSc at the University of Aveiro. The main objective is to develop a Shuffling AES capable of using shuffling keys to create different rounds in a 10-round AES encryption. The detailed assignment can be found here.

2 Generation of Keys

To start the project, we generate two keys: a shuffling key (SK) and an AES key based on two passwords. Using Password-Based Key Derivation Function 2 (PBKDF2), we generate a 128-bit AES key derived from a password, using a specified salt and SHA-256 hash. We also generate a 128-bit shuffling key with a different password from the AES key.

Listing 1: Key Generation Implementation in C

```c
void generate_keys_from_passwords(const char* password1, const char* password2, char*
    aes_key, char* sk) {
    unsigned char salt[8] = {0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0};
    int iterations = 10000;

    PKCS5_PBKDF2_HMAC(password1, strlen(password1), salt, sizeof(salt), iterations,
        EVP_sha256(), 16, aes_key);
    PKCS5_PBKDF2_HMAC(password2, strlen(password2), salt, sizeof(salt), iterations,
        EVP_sha256(), 16, sk);
}
```

3 AES

To understand the S-AES better, we first need to explain how AES works. AES (Advanced Encryption Standard) is a symmetric encryption algorithm that operates on a block of data, and it uses a key of 128, 192, or 256 bits. Depending on the key size, AES runs for 10, 12, or 14 rounds. These are the four sub-operations and the way they are gathered in order to cipher:

Operation	**Description**
SubBytes	Each byte is replaced using a predefined lookup table (S-Box), providing non-linear substitution to enhance security and resist cryptanalysis.
ShiftRows	Rows in the 4x4 matrix are shifted left by different offsets: • Row 1: No shift • Row 2: Shifted by 1 position • Row 3: Shifted by 2 positions • Row 4: Shifted by 3 positions
MixColumns	Each column in the matrix is transformed by multiplying it with a fixed polynomial modulo $x^4 + 1$, ensuring diffusion across the state matrix.
AddRoundKey	The state matrix is XORed with the round key (a part of the expanded AES key). This is the only operation directly involving the encryption key.
Key Expansion	The original AES key is expanded into multiple round keys using the Rijndael key schedule. The process involves: • Generating $N+1$ sets of round keys, where N is the number of rounds (10, 12, or 14). Applies the Rcon (Round Constant) for additional randomness.

Table 1: AES Operations Breakdown

AES Structure	**Operations**
Key Expansion	Original AES key is expanded into multiple round keys using the Rijndael key schedule.
Initial Round	AddRoundKey (with the initial round key from Key Expansion)
Main Rounds (9, 11, or 13 times)	SubBytes $\rightarrow$ ShiftRows $\rightarrow$ MixColumns $\rightarrow$ AddRoundKey (using round keys from Key Expansion)
Final Round (no MixColumns)	SubBytes $\rightarrow$ ShiftRows $\rightarrow$ AddRoundKey (using final round key)

Table 2: AES Cipher Structure with Key Expansion

AES Decryption Structure	**Operations**
Key Expansion	The original AES key is expanded into multiple round keys using the Rijndael key schedule, but in reverse.
Final Round (no Inverse MixColumns)	Inverse ShiftRows $\rightarrow$ Inverse SubBytes $\rightarrow$ AddRoundKey (using final round key)
Main Rounds (9, 11, or 13 times)	Inverse MixColumns $\rightarrow$ Inverse ShiftRows $\rightarrow$ Inverse SubBytes $\rightarrow$ AddRoundKey (using round keys from Key Expansion)
Initial Round	AddRoundKey (with the final round key from Key Expansion)

Table 3: AES Decryption Structure with Key Expansion

4 S-AES

The Shuffling AES (S-AES) builds on standard AES by introducing a key shuffling step. This key shuffling affects 1 of the 10 rounds of the SAES and key shuffling as well. The first 64 bits are used to create a pseudo-random permutation on the round keys. The other 64 bits are used to do a modified AddRoundKey (where SK XORs the entire round key) and to create a variant of the original SBox.

4.1 Selected Round

Starting by how the the selected round is selected. Using the first 2 bytes of SK, we shift left the first byte of SK and then realize a bitwise OR operation to combine in a 16-bit value. That value is then passed through a mod 10 to obtain a round between 0-9. This is a way to make sure we obtain the same round once we use the same SK. This method also goes with what was asked about the equal possibillity between every number.

Listing 2: Selecting the round in C

```
void select_round(unsigned char *SK, unsigned int *selected_round){
    unsigned char val = (SK[0] << 8) | SK[1];
    *selected_round = val % 10;

}
```

4.2 Shuffle S-box

Part of the modified selected round is creating a shuffled sbox that uses the original AES standard sbox to build a variant of the sbox using the SK to change the order of the bytes. Using this, at least half of the sbox bytes must change their position. To make this variant, we use the Fisher-Yates Shuffle method adapted to this situation. This method as granted always at least 250 bytes with their position changed.

Listing 3: Fisher-Yates shuffle to generate a key-dependent S-Box in C

```
void shuffle_sbox(unsigned char *sbox, unsigned char *key) {
      // Create a seed based on the key bytes
    uint64_t seed = 0;
    for (int i = 0; i < 16 && i < 8; i++) {
        seed = (seed << 8) | key[i];
    }

    for (int i = 255; i > 0; i--) {
        seed = (seed * 6364136223846793005ULL + 1); // LCG parameters

        int j = seed % (i + 1);

        unsigned char temp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = temp;
    }
}
```

4.3 Pseudo-Random Permutation of Round Keys

To perform the asked Pseudo-Random Permutation on the round keys, we also used a key dependent method on the first 64 sk bits very similiar to the used to generate the shuffled s-box (Fisher-Yates), selecting a pseudo-random index in j in the key based on sk and shuffle it.

Listing 4: Fisher-Yates shuffle to a round key in C

```
void getPseudoRandomPermo( unsigned char *sk, unsigned char *roundKey) {
    int roundKeyLength = 16;
    int skLength = 8;

    for (int i = roundKeyLength - 1; i > 0; i--) {

        int j = sk[i % skLength] % (i + 1);  // Ensure j is within [0, i]


        unsigned char temp = roundKey[i];
        roundKey[i] = roundKey[j];
        roundKey[j] = temp;
    }
}
```

4.4 Encryption

The order of the encryption of the S-AES is:

- Select the Modified Round
- Shuffle The S-Box
- Create The Expanded Key
  - It takes an initial encryption key and generates a larger set of keys (round keys), which are used in each round of the AES encryption algorithm.
  - Initially, it copies the original key into the ‘expandedKey‘ array.
  - It then iteratively generates additional bytes until the expanded key reaches the required length (‘nrkeychars‘).
  - During this process, every set of 4 bytes is derived from the previous 4 bytes in the expanded key.
- Starts the Encryption Rounds
  - Pseudo-Random Permutation of the 1st Round Key
  - 1st round: AddRoundKey

Listing 5: First Encryption Round of S-AES in C

```
        getPseudoRandomPermo(s1, roundKey);

        addRoundKey(block, roundKey);
```

- All other rounds are normal AES Encryption unless is the modified round, always executing the pseudo-random permutation of the N round key.

Listing 6: Encryption Rounds of S-AES in C

```
        getPseudoRandomPermo(s1, roundKey);

        if (i == selected_round) {

            s_subBytes(block, ssbox);
            shiftRows(block);
            mixColumns(block);
            s_addRoundKey(block, roundKey, sk);
        } else {
            subBytes(block);
            shiftRows(block);
            mixColumns(block);
            addRoundKey(block, roundKey);
        }
```

Listing 7: SubBytes of the Modified S-AES Round in C

```
        void s_subBytes(unsigned char *state, unsigned char *sbox)
        {
            for (int i = 0; i < 16; i++) {
                state[i] = sbox[state[i]];
            }
        }
```

Listing 8: AddRoundKey of the Modified S-AES Round in C

```
        void s_addRoundKey(unsigned char *state, unsigned char *roundKey ,
            unsigned char *sk)
        {
            for (int i = 0; i < 16; i++) {
                // Use sk[i % 8] to repeat the 64-bit SK across 128 bits
                state[i] ^= roundKey[i] ^ sk[i % 8];
            }
        }
```

- Pseudo-Random Permutation of the Last Round Key
- Last round: Similar to the Last Round of a Normal AES Encryption, using SubBytes, ShiftRows and AddRoundKey

Listing 9: Last Encryption S-AES Round in C

```
        subBytes(block);
        shiftRows(block);

        getPseudoRandomPermo(s1, roundKey);

        addRoundKey(block, roundKey);
```

4.5 Decryption

The order of the Decryption of the S-AES is:

- Calculate the Modified Round (Same as Encryption)
- Shuffle the S-Box and Calculate the Inverse S-box

Listing 10: Invert a S-box in C

```
    void invert_sbox(unsigned char *sbox, unsigned char *inverse_sbox) {
    memset(inverse_sbox, 0xFF, 256);

    for (int i = 0; i < 256; i++) {
        unsigned char value = sbox[i];

        if (value >= 256 || inverse_sbox[value] != 0xFF) {
            memset(inverse_sbox, 0xFF, 256);  // Reset inverse_sbox to indicate an
                 error
            return;
        }

        inverse_sbox[value] = i;
    }
}
```

- Create The Expanded Key (Same as Encryption)
- Starts the Decryption Rounds
  - Pseudo-Random Permutation of the Last Round Key
  - 1st decrytion round: Same as in AES, AddRoundKey, Inverse of Shift Rows and Inverse Of SubBytes

Listing 11: First Decryption Round of S-AES in C

```
        getPseudoRandomPermo(s1, roundKey );

        addRoundKey(block, roundKey);
        invShiftRows(block);
        invSubBytes(block);
```

- All other rounds are normal AES Decryption unless is the modified round, always executing the pseudo-random permutation of the N round key.

Listing 12: Decryption Rounds of S-AES in C

```
        for (i = rounds - 1; i > 0; i--){
            createRoundKey(expandedKeyS + 16 * i, roundKey);
            getPseudoRandomPermo(s1, roundKey );

            if (selected_round == i){
                s_addRoundKey(block, roundKey, sk);
                invMixColumns(block);
                invShiftRows(block);
                invSSubBytes(block, inv_sbox);
            } else {
                addRoundKey(block, roundKey);
                invMixColumns(block);
                invShiftRows(block);
                invSubBytes(block);
            }
        }
```

Listing 13: Inverse SubBytes of the Modified S-AES Round in C

```
        void invSSubBytes(unsigned char *state,unsigned char *sbox)
        {
            int i;

            for (i = 0; i < 16; i++)
                state[i] = sbox[state[i]];
        }
```

- Pseudo-Random Permutation of the First Round Key
- Last Decryption round: Similar to the Last Round of a Normal AES Decryption, using AddRoundKey

Listing 14: Last Decryption S-AES Round in C

```
        getPseudoRandomPermo(s1, roundKey );

        addRoundKey(block, roundKey);
```

Functions such as MixColumns or ShiftRows were all took of a public github here, that's why we decided to keep them out of the report.

5 S-AES NI

5.1 Assembly Instructions

Table 4: Summary of AES Intrinsics

Intrinsic	**Description and Syntax**
_mm_xor_si128	Performs bitwise XOR on two 128-bit registers. Commonly used in cryptographic algorithms like AES. Syntax: `__m128i _mm_xor_si128(__m128i a, __m128i b);`
_mm_aesenc_si128	Performs an AES encryption round on a 128-bit block using: • ShiftRows: Shifts rows for data diffusion. • SubBytes: Applies S-box substitution. • MixColumns: Mixes columns for further diffusion. • AddRoundKey: XORs state with round key. Syntax: `__m128i _mm_aesenc_si128(__m128i a, __m128i roundkey);`
_mm_aesenclast_si128	Executes the final AES encryption round, excluding MixColumns. It performs: Syntax: `__m128i _mm_aesenclast_si128(__m128i a, __m128i roundkey);`
_mm_aesdec_si128	Executes an AES decryption round on a 128-bit block, reversing encryption transformations: • InvShiftRows: Reverse row shifts. • InvSubBytes: Reverse substitution with inverted S-box. • AddRoundKey • InvMixColumns: Reverses MixColumns. Syntax: `__m128i _mm_aesdec_si128(__m128i a, __m128i roundkey);`
_mm_aesdeclast_si128	Executes the final AES decryption round without InvMix-Columns. Syntax: `__m128i _mm_aesdeclast_si128(__m128i a, __m128i roundkey);`
_mm_aeskeygenassist_si128	Assists in AES key expansion by generating a 128-bit key schedule for a given AES key. It uses a constant 'imm8' (an immediate 8-bit value) to perform Rcon (Round Constant) calculations and generate round keys. This intrinsic helps to automate the key schedule generation for each round in AES. Syntax: `__m128i _mm_aeskeygenassist_si128(__m128i a, const int imm8);`

5.2 Encryption

Function Format:

```
void cipher_saesNI(
    unsigned char key[16], unsigned char sk[16], unsigned char plaintext[16], int key_size,
    unsigned char cipher[16]
);
```

As it was explained earlier the first steps to start the encription are calculating the selected round and the shuffled substitution box. With that processed we can start the encription:

Listing 15: Load the variables in __m128i format

```
            __m128i key128 = _mm_loadu_si128((__m128i*)key);
            __m128i sk128 = _mm_loadu_si128((__m128i*)sk);
            __m128i plaintext128 = _mm_loadu_si128((__m128i*)plaintext);
            __m128i round_keys[11];
            __m128i x = plaintext128;
```

Firstly, we start with the generation of all keys using the **get_round_key** function provided in the project assignment.

Listing 16: Generation of all keys

```
    expand_key(&key128, round_keys);


void expand_key(__m128i *key, __m128i *round_keys) {
    round_keys[0] = *key;
    round_keys[1] = get_round_key(round_keys[0], _mm_aeskeygenassist_si128(round_keys
        [0], 0x01));
    round_keys[2] = get_round_key(round_keys[1], _mm_aeskeygenassist_si128(round_keys
        [1], 0x02));
    round_keys[3] = get_round_key(round_keys[2], _mm_aeskeygenassist_si128(round_keys
        [2], 0x04));
    round_keys[4] = get_round_key(round_keys[3], _mm_aeskeygenassist_si128(round_keys
        [3], 0x08));
    round_keys[5] = get_round_key(round_keys[4], _mm_aeskeygenassist_si128(round_keys
        [4], 0x10));
    round_keys[6] = get_round_key(round_keys[5], _mm_aeskeygenassist_si128(round_keys
        [5], 0x20));
    round_keys[7] = get_round_key(round_keys[6], _mm_aeskeygenassist_si128(round_keys
        [6], 0x40));
    round_keys[8] = get_round_key(round_keys[7], _mm_aeskeygenassist_si128(round_keys
        [7], 0x80));
    round_keys[9] = get_round_key(round_keys[8], _mm_aeskeygenassist_si128(round_keys
        [8], 0x1B));
    round_keys[10] = get_round_key(round_keys[9], _mm_aeskeygenassist_si128(round_keys
        [9], 0x36));
}
```

Before explaining the encryption process, it's better to refer some functions that we used.

- `s_subBytesNI`, `shiftRowsNI`, `mixColumnsNI`, and `s_addRoundKeyNI`, which apply the normal SAES round-specific transformations—SubBytes, ShiftRows, MixColumns, and s_AddRoundKey using the `__m128i` format. These functions were modified to ensure compatibility with the assembly instructions.
- `getPseudoRandomPermoNI`, a function used for key permutation, similar to the previously explained `getPseudoRandomPermo`, but designed to accept `__m128i` variables.
- `Round_block`, reorders a 128-bit block's data to match the expected SAES transformation structure.

To ensure that the same key and the shuffled key produce the same ciphertext in both implementations, the rounds are structured as follows:

- **Initial Round**: In this initial step, the key is prepared to ensure the correct output after performing the XOR operation with the plaintext.

```
            Round_block(&round_keys[0]);
            getPseudoRandomPermoNI(sk128, &round_keys[0]);
            Round_block(&round_keys[0]);

            x = _mm_xor_si128(x, round_keys[0]);
```

- **Main Rounds**: Each main round begins by preparing the round key for the upcoming transformations. For standard rounds, only the `_mm_aesenc_si128` instruction is applied.

  However, in the selected round, additional steps are included. These extra transformations and the specific sequence of function calls are necessary to achieve the same output as the "normal" S-AES implementation when the same key and shuffled key is used to cipher.

```
        for (int i = 1; i < 10; i++) {

            Round_block(&round_keys[i]);
            getPseudoRandomPermoNI(sk128,&round_keys[i]);
            Round_block(&round_keys[i]);

            if (i == selected_round) {

                Round_block(&round_keys[i]);
                Round_block(&x);

                s_subBytesNI(&x, ssbox);
                shiftRowsNI(&x);
                mixColumnsNI(&x);

                s_addRoundKeyNI(&x, &round_keys[i], &sk128);

                Round_block(&x);
            } else {
                x = _mm_aesenc_si128(x, round_keys[i]);
            }
        }
```

- **Final Round**: The final round completes the encryption by applying a final transformation to ‘x‘ with the last round key (_mm_aesenclas_si128).

  Before the last encryption step, we prepare the final round key the same way we did previously:

```
        Round_block(&round_keys[10]);
        getPseudoRandomPermoNI(sk128,&round_keys[10]);
        Round_block(&round_keys[10]);

        x = _mm_aesenclast_si128(x, round_keys[10]);
```

5.3 Decryption

Function Format:

```
void decipher_saesNI(
    unsigned char key[16], unsigned char sk[16], unsigned char ciphertext[16], int key_size,
    unsigned char plaintext[16]
);
```

Before starting the decryption, the same pré-processing steps as in the encryption must be taken. This includes using the shuffled key to rearrange the default substitution box, calculating the selected round, and generating all the required keys. (These steps were explained earlier.)

Additional functions include:

- `InvSubBytesNI`, `InvShiftRowsNI` and `InvMixColumnsNI`, which apply the inverse of the SAES round-specific transformations using `__m128i` format.

With all the necessary data collected, we can now begin the decryption process.

Listing 17: Load the variables

```
__m128i key128 = _mm_loadu_si128((__m128i*)key);
__m128i sk128 = _mm_loadu_si128((__m128i*)sk);
__m128i round_keys[11];

__m128i x = _mm_loadu_si128((__m128i*)ciphertext);
```

- **Initial Round**: As noted in the encryption process, the key is prepared for the initial XOR operation. However, in decryption, we begin with the final round key rather than the first, reversing the encryption sequence to correctly reconstruct the plain text.

```
Round_block(&round_keys[10]);
getPseudoRandomPermoNI(sk128,&round_keys[10]);
Round_block(&round_keys[10]);

x = _mm_xor_si128(x, round_keys[10]);
```

- **Other Rounds**

```
for (int i = 9; i > 0; i--) {

    Round_block(&round_keys[i]);
    getPseudoRandomPermoNI(sk128,&round_keys[i]);
    Round_block(&round_keys[i]);

    // (2)
    if (i == selected_round) {
        Round_block(&x);
        InvShiftRowsNI(&x);
        InvSubBytesNI(&x);

        Round_block(&round_keys[i]);
        s_addRoundKeyNI(&x, &round_keys[i], &sk128);
        InvMixColumnsNI(&x);
        InvShiftRowsNI(&x);
        InvSSubBytesNI(&x, inv_sbox);

        // Generate the key for the next round
        __m128i t = _mm_loadu_si128(&round_keys[i-1]);
        Round_block(&t);
        getPseudoRandomPermoNI(sk128,&t);

        // If selected round is 1 we have to make the last one here
        if (i - 1 == 0 ){
            x = _mm_xor_si128(x, t);
        }else {
            // End round so that
            x = _mm_xor_si128(x, t);
            InvMixColumnsNI(&x);
        }

        Round_block(&x);

    } else {
```

```
                    // Once we did an hand made round we have to skip the next one
                    if (i == selected_round -1 ) {
                        continue;
                    }

                    x = _mm_aesdec_si128(x,  _mm_aesimc_si128(round_keys[i]));
                    }
                }

            // If selected round is 1 we did the last one up there
            if (selected_round != 1){
                Round_block(&round_keys[0]);
                getPseudoRandomPermoNI(sk128, &round_keys[0]);
                Round_block(&round_keys[0]);

                x = _mm_aesdeclast_si128(x, round_keys[0]);
            }
```

As previously mentioned, we start by setting up the key to begin the rounds. Up until the selected round, we simply invoke the following instruction:

```
            x = _mm_aesdec_si128(x,  _mm_aesimc_si128(round_keys[i]));
```

When the algorithm reaches the `selected round`, a manual processing step is performed. Special attention is required if the selected round is 1, as it is the last round before the final one. The procedure for handling this selected round is shown in the code above, marked with comment (2). After processing the selected round manually, we must skip the following round.

For the final step, if the selected round is 1, decryption is complete. Otherwise, we need to apply the following instruction:

```
            x = _mm_aesdeclast_si128(x, round_keys[0]);
```

To better explain how the decryption process works, we provide part of two flow examples. The first one assumes the selected round is 1, while the second assumes the selected round is 2.

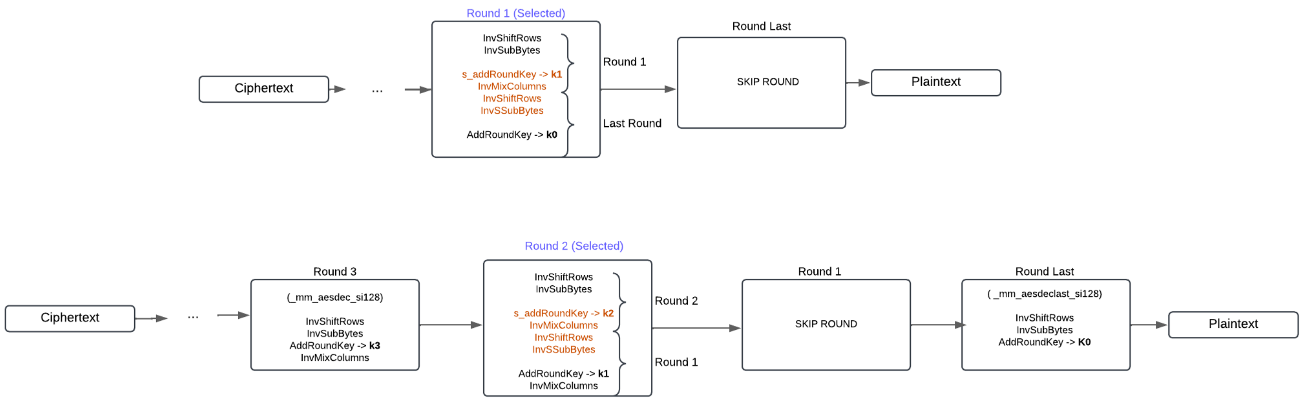


Figure 1: Example of part of the Decryption Flow

6 Results and Deliverable

Encryption and Decryption is in ECB mode and using PKCS7 padding. In the delivered code, execute in the bash:

```
./run_deliverable.sh
```

and then you can execute 3 different commands:

- Decryption
  - AES:

    ```
    echo -n "CipherText" | ./decrypt key
    ```

  - S-AES:

    ```
    echo -n "CipherText" | ./decrypt key sk
    ```

- Encryption
  - AES:

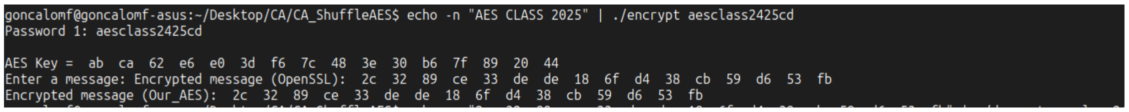
    ```
    echo -n "Plaintext" | ./encrypt key
    ```

  - S-AES:

    ```
    echo -n "Plaintext" | ./encrypt key sk
    ```

- Speed Test
  - Speed is Measuring the time needed to encrypt and decrypt a buffer of 4096 bits with random data. As we can see in Figure 6, the AES implemented by openSSL is much faster then the S-AES that we made, but using assembly instructions gets way faster.

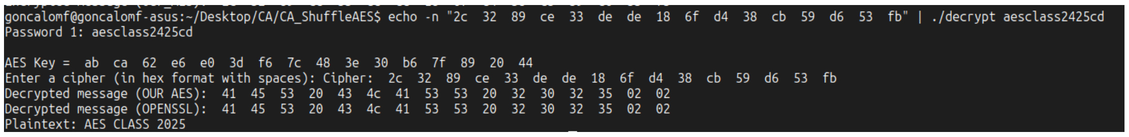
    ```
    ./speed_test
    ```



```
goncalomf@goncalomf-asus:~/Desktop/CA/CA_ShuffleAES$ echo -n "AES CLASS 2025" | ./encrypt aesclass2425cd
Password 1: aesclass2425cd

AES Key =  ab  ca  62  e6  e0  3d  f6  7c  48  3e  30  b6  7f  89  20  44
Enter a message: Encrypted message (OpenSSL):  2c  32  89  ce  33  de  de  18  6f  d4  38  cb  59  d6  53  fb
Encrypted message (Our_AES):  2c  32  89  ce  33  de  de  18  6f  d4  38  cb  59  d6  53  fb
```

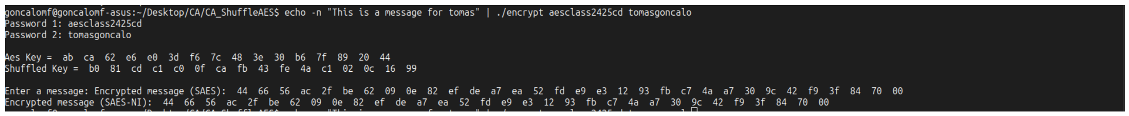
Figure 2: Example of an encryption using AES



```
goncalomf@goncalomf-asus:~/Desktop/CA/CA_ShuffleAES$ echo -n "2c  32  89  ce  33  de  de  18  6f  d4  38  cb  59  d6  53  fb" | ./decrypt aesclass2425cd
Password 1: aesclass2425cd

AES Key =  ab  ca  62  e6  e0  3d  f6  7c  48  3e  30  b6  7f  89  20  44
Enter a cipher (in hex format with spaces): Cipher:  2c  32  89  ce  33  de  de  18  6f  d4  38  cb  59  d6  53  fb
Decrypted message (OUR AES):  41  45  53  20  43  4c  41  53  53  20  32  30  32  35  02  02
Decrypted message (OPENSSL):  41  45  53  20  43  4c  41  53  53  20  32  30  32  35  02  02
Plaintext: AES CLASS 2025
```

Figure 3: Example of an decryption using AES

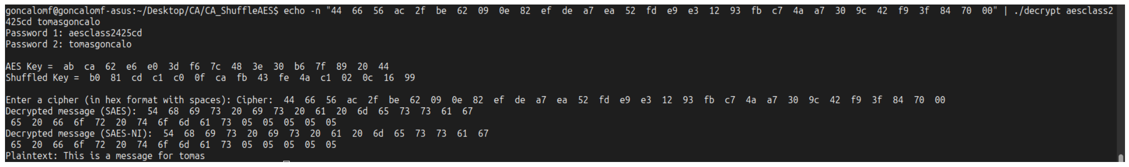


```
goncalomf@goncalomf-asus:~/Desktop/CA/CA_ShuffleAES$ echo -n "This is a message for tomas" | ./encrypt aesclass2425cd tomasgoncalo
Password 1: aesclass2425cd
Password 2: tomasgoncalo

Aes Key =  ab  ca  62  e6  e0  3d  f6  7c  48  3e  30  b6  7f  89  20  44
Shuffled Key =  b0  81  cd  c1  c0  0f  ca  fb  43  fe  4a  c1  02  0c  16  99

Enter a message: Encrypted message (SAES):  44  66  56  ac  2f  be  62  09  0e  82  ef  de  a7  ea  52  fd  e9  e3  12  93  fb  c7  4a  a7  30  9c  42  f9  3f  84  70  00
Encrypted message (SAES-NI):  44  66  56  ac  2f  be  62  09  0e  82  ef  de  a7  ea  52  fd  e9  e3  12  93  fb  c7  4a  a7  30  9c  42  f9  3f  84  70  00
```

Figure 4: Example of an encryption using SAES

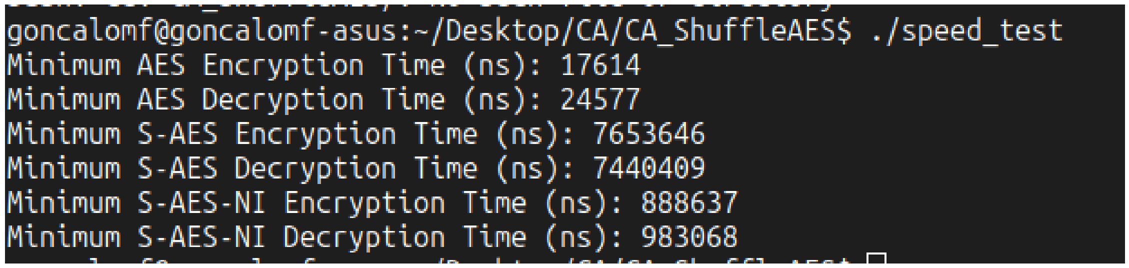


```
goncalomf@goncalomf-asus:~/Desktop/CA/CA_ShuffleAES$ echo -n "44  66  56  ac  2f  be  62  09  0e  82  ef  de  a7  ea  52  fd  e9  e3  12  93  fb  c7  4a  a7  30  9c  42  f9  3f  84  70  00" | ./decrypt aesclass2
425cd tomasgoncalo
Password 1: aesclass2425cd
Password 2: tomasgoncalo

AES Key =  ab  ca  62  e6  e0  3d  f6  7c  48  3e  30  b6  7f  89  20  44
Shuffled Key =  b0  81  cd  c1  c0  0f  ca  fb  43  fe  4a  c1  02  0c  16  99

Enter a cipher (in hex format with spaces): Cipher:  44  66  56  ac  2f  be  62  09  0e  82  ef  de  a7  ea  52  fd  e9  e3  12  93  fb  c7  4a  a7  30  9c  42  f9  3f  84  70  00
Decrypted message (SAES):  54  68  69  73  20  69  73  20  61  20  6d  65  73  73  61  67
 65  20  66  6f  72  20  74  6f  6d  61  73  05  05  05  05  05
Decrypted message (SAES-NI):  54  68  69  73  20  69  73  20  61  20  6d  65  73  73  61  67
 65  20  66  6f  72  20  74  6f  6d  61  73  05  05  05  05  05
Plaintext: This is a message for tomas
```

Figure 5: Example of an decryption using SAES



```
goncalomf@goncalomf-asus:~/Desktop/CA/CA_ShuffleAES$ ./speed_test
Minimum AES Encryption Time (ns): 17614
Minimum AES Decryption Time (ns): 24577
Minimum S-AES Encryption Time (ns): 7653646
Minimum S-AES Decryption Time (ns): 7440409
Minimum S-AES-NI Encryption Time (ns): 888637
Minimum S-AES-NI Decryption Time (ns): 983068
```

Figure 6: Speed test with 100000 iterations

References

Fisher-Yates Shuffling of SBoxes and Inspiration for the AES Code and Intel AES