University of Aveiro
Masters in Cybersecurity
Applied Cryptography

Project 1

Authors:

- José Luís Rodrigues Costa: 92996

Introduction

This project was developed for the applied cryptography class.
The objective is to implement a custom version of the AES algorithm, that is called S-AES. This new algorithm differs from the normal AES, because it uses a new key (SK) to perform a custom round, with different steps compared to the normal AES. Both AES and S-AES will use PKCS7 padding, ECB cipher mode and AES-NI Intel assembly instructions.
The requisites to run this project are the libssl-dev package.

How to use

As this project was developed in the C programming language. It can be compiled with “make” and executed with 3 different functionalities for both AES and S-AES. To choose which one to run, if the second key (SK) is provided, it will use S-AES, else the normal AES will be executed.

- To encrypt using stdin and returning the result in stdout by executing, for example, the following command to encrypt a string with S-AES:
  `echo “textToEncrypt” | ./encrypt “key1” “key2”`

- To decrypt using stdin and return the result in stdout by executing, for example, the following command to decrypt a string with hex values with S-AES:
  `echo “fc3bbe7e8e546cd5d72f438be1f19517” | ./decrypt “key1” “key2”`

- A useful command to test both the encryption and decryption, can be the following using S-AES:
  `echo “textToEncrypt” | ./encrypt “key1” “key2” | ./decrypt “key1” “key2”`

- A speed functionality is also available, it provides a way to test the encryption and decryption speed of a 4Kb buffer of random data and random keys with

the S_AES algorithm. It prints the lowest time for the encryption and decryption of the buffer, compared to a standard openssl library implementation of the normal AES.
To run, simply execute:
`./speed`

Key Generation

The key generation is a part of this project responsible for the generation of a 128 bit key, given the user plain text key, using the PKCS5_PBKDF2_HMAC_SHA1 function from the openssl library. It is applied to both keys (normal and SK), and generates a 128 bit hash that is going to be used for the S-AES and AES algorithm. These keys are only generated in the encrypt and decrypt executables, as the speed uses randomly generated 128 bit keys.

S-AES Explanation

The S-AES implementation only differs from the normal AES implementation in one round. This round is chosen based on the SK and executes a custom approach of the following steps:
SubBytes_S:

- For this step, the S-BOX of the normal AES is shuffled based on the second key provided. The substitutions happen in the same way as AES, but with the new S_S-BOX matrix.

ShiftRows_S:

- In this step, the byte shifts applied to each row are permuted with the SK.

MixColumns_S:

- In this step, the SK is used to add an offset to the index used to select the column of the matrix.

AddRoundKey_S:

- In this step, the SK should be used to add an offset to the index used in the corresponding round key matrix.

S-AES Code Implementation

For the code implementation, the S-AES algorithm needs to generate numbers in a range, based on a second key (SK), with the same probability for every number in that range.
To solve this problem, a random number generator was used. The seed of this generator is the SK. This random number generator can generate numbers in a specific range. If both the encryption and the decryption use the same seed, they will generate the same numbers used for the encryption and decryption steps.
This generator will be used to do the following steps:

- Generate the custom round (from 1 to 9)
- Generate the offset used in AddRoundKey_S (from 1 to 16)
- Generate the offset used in MixColumns_S (from 0 to 3)
- Shuffle a S_Shift_Rows array that will be used to permute the S_ShiftRows step
- Shuffle the S_box for the SubBytes_S step.

The algorithm starts with the setup_encrypt() and setup_decrypt functions(). These functions are responsible for computing the KeyExpansion and performing all the operations related to the number generator, including the calculation of the new inverted s_rs_box.

```
srand(val); // the seed is based on the s_key
S_Offsets[0] = GetValueFromKey(9,1);  // Round - working
S_Offsets[1] = GetValueFromKey(16,1); // AddRoundKey - working
S_Offsets[2] = GetValueFromKey(3,0);  // MixColumns - working
ResetInitialArrays();
shuffle(S_Shift_Rows,4);
shuffle(S_sbox,256);

initialize_inverse_aes_sbox(S_sbox,S_rsbox);
```

Picture 1

This implementation of the S-AES uses AES-NI Intel assembly instructions that provide a considerable performance gain while executing the normal AES algorithm. The S-AES differs in one round from the original AES algorithm.
The function that executes all the encryption steps of one 16 byte block is shown in the code snippet below:

```
static void Cipher_NI(void)
{
  uint8_t round = 0;
  state_NI = _mm_xor_si128(state_NI, RoundKey_NI[0]); // First Round
  for(round = 1; round < Nr; ++round){
    if(round == S_Offsets[0]){                        // Custom round selection
      _mm_storeu_si128((__m128i *) state, state_NI);  // Store state_NI matrix
      SubBytes_S();
      ShiftRows_S(S_Shift_Rows);
      MixColumns_S();
      AddRoundKey_S(round);
      state_NI = _mm_loadu_si128((__m128i *) state);  // Load state_NI matrix
    }else{
      state_NI = _mm_aesenc_si128(state_NI, RoundKey_NI[round]); // Normal AES round
    }
  }
  state_NI = _mm_aesenclast_si128(state_NI, RoundKey_NI[10]); // Last round
}
```

Picture 2

As we can see, on the modified round (S_Offsets[0]) the custom steps of the S-AES are executed, as explained in the last chapter (S-AES Explanation).

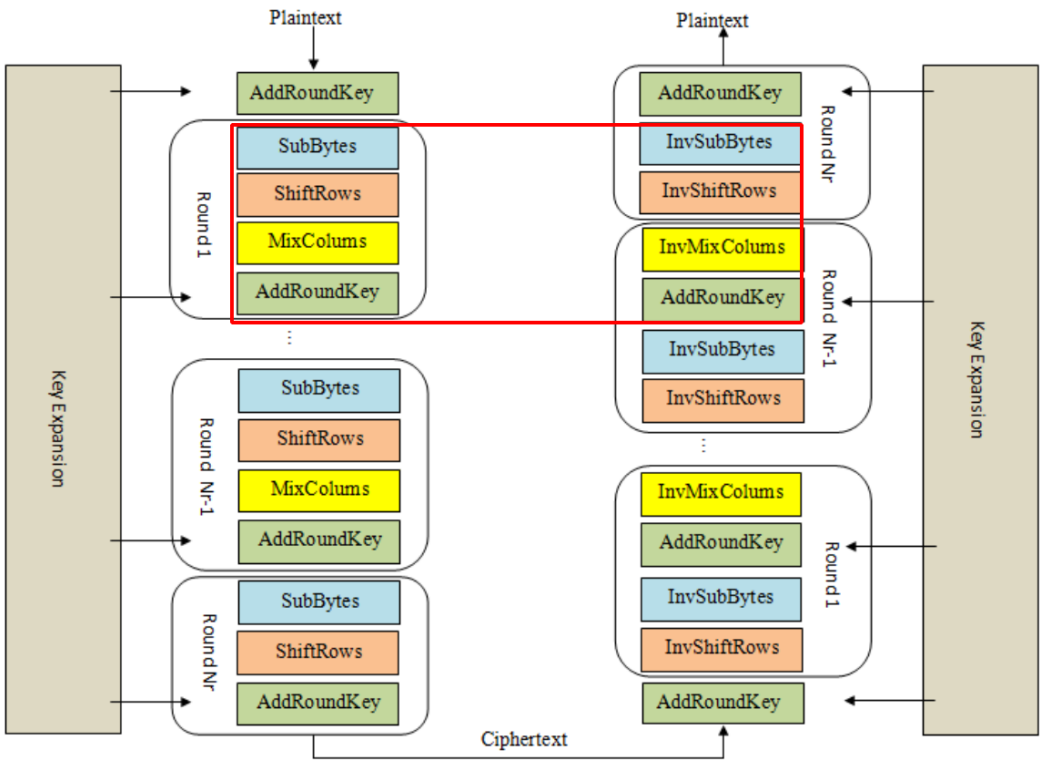
The ShiftRows_S step uses a S_Shift_Rows array initialized with [0,1,2,3], this means, for the position 0, the row is not going to rotate. For position 1, the row is going to be rotated by 1 offset, and so on.
A shuffle function that uses rand() is used to randomize this array, thus performing the described ShiftRows_S step (shuffled in picture 2).

The Sub_Bytes_S step uses the same shuffle function to randomize the original s_box, creating the new S_s_box, substituting the values on the new S_s_box and the corresponding inverted matrix (shuffled in picture 2).

The Mix_Columns_S step applies the offset (generated in picture 2) to the selected column in the data block.

The AddRoundKey_S step uses the offset (generated in picture 2) to shift the corresponding Round Key matrix.

The S-AES implementation with AES-NI instructions makes the decryption process a bit tricky, as the aes encryption and decryption rounds don't match.



Picture 3

As we can see in the illustration above, if the first round from the encryption is chosen as the modified round, the 4 steps do not correspond to the same decryption round. In the decryption, the InvSubBytes and InvShiftRows from the last round (round Nr) and the MixColumns and AddRoundKey from round 9 (round Nr-1) are the ones that correspond to the original steps from the first encryption round. As AES-NI only decrypts a full round, as shown in the illustration, the algorithm will have to manually decrypt 2 full rounds, with the correspondent modified steps in each. As can be seen in the code snippet below.

```
static void InvCipher_NI(void)
{
  int round=0;
  int c = 1;
  state_NI = _mm_xor_si128(state_NI, RoundKey_NI[10+0]); // First round
  for(round=Nr-1;round>0;round--)
  {
    if(round == S_Offsets[0]){                          // Modified round and Normal round
      _mm_storeu_si128((__m128i *) state, state_NI);     // Load state_NI matrix to state variable
      InvShiftRows();                                   // Normal ShiftRows
      InvSubBytes();                                    // Normal SubBytes
      AddRoundKey_S(round);                             // Modified AddRoundKey
      InvMixColumns_S();                                // Modified MixColumns
      InvShiftRows_S(S_Shift_Rows);                     // Modified ShiftRows
      InvSubBytes_S();                                  // Modified SubBytes
      if(round-1==0){                                   // If last round is randomly choosen
        AddRoundKey(0);
        state_NI = _mm_loadu_si128((__m128i *) state);
        break;
      }
      AddRoundKey(round-1);                             // Normal AddRoundKey
      InvMixColumns();                                  // Normal MixColumns
      round -= 1;                                       // Two rounds should be skipped in normal AES decryption
      c+=2;
      state_NI = _mm_loadu_si128((__m128i *) state);    // Save state variable to state_NI matrix
    }else{
      state_NI = _mm_aesdec_si128(state_NI, RoundKey_NI[10+c]); // Normal AES descryption
      c++;
    }
  }
  if(round-1 != 0){                                     // If last round is not randomly choosen
    state_NI = _mm_aesdeclast_si128(state_NI, RoundKey_NI[0]);
  }
}
```

Picture 4

In this code snippet we can see that inside the if where the modified round is applied, the algorithm decrypts two rounds. First 2 normal steps (InvShiftRows,SubBytes), then the 4 modified steps, and the last 2 normal steps (AddRoundKey, InvMixColumns), which performs the decryption of 2 full rounds.

Results

When testing with the speed executable, a computer with the following specs: Intel i7-8750h with 16gb RAM, managed to achieve a minimum time of 469993 ns (encrypting and decrypting) using the S-AES, while the Openssl AES implementation managed 29349 ns.

These results are very different, the S-AES implementation is slower, so another test was also conducted. Using the same speed executable, but comparing the S-AES

implementation and the AES-NI from acapola (repository in references). The results don't differ as much, the AES-NI achieved 145271 ns, and the S-AES 470165 ns. As we have the S-AES round both in the encryption and decryption, these results are expected.

References

This S-AES implementation was based on the original AES code by kokke:
https://github.com/kokke/tiny-AES-c
The PKCS7 padding was also based on the following implementation by erev0s:
https://erev0s.com/blog/tiny-aes-cbc-mode-pkcs7-padding-written-c
The NI-AES implementar was based on this NI-AES code by acapola:
https://gist.github.com/acapola/d5b940da024080dfaf5f