Project 1

Diogo Miguel de Almeida Silveira 85117

December 6, 2021

Contents

Abstract

This report describes the development of the S-AES encryption method as required for Project 1.

It begins with a brief introduction followed by a specification of the software implemented: the core cipher module and its applications for demonstration.

1 Introduction

Advanced Encryption Standard (AES) is currently the standard specification for symmetric encryption. It consists of a block cipher which comes in several versions with 10, 12 and 14 rounds, taking 128, 192 and 256-bit keys, respectively, and operating in several modes (e.g: Cipher Block Chain (CBC), Electronic Codeblock (ECB), etc).

This project aims to demonstrate an implementation of Shuffled AES (S-AES) which is a version of AES that takes a second key used to create a shuffled round. The 10-round (128-bit key) version of AES will be used (AES-128), in ECB mode, as the base implementation, and the S-AES will be built on top of it.

Then 3 applications of the cipher will be developed, for encryption, decryption and speed testing. Every piece of software in this project will be developed using Python 3.8.10.

2 Base AES-128 Implementation

The base for S-AES is a standard AES-128 cipher[1][2][3], in ECB mode, developed from scratch in pure Python, so that it would be possible to adapt the code to integrate the changes required for this project, namely the Shuffled Round and the Shuffle Key setup.

2.1 Key Setup

The cipher can take a key of any length, which is converted into a 128-bit (16-byte) key using a SHAKE128 hashing method from Python's *hashlib* library.

Then the key is expanded to an array of 11 round keys which is stored for later use. This expansion is made with the help of two lookup tables: Rijndael S-BOX and R-CON. These, as well as all the other lookup tables used are stored in their own dedicated .py for the sake of organization.

2.2 Encryption Rounds

The encryption rounds were built following the AES standards. The steps *AddRoundKey*, *SubBytes* and *ShiftRows* were relatively simple to implement while the *MixColumns* step required more research on how to build it. The final solution for this step was based on code from GitHub[4] adapted to use a set of lookup tables[5] to perform the Galois Multiplication instead of executing the operation during each round, which would take too much time.

3 S-AES Implementation

As already mentioned above, the S-AES was built on top of the base AES-128 cipher. A second key is passed to this version of AES which is meant to select a random Shuffle Round (from round 1 to round 9)

3.1 Key Setup

The key setup for S-AES is the same that is used for AES-128, with the difference that another key (the Shuffle Key) has to be processed.

Like the standard key, this one if first processed to obtain a 16-byte key, using a SHAKE128 hash from Python's *hashlib* library. Then, the integer values of its bytes are summed, as depicted in the code below:

```
# Sums all bytes of sk:
def get_sk_sum(self):
    sum = 0
    for i in self.sk:
        sum += i
    return sum
```

The result of this sum is then used to compute a set of offsets and other values used during the Shuffle Round, as well as which round will run as the Shuffle Round. The offsets are obtained by performing the remainder of the integer division of the sum computed before by values which are usually the length of the round variables that will be affected by the Shuffle Key (SK).

All these setup operations are executed before the encryption/decryption rounds start, to avoid performance penalties during these rounds.

3.2 Shuffle Round Selection

The Shuffle Round is selected by the SK, by getting the index between 1 and 9 from the remainder of the integer division of its sum by 9 (Number of possible rounds). All four steps of this round are changed and affected by the SK.

3.3 Shuffled *AddRoundKey*

This step is relatively simple. It uses one of the offsets computed to rotate the scheduled key meant to be used during this round and the XOR operation is performed with the result of this rotation.

3.4 Shuffled *ShiftRows*

For this step, the standard order of the shift offsets (0, 1, 2, 3) is shuffled, by using the sum of the SK bytes as the index for selecting an item from the list of all permutations of this order. This is performed during the key setup.

The permutations are computed by calling the *permutations* method from the Python's *itertools* library.

The same operations are executed for the inverse of this step, for decryption.

3.5 Shuffled *SubBytes*

During this step the a shuffled version of Rijndael S-BOX is used to perform the bytes' substitution. This new S-BOX is calculated during the key setup by passing the sum of the bytes from SK as seed to the *Random.shuffle* method of Python's *random* library:

```
def get_s_sbox(self):
    s_sbox = list(copy(const.SBOX))
    seed = self.sk_sum
    random.Random(seed).shuffle(s_sbox)
    return s_sbox
```

The reversed version of this S-BOX, used for the inverse of this step (for decryption), is computed immediately after, directly from the original one:

```
def get_s_inv_sbox(self):
    inv_s_sbox = [None for _ in range(len(self.s_sbox))]
    for i in range(len(inv_s_sbox)):
        inv_s_sbox[self.s_sbox[i]] = i
    return inv_s_sbox
```

Both steps, for encryption and decryption, are executed the same way as the standard ones, but using the previously computed S-BOX and its reverse, respectively.

3.6 Shuffed *MixColumns*

For this step, once again, an offset was used, this time for selecting the columns. Then, the *MixColumns* operations are normally performed. Note that it applies to the columns being selected but not to the resulting ones stored after the operation. Those are stored by the same order of execution:

```
...
# Get the current column:
# (j selects the column itself, i selects within the column)
for j in range(4):
    # Applies offset before operation:
    column.append(state[(((j + self.s_mix_columns_offset) % 4)*4+ i)])
self.mix_column(column)
# 'Paste' into the new table:
# (here, j selects within the column)
for j in range(4):
    state[j*4+i] = column[j]
...
```

The opposite happens in the inverse of this step, where the columns are selected by their order for the operation and the offset is used to store the results afterwards, thus going back to their original (before encryption) positions:

```
...
# Get the current column:
# (j selects the column itself, i selects within the column)
for j in range(4):
    column.append(state[j*4+i])
self.inv_mix_column(column)
# 'Paste' into the new table:
# (here, j selects within the column)
for j in range(4):
    # Applies offset after operation:
    state[(((j + self.s_mix_columns_offset) % 4)*4+ i)] = column[j]
 ...
```

4 Implementation

The S-AES cipher was implemented alongside a regular AES implementation from Python's *cryptography* library, which is used instead of the first one if the SK is not given as argument. A wrapper was built to manage the operations executed given the input, as well as some key processing operations (like hashing or encoding).

5 Applications

Three applications were developed to demonstrate the implementation of the cipher.

5.1 Encrypt

This application is a simple encryptor that takes two passwords (or one) as arguments and uses them to encrypt a file called *test_file*. Below is an example of the command used to call this application:

```
python encrypt.py <key1> <key2>
```

If *key2* is omitted the standard AES encryption is executed, else it will be S-AES.

5.2 Decrypt

This application is a simple decryptor that takes two passwords (or one) as arguments and uses them to decrypt a file called *test_file*. Below is an example of the command used to call this application:

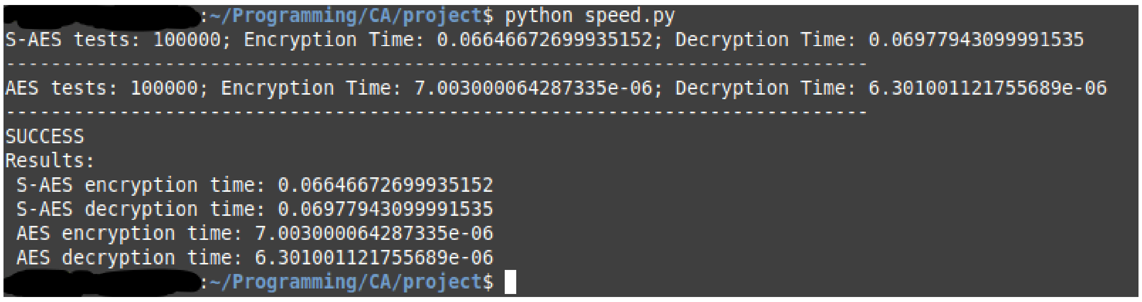
```
python decrypt.py <key1> <key2>
```

If *key2* is omitted the standard AES decryption is executed, else it will be S-AES.

5.3 Speed

This application aims to test the speed of S-AES and AES and display the best results for each one (both for encryption and decryption). It runs 100 000 tests for each version (AES and S-AES), generating a new key for each test. It takes more time to execute because S-AES is significantly slower than the implementation of AES used.

The results obtained are depicted (in nanoseconds) in the screenshot below:



```
[redacted]:~/Programming/CA/project$ python speed.py
S-AES tests: 100000; Encryption Time: 0.06646672699935152; Decryption Time: 0.06977943099991535
----------------------------------------------------------------------------
AES tests: 100000; Encryption Time: 7.003000064287335e-06; Decryption Time: 6.301001121755689e-06
----------------------------------------------------------------------------
SUCCESS
Results:
 S-AES encryption time: 0.06646672699935152
 S-AES decryption time: 0.06977943099991535
 AES encryption time: 7.003000064287335e-06
 AES decryption time: 6.301001121755689e-06
[redacted]:~/Programming/CA/project$
```

Figure 1: Results obtained from speed.py

References

[1] unknown, “The advanced encryption standard (aes) algorithm.” ”https://www.commonlounge.com/discussion/e32fdd267aaa4240a4464723bc74d0a5”.

[2] unknown, “Advanced encryption standard.” ”https://en.wikipedia.org/wiki/Advanced_Encryption_Standard”.

[3] unknown, “Aes encryption with python step by step.” ”https://femionewin.medium.com/aes-encryption-with-python-step-by-step-3e3ab0b0fd6c”.

[4] R. Chai, “Python aes implementation.” ”https://gist.github.com/raullenchai/2920069”.

[5] unknown, “Galois multiplication lookup tables.” ”https://en.wikipedia.org/wiki/Rijndael_MixColumns#Galois_Multiplication_lookup_tables”.