Homework #2
AES implementation

CNS Course Sapienza

Riccardo Prinzivalle, 1904064

October 30, 2020

1 Homework Goal

This homework contains a basic python implementation of AES algorithm. Throughout the work, it has been chosen to make a compromise: the implementation is in midpoint between complete Galois Field and precomputed look-up tables for every step. Both encryption and decryption are studied and implemented, then the operatives modes are implemente and finally it will illustrate a comparison with world known AES implementations.

2 Encryption

Before digging into the details of every operation performed, it is necessary to explain how the variables are stored in this implementation: the 4x4 state matrix is threated as single vector of 16 components, from which of course it is possible to build the matrix representation and viceversa, it was only a choice due to initial lack of knowledge of python language; every state block is a byte stored in hex values to simplify computations (as I discovered python stores hex values as int so in practice the bytes blocks are stored as int). Also the key is stored as an array of 16 bytes, and every bytes follows the same philosophy adopted for the state.
AES encryption can be decomposed into a sequence of the following four big blocks:

1. Byte Substitution
2. Shift Row
3. Mix Columns
4. Key Addition

The following subsections contains implementation details and choices made to build the single blocks.

2.1 Byte Substitution

The byte substitution block is build by two fuctions, one perform the single byte substitution operation (*subByteSingle*), while the other one (*subByte*) performs the byte substitution to all the byte of the state calling the previous function for every single byte. The basic operation of single byte substitution is performing simply by looking to a prestored table which contains all the values for the 256 combinations of 8 bit contained in one byte. This method is more efficient and more simpler to implement with respect to performing all the operations needed in GF(256).

2.2 Shift Row

The shift row block is a simple function wich perform a left row shift for every row of the state matrix; since the state is sorted as a plain array, it s necessary first to build the row of the matrix from the array, then perform the row shift and finally substitute back in the state array the shifted value. As written in AES standard, the shift is of one byte more very time one new row is processed (first row has no shift while last row has a circular shift of 3 bytes).

2.3 Mix Columns

The mix column block is based on an intermediate difficulty of implementation: the base idea is taken from [3], which is the central operation performed inside the *mixcolumns* function. In order to perform it, it is necessary to construct the columns from the array state and in the end rebuild the state vector form the columns. The central operation is based on another function called *xtime*, which together with the xor operations, is able to substitute all the GF operations. The xtime function implements the multiplication of the polynomial stored in the byte for x, its idea is taken from [2] and since the algorithm works with a byte implementation, it is necessary to use the final end in the return of the function to truncate the byte after the left shift.
To debug this section, I used the test vetors which can be found in [4].

2.4 Key Addition

The key addition layer basically uses 3 functions in this implementation: *keyAddition* perform the xor operation between the state and the subkey (the for is necessary to perform the operation bytewise, since the state is an array); *roundKey* generates the subkeys needed at every iteration of AES while *gFuntion* is an auxiliary function needed by *roundKey*.
The *gFunction* corresponds to function g of classical AES implementation, the shift and the byte substitution are based on previous functions or parts of them, while the xor operation is performed with a precomputed vector instead of computing at every round the specific byte to xor. In this way, the implementation is more efficient and simpler.

The *roundKey* function first joins 4 bytes at a time of the key in order to work with words, then it needs the auxiliary *gFunction*, then it performs the four cascaded xor and finally it rebuild the byte based key from the word.
To debug this section, it has been used a step by step subkey generation that can be found in [1].

3 Decryption

The decryption part of AES is simpler than encryption since it is possible to reuse at least the key addition layer; based on implementation choices, as those mades in this homework, it is possible to reuse also the *mixcolumn* idea in its inverse. The decryption is less efficient than its counterpart since it generates the subkeys at every round instead of generating it once and storing it, this has been chosen to speed up the implementation.

3.1 Mix Columns Inverse

The inverse of *mixcolumns* reuses part of its direct function together with a precomputation: the columns are preprocessed (fig. 1b) and then they are elaborated by the *mixcolumns* core (fig. 1a). All the matematical details on this choice can be found in chapter 4 of [3].

```
# mixcolumn
xorAll = column[0] ^ column[1] ^ column[2] ^ column[3]
temp = column[0]
column[0] = column[0] ^ xtime(column[0] ^ column[1]) ^ xorAll
column[1] = column[1] ^ xtime(column[1] ^ column[2]) ^ xorAll
column[2] = column[2] ^ xtime(column[2] ^ column[3]) ^ xorAll
column[3] = column[3] ^ xtime(column[3] ^ temp) ^ xorAll
```

(a) Core

```
# preprocessing
u = xtime(xtime(column[0] ^ column[2]))
v = xtime(xtime(column[1] ^ column[3]))
column[0] = column[0] ^ u
column[1] = column[1] ^ v
column[2] = column[2] ^ u
column[3] = column[3] ^ v
```

(b) Preprocessing

Figure 1: Mix columns base functions

3.2 Shift Row Inverse

This blocks perform the row shift to the right, in order to reverse the *shiftRow* operation. As before, from state it builds the necessary rows and at the end the state is rebuilt from each rows. In order to reverse the direct operation, the first row is not shifted while the others receive successive shift till the third row which undergoes a right shift of three elements.

3.3 Byte Substitution Inverse

This layer perform the inverse operation of Byte Substitution; it is based on two functions: *subByteInv* is just a wrapper to call the base function on every byte of the state; *subByteSingleInv* instead is the base funtion which performs the inverse Byte Substitution based on the inverse table seen before in the *subByteSingle*.

4 Operation Modes

Mass of empty crucible	7.28 g
Mass of crucible and magnesium before heating	8.59 g
Mass of crucible and magnesium oxide after heating	9.46 g
Balance used	#4
Magnesium from sample bottle	#1

5 Real World Standard Comparison

Mass of magnesium metal	= 8.59 g - 7.28 g
	= 1.31 g
Mass of magnesium oxide	= 9.46 g - 7.28 g
	= 2.18 g
Mass of oxygen	= 2.18 g - 1.31 g
	= 0.87 g

Because of this reaction, the required ratio is the atomic weight of magnesium: 16.00 g of oxygen as experimental mass of Mg: experimental mass of oxygen or $\frac{x}{1.31} = \frac{16}{0.87}$ from which, $M_{\mathrm{Mg}} = 16.00 \times \frac{1.31}{0.87} = 24.1 = 24\,\mathrm{g\,mol^{-1}}$ (to two significant figures).

References

[1] Aes example (8bit key and message). `https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf`.

[2] Understanding aes mix-columns transformation calculation. `https://piazza.com/class_profile/get_resource/kf2apkmqhrq4qw/kg6jf10s14j1tu`.

[3] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, Berlin, Heidelberg, 2002.

[4] K. C. Xintong. Aes mixcolumn. `https://en.wikipedia.org/wiki/Rijndael_MixColumns`.