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 operation modes are implemented and finally it will illustrate a comparison with world known AES implementation.

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 treated as single vector of 16 components, from which of course it is possible to build the matrix representation and vice versa, it was only a choice due to initial lack of knowledge of python language; every state component is a byte stored in hex values to simplify computations (as I discovered python stores hex values as int so in practice the byte blocks are stored as int). Also the key is stored as an array of 16 bytes, and every byte follows the same philosophy adopted for the state.
The encryption function is *encrypt*, and it is composed by a sequence of the following four big blocks:

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

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

2.1 Byte Substitution

The byte substitution block is build by two functions, one performs the single byte substitution operation (*subByteSingle*), while the other one (*subByte*) performs the byte substitution for all the bytes of the state calling the previous function for every single byte. The basic operation of single byte substitution is performed 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 which perform a left row shift for every row of the state matrix; since the state is sorted as a plain array, it is necessary first to build the row of the matrix from the array, then performs the row shift and finally substitutes back in the state array the shifted value. As written in AES standard, the shift is of one byte more every 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 [6], this idea is the core operation (fig. 1a) performed inside the *mixcolumns* function. In order to perform it, it is necessary to construct the columns from the state array and in the end rebuild the state vector from 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 times x, its idea is taken from [5] and since the algorithm works with a byte implementation, it is necessary to use the final 'and' in the return of the function to truncate the byte after the left shift.
To debug this section, I used the test vectors which can be found in [7].

2.4 Key Addition

The key addition layer basically uses 3 functions in this implementation: *keyAddition* performs 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 *gFunction* 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.

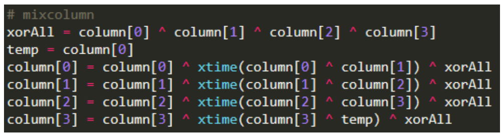
The *roundKey* function first joins 4 bytes of the key at a time in order to work with words, then it needs the auxiliary *gFunction*, then it performs the four cascaded xor and finally it rebuilds the byte based key from the words.
To debug this section, it has been used a step by step subkey generation that can be found in [2].

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 made in this work, 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. The decryption function is called *decrypt*.

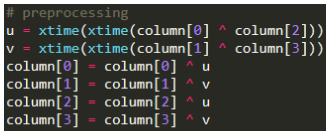
3.1 Mix Columns Inverse

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 mathematical details on this choice can be found in chapter 4 of [6].



```
# 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 block performs 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 performs 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 function which performs the inverse Byte Substitution based on the inverse table seen before in the *subByteSingle*.

4 Operation Modes

Operation modes allows to use a block cipher as AES in the case the message has length different from one block size: in this case, it is possible to encrypt multiple blocks or even deal with blocks of length different from 128 bit. In order to work with different blocks, we need to manipulate the input: in this work, it is assumed to receive a string of hex numbers as input, as in table 1, so it is necessary to have some input manipulation function.
The values in table 1 are taken from example of implementation of AES found on GitHub.

Input text	6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411e5fbc1191a0a52ef f69f2445df4f9b17ad2b417be66c37
Key	2b7e151628aed2a6abf7158809cf4f3c
IV (where needed)	5468617473206D79204B756E67204675

Table 1: Parameter used in testing of AES operation modes

4.1 Text to Blocks

First it is necessary the *text2blocks* function, which divides the hex string received as input into the blocks that can be encrypted by AES. In this step a problem arises: what to do with blocks whose length is different from block size? It has been chosen in this work to use the padding scheme PKCS#7, which in this case added a number of bytes to complete the block with value equal to the number of missing bytes; in the special case when the last block has length equal to block size, the padding scheme plans to add a new block composed by bytes whose value is equal to block size in bytes (16 in this case so a block composed by 10 in hex base). The padding is added during text to block conversion by a similar function as those explained before, the only difference is that it adds padding to the last block; this function is *text2blocksPadding.* To have a better understanding of why it was necessary to design two similar functions, there it is their main motivation: *text2blocksPadding* is used with plaintext as input, so it must add padding since text is going to be transformed and aligned to block dimension; *text2blocks* is used in decryption with ciphertext as input, so the padding has already been added in encryption and it is necessary to just split the text into blocks whose length is exactly equal to block size.

4.2 Blocks to Text

To obtain the text from an array of blocks, some extra functions are needed: *blocks2text* and *blocks2textPadding*, whose aim is the inverse of that of previous section. The first is used at the end of encryption phase to obtain the ciphertext from the array of encrypted blocks. Special mention goes to the second function, which works at the end of decryption to join plain blocks into the decrypted plaintext: in this case it is necessary to reverse and remove the padding scheme used, this operation is simpler when the last byte of last block is 16 in base ten since we simply remove the last block from the the array (and the corresponding list made to host the blocks undergoing manipulation); in the case the last byte read is different from 16, it is necessary to remove a number of couples (every byte is in hex, so two cipher must be removed) equal to the last byte read and this is done by a simple iterative *pop*.

4.3 Block manipulation

Since the state is a list of 16 int representing the 16 bytes, it is necessary to transform every block (which contains a string of hex values) into the int array needed by encryption function: this operation is performed by *block2int* function. The reverse function, *int2block*, allows to recover the block string from the list of 16 int which was used as state.

4.4 Implementation notes

All the operation modes have been implemented based on the slide of the course, CBC and CFB cause some problems since the following block to be encrypted needs to know the previous encrypted block, and due to the different implementation of *block2int* and *int2block*, it is necessary from the second block on to delete the spaces the function *int2block* introduces in order to have the right input for *block2int* in the following block computations.
Another problem is in the CTR, since for every block it is needed to increment the nonce, and a problem may arise when the increment arrives to the byte in position 0 in the block array and the current value is 255, I am not able to decide what to do in this situation, so I used a nonce which does not create this problem in the test (I know this is not the right way to do it). Furthermore, this implementation uses a 128 bit nonce; this is because initially I wasn't able to find a standard and I thought the nonce was all the 128 bit and it is incremented; then I found that *pycryptodome*, the library chosen as standard for the comparison, implemented a 64 bit nonce plus a 64 bit counter and they are put one at the side of the other to build a 128 bit IV. Both these standard are documented in [4], so I had chosen to stay firm in my choice, so experimental result will be different. Perhaps, on further digging on the net, I found that it is possible to use an IV of 128 bit where the left 96 bits are nonce and the remaining 32 bits on the right are dedicated to counter, this is specified in [1].
Instead, ECB and OFB were simpler to implement due to their simpler algo-

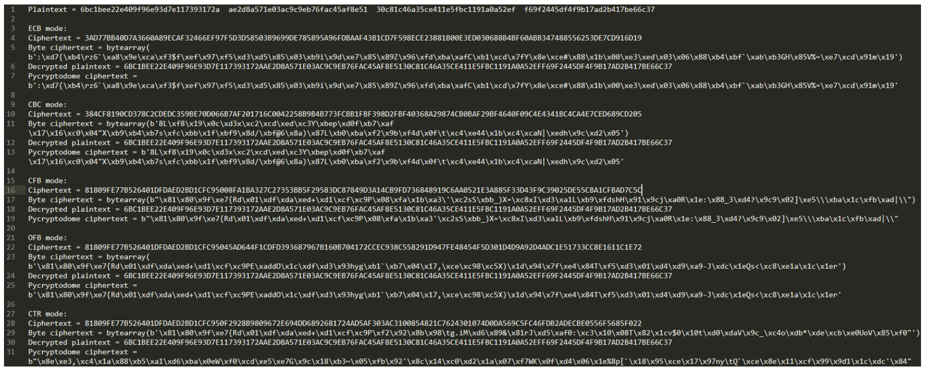
rithm.
A note on the implementation of CFB: it can be implemented choosing a variable number of bits of ciphertext to xor in the following iteration, in this case, to simplify the implementation, it has been chosen to xor the complete block.

5 Real World Standard Comparison

As validation and comparison of the implementation in this report, it has been chosen *PyCryptoDome* as test library.

5.1 Validation

All the operation modes asked in the homework are available in *PyCryptoDome*, so as an initial validation all operational modes are implemented. To have a fair comparison, the padding scheme is added manually in the case of library test in order to obtain also the last block equal to the implementation proposed.



```
Plaintext = 6bc1bee22e409f96e93d7e117393172a  ae2d8a571e03ac9c9eb76fac45af8e51  30c81c46a35ce411e5fbc1191a0a52ef  f69f2445df4f9b17ad2b417be66c37

ECB mode:
Ciphertext = 3AD77BB40D7A3660A89ECAF32466EF97F5D3D58503B9699DE785895A96FDBAAF43B1CD7F598ECE23881B00E3ED030688B4BF60ABB347488556253DE7CD916D19
Byte ciphertext = bytearray(
b':\xd7{\xb4\rz6`\xa8\x9e\xca\xf3$f\xef\x97\xf5\xd3\xd5\x85\x03\xb9i\x9d\xe7\x85\x89Z\x96\xfd\xba\xafC\xb1\xcd\x7fY\x8e\xce#\x88\x1b\x00\xe3\xed\x03\x06\x88\xb4\xbf`\xab\xb3GH\x85V%=\xe7\xcd\x91m\x19')
Decrypted plaintext = 6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C37
Pycryptodome ciphertext =
b':\xd7{\xb4\rz6`\xa8\x9e\xca\xf3$f\xef\x97\xf5\xd3\xd5\x85\x03\xb9i\x9d\xe7\x85\x89Z\x96\xfd\xba\xafC\xb1\xcd\x7fY\x8e\xce#\x88\x1b\x00\xe3\xed\x03\x06\x88\xb4\xbf`\xab\xb3GH\x85V%=\xe7\xcd\x91m\x19'

CBC mode:
Ciphertext = 384CF8190CD378C2CDEDC359BE70D066B7AF201716C0042258B9B4B773FCBB1FBF398D2FBF40368A29874CB0BAF29BF4640F09C4E4341BC4CA4E7CED689CD205
Byte ciphertext = bytearray(b'8L\xf8\x19\x0c\xd3x\xc2\xcd\xed\xc3Y\xbep\xd0f\xb7\xaf
\x17\x16\xc0\x04"X\xb9\xb4\xb7s\xfc\xbb\x1f\xbf9\x8d/\xbf@6\x8a)\x87L\xb0\xba\xf2\x9b\xf4d\x0f\t\xc4\xe44\x1b\xc4\xcaN|\xedh\x9c\xd2\x05')
Decrypted plaintext = 6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C37
Pycryptodome ciphertext = b'8L\xf8\x19\x0c\xd3x\xc2\xcd\xed\xc3Y\xbep\xd0f\xb7\xaf
\x17\x16\xc0\x04"X\xb9\xb4\xb7s\xfc\xbb\x1f\xbf9\x8d/\xbf@6\x8a)\x87L\xb0\xba\xf2\x9b\xf4d\x0f\t\xc4\xe44\x1b\xc4\xcaN|\xedh\x9c\xd2\x05'

CFB mode:
Ciphertext = 81809FE77B526401DFDAED2BD1CFC95008FA1BA327C27353BB5F29583DC87849D3A14CB9FD736848919C6AA0521E3A885F33D43F9C39025DE55CBA1CFBAD7C5C
Byte ciphertext = bytearray(b"\x81\x80\x9f\xe7{Rd\x01\xdf\xda\xed+\xd1\xcf\xc9P\x08\xfa\x1b\xa3\'\xc2sS\xbb_)X=\xc8xI\xd3\xa1L\xb9\xfdshH\x91\x9cj\xa0R\x1e:\x88_3\xd4?\x9c9\x02]\xe5\\\xba\x1c\xfb\xad|\\")
Decrypted plaintext = 6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C37
Pycryptodome ciphertext = b"\x81\x80\x9f\xe7{Rd\x01\xdf\xda\xed+\xd1\xcf\xc9P\x08\xfa\x1b\xa3'\xc2sS\xbb_)X=\xc8xI\xd3\xa1L\xb9\xfdshH\x91\x9cj\xa0R\x1e:\x88_3\xd4?\x9c9\x02]\xe5\\\xba\x1c\xfb\xad|\\"

OFB mode:
Ciphertext = 81809FE77B526401DFDAED2BD1CFC95045AD644F1CDFD393687967B160B704172CCEC938C558291D947FE48454F5D301D4D9A92D4ADC1E51733CC8E1611C1E72
Byte ciphertext = bytearray(
b'\x81\x80\x9f\xe7{Rd\x01\xdf\xda\xed+\xd1\xcf\xc9PE\xaddO\x1c\xdf\xd3\x93hyg\xb1`\xb7\x04\x17,\xce\xc98\xc5X)\x1d\x94\x7f\xe4\x84T\xf5\xd3\x01\xd4\xd9\xa9-J\xdc\x1eQs<\xc8\xe1a\x1c\x1er')
Decrypted plaintext = 6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C37
Pycryptodome ciphertext =
b'\x81\x80\x9f\xe7{Rd\x01\xdf\xda\xed+\xd1\xcf\xc9PE\xaddO\x1c\xdf\xd3\x93hyg\xb1`\xb7\x04\x17,\xce\xc98\xc5X)\x1d\x94\x7f\xe4\x84T\xf5\xd3\x01\xd4\xd9\xa9-J\xdc\x1eQs<\xc8\xe1a\x1c\x1er'

CTR mode:
Ciphertext = 81809FE77B526401DFDAED2BD1CFC950F2928B9809672E694DD6892681724AD5AF303AC3100854821C7624301074D0DA569C5FC46FDB2ADECBE0556F5685F022
Byte ciphertext = bytearray(b'\x81\x80\x9f\xe7{Rd\x01\xdf\xda\xed+\xd1\xcf\xc9P\xf2\x92\x8b\x98\tg.iM\xd6\x89&\x81rJ\xd5\xaf0:\xc3\x10\x08T\x82\x1cv$0\x10t\xd0\xdaV\x9c_\xc4o\xdb*\xde\xcb\xe0UoV\x85\xf0"')
Decrypted plaintext = 6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C37
Pycryptodome ciphertext =
b"\x8e\xe3,\xc4\x1a\x88\xb5\xa1\xd6\xba\x0eW\xf0\xcd\xe5\xe7G\x9c\x18\xb3~\x05\xfb\x92'\x8c\x14\xc0\xd2\x1a\x07\xf7WK\x0f\xd4\x06\x1e%8p[`\x18\x95\xce\x17\x97ny\tQ`\xce\x8e\x11\xcf\x99\x9d1\x1c\xdc'\x84"
```

Figure 2: Output comparison with *PyCryptoDome*

As it is possible to see in Fig. 2, the first four operation modes gives the same result as the implementation proposed, while the last mode, which is CTR, gives a different result since the nonce vector in *PyCryptoDome* library is only of 64 bit and the remaining 64 bits of IV are dedicated to counter, while in the implementation proposed the nonce and counter are brought together in a 128 bit vector.

5.2 Throughput Comparison

To have a fair comparison, the implementation proposed has been tested together with the *PyCryptoDome* library on 3 files of fixed dimensions: 1KB, 100KB and 10MB. The code for file processing is based on [3]. As specified on the request, only 3 operation modes are tested, in this case it has been chosen to use ECB, OFB and CBC. To optimize time, the test are run a different number

of times: the 1KB file test is run 100 times, the 100KB is run 10 times and the 10MB test is run 5 times only, since the proposed implementation requires around 40 minutes to run the most intensive test.
The most efficient encryption implementation is ECB, which needs around 20 minutes to encrypt the 10MB files 5 times, it results in a medium time of 3.79 minutes and a throughput of 45KB/s; the other have smaller but similar throughput: 44.5KB/s for OFB and 43.6KB/s for CBC. These values were computed in the case of 10MB file. The other encryption throughputs can be seen in table 2.

Encryption	1KB	100KB	10MB
ECB	45,22 KB/s	45,17 KB/s	45 KB/s
ECB PyCryptoDome	7,5 MB/s	325 MB/s	714 MB/s
OFB	44,04 KB/s	44,28 KB/s	44,5 KB/s
OFB PyCryptoDome	6,97 MB/s	244,1 MB/s	433,92 MB/s
CBC	43,20 KB/s	43,42 KB/s	43,6 KB/s
CBC PyCryptoDome	6,96 MB/s	195,28 MB/s	442,49 MB/s

Table 2: AES encryption throughput

As seen in table 2, the library implementation is much more faster. That is normal since the library has undergone different improvements during its development. There's a thing in *PyCryptoDome* library that astonish me: growing the file dimension, also the throughput increases. Similar comparison happens in the decryption phase as shown in table 3; in this case it is expected a lower throughput since the decryption function is less optimized and it wastes computational time not storing the subkeys but computing them in every round.
The implementation proposed has a strange behavior in decryption: while in encryption the three modes chosen present at maximum a difference of 2 KB/s, in decryption the OFB is the only one having performances in line with encryption, while the other two modes has more or less half of the performance with respect to their counterpart in encryption.

6 Conclusion

Even if the implementation proposed is much slower than the standard library chosen as comparison, it works well and it does its dirty job. The code may

Decryption	1KB	100KB	10MB
ECB	22,14 KB/s	22,3 KB/s	22,287 KB/s
ECB PyCryptoDome	7,5 MB/s	488 MB/s	2,12 GB/s
OFB	44,61 KB/s	45,05 KB/s	45,26 KB/s
OFB PyCryptoDome	6,95 MB/s	244,07 MB/s	454,4 MB/s
CBC	22,01 KB/s	22,23 KB/s	22,43 KB/s
CBC PyCryptoDome	6,97 MB/s	244,08 MB/s	480,66 MB/s

Table 3: Implemented AES decryption throughput

not be much optimized or it can have been written in a better way but my knowledge in python is not so huge, but this was an opportunity to dust and improve my experience.

References

[1] 64bit nonce in aes-ctr, why not 128? `https://crypto.stackexchange.com/a/27000`.

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

[3] Python encryption and decryption with pycryptodome. `https://nitratine.net/blog/post/python-encryption-and-decryption-with-pycryptodome/`.

[4] Recommendations for block cipher modes of operation. `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf`.

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

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

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