Assignment: 5

Date of Assignment: 26 Sept 2017

Date of Submission: 8 Oct 2017 Time: 5:30 pm

Your assignment is to implement AES-256, meaning the AES algorithm with 256-bit keys.

Make sure that you clearly identify the members of your team (one or two people from your batch only).

Make sure that all members of your team understand the code and contribute equally, as much as possible, to its development.

Your program should be able to do each of the following, in accordance with the option command line argument:

1. read in the file of Hex lines and output the corresponding ciphertext in Hex;
2. read in an encrypted file and output the corresponding plaintext.

The Plaintext is:
00 44 88 CC
11 55 99 DD
22 66 AA EE
33 77 BB FF

The CipherKey is:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

1. Generate the subkeys and display in one file

2. Show your result of each step in each round and store in separate files for encryption as well as decryption.

CS361 Assignment 4--AES Crypto

Due: Friday, July 3, 2015 by midnight

Deliverables: You will be submitting your code electronically. Make sure that you clearly identify the members of your team (one or two people).

Your assignment is to implement AES-256, meaning the AES algorithm with 256-bit keys.

Make sure that all members of your team understand the code and contribute equally, as much as possible, to its development.

Below are the specific instructions for this assignment.

1. Please provide a README.txt file including your program timing information, along with the usual stuff.

2. The primary file name for this assignment is `AES.java`. The main method must exist in `AES.java`. Students can add more files as they wish but all files must be submitted.

3. The assignment must be compiled by `javac *.java`.

4. The assignment must be executed by the following command:

   ```
   java AES option keyFile inputFile
   ```

   where `keyFile` is a key file, `inputFile` (no extension) names a file containing lines of plaintext, and option is "e" or "d" for encryption and decryption, respectively. `keyFile` contains a single line of 64 hex characters, which represents a 256-bit key. The `inputFile` should have 32 hex characters per line. (However, see the procedures below for malformed or short lines.)

5. If option "e" is given, `inputFile` is encrypted with the key from `keyFile` and the program outputs an encrypted file named by adding extension ".enc" to the `inputFile`. If option "d" is given, `inputFile` is decrypted with a key from `keyFile` and the program outputs an encrypted file named by adding an extension ".dec" to the `inputFile`. Please be sure that your encrypted file

matches follows the same input format and your decrypted file must match your original plain text (modulo dealing with any malformed or short lines).

As an example, if you run:

```
java AES e key plaintext
```

The output should be in file `plaintext.enc`.

If you then run:

```
java AES d key plaintext.enc
```

The output would be in `plaintext.enc.dec`, and should match the original `plaintext` input file. Also, `plaintext.enc` must follow AES standards specified in the assignment page.

Background

Several former students described this exercise as the "most useful programming assignment" they had in their undergraduate careers, and one they mention in resumes and interviews with prospective employers. I hope that you find it useful as well.

The Advanced Encryption Standard (AES) is a very important commercial block cipher algorithm, designed to replace the earlier DES.

AES uses repeat cycles or ``rounds.'' There are 10, 12, or 14 rounds for keys of 128, 192, and 256 bits, respectively. The input text is represented as a 4 x 4 array of bytes. The key is represented as a 4 x n array of bytes, where n depends on the key size.

Each round of the algorithm consists of four steps:

1. **subBytes:** for each byte in the array, use its value as an index into a fixed 256-element lookup table, and replace its value in the state by the byte value stored at that location in the table. You can find the table and the inverse table on the web.
2. **shiftRows:** Let Ri denote the ith row in state. Shift R0 in the state left 0 bytes (i.e., no change); shift R1 left 1 byte; shift R2 left 2 bytes; shift R3 left 3 bytes. These are circular shifts. They do not affect the individual byte values themselves.
3. **mixColumns:** for each column of the state, replace the column by its value multiplied by a fixed 4 x 4 matrix of integers (in a particular Galois Field). This

is the most complex step. Find details at any of several websites. Note that the inverse operation multiplies by a different matrix.

4. **addRoundkey:** XOR the state with a 128-bit round key derived from the original key K by a recursive process.

For encryption, the final round is slightly different from the others and there is an additional addRoundKey. For decryption, the initial round is slightly different and there is an additional addRoundKey.

The Assignment

Your assignment is to implement AES-256. That means that you will use 128-bit input, 256-bit key, 14 rounds. While you are debugging, I'd suggest that you hard code the key and input block. After that, you should input them from a file as specified above.

There is a tremendous amount of information on the web about the AES algorithm. You can find it by searching for "Rijndael" or "Advanced Encryption Standard" on the web. The AES standard is here: AES standard. Any questions about the algorithm can be resolved by looking there. There is also a book, The Design of Rijndael by Joan Daemen and Vincent Rijmen that describes the algorithm in detail. One student found the following website particularly helpful in understanding AES. AES-128 **Note that the animation is for AES-128; AES-192 and AES-256 have some differences.**

Another student found the following two sites helpful: info on Key Schedule and info on mixColumns.

In past semesters, a significant number of folks have had problems with the key expansion algorithm. The algorithm is different depending on the key size. Apparently the Wikipedia page on AES has a pretty good description of the key generation. (However, one semester as a prank one of the students in the class changed one value in the S-box on Wikipedia and caused a lot of grief to other students.) Here is the description from the Rijndael book on key expansion: Key Expansion. Note that the algorithm for 256-bit key expansion is the one on page 46. The one for 128-bit key expansion is on page 45.

Use Electronic Code Book (ECB) mode. That is, encode each block separately, without chaining blocks together.

Note: mixColumns is by far the most difficult step of this algorithm. If you can't figure out how to do it, ask me and I'll give you my mixColumns code without penalty.

There are implementations of AES on the web and a reference implementation in the back of the Daemen/Rijmen book. *Don't use them. You are expected to code your own implementation. Also, do not consider using a friend's implementation from a previous semester. The TA has all of those and will check that you have not used an old submission. It's far better for your grade if you skip this assignment rather than to be caught cheating.*

After you have completed your coding of the algorithm, turn off all unnecessary IO, and encrypt a large file. That file should be a series of lines, each containing a single 128-bit number in hex format. That is, you do not have to worry about encrypting an ASCII file or other file type. I'm less concerned that you can do that, than that you have the algorithm correct.

Just to be clear, your input file might start with the following lines:

```
0A935D11496532BC1004865ABDCA4295
00112233445566778899AABBCCDDEEFF
   ....
```

You'll read in a line, converting from Hex to binary for storage into your state array. Apply the AES algorithm to encrypt the string as stored, and write out the ciphertext in Hex notation to the output file. When decrypting you'll reverse the process. If any line contains non-Hex characters, skip that line. You do not have to generate any output for malformed input. If any line is less than 32 Hex characters (128-bits) pad on the right with zeros. If the line is longer than 32 Hex characters, only consider the first 32. Treat the padded/truncated line as the input; you don't need to reflect in the output that you padded or truncated the line. Your program should be able to deal with uppercase, lowercase or mixed input.

The key is stored in a separate file, but in the same format. For example, for a 256-bit case, your key file might have the following single line (correspondingly shorter for the 128-bit and 192-bit case):

```
FFEEDDCCBBAA00998877665544332211FFEEDDCCBBAA00998877665544332211
```

Your program should be able to do each of the following, in accordance with the option command line argument:

1. read in the file of Hex lines and output the corresponding ciphertext in Hex;
2. read in an encrypted file and output the corresponding plaintext.

Take measurements of the throughput. That is, how many MB/sec can you encrypt using your implementation? Then decrypt the file and measure the throughput. Compare encryption and decryption speeds.

The following is a simple Python program to generate 100 input test vectors: Test Vector Generator. However, it won't generate ill-formed test vectors, except that some may be short if there are leading zeros. Modify the call statement to generate more vectors or vectors of length other than 32 hex digits.

Don't get stumped by the mix-columns step. Try to understand how it works, but if you just can't figure it out, ask me for the code. You won't be penalized.

Extra Credit

What's above is all that you must do to complete the assignment. However, there are two extra steps you can complete for up to two extra points (1 for each part). In your README file, describe what you did to improve your program in either of the following two ways.

1. Modify your program so that it can run any of AES-128, AES-192 or AES-256. Your program should take an additional flag as input to say which mode to run in. Note that the key expansion algorithm is slightly different in these cases.
2. Instrument your program to run in either of CBC or ECB mode.

Below are the specific instructions for the extra credit component of this assignment:

- Provide a README.txt file including the string "Extra Credit".

- Your extra credit assignment should be executed as follows.
- `java AES option [length] [mode] keyFile inputFile`

  where option is "e" or "d"; length is "-length 128" for 128 bits, length is "-length 192" for 192 bits or "-length 256" for 256 bits; mode is "-mode ecb" for ECB mode or "-mode cbc" for CBC mode;`keyFile` is a key file containing a key of the appropriate length; and `inputFile` is an input file to encrypt or decrypt depending on `option`.

- The default length and mode are 128 and ECB if they are not given.

Test Output for AES-128

Below is some test output from my program. All of this data is in hexadecimal notation. It uses plaintext and key consisting of all 0's. Also shown is the result of each step of the transformation. You don't have to produce output that looks like this; it's here to allow you to see the various transformations so that you can check whe ther your program is behaving as it should.

Make sure that you are getting the correct intermediate values of the array. It's easy to write an algorithm that inverts, but is not AES. It's not enough that you get the original input if you encrypt and then decrypt.

```
The Plaintext is:
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00

The CipherKey is:
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00

The expanded key is:
00000000 62626262 9BF99BF9 9069F20B EE87757E 7FF88DF3 EC14996A 2135ACC6
0E3B9751 B18A1D4C B43E236F
00000000 63636363 98FB98FB 976CF40F 066A9E91 2E44DA4B 6125FFB4 7550AF1B
F9A9061D D47D7B66 EF92E98F
00000000 63636363 98FB98FB 34CF57AC DA1542EE 2B3E7C92 4B75099B 17626BF0
03610AFA D8B9B349 5BE25118
00000000 63636363 C9AAC9AA 50FA3399 7B81B22B 8809BB90 858C37A7 870B3C9B
3338049F E2DADE41 CB11CF8E

After addRoundKey(0):
00000000000000000000000000000000
After subBytes:
63636363636363636363636363636363
After shiftRows:
63636363636363636363636363636363
After mixColumns:
63636363636363636363636363636363
After addRoundKey(1):
01000000010000000100000001000000
After subBytes:
7C6363637C6363637C6363637C636363
After shiftRows:
7C6363637C6363637C6363637C636363
After mixColumns:
5D7C7C425D7C7C425D7C7C425D7C7C42
After addRoundKey(2):
C6E4E48BA48787E8C6E4E48BA48787E8
After subBytes:
B469693D4917179BB469693D4917179B
After shiftRows:
B417699B4969173DB417699B4969173D
After mixColumns:
B8BAC794039F49DFB8BAC794039F49DF
After addRoundKey(3):
282DF3C46AF386254A4E90A70890E546
After subBytes:
34D80D1C020D443FD62F605C3060D95A
After shiftRows:
```

```
340D605A022FD91CD6600D3F30D8445C
After mixColumns:
45D41785B030A0C8253EED720B0B8474
After addRoundKey(4):
ABD2CDFE375AB54950A0AFC0759A6A5F
After subBytes:
62B5BDBB9ABED53B53E079BA9DB802CF
After shiftRows:
62BE79CF9AE002BB53B8BD3B9DB5D5BA
After mixColumns:
AB4164E4ADFCA83AF3DFC7868A324CB3
After addRoundKey(5):
D46F4F6C55B896337E05BB3D7979DE23
After subBytes:
48A88450FC6C90C3F36BEA27B6B61D26
After shiftRows:
486CEA26FC6B1D50F3B684C3B6A89027
After mixColumns:
E8938112135D5DC97BD008A123714CB7
After addRoundKey(6):
04F2CA9707782845E22F019649C5D710
After subBytes:
F2897488C5BC346E98157C903BA60ECA
After shiftRows:
F2BC7CCAC5150E8898A6746E3B893490
After mixColumns:
96DFF34228754F44C03D64BD52FE71CB
After addRoundKey(7):
B7AAE4C51D252D4F6C920F8194E58150
After subBytes:
A9AC69A6A43FD884504F760C22D90C53
After shiftRows:
A93F7653A44F0CA650D9698422ACD80C
After mixColumns:
2D1E8F0F288802E33DC6CC537F1E310A
After addRoundKey(8):
23E78C3C132163DBAAC0C6572E03CB95
After subBytes:
269464EB7DFDFBB9ACBAB45B317B1F2A
After shiftRows:
26FDB42A7DBA1FEBAC7B64B93194FB5B
After mixColumns:
CE2AD677DBD8DFEF134FCF99654FA58A
After addRoundKey(9):
7FFE0E9551A566350E347C472929ECCB
After subBytes:
D2BBAB2AD1063396AB1810A0A5A5CE1F
After shiftRows:
D206101FD118CE2AABA5AB96A5BB33A0
After addRoundKey(10):
66E94BD4EF8A2C3B884CFA59CA342B2E
The ciphertext:
66 EF 88 CA
E9 8A 4C 34
4B 2C FA 2B
D4 3B 59 2E
```

The Decryption (fixed from the earlier version):

After addRoundKey(10):
D206101FD118CE2AABA5AB96A5BB33A0
After invShiftRows:
D2BBAB2AD1063396AB1810A0A5A5CE1F
After invSubBytes:
7FFE0E9551A566350E347C472929ECCB
After addRoundKey(9):
CE2AD677DBD8DFEF134FCF99654FA58A
After invMixColumns:
26FDB42A7DBA1FEBAC7B64B93194FB5B
After invShiftRows:
269464EB7DFDFBB9ACBAB45B317B1F2A
After invSubBytes:
23E78C3C132163DBAAC0C6572E03CB95
After addRoundKey(8):
2D1E8F0F288802E33DC6CC537F1E310A
After invMixColumns:
A93F7653A44F0CA650D9698422ACD80C
After invShiftRows:
A9AC69A6A43FD884504F760C22D90C53
After invSubBytes:
B7AAE4C51D252D4F6C920F8194E58150
After addRoundKey(7):
96DFF34228754F44C03D64BD52FE71CB
After invMixColumns:
F2BC7CCAC5150E8898A6746E3B893490
After invShiftRows:
F2897488C5BC346E98157C903BA60ECA
After invSubBytes:
04F2CA9707782845E22F019649C5D710
After addRoundKey(6):
E8938112135D5DC97BD008A123714CB7
After invMixColumns:
486CEA26FC6B1D50F3B684C3B6A89027
After invShiftRows:
48A88450FC6C90C3F36BEA27B6B61D26
After invSubBytes:
D46F4F6C55B896337E05BB3D7979DE23
After addRoundKey(5):
AB4164E4ADFCA83AF3DFC7868A324CB3
After invMixColumns:
62BE79CF9AE002BB53B8BD3B9DB5D5BA
After invShiftRows:
62B5BDBB9ABED53B53E079BA9DB802CF
After invSubBytes:
ABD2CDFE375AB54950A0AFC0759A6A5F
After addRoundKey(4):
45D41785B030A0C8253EED720B0B8474
After invMixColumns:
340D605A022FD91CD6600D3F30D8445C
After invShiftRows:
34D80D1C020D443FD62F605C3060D95A
After invSubBytes:
282DF3C46AF386254A4E90A70890E546
After addRoundKey(3):

```
B8BAC794039F49DFB8BAC794039F49DF
After invMixColumns:
B417699B4969173DB417699B4969173D
After invShiftRows:
B469693D4917179BB469693D4917179B
After invSubBytes:
C6E4E48BA48787E8C6E4E48BA48787E8
After addRoundKey(2):
5D7C7C425D7C7C425D7C7C425D7C7C42
After invMixColumns:
7C6363637C6363637C6363637C636363
After invShiftRows:
7C6363637C6363637C6363637C636363
After invSubBytes:
01000000010000000100000001000000
After addRoundKey(1):
63636363636363636363636363636363
After invMixColumns:
63636363636363636363636363636363
After invShiftRows:
63636363636363636363636363636363
After invSubBytes:
00000000000000000000000000000000
After addRoundKey(0):
00000000000000000000000000000000
The decryption of the ciphertext:
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00

The decryption of the ciphertext:
00000000000000000000000000000000
```

Test Output for AES-256

Below is some test output from my program. Again the data is in hexadecimal notation. It uses plaintext 00112233445566778899AABBCCDDEEFF and key consisting of all 0's. Also shown is the result of each step of the transformation.

```
The Plaintext is:
00 44 88 CC
11 55 99 DD
22 66 AA EE
33 77 BB FF

The CipherKey is:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

The expanded key is:
00000000 00000000 62626262 AAAAAAAA 6F0D6F0D 7DD77DD7 535E313C 96413CEB
9EC0F1CD 2B6A56BD 64A45598 6D0751EC E743168E 747322CE 105345CB
```

```
00000000 00000000 63636363 FBFBFBFB 6C0F6C0F 8D768D76 545B3738 8AFC7107
AAF1C6FE 31CDBCBB 06F731CF BB76CA71 B04776B9 ED9B5120 F8BFC970
00000000 00000000 63636363 FBFBFBFB 6C0F6C0F 8D768D76 EDE28E81 81F77A0C
8F6DE362 2BDCA6AA FD907311 A975D379 E8780B1A 0B7EADD4 0A727963
00000000 00000000 63636363 FBFBFBFB CFACCFAC 6A916A91 C16DA20E C1503AAB
2845E7E9 DF8FB51E 5217F019 0B84312F 9C8B7B62 A125143B 179CE785

After addRoundKey(0):
00112233445566778899AABBCCDDEEFF
After subBytes:
638293C31BFC33F5C4EEACEA4BC12816
After shiftRows:
63FCAC161BEE28C3C4C193F54B8233EA
After mixColumns:
6379E6D9F467FB76AD063CF4D2EB8AA3
After addRoundKey(1):
6379E6D9F467FB76AD063CF4D2EB8AA3
After subBytes:
FBB68E35BF850F38956FEBBFB5E97E0A
After shiftRows:
FB85EB0ABF6F7E3595E98E38B5B60FBF
After mixColumns:
98C6AD6C9FD673A1A7ED33B3006CC718
After addRoundKey(2):
FAA5CE0FFDB510C2C58E50D0620FA47B
After subBytes:
2D068B7654D5CA25A6195370AA764921
After shiftRows:
2DD55321541949 76A6768B25AA06CA70
After mixColumns:
4C483DB3BCCB454063E9B246FF93B3C9
After addRoundKey(3):
E6B3C6481630BEBBC91249BD55684832
After subBytes:
8E6DB4524704AEEADDC93B7AFC455223
After shiftRows:
8E043B2347C95252DD45B4EAFC6DAE7A
After mixColumns:
13E899F0CE6ADCF6307ACE4280B55828
After addRoundKey(4):
7C84F53FC365D35A5F16A28D8DBA5784
After subBytes:
105FE6752E4D66BECF473A5D5DF45B5F
After shiftRows:
104D3A5F2E475B75CFF4E6BE5D5F665D
After mixColumns:
929BC8F9BB384084DAB3353F60142964
After addRoundKey(5):
EF1645936C4E3615A73EB855B7625FF5
After subBytes:
DF476EDC502F05595CB26CFCA9AACFE6
After shiftRows:
DF2F6CE650B2CFDC5CAA6E59A94705FC
After mixColumns:
5ED319EE7EB9182E6AF8C19279D4FB41
After addRoundKey(6):
0D87F42F20E2FA435BCF4F3045EC7A4F
```

```
After subBytes:
D717BF15B7982D1A398A84046ECEDA84
After shiftRows:
D7988484B78ADA1539CEBF1A6E172D04
After mixColumns:
06EFCB6D3FD8ADB89E7EBC0ECC332F80
After addRoundKey(7):
90654AAC7E245AE8A20FC6342734232B
After subBytes:
604DD691F336BE9B3A76B418CC1826F1
After shiftRows:
6036B4F1F37626913A18D69BCC4DBE18
After mixColumns:
DF3A2DDBD0E4616711F023ADF297CE8C
After addRoundKey(8):
4190A2F310150C22E036C04A3F69AC65
After subBytes:
83603A0DCA59FE93E105BAD675F9914D
After shiftRows:
8359BA4DCA05910DE1F93A937560FED6
After mixColumns:
01A962E71C65E1CB60D5C2C6627A93B6
After addRoundKey(9):
2A98493876A83D4436696473DFC139A8
After subBytes:
E5463B0738C2271B05F9438F9E7812C2
After shiftRows:
E5C243C238F9120705783B1B9E46278F
After mixColumns:
0D7DFC2A75E0ECADA2A3267A45F41CDD
After addRoundKey(10):
697B0178D1177CBAF792558ADD3B0DC4
After subBytes:
F9217CBC3EF010F4684FFC7EC1E2D71C
After shiftRows:
F9F0FC1C3E4FD7BC68E27CF4C121107E
After mixColumns:
0201CE24C67E1BB965C775D594CD4295
After addRoundKey(11):
6FBA672FC1086E3D340DA6E478BC3BBA
After subBytes:
A8F4851578309F2718D72469BC65E2F4
After shiftRows:
A83024F478D7E21518658527BCF49F69
After mixColumns:
CB50D70465E54F973D610586929CD666
After addRoundKey(12):
2CE03F9826A2371C2B170EFD1C25CC04
After subBytes:
71E17546F73A9A9CF1F0AB549C3F4BF2
After shiftRows:
713AABF2F7F04B46F13F759C9CE19A54
After mixColumns:
F5110BFDF3975B35518C9B61D5A4AE6C
After addRoundKey(13):
81FC005C800C251073DD36751B847A57
After subBytes:
```

```
0CB0634ACDFE3FCA8FC1059DAF5FDA5B
After shiftRows:
0CFE055BCDC1DA4A8F5F63CAAFB03F9D
After addRoundKey(14):
1C060F4C9E7EA8D6CA961A2D64C05C18
The ciphertext:
1C 9E CA 64
06 7E 96 C0
0F A8 1A 5C
4C D6 2D 18
```

The Decryption Phase

```
After addRoundKey(14):
0CFE055BCDC1DA4A8F5F63CAAFB03F9D
After invShiftRows:
0CB0634ACDFE3FCA8FC1059DAF5FDA5B
After invSubBytes:
81FC005C800C251073DD36751B847A57
After addRoundKey(13):
F5110BFDF3975B35518C9B61D5A4AE6C
After invMixColumns:
713AABF2F7F04B46F13F759C9CE19A54
After invShiftRows:
71E17546F73A9A9CF1F0AB549C3F4BF2
After invSubBytes:
2CE03F9826A2371C2B170EFD1C25CC04
After addRoundKey(12):
CB50D70465E54F973D610586929CD666
After invMixColumns:
A83024F478D7E21518658527BCF49F69
After invShiftRows:
A8F4851578309F2718D72469BC65E2F4
After invSubBytes:
6FBA672FC1086E3D340DA6E478BC3BBA
After addRoundKey(11):
0201CE24C67E1BB965C775D594CD4295
After invMixColumns:
F9F0FC1C3E4FD7BC68E27CF4C121107E
After invShiftRows:
F9217CBC3EF010F4684FFC7EC1E2D71C
After invSubBytes:
697B0178D1177CBAF792558ADD3B0DC4
After addRoundKey(10):
0D7DFC2A75E0ECADA2A3267A45F41CDD
After invMixColumns:
E5C243C238F9120705783B1B9E46278F
After invShiftRows:
E5463B0738C2271B05F9438F9E7812C2
After invSubBytes:
2A98493876A83D4436696473DFC139A8
After addRoundKey(9):
01A962E71C65E1CB60D5C2C6627A93B6
After invMixColumns:
8359BA4DCA05910DE1F93A937560FED6
After invShiftRows:
```

83603A0DCA59FE93E105BAD675F9914D
After invSubBytes:
4190A2F310150C22E036C04A3F69AC65
After addRoundKey(8):
DF3A2DDBD0E4616711F023ADF297CE8C
After invMixColumns:
6036B4F1F37626913A18D69BCC4DBE18
After invShiftRows:
604DD691F336BE9B3A76B418CC1826F1
After invSubBytes:
90654AAC7E245AE8A20FC6342734232B
After addRoundKey(7):
06EFCB6D3FD8ADB89E7EBC0ECC332F80
After invMixColumns:
D7988484B78ADA1539CEBF1A6E172D04
After invShiftRows:
D717BF15B7982D1A398A84046ECEDA84
After invSubBytes:
0D87F42F20E2FA435BCF4F3045EC7A4F
After addRoundKey(6):
5ED319EE7EB9182E6AF8C19279D4FB41
After invMixColumns:
DF2F6CE650B2CFDC5CAA6E59A94705FC
After invShiftRows:
DF476EDC502F05595CB26CFCA9AACFE6
After invSubBytes:
EF1645936C4E3615A73EB855B7625FF5
After addRoundKey(5):
929BC8F9BB384084DAB3353F60142964
After invMixColumns:
104D3A5F2E475B75CFF4E6BE5D5F665D
After invShiftRows:
105FE6752E4D66BECF473A5D5DF45B5F
After invSubBytes:
7C84F53FC365D35A5F16A28D8DBA5784
After addRoundKey(4):
13E899F0CE6ADCF6307ACE4280B55828
After invMixColumns:
8E043B2347C95252DD45B4EAFC6DAE7A
After invShiftRows:
8E6DB4524704AEEADDC93B7AFC455223
After invSubBytes:
E6B3C6481630BEBBC91249BD55684832
After addRoundKey(3):
4C483DB3BCCB454063E9B246FF93B3C9
After invMixColumns:
2DD5532154194976A6768B25AA06CA70
After invShiftRows:
2D068B7654D5CA25A6195370AA764921
After invSubBytes:
FAA5CE0FFDB510C2C58E50D0620FA47B
After addRoundKey(2):
98C6AD6C9FD673A1A7ED33B3006CC718
After invMixColumns:
FB85EB0ABF6F7E3595E98E38B5B60FBF
After invShiftRows:
FBB68E35BF850F38956FEBBFB5E97E0A

```
After invSubBytes:
6379E6D9F467FB76AD063CF4D2EB8AA3
After addRoundKey(1):
6379E6D9F467FB76AD063CF4D2EB8AA3
After invMixColumns:
63FCAC161BEE28C3C4C193F54B8233EA
After invShiftRows:
638293C31BFC33F5C4EEACEA4BC12816
After invSubBytes:
00112233445566778899AABBCCDDEEFF
After addRoundKey(0):
00112233445566778899AABBCCDDEEFF
The decryption of the ciphertext:
00 44 88 CC
11 55 99 DD
22 66 AA EE
33 77 BB FF

The decryption of the ciphertext:
00112233445566778899AABBCCDDEEFF
```