EECS 4980 / 5980 Programming Assignment
Due Thursday, March 23, 2017 at 11:59:59 PM.

> ***NOTE***: Under no circumstances are you allowed to use *any* code from *any* source other than what you individually create from scratch, or are provided by the instructor. You may, of course, use any information in the text and the lecture slides, including the links provided on AES. Source code for AES is readily available from the web; *don't use any of it, even as a reference (except as explicitly allowed). If I believe you didn't write your code, I'll ask you to explain it, potentially line-by-line, to prove to me that it's yours and yours alone.*

Your assignment is to write a (single) program to do AES encryption and decryption. Use Visual Studio 2015 to code your **Win32 Console Application** in C/C++. Call your program "AES".

The command-line syntax for your program will be:

```
AES <-action> <key> <mode> <infile> <outfile>
```

All five parameters are required. The `<action>` will either be "`-d`" or "`-e`" (case insensitive). If the action is "`-d`", you are to decrypt the ciphertext file specified by `<infile>` and place the resulting plaintext into `<outfile>`. If the action is "`-e`", you are to encrypt the file specified by `<infile>` and place the ciphertext into `<outfile>`. The `<key>` is specified as either 32 hex digits (case-insensitive), or a string of up to 16 characters enclosed in single quotes (note: if your 16-character key contains embedded spaces (which IS allowed), the whole thing will have to be enclosed in **double** quotes). The `<mode>` parameter is to be "ECB" or "CBC" (case-insensitive), and you are to implement both ECB and CBC modes as with the second DES project (obviously, the IV for AES will need to be longer than it was for DES, but use the same approach – select it randomly, encrypt it with the key (ECB), and then use the IV for the remainder of the stream in CBC mode).

Examples:

```
AES –E 0123456789ABCDEF0123456789ABCDEF ECB plaint.txt cipher.txt
AES –D 0123456780ABCDEF0123456789abcdef ECB cipher.txt plain(2).txt
AES –e 'Pa$$wordPa$$w0rd' cbc in.txt out.txt
AES –d 'Pa$$wordPa$$w0rd' cbc out.txt in2.txt
AES –E "'Some 128-Bit Key'" ECB in.txt out.enc
```

AES works with 128-bit (16-byte) blocks at a time. If the length of the file you wish to encode does not happen to be an even multiple of 16, then you will have to pad the final block to make 128 bits. You will handle padding as follows:

Your program should be written to handle 31-bit file lengths (0 – 2 billion bytes). The first thing your program should do is establish the length of `<infile>`, and use that file size as the right 32 bits of the first block to encrypt. The upper 96 bits should be randomly generated garbage (we'll discard it later). Encrypt the first block (containing the file length) and write that block to `<outfile>`.

The second block you produce will be the actual first block from the data in `<infile>`. If, in the final block of the data file, you have to pad the file to get an even multiple of 16 bytes, fill the padding bytes with random characters (0-255). Do not pad with static data (like all zeroes or all 0xFF)

When you decrypt, read the first block and decrypt it. Mask off the upper 12 bytes, leaving a 32-bit file size. Then process the remaining blocks. When you get to the last block, if the file size is not a multiple of 16 bytes, output only the necessary number of bytes (and discard the remaining garbage).

The result of this padding scheme is that your ciphertext file will be 16 to 31 bytes longer than your plaintext file. If the original file was a multiple of 16 bytes in length, the ciphertext file will be 16 bytes longer to accommodate the 128-bit file size block. If you have to pad with 1 to 15 bytes to get an even multiple of 16 bytes, then your ciphertext file will be 17 to 31 bytes longer than the plaintext file.

When you decrypt a ciphertext file, your decrypted file should be bit-identical to the original plaintext file – no extra characters, etc.).

The Windows command-line program FC (File Compare) can be used with the "/B" option to compare two files at the binary level:

```
>FC /B file1 file2
```

Submit your code as a ZIP or 7-Zip archive (bzip, gzip, tar, rar, etc. are not acceptable) of the entire project tree (including all source and binary subdirectories). Post your submission to Blackboard. Make sure your code is commented, and that the comments include your name!

Please place in your ZIP/7-Zip archive a (single) PDF of (all of) your code. If your source code is in multiple files, then gather them all up into the PDF. There should be a SINGLE file (this PDF) in your archive that comprises all I should need to print in order to see all of your code. I should not have to go digging through your directories to gather up your source files.

Your code should be well-designed (appropriately modularized) and well-documented. There is no reason for shoddy documentation; in fact, extra documentation will probably help you write the code. Resist the temptation to code it all with minimal / no comments and then add the documentation later.

The comments in your code should demonstrate to me that you understand how the code works.

If you have any questions, or run into difficulties, e-mail me as soon as possible. Do not delay in starting your work; there's a lot to do here. None of it is particularly difficult, but as they say, "the devil is in the details." Debugging crypto code is particularly challenging, because, even when it works right, the output looks like garbage.

DO NOT use bitsets or any other STL type or extension to C/C++. It should go without saying that you may not use a crypto library. You should implement the state as `unsigned char[4][4]`

Think in terms of efficiency. Your completed code should run quickly (after all, this IS code that would be built-in to system utilities; we don't want to wait while we number-crunch our files just to encrypt / decrypt them!). I am providing a 5 MB test file, SHAKESPEARE.TXT, which contains the complete works of Shakespeare's plays. Your program should encrypt / decrypt this file in well under 10 seconds, depending on your CPU and speed. On my 5-year-old desktop PC, my AES code process the Shakespeare file in about 3 seconds if it's on the hard drive; about double that on a flash drive.

Graduate Students – you are required to be held to a higher performance standard than the undergraduates, and you also take fewer classes per semester, leaving you more time to devote to each individual class. In light of this, you are required to implement not only 128-bit keys, but you are to also support 192- and 256-bit keys (of course, this changes the length of the keys you have to support in the command-line parameters)