AES Readme

Joseph Tyler Manahan

October 2018

1 Overview

This implementation of AES is written entirely in *Python 3.6.1*, following the *NIST* standard in *ECB* mode. Source code complies with the *PEP8* style-guide.

1.1 Usage

AES.py can be invoked directly from the CLI with the following usage. Arguments may be passed in any order, so long as all arguments are present.

```
python3 AES.py –keysize {128 | 256} –keyfile $KEYFILE
–inputfile $INPUTFILE –outputfile $OUTFILE
–mode {encrypt | decrypt}
```

2 Structure

AES.py is comprised of multiple sub-classes that serve to abstract the various structures present in the Rijndael algorithm.

2.1 AES

The *AES* class contains the *encrypt()* and *decrypt()* methods, which are called on an instance of the class in order to perform cryptographic operations. The other methods in the class are intended to be used during the process of encryption and decryption. The class also contains several constant arrays that are used in the sub-bytes and mix-columns steps, as well as their inverse and *RCON*.

2.1.1 Block

This is the centerpiece of the implementation. *Blocks* are an abstraction for a 4x4 matrix of bytes. They are used to represent rounds in a key schedule, as well as states for encrypting and decrypting. The *Block* class allows for easy manipulation of the data within the matrix through the use of the *Row* and *Column* subclasses.

2.1.2 Key

The *Key* class abstracts the process of generating and referencing the round keys. Each round is represented by a *Block*, with a single key containing as many blocks (plus one) as there are rounds in the schedule. The class can expand 128-bit and 256-bit keys.

3 Program Flow

When *AES.py* is invoked on the command line, the *__main__* method parses CLI arguments and passes them to the *main()* method, which constructs an *AES* instance. On construction, the key defined by the keyfile is expanded by the *Key* class. For a 128-bit *Key*, a single *Block* is read in from the keyfile to create the cypher key, whereas a 256-bit key requires two *Blocks* of keyfile to begin. After the original *Blocks* are generated, the following round keys are appended to the rounds array of the *Key*. Depending on the mode selected, the program executes either the *encrypt()* or *decrypt()* method on the *AES* class instance created by the CLI arguments. When the *encrypt()* or *decrypt()* method is invoked, a *ThreadPoolExecutor* is created to manage the number of threads created by the process. After that, the inputfile is opened in read-bytes mode. The filesize in bytes is then determined in order to calculate how many *Blocks* will need to be written. For each *Block* of the input, a thread is generated that will run the appropriate (*encryptBlock()* or *decryptBlock()*) method. These methods follow the pseudo-code defined in the *NIST* specification, ultimately encrypting or decrypting the *Block* or 'state', and returning a *bytes* object that represents the 16 bytes within. After the thread is created, it is pushed onto a queue of threads and calls up on a semaphore that acts as a bounded buffer - ~~waking the *writeBlocks()* method.~~* When encrypting, the last *Block* that is enqueued will always contain some padding, while no other *Block* will contain any padding. If the last *Block* in the inputfile lands perfectly on a 16-byte boundary, an extra block is created containing 16 bytes of padding. The number of bytes padded is denoted in *CMS* format, meaning that any the value of any padded byte will be the number of padded bytes, with the bottom right byte of the padded *Block* ensured to be padded. When decrypting, the last *Block* dequeued will always contain some padding. The *writeBlocks()* method is designed to take a parameter denoting the number of *Blocks* that are padded in the queue, so that it may account for the padded bytes accordingly. This is only ever used when decrypting, since encrypted *Blocks* need to be written out fully, while decrypted *Blocks* should only write the number of bytes present in the original file.

* - This feature is deprecated but still present, as the program no longer "streams" data to and from files.

4 Conclusions

The implementation was originally designed to be multi-threaded, however, during testing, it was discovered that using a single thread outperformed any greater number of threads by an order of magnitude. This could be due to the limited memory capacity of the MacBook that it was developed and tested on, though it is incredibly likely that this is due to inefficient code. Some of these inefficiencies are likely due to the fact that older builds of this program "streamed" data from files in order to prevent memory overload. Threading would have been more helpeful in this case, as it would have allowed *Blocks* to be encrypted while other *Blocks* were being written. Although the "streaming" design was ultimately scrapped as the process was excruciatingly slow, it may be the case that some sections of code were not optimized for the new design. As this was my first Python program, I have yet to research all of the time complexities for standard functions and operations.

TL;DR: *Please don't shame me for my shitty code, Asper.*