Python Implementation of Advanced Encryption Standard

Minghui Liu

December 12, 2016

Abstract

This project implements Advanced Encryption Standard(AES) using python for learning purposes. It uses fixed block size of 128 bits and fixed key size of 256 bits and supports three modes of operations: ECB, CBC and OFB. The correctness of this implementation is verified by comparing encryption and decryption results with NIST known test vectors. The performance of this project is compared to the implementation in pyCrypto, and is found to be significantly slower.

1 Introduction

This project is a programming project aiming to implement AES using python for learning purposes. The design goal of this project is to break down and implement AES in a way that makes studying the internals of AES easier. It was not designed for real world production use and will not have the same level of security or performance as libraries used in production.

2 Implementation Details

This project is written in Python 2.7. It is strongly recommended that you run this project with Python 2.7 as you might get errors using Python 3.

This implementation uses 128-bit fixed block size and 256-bit fixed key size. It supports three modes of operation, Electronic Code Book(ECB), Cipher Block Chaining(CBC) and Output Feedback (OFB). The use of ECB is strongly discouraged as it leaks information about the plain text. It is only included for experiment.

This project has a single file, aes.py, and a single class called AES that encompasses everything. Users need to instantiate an instance of the AES class with a key to do encryption or decryption. All AES round functions are written according to AES standard. Note that AES standard defined AES stable table using column major order. To make coding easier, a transpose function is written to convert state from column major to row order, which is easier to program in Python.

```
def transpose(state):
  # return tranposed state
```

Since AES is a block cipher, inputs need to be padded so that its size is a multiple of AES block size. This implementation uses PKCS7 padding scheme to pad inputs.

```
def appendPadding(bytes):
  # pad input to multiples of block size

def stripPadding(bytes):
  # strip padding
```

Note that PKCS7 always pads input even if the input size is a multiple of block size, in which case PKCS7 appends an extra block to input.

3 How to use this project

3.1 Use the command line interface

This implementation comes with a command line interface for encrypting and decrypting files using password. To see help on how to use the command line interface, run `python aes.py -h` in terminal. To encrypt a file, run `python aes.py -e [inputfile] -o [outputfile]`. The output file name is optional and `[inputfile].aes` will be used if you don't supply output file name. To decrypt a file, run `python aes.py -d [inputfile] -o [outputfile]`. The output file name is also optional here and `decrypted_[inputfile]` will be used by default. You will need to input a password to encrypt or decrypt a file.

3.2 Use it in your project

To use this implementation, place `aes.py` in the same directory as your python project and include the `AES` class from `aes`.

```
from aes import AES
```

Then your need to create a key. Note that the key must be in the form of a byte array (a list of integers between 0 and 255). You can create a random key using any random number generator or create a key from password using `passwordToKey` function in `AES` class.

```
# generate key using random integer generator
from random import randint
key = [randint(0, 255) for _ in xrange(32)]

# generate key using OS random string generator and convert to byte array
import os
key = map(ord, os.urandom(32))

# generate key from password using passwordToKey function
key = AES.passwordToKey("p@33w0rd")
```

Next you need to create an AES instance using the key you just generated.

```
cipher = AES(key)
```

To encrypt a file, call your instance's `encrypt` method and specify Mode of operation. Valid choices are `"ECB"`, `"CBC"` and `"OFB"`. For CBC mode and OFB mode, you can supply your own Initialization Vector. If you don't then a random one will be generated.

```
# encrypt using ECB Mode
cipher.encrypt(plaintext, "ECB")

# encrypt using CBC Mode and random IV
cipher.encrypt(plaintext, "CBC")

# encrypt using CBC Mode and suply your own IV
myIV = map(ord, os.urandom(16))
cipher.encrypt(plaintext, "CBC", iv=myIV)
```

To decrypt, use your instance's `decrypt` method.

```
# decrypt using CCB Mode
cipher.decrypt(ciphertext, "CBC")
```

To encrypt or decrypt files, use your instance's `encryptFile` and `decryptFile` methods. You need to specify an input file path. Output file name is optional as this implementation will give a default output name if you don't supply one. Mode and IV are also optional and will default to CBC and a random generated IV.

```
# encrypt test.pdf
cipher.encryptFile("test.pdf")
```

```
# encrypt to test.pdf.aes using CBC Mode and suply your own IV
myIV = map(ord, os.urandom(16))
cipher.encrypt("test.pdf", "test.pdf.aes", "CBC", iv=myIV)

# decrypt test.pdf.aes
# output file name will be decrypted_test.pdf
cipher.decrypt("test.pdf.aes")
```

4 Correctness

The correctness of this implementation is verified again NIST known AES test vectors. The tests are defined in `correctness.py`. To run the tests, open a terminal window and run `python correctness.py`.

5 Performance

The speed of this implementation is measured against `pyCrypto`, a widely used Crypto library for Python. All tests are performed on the author's laptop. Both encryption and decryption time for long randomly generated strings are measured for both implementations and compared. To run the performance test, run `python performance.py`.

Mode	Size	Encryption Time(seconds)		Decryption Time(seconds)	
		pyCrypto	This Implementation	pyCrypto	This Implementation
ECB	64KB	0.001	12.045	0.001	12.683
	256KB	0.002	41.764	0.002	43.462
	1MB	0.009	174.777	0.009	218.034
CBC	64KB	0.001	11.113	0.001	10.885
	256KB	0.003	43.296	0.003	42.9
	1MB	0.021	191.320	0.022	183.890
OFB	64KB	0.001	10.059	0.001	11.022
	256KB	0.003	45.343	0.003	45.468
	1MB	0.008	191.545	0.01	207.462

Table 1: Performance comparison.

As evident from Table 1 and 2, this implementation is significantly slower than `pyCrypto`, which is expected. Most of the operations in pyCrypto are vectorized to matrix multiplication, which makes the code run much faster.

References

[BBF+02] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient software implementation of aes on 32-bit platforms. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 159–171. Springer, 2002.

[BI02] Lawrence E Bassham III. The advanced encryption standard algorithm validation suite (aesavs). *NIST Information Technology Laboratory*, 2002.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[Mag16] DI Magagement. Using padding in encryption, 2016. [Online; accessed 30-March-2016].

[RD01] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[Sel10] Douglas Selent. Advanced encryption standard. *Rivier Academic Journal*, 6(2):1–14, 2010.

[Sta01] NIST-FIPS Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:1–51, 2001.

[Wik16a] Wikipedia. Advanced encryption standard — wikipedia, the free encyclopedia, 2016. [Online; accessed 9-December-2016].

[Wik16b] Wikipedia. Aes implementations — wikipedia, the free encyclopedia, 2016. [Online; accessed 7-November-2016].

[Wik16c] Wikipedia. Block cipher mode of operation — wikipedia, the free encyclopedia, 2016. [Online; accessed 9-December-2016].

[Wik16d] Wikipedia. Padding (cryptography) — wikipedia, the free encyclopedia, 2016. [Online; accessed 11-July-2016].