Block Ciphers

Task 1

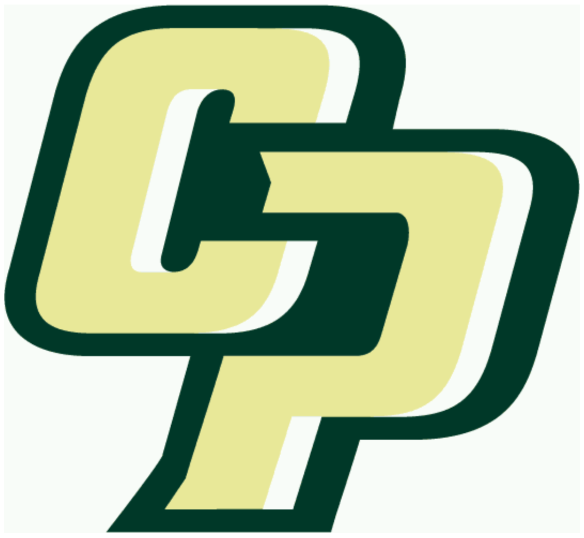
```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

blockSizeBytes = 16

def padByteArray(array):
   if (len(array) % blockSizeBytes != 0):
       newLen = (len(array) // blockSizeBytes + 1) * blockSizeBytes
       addedBytes = newLen - len(array)
       padding = bytearray(addedBytes.to_bytes(1, 'little')) * addedBytes
       # print(padding)
       # print(array)
       return array + padding
   else:
       return array


def xorByteArray(arr1, arr2):
   if len(arr1) != len(arr2):
       print("ERROR: arrays are not of equal length")
       return
   newArray = bytearray()
   for i in range(len(arr1)):
       newArray.append(arr1[i] ^ arr2[i])
   return newArray


def ECBEncryptBinary(key, plaintext):
   if len(key) != blockSizeBytes:
       print("ERROR: key is not of length " + str(blockSizeBytes))
       return

   plaintext = padByteArray(plaintext)

   cipher = Cipher(algorithms.AES(key), modes.ECB())
   encryptor = cipher.encryptor()
   cipherText = bytearray()

   for i in range(0, len(plaintext) // blockSizeBytes):
       cipherTextBuf = encryptor.update(plaintext[i * blockSizeBytes: i *
blockSizeBytes + blockSizeBytes])
       cipherText += cipherTextBuf

   return cipherText
```

```
def CBCEncryptBinary(key, initVector, plaintext):
   if len(key) != blockSizeBytes:
       print("ERROR: key is not of length " + blockSizeBytes)
       return
   if len(initVector) != blockSizeBytes:
       print("ERROR: key is not of length " + blockSizeBytes)
       return

   plaintext = padByteArray(plaintext)

   cipher = Cipher(algorithms.AES(key), modes.ECB())
   encryptor = cipher.encryptor()
   cipherText = bytearray()
   lastBlock = bytearray(initVector)

   for i in range(0, len(plaintext) // blockSizeBytes):
       prePlainText = plaintext[i * blockSizeBytes: i * blockSizeBytes +
blockSizeBytes]
       preCipherText = xorByteArray(prePlainText, lastBlock)
       cipherTextBuf = encryptor.update(preCipherText)
       lastBlock = cipherTextBuf
       cipherText += lastBlock

   return cipherText


def ECBDecryptBinary(key, cipherText):
   if len(key) != blockSizeBytes:
       print("ERROR: key is not of length " + blockSizeBytes)
       return

   plaintext = bytearray()
   cipher = Cipher(algorithms.AES(key), modes.ECB())
   decryptor = cipher.decryptor()

   for i in range(0, len(cipherText) // blockSizeBytes):
       plaintext += decryptor.update(cipherText[i * blockSizeBytes: i *
blockSizeBytes + blockSizeBytes])

   return plaintext


def CBCDecryptBinary(key, initVector, cipherText):
   if len(key) != blockSizeBytes:
       print("ERROR: key is not of length " + blockSizeBytes)
       return
   if len(initVector) != blockSizeBytes:
       print("ERROR: key is not of length " + blockSizeBytes)
       return

   plainText = bytearray()
   cipher = Cipher(algorithms.AES(key), modes.ECB())
   decryptor = cipher.decryptor()
   lastBlock = initVector

   for i in range(0, len(cipherText) // blockSizeBytes):
       prePlaintext = decryptor.update(cipherText[i * blockSizeBytes: i *
blockSizeBytes + blockSizeBytes])
       newPlaintext = xorByteArray(prePlaintext, lastBlock)
       lastBlock = cipherText[i * blockSizeBytes: i * blockSizeBytes +
blockSizeBytes]
       plainText += newPlaintext

   return plainText


def encryptBMP(filepath):
   key1 = os.urandom(blockSizeBytes)
   key2 = os.urandom(blockSizeBytes)
   initVector = os.urandom(blockSizeBytes)

   with open(filepath, 'rb') as ecbInput:
       txt = bytearray(ecbInput.read())

   # print(txt)
   header = txt[0:54]
   ecbEncryptedMessage = ECBEncryptBinary(key1, txt)
   ecbEncryptedMessage = header + ecbEncryptedMessage[54:]

   with open('ECB.bmp', 'wb+') as file:
       file.write(ecbEncryptedMessage)

   # print(txt)
   header = txt[0:54]
   cbcEncryptedMessage = CBCEncryptBinary(key2, initVector, txt)
   cbcEncryptedMessage = header + cbcEncryptedMessage[54:]

   with open('CBC.bmp', 'wb+') as file:
       file.write(cbcEncryptedMessage)


def encryptFile(filepath):
   key1 = os.urandom(blockSizeBytes)
   key2 = os.urandom(blockSizeBytes)
   initVector = os.urandom(blockSizeBytes)

   with open(filepath, 'rb') as ecbInput:
       txt = bytearray(ecbInput.read())

   # print(txt)
   header = txt[0:54]
   ecbEncryptedMessage = ECBEncryptBinary(key1, txt)
   ecbEncryptedMessage = header + ecbEncryptedMessage[54:]

   with open('ECB.bmp', 'wb+') as file:
       file.write(ecbEncryptedMessage)
```

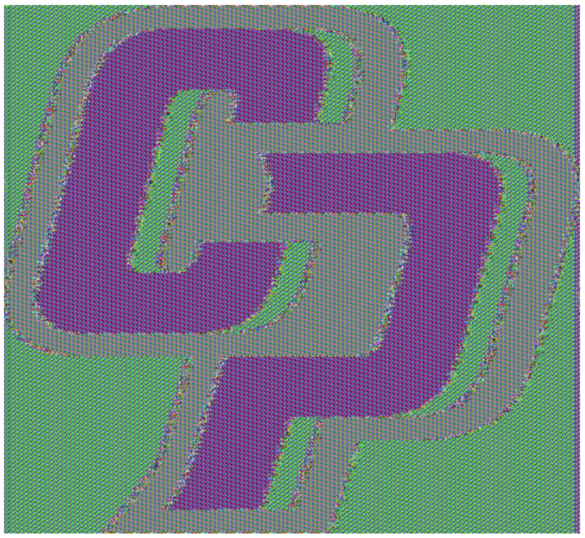
```
    # print(txt)
    header = txt[0:54]
    cbcEncryptedMessage = CBCEncryptBinary(key2, initVector, txt)
    cbcEncryptedMessage = header + cbcEncryptedMessage[54:]

    with open('CBC.bmp', 'wb+') as file:
        file.write(cbcEncryptedMessage)

    ecbDecryptedMessage = ECBDecryptBinary(key1, ecbEncryptedMessage)
    cbcDecryptedMessage = CBCDecryptBinary(key2, initVector, cbcEncryptedMessage)

    with open('ECBdec.bmp', 'wb+') as file:
        file.write(header + ecbDecryptedMessage[54:])

    with open('CBCdec.bmp', 'wb+') as file:
        file.write(header + cbcDecryptedMessage[54:])


encryptFile('cp-logo.bmp')
# encryptBMP('cp-logo.bmp')
```

cp-logo.bmp

Original Image



ECB (Encryption and Decryption)

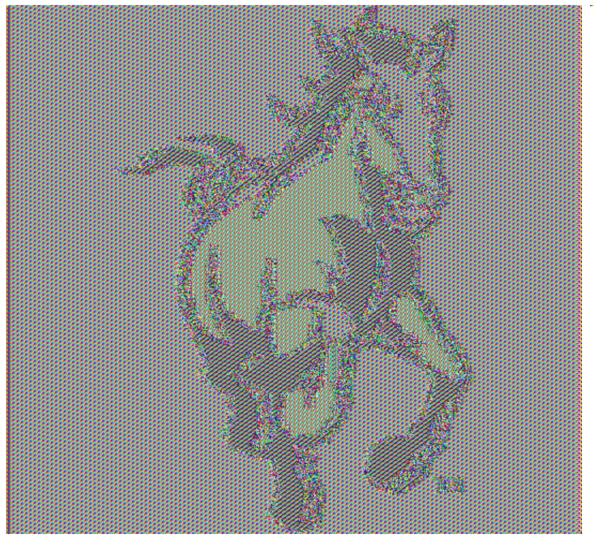


CBC (Encryption and Decryption)

mustang.bmp

Original Image

ECB (Encryption and Decryption)



CBC (Encryption and Decryption)

Question 1: For Task 1, when using ECB mode, the encryption altered the original image, but outlines of the original image still remained. When using CBC mode, the encryption altered the original image with no hints of the original image reamaining whatsoever. As seen from these images, CBC is a much more effective mode for encryption because each image block is not only sequentially encrypted with an AES key (i.e. ECB mode) but also XORed with the previous block, making a more unique ciphertext and removing any hints of identical image blocks being encrypted.

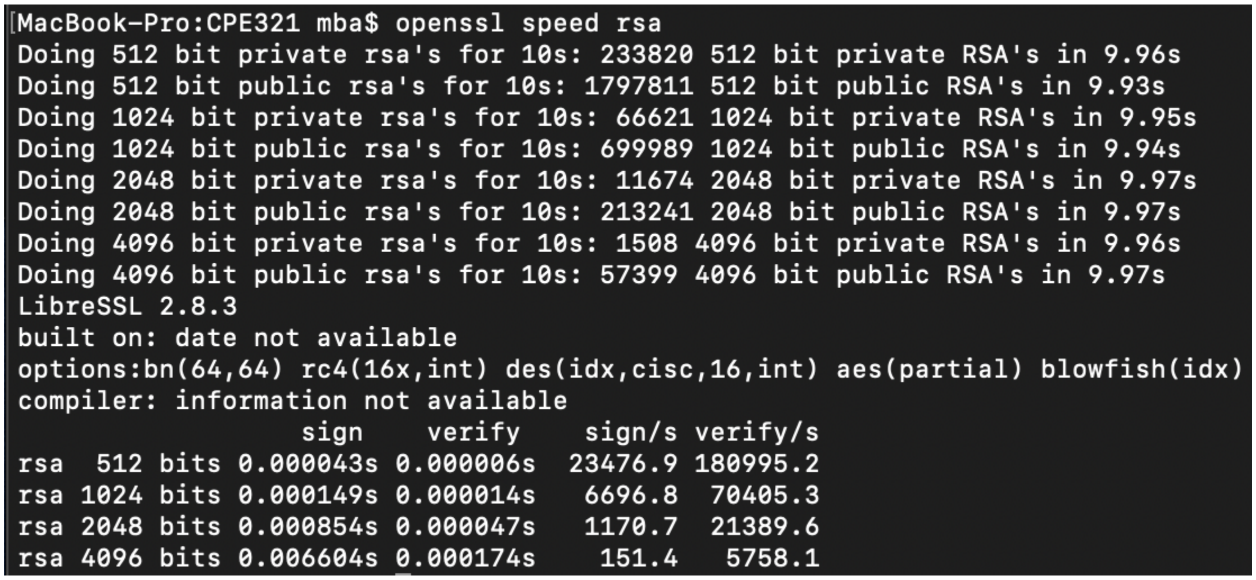
Task 2

```
from blockEncryptions import padByteArray
from blockEncryptions import CBCEncryptBinary
from blockEncryptions import CBCDecryptBinary
import os

numUsers = 456
numSessions = 31337
blockSizeBytes = 16

def urlEncodeString(string:str):
   string = string.replace(';', '%3B')
   string = string.replace('=', '%3D')
   return string

def CDCattack(targetString, encryptedMsg, decryptedMsg):
   newStr = bytearray()
   for i, char in enumerate(targetString.encode()):
       decChar = encryptedMsg[i] ^ decryptedMsg[i+ blockSizeBytes]
       newChar = decChar ^ char
       newStr.append(newChar)
   return newStr + encryptedMsg[len(targetString):]

def submit(userData):
   key = os.urandom(blockSizeBytes)
   iv = os.urandom(blockSizeBytes)
   userData = urlEncodeString(userData)
   userString =
"userid="+str(numUsers)+';userdata='+userData+';session-id='+str(numSessions)
   userString = padByteArray(userString.encode())
   cipherText = CBCEncryptBinary(key, iv, userString)

   return (key, iv, cipherText)

def verify(key, iv, cipherText):
   plaintext = CBCDecryptBinary(key, iv, cipherText)
   return b';admin=true;' in plaintext


result = submit("asdf;alksdjf;alkjfasdlfkjf")
decryptedStr = CBCDecryptBinary(result[0], result[1], result[2])
encryptedAttack = CDCattack(";admin=true;", result[2], decryptedStr)
print(verify(result[0], result[1], encryptedAttack))
```

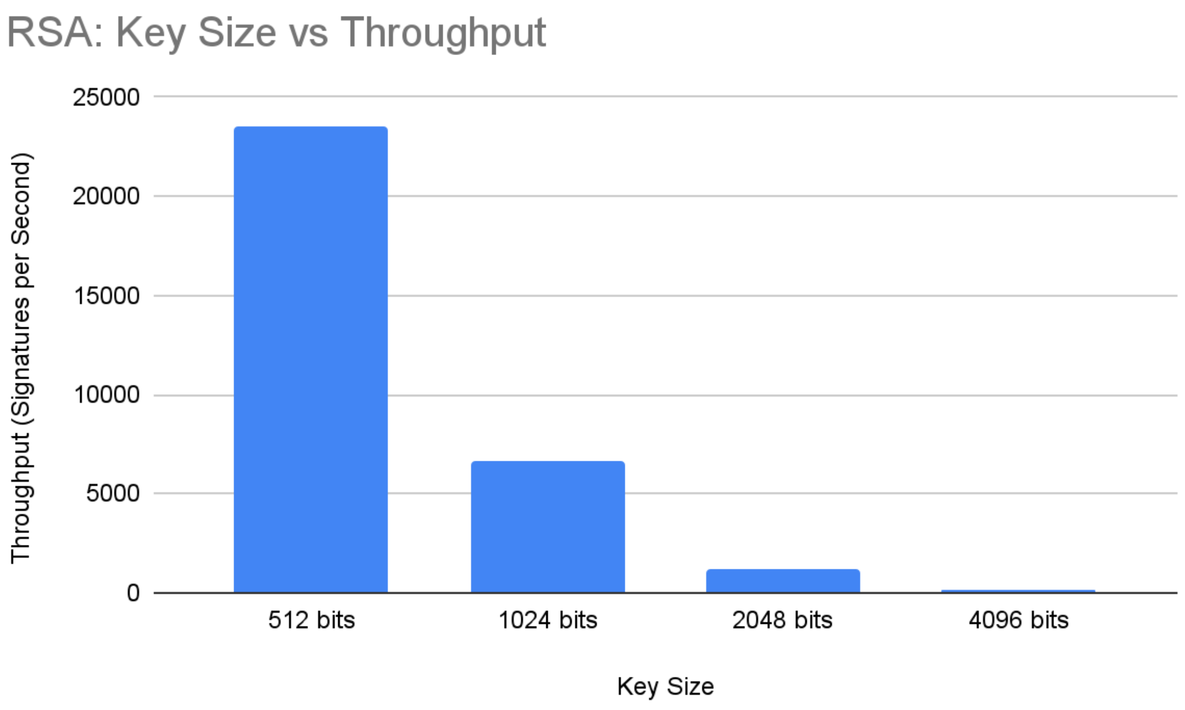
Question 2: This attack is possible because CBC block encryption is susceptible to message extension attacks; by messing around with the tag and a new portion you want to add to the message, you can end up with an equivalent tag using different messages while having the same key. Some ways to prevent this include adding message length to an encrypted tag and comparing actual length with the expected length or using an HMAC algorithm which similarly verifies integrity.

Task 3

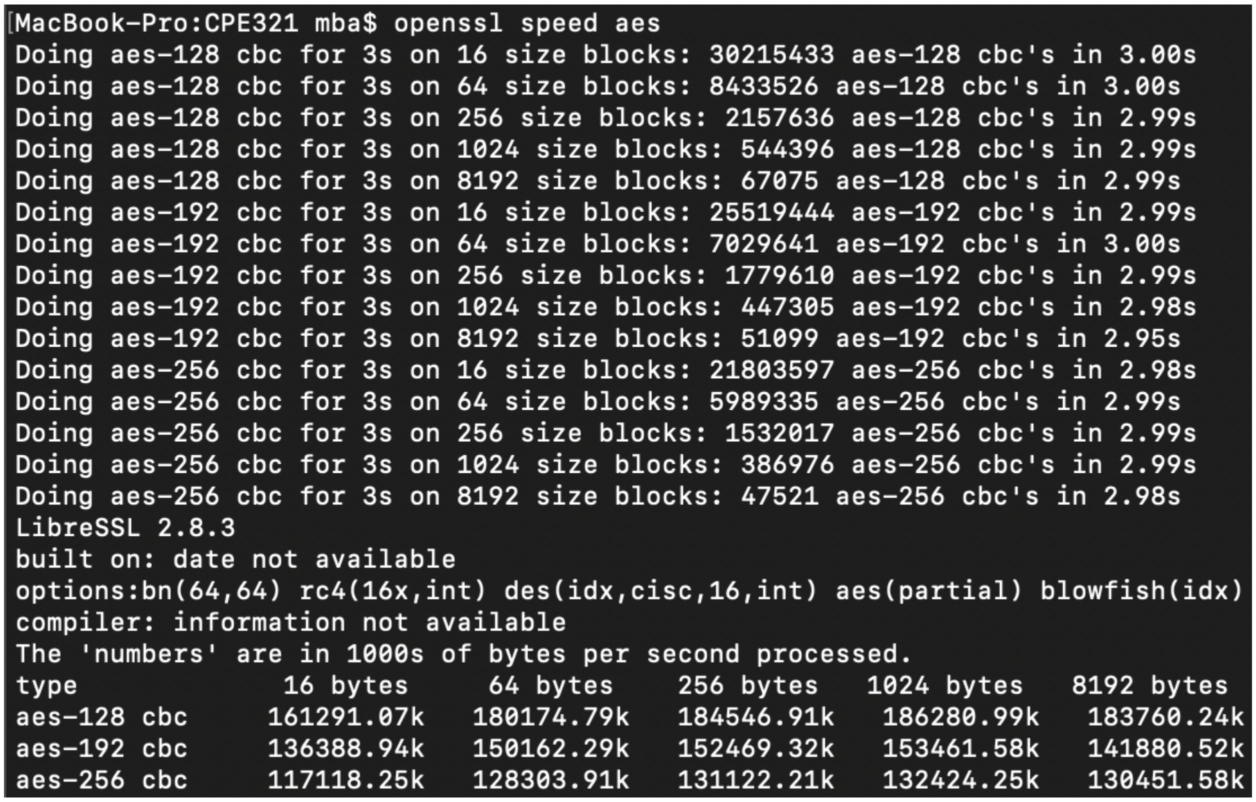
RSA



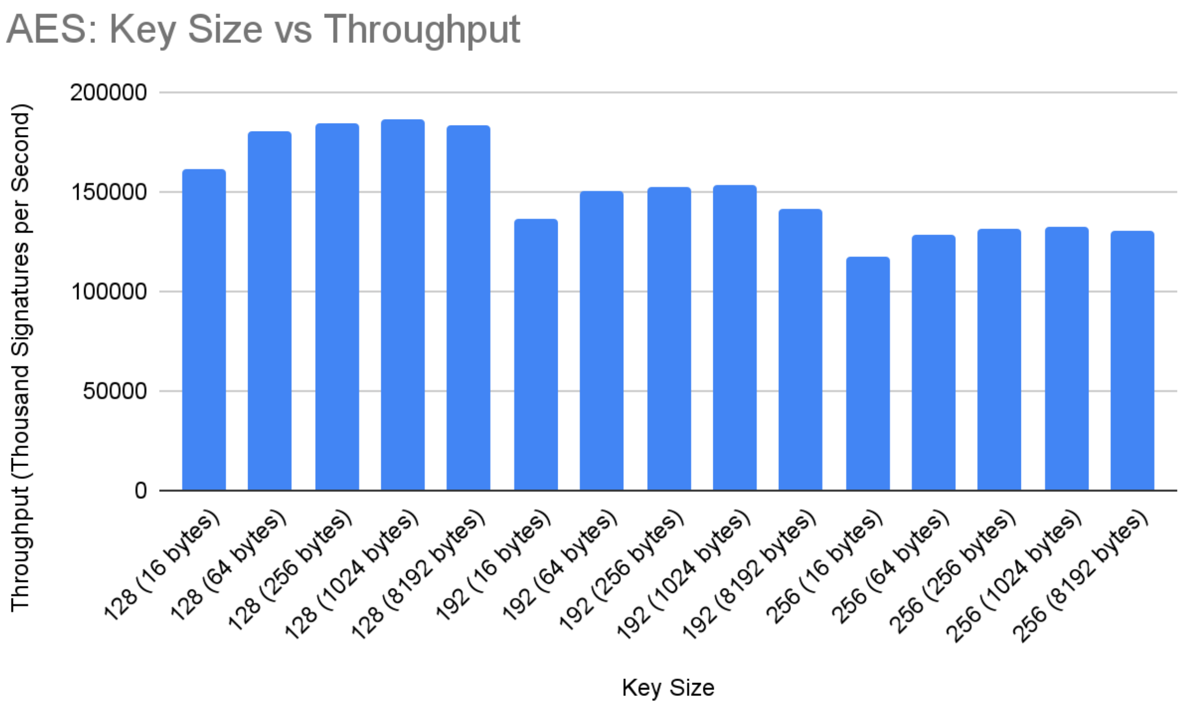
```
MacBook-Pro:CPE321 mba$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 233820 512 bit private RSA's in 9.96s
Doing 512 bit public rsa's for 10s: 1797811 512 bit public RSA's in 9.93s
Doing 1024 bit private rsa's for 10s: 66621 1024 bit private RSA's in 9.95s
Doing 1024 bit public rsa's for 10s: 699989 1024 bit public RSA's in 9.94s
Doing 2048 bit private rsa's for 10s: 11674 2048 bit private RSA's in 9.97s
Doing 2048 bit public rsa's for 10s: 213241 2048 bit public RSA's in 9.97s
Doing 4096 bit private rsa's for 10s: 1508 4096 bit private RSA's in 9.96s
Doing 4096 bit public rsa's for 10s: 57399 4096 bit public RSA's in 9.97s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000043s 0.000006s  23476.9 180995.2
rsa 1024 bits 0.000149s 0.000014s   6696.8  70405.3
rsa 2048 bits 0.000854s 0.000047s   1170.7  21389.6
rsa 4096 bits 0.006604s 0.000174s    151.4   5758.1
```



AES



```
[MacBook-Pro:CPE321 mba$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 30215433 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 64 size blocks: 8433526 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 2157636 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 1024 size blocks: 544396 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 8192 size blocks: 67075 aes-128 cbc's in 2.99s
Doing aes-192 cbc for 3s on 16 size blocks: 25519444 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 64 size blocks: 7029641 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 256 size blocks: 1779610 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 1024 size blocks: 447305 aes-192 cbc's in 2.98s
Doing aes-192 cbc for 3s on 8192 size blocks: 51099 aes-192 cbc's in 2.95s
Doing aes-256 cbc for 3s on 16 size blocks: 21803597 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 64 size blocks: 5989335 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 256 size blocks: 1532017 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 1024 size blocks: 386976 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 8192 size blocks: 47521 aes-256 cbc's in 2.98s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc     161291.07k   180174.79k   184546.91k   186280.99k   183760.24k
aes-192 cbc     136388.94k   150162.29k   152469.32k   153461.58k   141880.52k
aes-256 cbc     117118.25k   128303.91k   131122.21k   132424.25k   130451.58k
```



Question 3: AES is much more consistent in its throughput as key size increases where as RSA very quickly drops off in its throughput as key size increases. For instance, RSA has around 23000 signatures per second at 512 bits but about 1000 signatures at 2048 bit. On the other hand, AES has about 160000k signatures per second at 16 bytes and 130000k signatures at 8192 bytes. Based on this, it seems AES is much more scalable.