Diego Garay, Salvador Gutierrez
Bret Hartman
CPE 321-03
February 25th 2022

Block Ciphers

Questions

1. ***For Task 1, viewing the resulting ciphertexts, what do you observe? Are you able to derive any useful information about either of the encrypted images? What are the causes for what you observe?***

   When reviewing the resulting ciphertexts, it's clearly apparent how the two different encryptions differ in obscuring the original information. The resulting ciphertext for ECB helps to showcase the weakness of encrypting all information with the same key, as information leakage is visibly present. Because our BMP files have blocks of repeated messages(in our case, repeating colors), even though they are "encrypted", a rough image can still be made out. When it comes to CBC, it is trivial to "see" the advantage CBC has over ECB when it comes to information encryption. Even though there is an equal amount of repeated information, because CBC does not use the same key for every block to be encrypted, it looks a lot more random than ECB.

2. ***For Task2, why is this attack possible? What would this scheme need in order to prevent such attacks?***

   Because in CBC encryption every block of ciphertext is used as the key to encrypt the following block of ciphertext, if we can flip a bit in the original ciphertext message, then

eventually the flipped bit will corrupt the original plaintext message. To be specific, because each block of cipher text is XOR'd with the next ciphertext block, if we can determine how many blocks of information our message is, we can corrupt the ciphertext in a way that lets us choose what plaintext information we change. In the context of our task, if our message is only ";admin=true;" then with a little mathematical magic(determining what value to XOR with our selected block indices), then even though the resulting ciphertext is "%3Badmin%3Dtrue%3B" we can attack the bits representing "%3B" and "%3D" to decode the URL encoding and place the actual ";" and "=" characters.
A trivial, but computationally expensive solution to solve this problem, would be to re-encrypt the decrypted plaintext, and then compare the new ciphertext with the original ciphertext. If they are equal, no information was changed, but if they're not, then we know we have been the victims of a byte-flipping attack.

3. ***For Task3, how do the results compare? Make sure to include the plots in your report.***

   Based on the plots, it is apparent that AES signatures can be created at a much faster rate than RSA signatures can be created. When it comes to performance drop off, for the most part all AES signatures increase in complexity after a block size of 256 is chosen. For RSA, public signature generation does not increase greatly in complexity until around a bit length of 7680. On the other hand, for private signatures, there is a sharp increase in signature complexity once a bit length of 2048 is chosen. Overall, it seems like RSA signature generation is more computationally expensive than AES signature generation.

Task 1:

Steps:

1. Take in the name of the BMP file either as a program argument or as a python input.
2. Open said BMP file using Pillow's "Image.open(...)" function
3. Use Pillow's "Image.convert("RGB")" function to change the BMP file to an array of RGB colors. Subsequently, use Python's "List.toBytes()" function to create a byte array of all the RGB information
4. Once we have our byte array, we have to pad this information according to #PKCS 7 standards, as this is how PyCryptodome's AES class expects us to send information to it. Multiple ways this can be achieved.
5. Send our padded information to PyCryptodome's function "AES.new(KEY, MODE, (IV):Optional in the case of ECB)" to encrypt our padded information.
6. Once we encrypted our information, we had to reverse steps 1-3 to produce our new BMP image. To start, we have to take our encrypted information and revert it back to an RGB image. It's some pretty clever mathematics that we found on the internet (StackOverflow forums for how to use Pillow), because this is seemingly a common enough problem that people run into.
7. Use Pillow's "Image.new(MODE, SIZE)" function to create a new image template, where MODE and SIZE are the same as the original BMP file.
8. Use Pillow's "Image.putdata(RGB_ENCRYPTEDINFO)" to write the encrypted image data to the new image we just started making, and viola! We're done.

9. Optional: Use Pillow's "Image.save(NAME, FILETYPE)" function to save the result of our work into a new image. Really useful for testing purposes.

Description: Okay, so it looks like this lab will be a lot different than the other labs we've done. We're actually writing some serious code and this is kinda fun. Given that we can't use built-in methods, we should start by reading the **PyCryptodome**(suggested Cryptography library) documentation to see if there are any hints on where to start…. Okay, from what we're seeing, our block sizes are 16-bytes and the keys can be 128, 192, 256 bits long. It also looks like we're going to have to implement PCKS#7 padding to ensure that my blocks are the correct size.

NOTE: There was a lot of headache trying to figure out how to work with the headers of BMP files, but eventually Sal figured out that we can use the **Pillow**(PIL Fork) library for Python to make our lives easier.

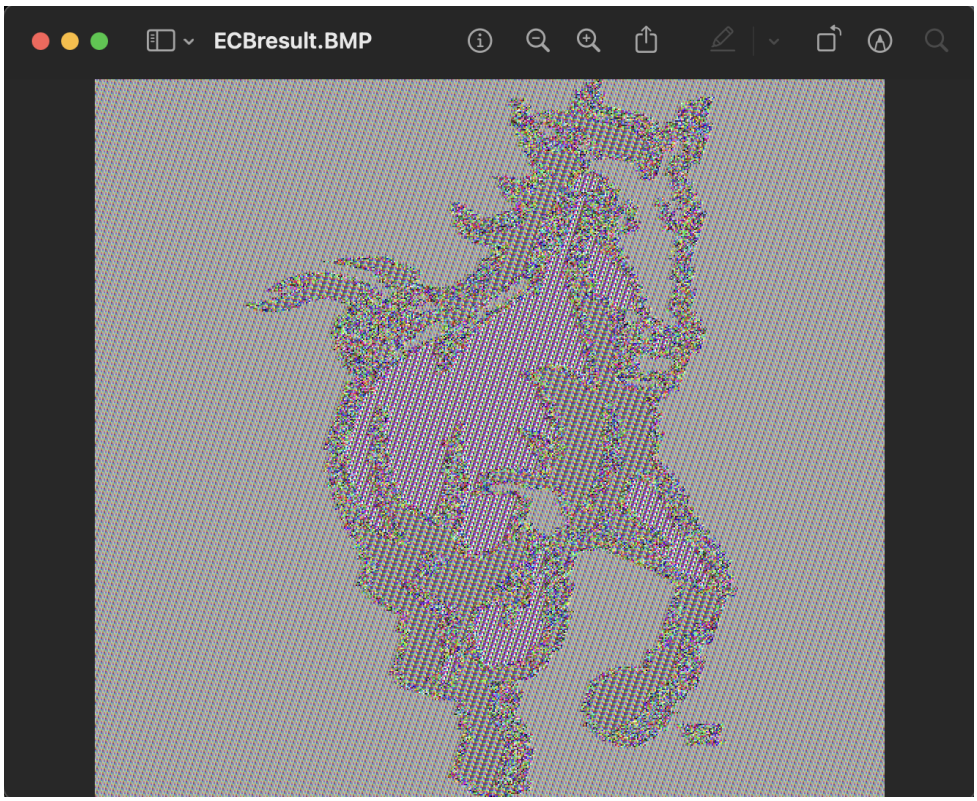
Okay, utilizing pillow, we have managed to convert the BMP file into a byte array of all of the RGB values of the. Now that we have an array of bytes, we can finally send this information to our CBC/ECB functions to perform the encryption. Once our encrypted information is returned to us, we gotta perform some mathematical magic to convert it back into RGB. Once this is done, we can just use **Pillow** to do the rest of the image conversion stuff for us. The hardest parts of this lab are definitely gonna be the #PKCS 7 padding and the ByteArray to RGB/Pixel conversion….

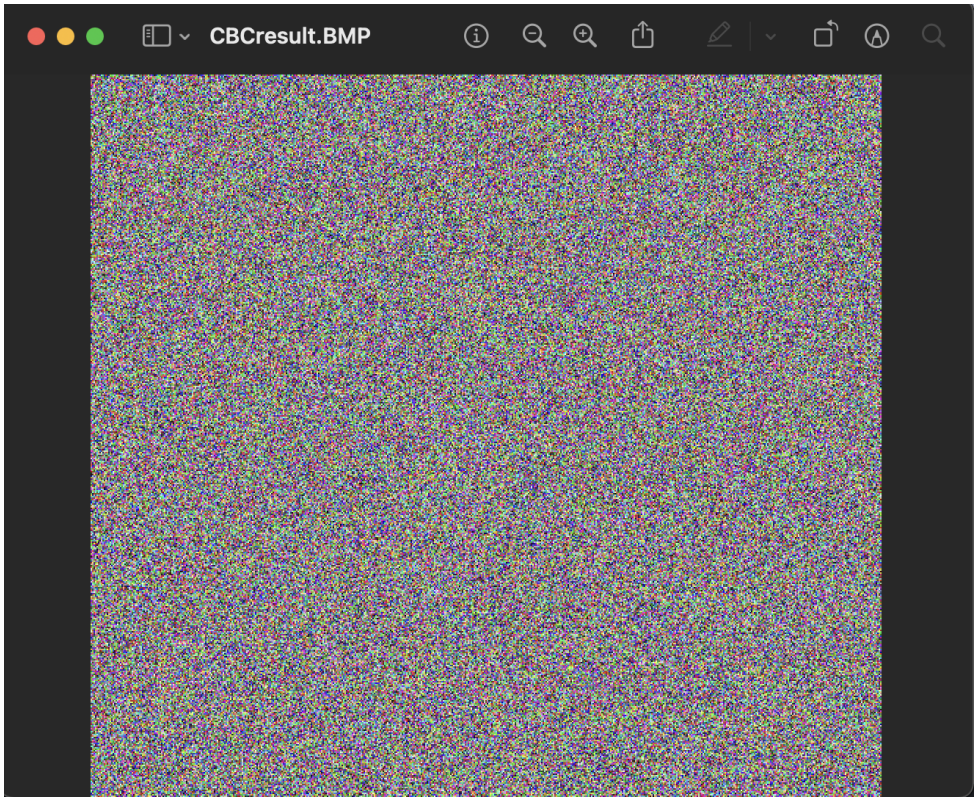
CODE :

```
from PIL import Image
import sys
from cbc import CBC
from ecb import ECB
from confid import condifidentialityLimits

def main():
   if len(sys.argv) >= 2:
       infile = sys.argv[1]
       task1(infile)

def task1(infile):
   try:
       im = Image.open(infile, mode="r")
   except:
       print(f"That is not a valid file.")
       return
   info = im.convert("RGB").tobytes()
   ecbEncryptedInfo = ECB(info)
   cbcEncryptedInfo = CBC(info)

   createNewBMP(im,ecbEncryptedInfo, len(info),"ECB")
   createNewBMP(im,cbcEncryptedInfo, len(info),"CBC")

def createNewBMP(img,encryptedInfo,ogLen,encType):
   newImage = to_RGB(encryptedInfo[:ogLen])
   im2 = Image.new(img.mode, img.size) #create an image that is the same specs as the
original image
   im2.putdata(newImage)
   im2.save(encType + "result.BMP", "BMP")

#maps the original image to RGB information
# a lot of magic happens here, just bask in the glory that is python
def to_RGB(information):
   r, g, b = tuple(
       map(
         lambda d:
           [information[i] for i in range(0,len(information)) if i % 3 == d], [0,1,2])
       )
   pixels = tuple(zip(r,g,b))
   return pixels
```

```python
#EBC.py/CBC.py
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

def ECB(info):
    cipher_key = get_random_bytes(16)
    ogLen = len(info) # we must save the original length of information
    blockLen = 16
    paddedInfo = pad(info,blockLen)
    encryptedInfo = aesEcbEncryption(cipher_key, paddedInfo)
    return encryptedInfo

def CBC(info,cipher_key=get_random_bytes(16), iv=get_random_bytes(16)):
    ogLen = len(info) # we must save the original length of information
    blockLen = 16
    paddedInfo = pad(info,blockLen)
    mode = AES.MODE_CBC
    encryptedInfo = aesCbcEncryption(cipher_key, paddedInfo, iv, mode)
    return encryptedInfo

# PKCS #7 species that the value of each added byte is the number of bytes that are
added.
def pad(information,block_length):
    info_len = len(information)
    # b"\x00 is da number 1"
    # 1 * k - (lth mod k) <= from RFC 5652 6.3
    pad = b"\x00" * (block_length - (info_len%block_length))
    return (information + pad)

#takes a cipher key, and information to encrypt with ECB using AES
#NOTE: the information being passed in must be padded
def aesEcbEncryption(key, information, mode=AES.MODE_ECB):
    aes = AES.new(key,mode)
    new_info = aes.encrypt(information)
    return new_info

#takes a cipher key, and information to encrypt with ECB using AES
#NOTE: the information being passed in must be padded
def aesCbcEncryption(key, information, iv, mode=AES.MODE_CBC):
    aes = AES.new(key,mode,iv)
    new_info = aes.encrypt(information)
    return new_info
```

OUTPUT :





Task 2:

Steps:

1. Import the CBC code that was created for Task1, with a given initial cipherkey/initialization vector
2. Prompt the user for the username/information/query they would like to send the "web server"
3. Create the Submit() function:
    a. URL encode the user given string, this can easily be achieved through the built in Python module "urllib.parse"
    b. Prepend the string with "userid=456;userdata=" and append the string with ";session-id=31337".
    c. Use Python's "Bytes(STRING, "utf-8")" function to create a ByteArray from the new string created
    d. Send this ByteArray to our CBC function with our generated Key and IV
4. Create the Verify() function:
    a. Use AES to create a new AES object "AES.new(KEY, MODE, IV)"
    b. Use the newly created AES object to decrypt out encoded query into plaintext "OBJECT.decrypt(ENCODED QUERY).decode("utf-8")"
    c. Search this new plaintext for the substring ";admin=true;" and return a boolean value as the result
5. Determine what string we will pass to try and corrupt the ciphertext(";admin=true;" would be the easiest, but isn't the only option). Once we know what string, we have to do fancy math to determine what positions "%3D" and "%3B" are in, and then XOR them with the appropriate bit so that way

once the encryption is done, they go from URL encoded to actual characters.

6. Send this string through Submit(), and once we get our new encoded string, send this to Verify() to see if we get the same results.

Description: This portion of the Assignment was a little more difficult than expected. While we could figure out multiple ways to flip the bytes, we ran into a lot of trouble trying to return this altered information back to CBC function. There was a constant issue where **PyCryptodome**'s functions could tell that we messed with the ciphertext bits and wouldn't accept these valid ByteArrays as valid inputs. So after trying to refactor our implementation, and then even trying to modify/borrow other implementations online, we did some 200 level IQ plays….

CODE:

```
import urllib.parse #god bless the Python gods for having a built-in function for just
about anything
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from cbc import CBC

def condifidentialityLimits():
   block_length = AES.block_size
   intial_iv = get_random_bytes(block_length)
   intial_cipher_key = get_random_bytes(block_length)
   inputQuery = input("Username: ")
   encodedQuery = submit(inputQuery, intial_cipher_key, intial_iv)
   print(f"Non-Byte Flipped result: {verify(encodedQuery, intial_cipher_key,
intial_iv)}")
   #alright, now time to preform a byte flipping attack
   #we can't flip too many bits or it won't work
   res =  byteFlipAttack(4, 14, encodedQuery)
   attack = verify(res, intial_cipher_key, intial_iv)
   print(f"Byte Flipped result: {attack}")
```

```
# str -> URL encoded and AES-128-CBC encrypted string
def submit(query,cipher_key,iv):
   prependStr = "userid=456;userdata="
   appendStr = ";session-id=31337"
   # %3B is the URL encoding of ";" --- %3D is the URL encoding of "="
   encodedQuery = urllib.parse.quote(query)
   urlEncodedQuery = prependStr + encodedQuery + appendStr
   #our CBC function expects our answer to be in bytes
   bytesQuery = bytes(urlEncodedQuery, 'UTF-8')
   cbcEncodedQuery = CBC(bytesQuery, cipher_key, iv)
   return cbcEncodedQuery

def verify(encodedQuery, c_key, ivec):
   cipher = AES.new(c_key, AES.MODE_CBC, ivec)
   try: #bask in the glory that is python.....
       plaintext = cipher.decrypt(encodedQuery).decode('utf-8')
       #anytime I try to flip a bit, I get the following error so here's to this....
       #UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb4 in position 1:
invalid start byte
   except:
       return True
   #HINT: since all ";" & "=" are URL encoded, they don't show up, theoretically
impossible to make this function return true
   res = ";admin=true;" in  plaintext
   return res

# will flip the provided block at the Index provided within encodedQuery with provided
bit
def byteFlipAttack(blockIdx, bit, encodedQuery):
   bytesArr = []
   for i in range(len(encodedQuery)):
       if i != blockIdx:
           num = encodedQuery[i] # when we grab the byte, it becomes an int
       else:
           b = encodedQuery[i]
           flippedBit = (b ^ bit)
           num = flippedBit
       temp = num.to_bytes(1, byteorder="big")
       bytesArr.append(temp)

   bStr = b"".join(bytesArr)
   return bSt
```

Output:

```
(base) Eggbook-Pro:block_ciphers diegogaray$ ./run.sh
~~~Running python encryption files~~~
Usage for encryption: ./run.sh [filename].bmp
Username: dgaray;admin=true;
Non-Byte Flipped result: False
Byte Flipped result: True
~~~Finished running!~~~
(base) Eggbook-Pro:block_ciphers diegogaray$
```

Task 3:

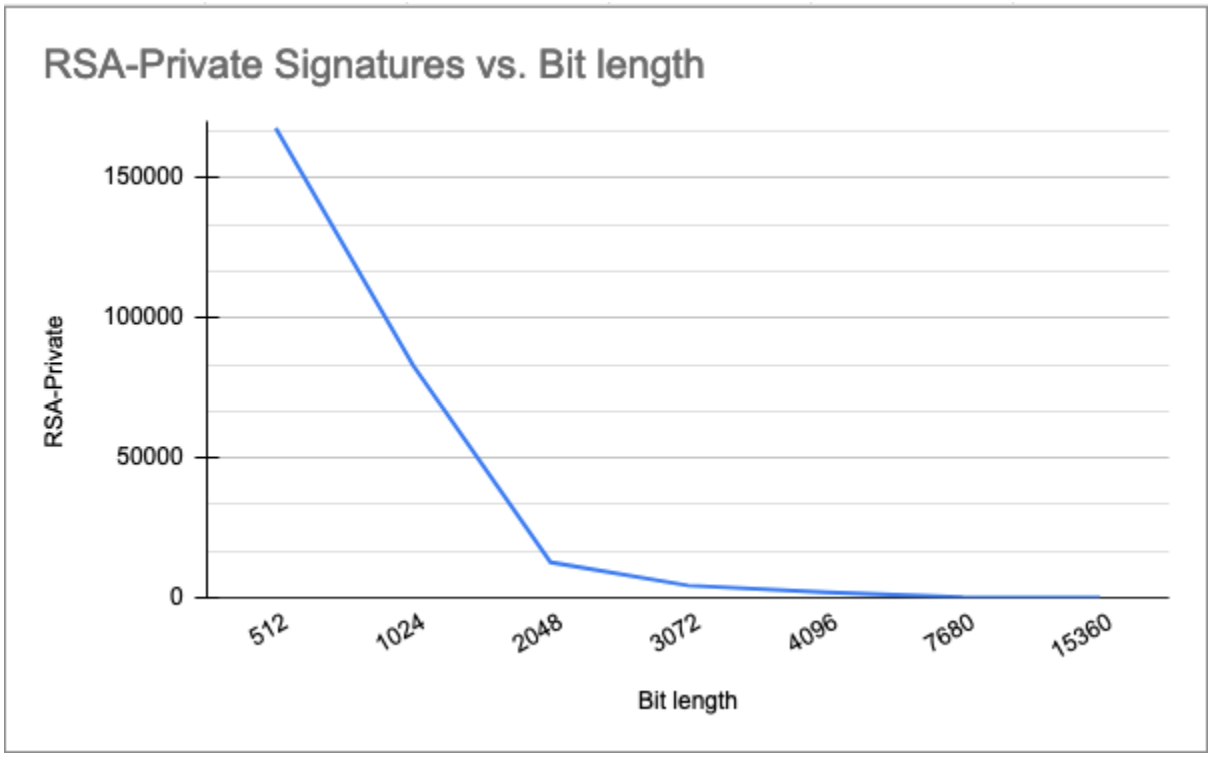
Steps:

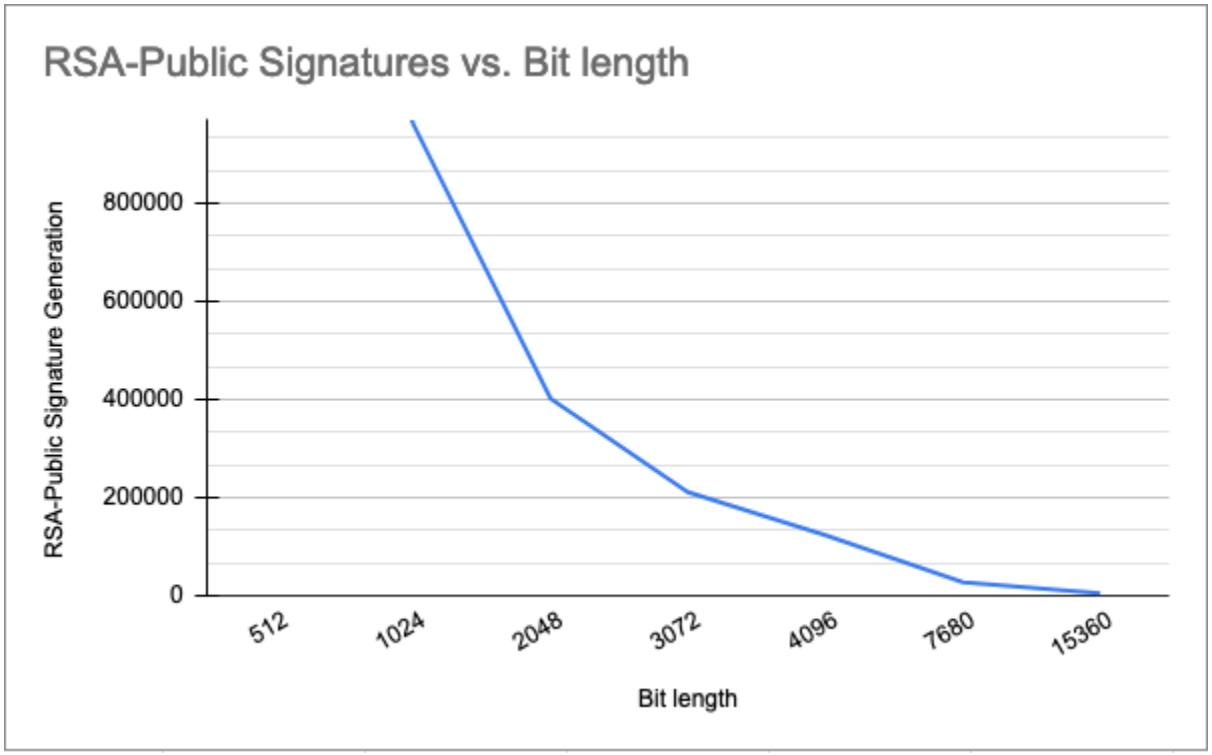
1. Use homebrew(sorry Window users) to install "openssl"
2. Once installed, within a terminal, type in openssl speed rsa to perform the RSA tests. Store the resulting numbers in an excel sheet. Type in "openssl speed aes" to perform the AES tests. Store these resulting numbers in the excel sheet being used to store the RSA numbers.
3. Using Excel/Google Sheets, create charts to show the differences between these methods on encryption.

Description: Okay, so compared to Tasks 1 & 2, this was really easy. We just have to run these tests and plot the numbers to compare them. If I didn't have experience doing this already though, this would be a bit tedious. I did this by hand, but I am sure there are some fancy ways of using **GREP** and the terminal to plot these out automatically, but for now manually is a-okay.

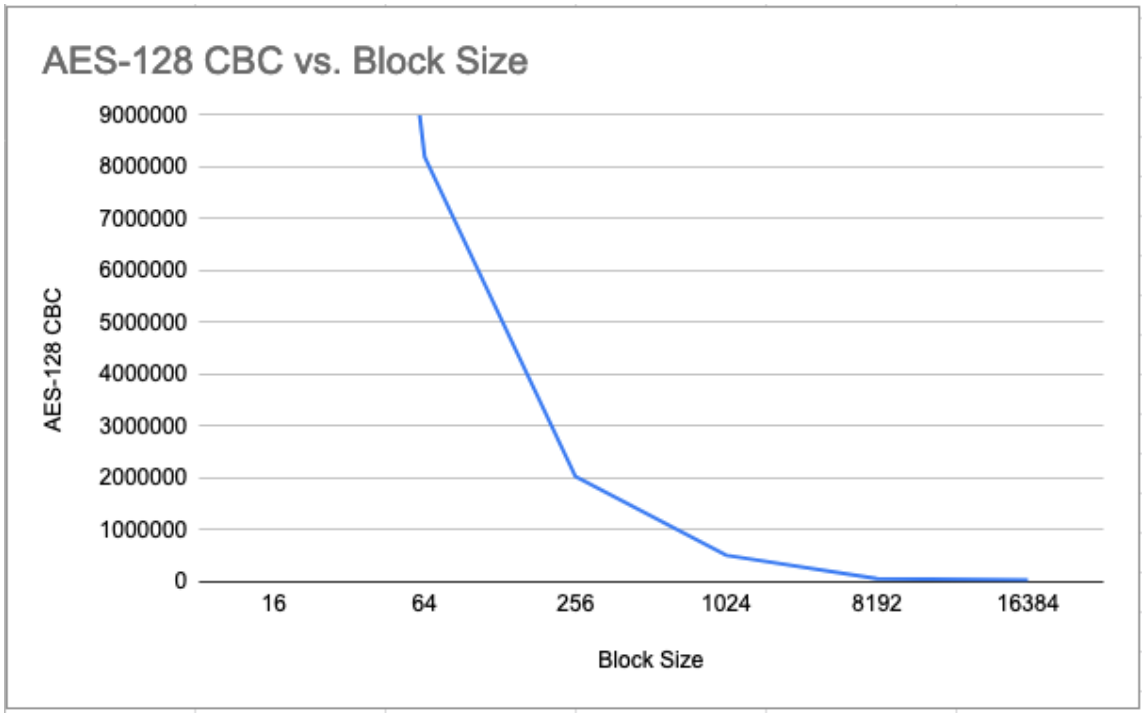
PLOTS:

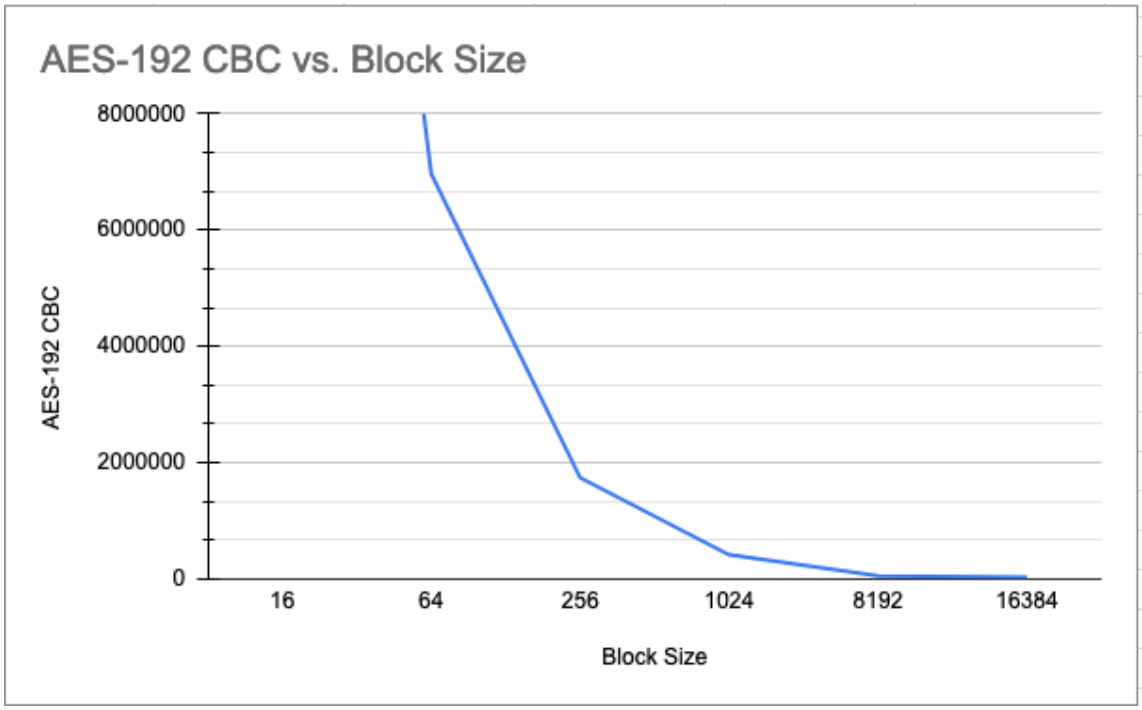
RSA RESULTS:

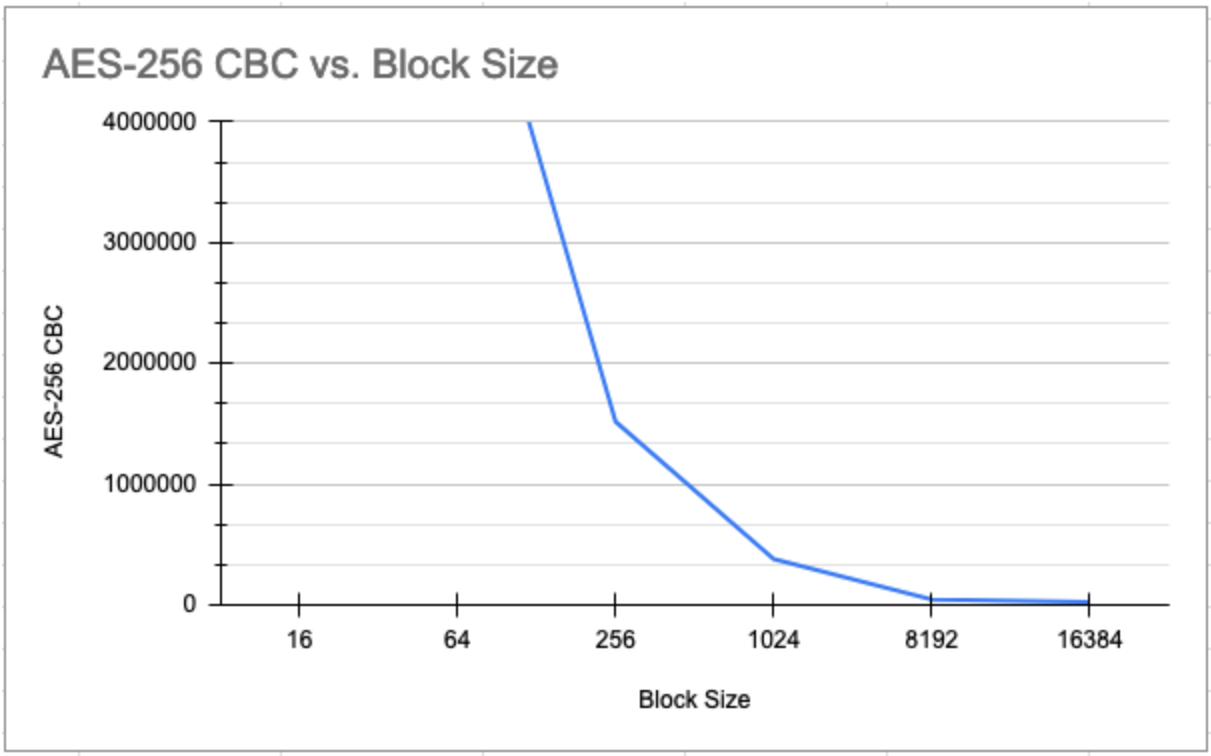




AES RESULTS:


AES-128 CBC vs. Block Size
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
AES-128 CBC
16
64
256
1024
8192
16384
Block Size


AES-192 CBC vs. Block Size
8000000
6000000
4000000
2000000
0
AES-192 CBC
16
64
256
1024
8192
16384
Block Size



CODE/TERMINAL OUTPUT:

RSA:

```
diegogaray (e) base ~ Documents GitHub block_ciphers openssl speed rsa
Doing 512 bits private rsa's for 10s: 167787 512 bits private RSA's in 9.62s
Doing 512 bits public rsa's for 10s: 1832353 512 bits public RSA's in 9.65s
Doing 1024 bits private rsa's for 10s: 82980 1024 bits private RSA's in 9.71s
Doing 1024 bits public rsa's for 10s: 961848 1024 bits public RSA's in 9.23s
Doing 2048 bits private rsa's for 10s: 12736 2048 bits private RSA's in 9.56s
Doing 2048 bits public rsa's for 10s: 402365 2048 bits public RSA's in 9.75s
Doing 3072 bits private rsa's for 10s: 4442 3072 bits private RSA's in 9.83s
Doing 3072 bits public rsa's for 10s: 211392 3072 bits public RSA's in 9.80s
Doing 4096 bits private rsa's for 10s: 1944 4096 bits private RSA's in 9.82s
Doing 4096 bits public rsa's for 10s: 124107 4096 bits public RSA's in 9.62s
Doing 7680 bits private rsa's for 10s: 184 7680 bits private RSA's in 9.25s
Doing 7680 bits public rsa's for 10s: 28040 7680 bits public RSA's in 8.08s
Doing 15360 bits private rsa's for 10s: 24 15360 bits private RSA's in 7.34s
Doing 15360 bits public rsa's for 10s: 5635 15360 bits public RSA's in 7.35s
OpenSSL 1.1.1i  8 Dec 2020
built on: Wed Dec  9 17:53:09 2020 UTC
```

```
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000057s 0.000005s  17441.5 189881.1
rsa 1024 bits 0.000117s 0.000010s   8545.8 104208.9
rsa 2048 bits 0.000751s 0.000024s   1332.2  41268.2
rsa 3072 bits 0.002213s 0.000046s    451.9  21570.6
rsa 4096 bits 0.005051s 0.000078s    198.0  12900.9
rsa 7680 bits 0.050272s 0.000288s     19.9   3470.3
rsa 15360 bits 0.305833s 0.001304s      3.3    766.7
```

AES:

```
diegogaray (e) base ~ Documents GitHub block_ciphers openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 32476561 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 64 size blocks: 8193854 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 256 size blocks: 2032135 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 1024 size blocks: 512235 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 8192 size blocks: 63972 aes-128 cbc's in 2.93s
Doing aes-128 cbc for 3s on 16384 size blocks: 32239 aes-128 cbc's in 2.94s
Doing aes-192 cbc for 3s on 16 size blocks: 27510602 aes-192 cbc's in 2.92s
Doing aes-192 cbc for 3s on 64 size blocks: 6966907 aes-192 cbc's in 2.93s
Doing aes-192 cbc for 3s on 256 size blocks: 1739199 aes-192 cbc's in 2.94s
Doing aes-192 cbc for 3s on 1024 size blocks: 416862 aes-192 cbc's in 2.88s
Doing aes-192 cbc for 3s on 8192 size blocks: 54178 aes-192 cbc's in 2.92s
Doing aes-192 cbc for 3s on 16384 size blocks: 27075 aes-192 cbc's in 2.92s
Doing aes-256 cbc for 3s on 16 size blocks: 23924801 aes-256 cbc's in 2.93s
Doing aes-256 cbc for 3s on 64 size blocks: 6040483 aes-256 cbc's in 2.92s
Doing aes-256 cbc for 3s on 256 size blocks: 1517083 aes-256 cbc's in 2.94s
Doing aes-256 cbc for 3s on 1024 size blocks: 379230 aes-256 cbc's in 2.93s
Doing aes-256 cbc for 3s on 8192 size blocks: 43813 aes-256 cbc's in 2.72s
Doing aes-256 cbc for 3s on 16384 size blocks: 23504 aes-256 cbc's in 2.93s
OpenSSL 1.1.1i  8 Dec 2020
```

```
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
aes-128 cbc     177346.41k   178978.38k   177551.73k   179020.01k   178859.60k   179661.15k
aes-192 cbc     150743.02k   152178.17k   151440.46k   148217.60k   151995.27k   151916.71k
aes-256 cbc     130647.38k   132394.15k   132099.74k   132536.35k   131954.45k   131429.88k
```