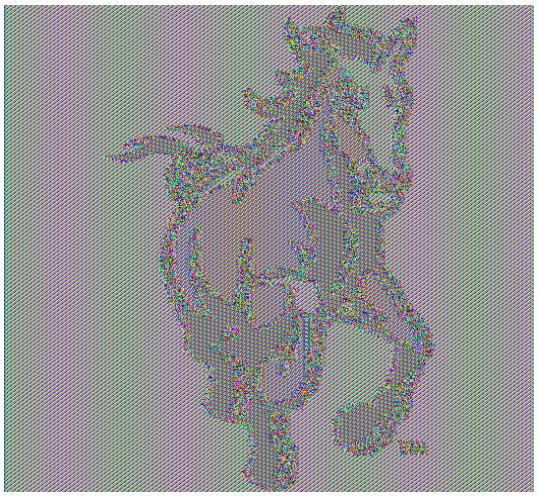
CSC321 Block Cipher Report

Group 4: Jaron Li, Ben Hinchliff, Adrian Nguyen, Joe Waltz

Background:
In this assignment, we looked at both Electronic Codebook Mode (ECB) and Cipher Block Chaining Mode (CBC), implemented our versions of them, and saw how effective they were in terms of their encryption. Along with that, we looked into the limits of block cipher as we emulated a web server and looked to have the verify() function return true by modifying the ciphertext produced by the submit() function to have the pattern ";admin=true;" injected. Lastly, we looked at the relationships between a RSA key size and the throughput for each function along with a block size and the throughput of AES key sizes.

Task 1:
In this task, we wrote both ECB and CBC modes of encryption from scratch while utilizing a Python cryptography package called PyCryptodome. To do this, we first generated both a random key and a random IV of 16 bytes as the ECB will require the key and CBC needing both the key and IV. For the ECB function, we started by creating an AES cipher object by utilizing the random key made along with "AES.MODE_ECB", a mode of operation. We then padded the plaintext to make sure its length was an exact multiple of the block size to ensure the plaintext can then be properly divided into blocks. Once that was completed, we went through the plaintext by intervals of 16 bytes to represent a block, and encrypted each block by using ".encrypt()" from PyCryptodome. We put these together to create a cipher text. This would conclude how we wrote ECB encryption from scratch. For CBC encryption, it was pretty similar to EBC with the difference being using the random IV we made along with utilizing an XOR function we made. The XOR function we created took in two blocks and iterated through the blocks and performed a byte-wise XOR operation on the bytes in the same locations of the two blocks. This is essential as this function then gives us the block that we would actually encrypt. To add on to the XOR function we created, it took in the current block we were looking at and performed XOR operation on the previous block we looked at, which is why we needed a random IV in the start as it would represent the previous block when the current block we were looking at was the first block. We then applied both ECB and CBC modes to a bmp file of a Cal Poly mustang logo, where we saw the CBC encryption do a better job of hiding the image compared to the ECB encryption.



(ECB encryption performed above)

(CBC encryption performed above)

Task 2:

In this task, we wrote submit() and verify() functions to emulate a web server to see what the limits of block cipher were. In the submit() function, we took in a random string and prepended “userid=456; userdata=” and appended “;session-id=31337”. We then encoded this new string with a python library called urllib, and we then encrypted this text with the CBC function we wrote in the last task. We then used verify to call a CBC decryption function we wrote to decrypt what we did in submit() where we then checked to see if “;admin=true;” was in the plain text. With submit() being written correctly, then verify() should return false as in the submit with the encoder with special characters, “;admin=true;” shouldn’t be able to be in the plaintext after the decryption. However, we are able to actually modify the ciphertext and get verify() to return true by doing some bit flipping and understanding how CBC works. We were able to understand that modifying a ciphertext block will affect how the next plaintext block would be decrypted. For this, we put ourselves in the minds of what a hacker would do and since we would always be in control of the string we would input, we were working with specifically “AAAAAAAAAAAAXadminYtrueX”. We chose this string as it would allow for this string to live within one block rather than two different blocks such as the string “XadminYtrueX”, which would make bit flipping a lot easier. Once we passed this string through submit() and had it encrypted into ciphertext, we then passed this into a flip_bit() function we wrote. We knew the byte locations of where this string would live and we knew which byte locations to target, which happened to be at locations 33, 39, and 44 as for locations

33 and 44, we wanted to change the “X” to a “;” and for location 39, we wanted to change the “Y” to a “=”. Once we knew these locations and knowing how CBC worked, we were able to identify which locations to target in the previous block through a little bit of math and once we found these locations, we were able to perform XOR operations and once that was complete, we were able to pass in the new ciphertext into verify() and once that was decrypted, we were able to successfully inject “;admin=true;” and have verify() return true. Throughout this process, we were able to see the limits of block cipher within the web server emulation.

```
def flip_bit(ciphertext):
    modify_cipher = bytearray(ciphertext)
    byte_pos = [33, 39, 44]
    for x in byte_pos:
        block = x // 16
        index = x % 16
        prev = (block - 1) * 16 + index
        if index == 1:
            modify_cipher[prev] ^= (ord("X") ^ ord (";"))
        elif index == 7:
            modify_cipher[prev] ^= (ord("Y") ^ ord ("="))
        elif index == 12:
            modify_cipher[prev] ^= (ord("X") ^ ord (";"))
    return bytes(modify_cipher)
```

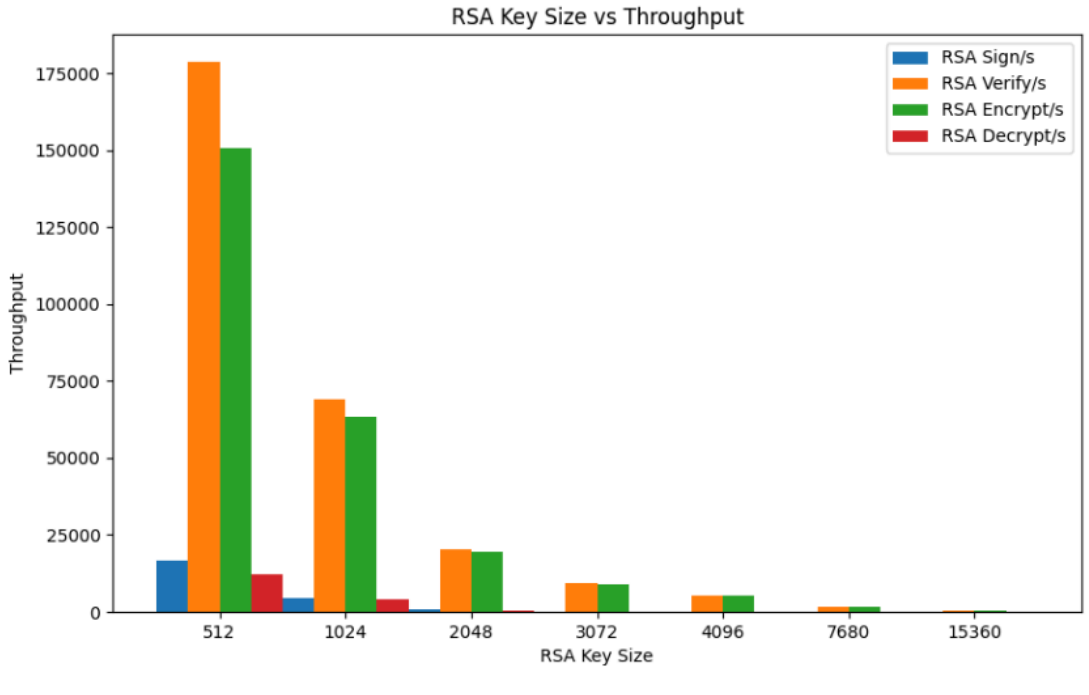
(Code above is our flip_bit() function)

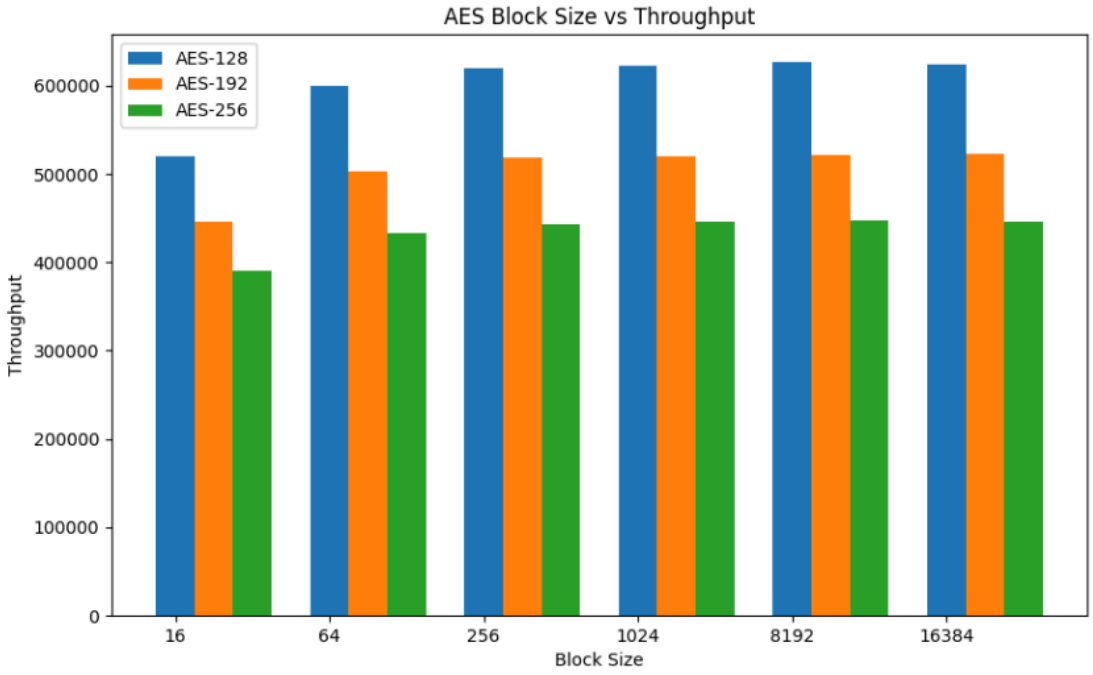
```
Encrypting: b'userid=456; userdata=AAAAAAAAAAAAXadminYtrueX;session-id=31337'
Decrypted: b'userid=456; userdata=AAAAAAAAAAAAXadminYtrueX;session-id=31337'
verify function returned false
Decrypted: b'userid=456; user\xd8\xf6$\xad\x1a\xee\x05\xde\xd5F\xb40)\x91\x96\x83A;admin=true;;session-id=31337'
verify with flip bit function returned true
```

(Results of verify() before and after calling flip_bit())

Task 3:

In this task, we were able to make “Openssl speed RSA” and “Openssl speed AES” calls to see what the difference looks like when it comes to performance between public and symmetric key algorithms. By making these calls, we were able to get values to compare block size to the throughput of AES key sizes along with RSA key sizes to throughput for different RSA functions. We were also able to graph these out to get a better visual representation of these comparisons.


RSA Key Size vs Throughput
RSA Sign/s
RSA Verify/s
RSA Encrypt/s
RSA Decrypt/s
Throughput
175000
150000
125000
100000
75000
50000
25000
0
512
1024
2048
3072
4096
7680
15360
RSA Key Size


AES Block Size vs Throughput
AES-128
AES-192
AES-256
Throughput
600000
500000
400000
300000
200000
100000
0
16
64
256
1024
8192
16384
Block Size

Questions:

1. In task 1, we observed that CBC mode encrypts a lot better than ECB mode as in the images, the mustang logo is completely blurred out in CBC while it's still very visible in ECB. The reason why this occurs is because with ECB, since each block is encrypted separately, it won't affect how other blocks look like. With this, if multiple blocks of plaintext look the same, then it means they will be encrypted the same. Therefore, the ciphertext blocks will then look the same. This means that patterns will remain in the ciphertext, therefore we can still see the mustang logo. With CBC however, since we are utilizing XOR and XOR'ing the current block with the previous block, the way each block is encrypted will be different, even if the blocks may be identical. By doing this, it doesn't allow patterns that were in the plaintext to show up in the ciphertext, which is why we can't see the mustang logo. Therefore, we can conclude CBC mode encrypts better than ECB mode.
2. We could do this attack because we knew what input string we wanted to pass in and we knew which characters we wanted to swap out. Since we also know what was going to be prepended and appended to the input string, we can construct this string to live within just one block to make bit flipping easier and we can inject most of the string we want in there while having placeholder characters replace the special characters at first, where we would then perform XOR on the placeholder characters with the special characters we want, allowing us to perform the character swap and being able to fully inject ";admin=true;". To prevent such attacks, we can possibly utilize a Message Authentication Code (MAC) to ensure the integrity and authenticity and so others can't impersonate the senders.
3. We can identify many relationships within these graphs shown earlier in the report under task 3. In the RSA Key Size vs Throughput graph, we can identify as the key size gets bigger, the amount of time it takes to do each operation starts to become shorter. We can also identify that RSA operations such as encrypt/s and verify/s need a longer time to perform compared to RSA operations such as decrypt/s and sign/s. For the AES Block Size vs Throughput graph, we can identify that aes-128-cbc has the highest throughput compared to aes-192-cbc and aes-256-cbc. We can also see an increase in throughput from block size 16 to 64, and then all the block sizes after 64 start to remain the same.