Symmetric Key Cryptography Implementation

Overview

- In this lab assignment, I utilized the PyCryptodome cryptography package in python to build the following two symmetric key cryptography security modes.
    - Electronic Codebook Mode (ECB)
    - Cipher Block Chaining Mode (CBC)
- I also explored vulnerabilities of block ciphers, notably in the CBC mode through byte flipping.
- Finally I studied the performance between Public vs Symmetric key algorithms.

Task 1: Modes of Operation:

- In this task, I implemented my own versions of ECB and CBC modes of encryption that encrypts the body of an image into a newly encrypted .bmp file.
- **ECB:** Each 128 bit block of plain text is encrypted independently

    i. Initialize a cipher object:

```
cipher_machine = AES.new(secret_key, AES.MODE_ECB)
```

    - This object will be used to encrypt blocks of data 128 bits at a time.

    ii. **Apply Padding** to the data we are encrypting: Since, AES only allows us to encrypt 128 bits at a time, we must apply padding to ensure that the length of our data is divisible by 128 bits / 16 bytes.

```
def apply_pkcs7_padding(msg):
  # Padding is necessary for msgs not a multiple of block size
  block_length = 16  # AES block size in bytes
  padding_amount = block_length - len(msg) % block_length
  if padding_amount == 0:
      padding_amount = block_length
  padding = bytes([padding_amount] * padding_amount)
  return msg + padding
```

Usage:

```
ready_text = apply_pkcs7_padding(plain_text)
```

iii. Loop through 128 bits at a time and encrypt the block.

```
encrypted_msg = b''
for i in range(0, len(ready_text), 16):
    block = ready_text[i:i+16]
    encrypted_msg += cipher_machine.encrypt(block)
return encrypted_msg
```

- **CBC:** Each 128 bit block of plain text will now be encrypted based on the result of its previous encrypted block XOR with its current plain text block. This introduces dependencies between blocks.

  i. **Initialize a cipher object**

  ii. **Apply Padding**

  iii. **Initialization Vector (IV):** Set the initial previous block to begin at the IV generated with AES.

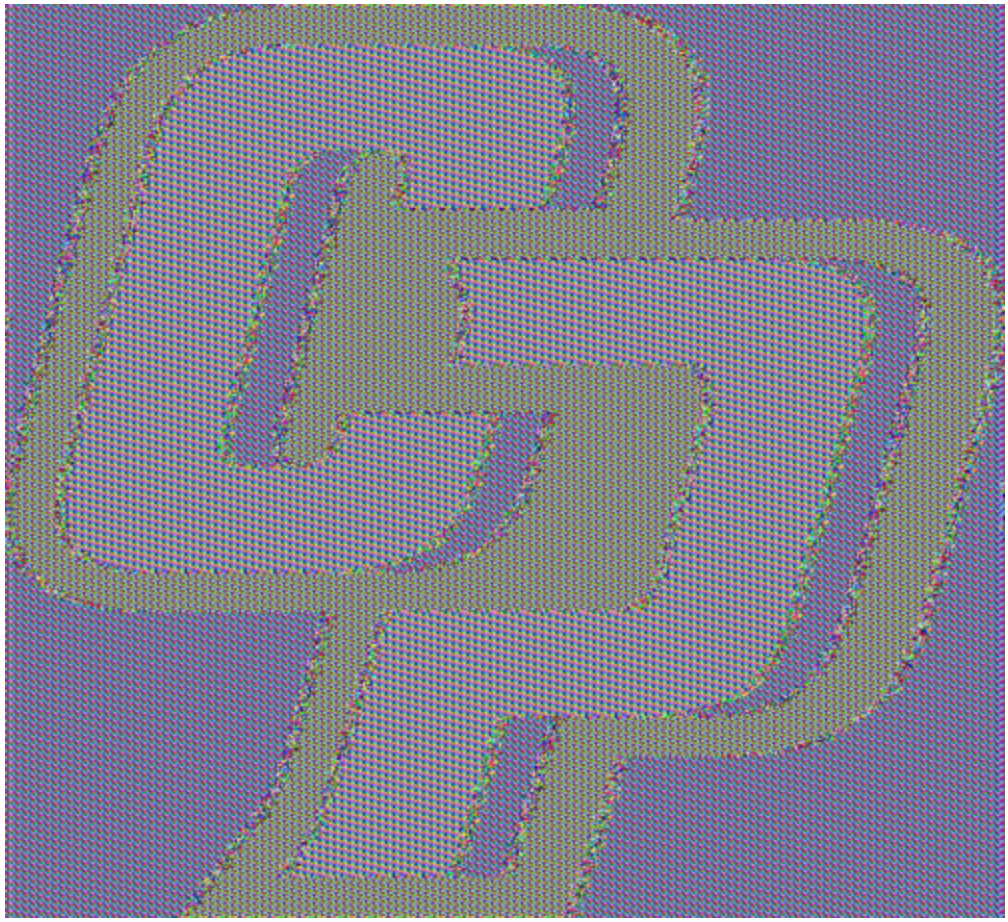
  iv. Loop through 128 bits at a time and encrypt the block based on the result of the XOR.

```
def encrypt_with_cbc_mode(plain_text, secret_key, iv):
    # CBC mode encryption, where each block depends on the previous one
    cipher_machine = AES.new(secret_key, AES.MODE_ECB)
    ready_text = apply_pkcs7_padding(plain_text)
    previous_cipher_block = iv
    encrypted_msg = b''
    for i in range(0, len(ready_text), 16):
        # a. Set the current block as the next sub array
        block = ready_text[i:i+16]
        # b. XOR our current block with the previous encryped block
        block_for_encryption = perform_xor(previous_cipher_block, block)
        # c. Encrypt this block now as our new cipher block
        cipher_block = cipher_machine.encrypt(block_for_encryption)
        # d. Append the cipher block to our encrypted message
        encrypted_msg += cipher_block
        # e. Assign our new previous block to the cipher block just created to pre
        previous_cipher_block = cipher_block
    return encrypted_msg
```

  - **Observations:**

- I used both of these modes to encrypt the following image

- The following image represents the EBC Mode:



    - Notice how the EBC mode doesn't do a good job at encrypting this data as the the same type of byte (a white pixel) will map to the same encrypted data leading to a pattern the human eye can easily distinguish and notice the pattern instantly.

- The following image represent CBC Mode:

- The CBC does a much better job at introducing noise to the image making the visual picture look unrecognizable due to the fact that each cipher block encrypted is entirely dependent on the previous cipher text block allowing for new encryption mapping.

Task 2: Limits of confidentiality

- This task was split up into 3 different parts where I began to test the limits of block ciphers.
- **Part1: Submit Implementation**
  - This function will take as input a string provided by the user and do the following:
    a. URL Encode the user data
    b. Format the string by prepending and appending given substr values.
    c. Apply padding.
    d. Encrypt with the cbc function created in task 1.

```
def submit(userdata, key, iv):
    # 1.
```

```
    encoded_userdata = urllib.parse.quote(userdata)
    # 2.
    formatted_data = f"userid=456;userdata={encoded_userdata};session-id=31337"
    # 3.
    formatted_data = formatted_data.encode('utf-8')
    # 4.
    ciphertext = encrypt_with_cbc_mode(formatted_data, key, iv)

    return ciphertext
```

- The goal for thisd function is to make it impossible for a user to provide a string that will result in `verify()` returning true.

- **Part 2: Verify Implementation**
  - This function should take as input the resulting cipher text of the submit function and do the following:

    a. Decrypt the cipher text string.

    b. Parse the bit string for the substring **';admin=true;'**

    c. Return true or false on whether the string exists.

```
def verify(ciphertext, key, iv):
    try:
        # 1.
        decrypted_data = decrypt_with_cbc_mode(ciphertext, key, iv)
        print(f"Decrypted data: {decrypted_data}")
        # 2. Directly search for the byte pattern of ';admin=true;'
        return b';admin=true;' in decrypted_data
    except Exception as e:
        print(f"Unexpected error during decryption: {str(e)}")
    return False
```

- **Part 3: Byte Flip Attack**
  - This was by far the most difficult part in the lab as it took a strong understanding of how you can modify the next ciphertext block by flipping a bit in the current ciphertext block by strategically using xor.
  - **Difficulties:** Since the user text, always encodes chars ";" and "=", it is not feasible to just send the target string as **";admin=true;"**
  - **Solution:** I set my string to be the following target string: `target_string =`

"000000000000sadminetrue"

- This resulted in the following cipher blocks:
  - Block 0: userid=456;userd
  - Block 1: ata=000000000000
  - Block 2: sadminetrue;sess
- So the goal, was to flip 2 of the bits in block 1 using XOR to result in the following
  a. Flip the "s" in block 2 to ";"
  b. Flip "e" in between "admin" and "true" to "="
- The result of this would be the correct formatted string, which would cause verify to return true.

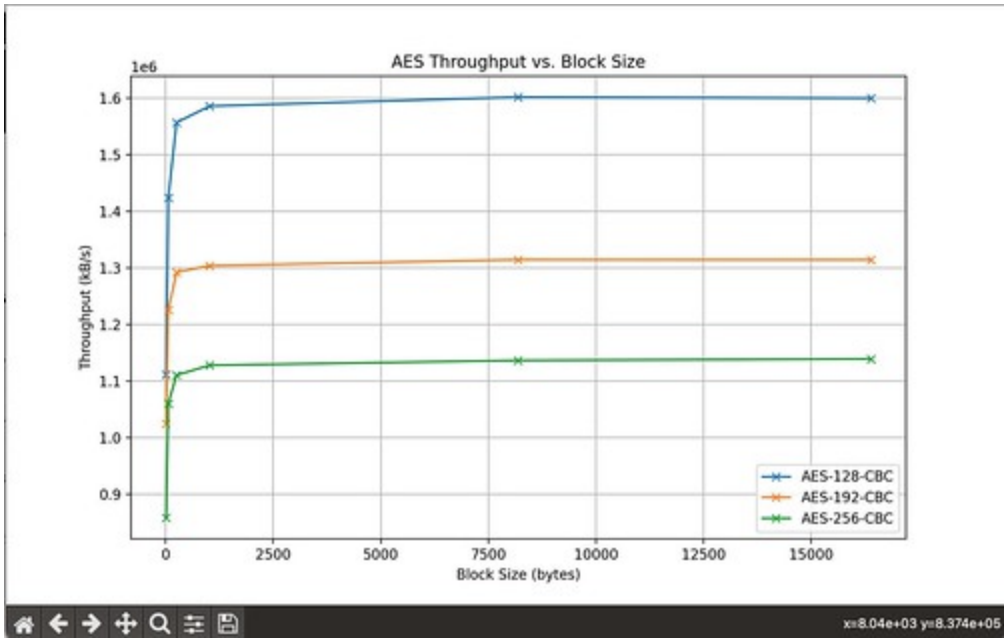
- **Code:**

```
def attack(key, iv):
    target_string = "000000000000sadminetrue"

    ciphertext = submit(target_string, key, iv)
    print(f"Original Cipher: {ciphertext}")

    cipher_array = bytearray(ciphertext)

    mask1 = 0x73 ^ 0x3b  # 's' to ';' #  Correct masks based on the XOR differen
    mask6 = 0x65 ^ 0x3d  # 'e' to '='

    # Applying the masks to the appropriate positions in the first block of ciph
    # which influences the second block's decryption
    cipher_array[16] ^= mask1  # Apply mask at the start of the second block
    cipher_array[16 + 6] ^= mask6  # Apply mask at the sixth position of the sec

    print(f"Modified Cipher Array: {cipher_array}")

    # Verify if the attack was successful
    is_admin = verify(cipher_array, key, iv)
    print("Modified ciphertext is admin:", is_admin)
```

Task 3: Performance comparison

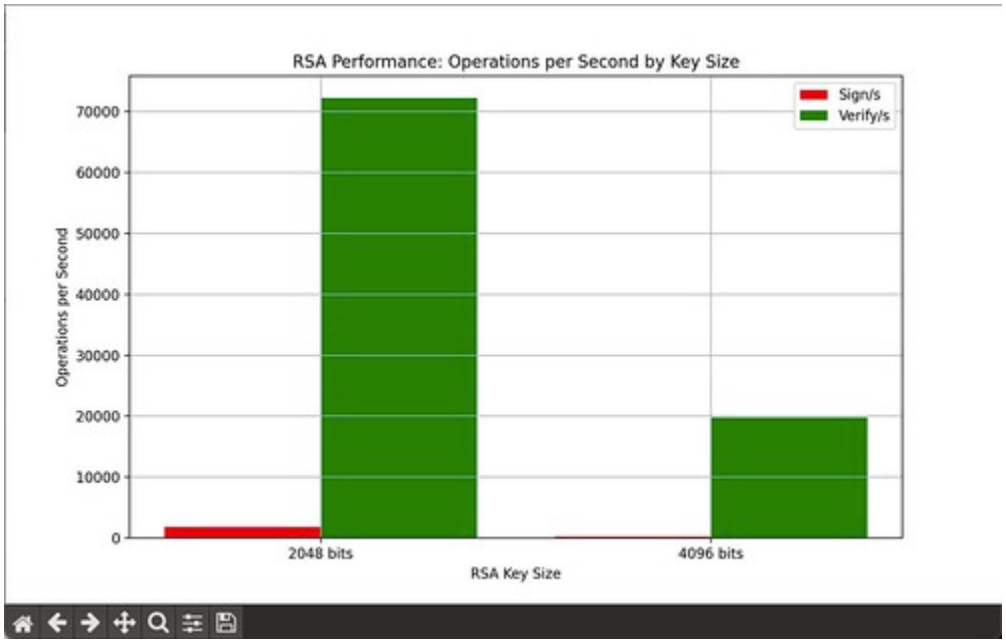
- In this task, I will measure the performance between public and symmetric key algorithms using SSL. Once I gain the measurements, I will use MatPlotLib to plot the results and create 2 graphs:

i. block size vs. throughput for the various AES key sizes:



- Performance slightly decreasees as the key size increases because there is more computation required for the larger keys.

ii. RSA key size vs. throughput for each RSA function:



- The most shocking part of this plot is the difference in speed between verifycation and signing, As you can see private key operations are a lot more expensive than public key operations.

QnA

1. For task 1, looking at the resulting cipher texts, 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?
    - **Answer:** I observed that the ECB mode does not do a good job at encrypting the image's data. You can still observe what the overall image is due to how `identical plaintext blocks map to identical ciphertext blocks` leading to the human eye being able to easily notify what the image is. In the CBC mode, it the image was encrypted in a way that made it indistinguishable from the original photo due to how every encrypted block depends on the previous cipher blocks before it leading to a better encrypted image.
2. For task 2, why this attack possible? What would this scheme need in order to prevent such attacks?
    - **Answer:** A `bit flipping attack` was possible because of how the cipher block is encrypted based on the previous cipher block before it using XOR. An attacker knowing this could easily write an input that will lead to the next cipher block being modified in their favor. This is a problem with the `integrity` of our message as an attacker can now alter the ciphertext.
    - **How to Prevent:** We could implement integrity checks to make sure that any changes to the data, will be caught before we decrypt the data.
3. For task 3, how do the results compare? Make sure to include the plots in your report.
    - **Answer:** The performance analysis in task 3 between AES and RSA had some interesting findings. From these results, you can see that AES is a very fast encryption method where there are speeds greater than 1000 MB/s. It is much faster than RSA, which is why is is used in protocols like TLS. RSA has a much slower throughput, especially in the signing operation for a 4096-bit key size that performs at around 300 operations per second. There is also a notable performance decrease when the key size doubles from a 2048-bit size key to a 4096-bit key. Where the verifies per second drops from 72257.8 to 19769.9. Notice how this is more than double, this is because RSA operations scale non-linearly with respect to the size of the key.