2.1

python test\_cbc.py -p [plaintext\_hex] -k [key\_hex] -iv [iv\_hex]

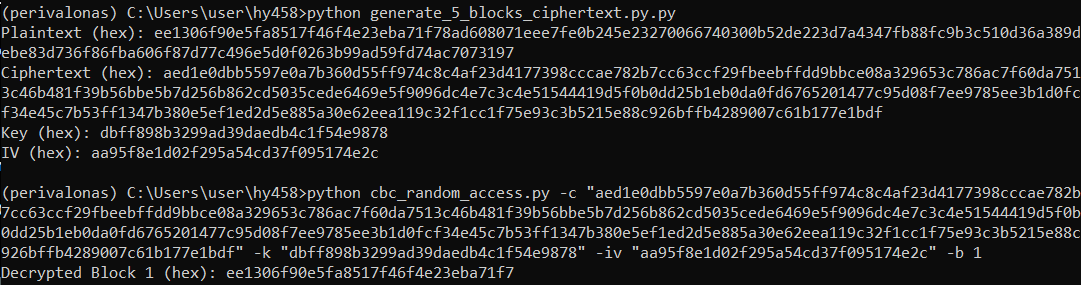
The script will compare my implementation with the library's and print results.

2.2

python generate\_5\_blocks\_ciphertext.py

python cbc\_random\_access.py -c [ciphertext\_hex] -k [key\_hex] -iv [iv\_hex] -b [block\_num]

This is possible because each block's encryption depends on the previous ciphertext block and the XOR operation used in the encryption process.



2.3

python cbc\_attack.py

Since the IV is equal to the key, the decryption process involves XORing the ciphertext block with the key/IV.

* P1 = D(C1) XOR IV
* P2 = D(0) XOR C1
* P3 = D(C1) XOR 0

P2 is the XOR of D(0) and C1. Since the attacker knows the value of C1 (it was part of the intercepted message), they can XOR P2 with C1 to obtain D(0). Now that the attacker has D(0), they can XOR it with the IV to obtain the original message block: Original Message = D(0) XOR IV

2.4

python gcm\_basic.py -e -t <plaintext\_in\_hex> -k <key\_in\_hex>

python gcm\_basic.py -d -t <ciphertext\_in\_hex> -k <key\_in\_hex> -iv <iv\_in\_hex> -g <tag\_in\_hex>

The provided script implements AES-128-GCM encryption and decryption in Python using the cryptography library. It supports command-line execution with options for encryption (-e) or decryption (-d). The user can provide input in hexadecimal format for plaintext, key, initialization vector (IV), and authentication tag. The script prints the encryption or decryption steps along with the final ciphertext, IV, and tag.

2.5

python gcm\_destroy.py -p [plaintext in string] -k <key\_in\_hex> -e [what to destroy]

Altering any of the encryption elements (ciphertext, IV, key, tag) typically results in a decryption failure, providing a level of security and integrity. If an attacker attempts to modify any of these elements, the decryption process will fail, preventing unauthorized access or tampering of data. Modifying the ciphertext makes the decryption process fail, which is a desirable outcome in cryptographic systems as it prevents unauthorized alterations to the data.This behavior is a fundamental aspect of authenticated encryption schemes like AES-GCM.

2.6

python gcm\_extended.py -e -t <plaintext\_hex> -a <associated\_data\_hex> -k <key\_hex>

python gcm\_extended.py -d -t <ciphertext\_hex> -a <associated\_data\_hex> -k <key\_hex> -iv <iv\_hex> -g <tag\_hex>

python gcm\_extended.py -c -t <ciphertext\_hex> -a <associated\_data\_hex> -k <key\_hex> -iv <iv\_hex> -g <tag\_hex> -m <elements\_to\_corrupt>

Ciphertext Corruption: Modifying the ciphertext results in a decryption failure due to the authentication process.

Associated Data Corruption: Any alteration to the associated data leads to a failed decryption, demonstrating the importance of data integrity.

IV Corruption: Corrupting the IV prevents successful decryption as the IV is crucial for producing different ciphertexts with the same key.

Tag Corruption: Modifying the tag causes authentication failure during decryption.

Combinations of Corruptions: Corrupting multiple elements increases the difficulty of successful decryption, emphasizing the importance of protecting all encryption components.

GCM's ability to detect tampering makes it ideal for ensuring the integrity of transmitted data. By including non-secret information (e.g., a checksum or metadata) as associated data during encryption, we can guarantee that the data arrives at its destination undamaged. This is particularly valuable for scenarios where data integrity is critical, even if the data itself doesn't need to be kept secret.

2.7

python gcm\_decrypt.py -c <ciphertext\_hex> -key <key\_hex> -iv <iv\_hex> -i <block\_index>

In GCM (Galois/Counter Mode), integrity is guaranteed through the use of a Message Authentication Code (MAC) called the authentication tag. The authentication tag is generated during encryption and is based on both the ciphertext and additional data (if any). It ensures that the ciphertext has not been tampered with during transmission or storage.

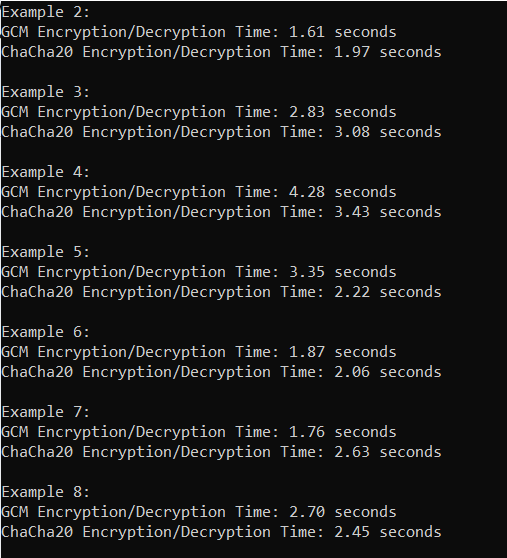
However, when you access GCM-encrypted data in a random order, as long as you provide the correct key, IV, and block index, GCM decryption will still guarantee integrity for the specific block you are decrypting. If the ciphertext for that block has been tampered with, the authentication tag will fail to verify, and decryption will result in an error.

So, the integrity of individual blocks is still guaranteed as long as the authentication tags for each block remain valid.

2.8

python compare.py

ChaCha20 is designed to be computationally efficient, making it faster in software implementations compared to AES-GCM. It uses simple bitwise operations and integer arithmetic, which are generally faster on most CPUs.



Above we can observe that with random 100mb data sometimes the ChaCha20 is faster but that is not always the case. I couldn’t plot the results because of some library dependencies.

2.9

python decryptor.py -d <estimated\_time> -c <ciphertext> -t <tag>

This script will try various seed values to decrypt the ciphertext within a specified timeframe and print the results of each attempt.

2.10

python bonus.py

In my GCM implementation, I used AES-ECB for encryption and decryption. The AES key was derived into subkeys H and C. Encryption involved creating a counter block, incrementing it for each block, and XORing with plaintext. Authentication used HMAC-SHA256 with a key derived from H. Decryption mirrored encryption.