Understanding the Galois Counter Mode (GCM)

HW3 - CNS Sapienza

Giulia Muscarà 1743261

November 21, 2019

1 Objectives

The purpose of this report was to analyse the Galois Counter operating mode in its functioning and in terms of performance when encrypting and decrypting variable size files, comparing it to Cipher block chaining operating mode. LibreSSL was used during the experiments because GCM is only supported in the said fork.

2 Approach

To carry out the analysis, the commands were executed on a scriptable Bash shell running on a Lubuntu virtual box, as it provides a fast and lightweight operating system with a clean and easy-to-use user interface.
The version of LibreSSL that was used is 2.1.6.
Comparisons were made with respect to aes-256 cipher, executing three rounds of encryption and decryption for both the analyzed operating modes, keeping track of the average running times. Tests were made on three text files and three images to highlight the differences between these two types of data. Text files were randomly generated setting the desired size: one file of 100 kB, one of 1 MB and one of 10 MB, in order to monitor how the encryption and decryption speeds would vary with respect to the variation of the files' size.
All the keys and initialization vectors used during the process were randomly generated as well, using the "rand" command, used in Fig.1, and specifying the desired length of the output each time.

```
user@user-VirtualBox:~$ openssl rand -hex 32
17f75ca740be1d053b1d88aa9a20397a4c8af9c8ade91ccae5fd9789085142ba
user@user-VirtualBox:~$ openssl rand -hex 16
94cb0f07b4ebfad428edf2c459cc89fa
```

Figure 1: Pseudo-random bytes generation

The encryption and decryption times were recorded using the bash shell's primitive *time*, that runs the associated command and reports the system resource usage. Most information shown by *time* is derived from the system call *wait3*. As shown in the example below, the command outputs real, user and system elapsed time but the reported measures will be referred to user time, as it is the amount of CPU time spent in user-mode code within the process. The example also shows the command "openssl enc", used to encrypt a file with the option "-e" and to decrypt with the option "-d" using a specific cipher, operating mode, key and initialization vector. The salt was not specified in the example as the command "-salt" automatically uses a randomly generated salt.

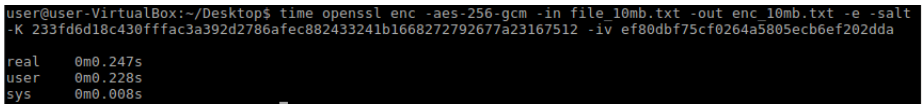

Figure 2: Example of time primitive usage

3 Results

Table 1: AES-256-cbc

File	Average encryption time (ms)	Average decryption time (ms)
100 kB txt	8,00	5,33
1 mB txt	8,00	6,67
10 mB txt	108,00	102,67
100 kB image	5,33	5,33
1 mB image	16,00	21,33
10 mB image	108,00	116,00

Table 2: AES-256-gcm

File	Average encryption time (ms)	Average decryption time (ms)
100 kB txt	8,00	6,67
1 mB txt	28,00	26,67
10 mB txt	192,00	201,33
100 kB image	8,00	8,00
1 mB image	28,00	24,00
10 mB image	214,67	205,33

In the tables above the results of the experiment were recorded, for each pair of cipher and operating mode and for each file size and data type. The average encryption and decryption times were calculated on three trials for every measure, executing the script thrice.

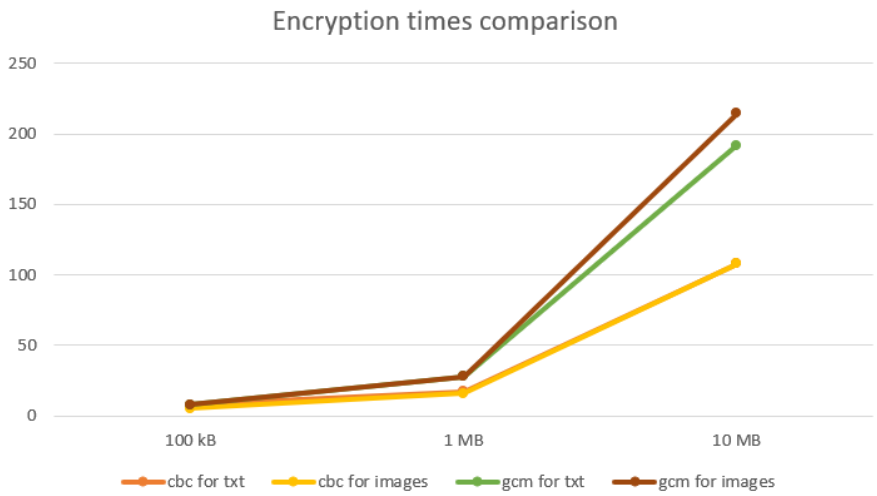

Figure 3: Encryption times comparison

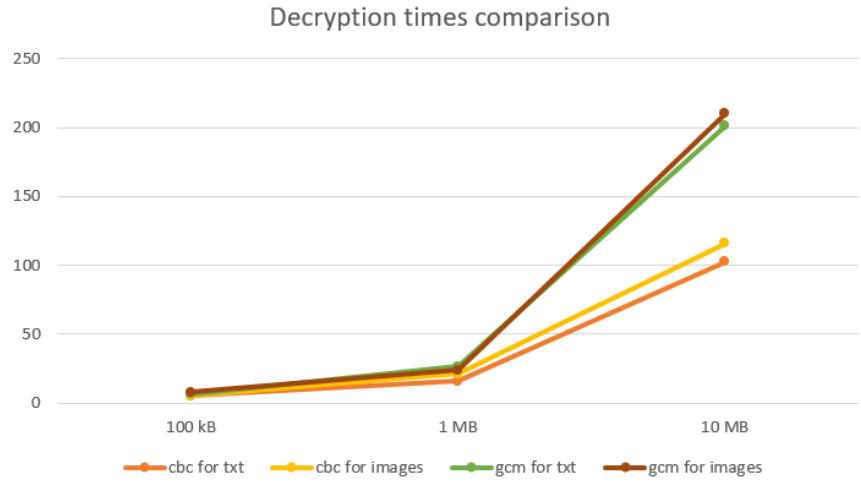

Figure 4: Decryption times comparison

4 Aes-256-gcm vs aes-256-cbc

As shown in the graphs in Fig.3 and Fig.4, for both encryption and decryption, cbc proved to be faster than the Galois counter mode and, as the file size increases, also encryption and decryption times do, keeping speed ratio approximately constant. Encryption and decryption processes tend to be slower for images than for txt files, regardless of the operatin mode used.
Indeed, cbc decryption allows parallelization, given that each block is xored with the previous ciphertext block, already available before starting the process as input. Also, being cbc decryption process simpler than gcm decryption, it is feasible that the former turned out to be the fastest.
On the other hand, cbc encryption does not allow parallelization nor preprocessing as each block must be xored with the previously encrypted one. Even though cbc encryption was expected to be slower than gcm encryption, the result was the opposite. This may be due to the fact that gcm encryption process involves the production of the authentication tag and is thus more complex if it is not pipelined and parallelized in the used hardware architecture. In optimal conditions, the encryption process should be faster when using gcm.
For both modes, encryption and decryption times are proportional to the lenght of the input data. As far as security is concerned, under the assumption a unique initialization vector is chosen for each new encryption made with the same key, gcm is proven to be cryptographically secure. However, if the assumption was missing, gcm would be exposed to a chosen IV attack.
So both cbc and gcm are proven to be secure, as long as they are adopted in suitable conditions. Given that gcm also provides authentication, it could be chosen over cbc in the case that messages or additional data authentication was a requirement. On the other hand, if there was the need to use an operating mode compatible with a more basic hardware, cbc would provide compatibility and, in this case, it would be faster.