**Final Project: AES-GCM Encryption Application**

Subject: Informational Security  
Student: Javier González Casares  
Date: 10/04/2025

**1. Introduction: Why This Project?**

When it came time to choose a project, there were several options, but cryptography has always caught my attention. I wanted to do something that wasn't just theoretical, but had a practical application and would allow me to better understand how security works in the real world.

Within cryptography, AES is like the gold standard today, used by everyone from your web browser to the government. So learning AES seemed fundamental to me.

But I wouldn't stick with just basic AES. Researching a bit, I saw that AES is used in different "modes of operation." Some only encrypt (like ECB or CBC), but then you need to add something else (a MAC) to ensure no one has modified the encrypted message along the way. Then I discovered GCM (Galois/Counter Mode). GCM does both things at once: it encrypts the data AND also generates an authentication "tag". This tag allows you to verify if the message you receive is exactly the same as the one that was sent, and if it was encrypted with the correct key. It's like having encryption and an authenticity seal in a single package.

Knowing that GCM is used in such important things as TLS (what makes your HTTPS connection secure), SSH, and other protocols convinced me that it was an interesting challenge.

So the goal was: to create a simple application with a graphical interface that would allow anyone to encrypt and decrypt text using AES-GCM, with me understanding each step of the process underneath, thanks to the documentation provided by the professor and the internet.

**2. What Does the Application Do? (Features)**

The application allows you to do the following:

* Encrypt text: You write or load a text, and the application encrypts it using AES-128 in GCM mode.
* Generate Key and Nonce: For each encryption, it automatically generates a 16-byte secret key (AES-128) and a "Nonce" (a 12-byte random number that should not be repeated with the same key).
* Show Results: It shows you the generated key (very important to save it!) and the encryption result, which includes the Nonce, the encrypted text, and the authentication tag, all encoded in Base64 for easy copying.
* Decrypt text: You paste the encryption result (Nonce|Ciphertext|Tag) and the key you used, and the application tries to decrypt it.
* Verify Integrity: The most important part of decryption: before showing you the original text, it recalculates the authentication tag and compares it with the one that came in the encrypted data. If they don't match, it means the data was altered (or the key is incorrect), and it will give you an error instead of showing corrupted data.
* Graphical Interface: It has a simple window with two tabs (Encrypt / Decrypt) made with Tkinter.
* Load/Save: It allows loading text from .txt files and saving the results (encrypted or decrypted text) in files as well.

**3. How Does It Work Inside? (The Algorithms)**

This is where it gets interesting and sometimes complicated. The application is based on two main components: AES and the GCM mode.

**3.1. AES (Advanced Encryption Standard)**

AES is the algorithm that actually does the work of "scrambling" the data so that it cannot be understood without the key. It works on fixed 16-byte blocks. To encrypt a block, it follows these steps several times (called "rounds"):

1. KeyExpansion (Key Expansion): This was a bit difficult for me at first. The original key you give it (16 bytes for AES-128) is not used directly in all rounds. First, it's "expanded" to generate a set of "subkeys" or "round keys," one for each round plus an initial one. This process uses byte rotations, substitutions with the S-Box (see point 2), and XOR operations with constants called Rcon. It's a specific algorithm to ensure that each round key is different and secure.
2. AddRoundKey (Add Round Key): This is the only step that uses the key directly! An XOR operation is performed between the current data block and the corresponding round key. It's done once at the beginning and at the end of each main round.
3. SubBytes (Substitute Bytes): Each of the 16 bytes of the block is substituted by a different byte, using a fixed lookup table called S-Box. This table is designed to be non-linear and is fundamental to security because it breaks the simple relationship between input and output.
4. ShiftRows (Shift Rows): The bytes of the block are imagined as a 4x4 matrix. In this step, rows 1, 2, and 3 of that matrix are cyclically shifted to the left (row 0 doesn't move, row 1 shifts 1 byte, row 2 shifts 2, and row 3 shifts 3). This helps mix the data between columns.
5. MixColumns (Mix Columns): This step is mathematically dense. Each column of the 4x4 matrix is transformed by combining its 4 bytes. It uses multiplications and additions in a special finite field called GF(2^8) (Galois mathematics!). The goal is that a change in a single input byte affects multiple output bytes in the same column, spreading the influence of each bit.

These steps (SubBytes, ShiftRows, MixColumns, AddRoundKey) are repeated several times (10 rounds for AES-128). The last round is special: it omits the MixColumns step.

To decrypt, the whole process is done in reverse: the round keys are used in reverse order and the inverse operations are applied: InvShiftRows, InvSubBytes, AddRoundKey (XOR is its own inverse), and InvMixColumns.

**3.2. GCM (Galois/Counter Mode)**

GCM is the "mode of operation" that uses AES to encrypt long messages and, most importantly, to authenticate them. It combines two processes: GCTR for encryption and GHASH for authentication.

**GCTR (Counter Mode for Encrypt/Decrypt):**

1. First, a "Nonce" (IV) is needed. For GCM, it's normal to use 12 bytes. From this Nonce, an initial counter block J0 is created (basically, the Nonce followed by 0x00000001).
2. GCTR encryption doesn't encrypt the plaintext directly with AES. Instead, it encrypts successive counter blocks. The first counter to be encrypted is ICB = J0 + 1.
3. ICB is encrypted with AES: AES(Key, ICB). The result is called "keystream".
4. An XOR is performed between the first plaintext block and the first keystream block to obtain the first ciphertext block.
5. The counter is incremented (ICB + 1), encrypted with AES to get the next keystream block, and XORed with the second plaintext block.
6. This is repeated until all the text is encrypted. If the last block is shorter, only the necessary part of the keystream is used.

Advantage: Decryption is exactly the same process: you encrypt the counters and XOR with the ciphertext to recover the plaintext. Also, since each block is encrypted independently, it can be done in parallel!

Key Function: \_gctr in [gcm.py](http://gcm.py)....

**GHASH (Hash Function for Authentication):**  
Here is where the "magic" (and complexity) of Galois comes in. GHASH creates a "digest" or "fingerprint" of the data.

1. It needs an H key, which is calculated once by encrypting a 16-byte block of zeros with AES: H = AES(Key, 0x00...00).
2. GHASH processes two types of data:
   * AAD (Additional Associated Data): Data you want to authenticate but not encrypt (e.g., network headers). My GUI doesn't ask for them, so I use them empty (b'').
   * Ciphertext (C): The result obtained from GCTR.
3. The AAD and ciphertext (both with zero padding if necessary to be multiples of 16 bytes) are processed in 16-byte blocks.
4. An accumulator value Y is maintained (starts at 0). For each block X (whether from AAD or C):
   * XOR is done: Y = Y ^ X
   * Multiplied in the Galois field GF(2^128): Y = Y \* H (using the \_galois\_multiply function).
5. After processing all AAD and C blocks, a final special block containing the length (in bits) of the AAD and ciphertext is processed.
6. The final value of Y is the result of GHASH.... Key Function: \_ghash in [gcm.py](http://gcm.py). The \_galois\_multiply multiplication is its fundamental piece.

**Calculation of the Tag (Authentication Tag):**

1. The Tag is not directly the result of GHASH!
2. To get the final Tag, first the initial counter block J0 is encrypted with AES: E(K, J0).
3. Then, an XOR is performed between the GHASH result and E(K, J0):
   * Tag = GHASH(AAD, C) ^ E(K, J0)
4. This 16-byte Tag is sent along with the Nonce and the ciphertext.

**Verification when Decrypting:**

1. The receiver has the Key, Nonce, Ciphertext, and received Tag.
2. They calculate H and J0 just as the sender did.
3. They calculate GHASH(AAD\_received, C\_received) using H.
4. They calculate E(K, J0).
5. They calculate the expected tag: Tag\_expected = GHASH\_calculated ^ E(K, J0).
6. They compare Tag\_expected with the received Tag. If they are identical, the verification is successful! The message is authentic and they can proceed to decrypt the ciphertext using GCTR. If they don't match, an error is thrown and decryption stops.

**4. Challenges During Development**

Implementing this from scratch wasn't trivial. I encountered several obstacles:

* The Galois Multiplication (GF(2^128)): Ugh! Understanding the theory behind finite fields and how to implement binary multiplication with polynomial reduction was, by far, the most complicated part. The \_galois\_multiply function needs to handle 128-bit integers and correctly apply the logic of shifts and XORs with the 0xE1... polynomial. I based it on explanations and examples I found, and had to debug it very carefully (using print to see intermediate values) until it worked as expected.
* The Complete GCM Flow: Making sure all the GCM steps (calculating H, J0, ICB=J0+1, calling GCTR, calling GHASH with AAD and C, encrypting J0, doing the XOR for the tag) were done in the correct order and with the correct data was a mess at first. It was easy to confuse J0 with ICB, or what to encrypt for the final tag. I had to reread the specification (NIST.FIPS.197) several times.
* Counter Handling: The GCM specification says that the counter is incremented in a specific way (only the last 32 bits). Implementing increment\_counter correctly, handling the possible overflow of those 32 bits, required attention.
* Debugging: Debugging cryptographic algorithms is difficult because a small error can completely change the result and it's not obvious where the failure is. I used many temporary prints (some are still commented in the code as # print(...)) to track the values of blocks, counters, and intermediate GHASH results.
* AES (MixColumns and KeyExpansion): Although there are more resources on AES, understanding the mathematics of MixColumns well and following the exact KeyExpansion algorithm also took time and debugging.

**5. Project Structure**

The code is organized into four Python files:

* [main.py](http://main.py): The entry point. It simply creates the main window of the application and starts it. It contains no encryption logic.
* [gui.py](http://gui.py): Defines the EncryptionApp class that creates the entire graphical interface using Tkinter (tabs, text boxes, buttons). It handles button events (encrypt, decrypt, load, save) and calls the functions from [gcm.py](http://gcm.py) to do the real work. It also handles Base64 encoding/decoding and output formatting.
* [gcm.py](http://gcm.py): Contains the GCM class. It implements the main logic of the GCM mode: the encrypt and decrypt functions, and the auxiliary functions \_gctr, \_ghash, and \_galois\_multiply. It uses the AES class from the other file.
* [aes.py](http://aes.py): Contains the AES class. It implements the basic AES algorithm (encryption and decryption of a single 16-byte block). It includes all round operations (\_sub\_bytes, \_shift\_rows, \_mix\_columns, \_add\_round\_key) and key expansion (\_key\_expansion).

**6. How to Run the Application**

Requirements: You need to have Python 3 installed on your system. No external libraries need to be installed, as everything (including Tkinter for the GUI) usually comes with Python.

Download the files: Make sure you have all four files ([main.py](http://main.py), [gui.py](http://gui.py), [gcm.py](http://gcm.py), [aes.py](http://aes.py)) in the same folder.

Run from terminal: Open a terminal or command line, navigate to the folder where you saved the files and run the following command:

python main.py

Usage: The application window will open.

* To encrypt: Go to the "Encrypt" tab, type your text, click "Encrypt". Copy and save the key that appears! Also copy the encrypted result.
* To decrypt: Go to the "Decrypt" tab, paste the encrypted result in the upper field and the key (which you saved before) in the key field. Click "Decrypt". If everything is correct, the original text will appear.

**7. Design Decisions and Additional Notes**

* AES-128: I decided to implement only AES-128 (16-byte keys) to simplify the project a bit, although the structure of the [aes.py](http://aes.py) code is prepared to be adapted to AES-192 or AES-256 by changing the number of rounds and the key expansion logic.
* 12-byte Nonce/IV: I focused on the standard GCM case with 12-byte (96-bit) IVs because it's more efficient and simplifies how the initial counter J0 is calculated.
* No AAD in the GUI: Although the [gcm.py](http://gcm.py) implementation does have the auth\_data parameter, I didn't add a field in the graphical interface to input it, to keep it simpler. Currently, empty AAD is always used.
* Output/Input Format: I chose to use Base64 for the key, nonce, ciphertext, and tag because it's a standard format for representing binary data as text, easy to copy and paste. I used the | character as a separator between Nonce, Ciphertext, and Tag in the output string because it's an uncommon character in Base64 and easy to split (split('|')).
* Text Encoding: I always use UTF-8 to convert between the text the user writes and the bytes the encryption algorithm needs. It's the most common standard today.
* GUI Library: I used Tkinter because it comes integrated with Python and I didn't want to add external dependencies to the project.