### License Statement

This is a Python version of AES-GCM. It is a modification of an AES and an AES-GCM program posted by Cyrill Gössi at https://github.com/cgossi/fundamental_cryptography_with_python/tree/main . Cyrill graciously allowed the use of his programs for modifications. However, only use for testing, education, demonstration and research is allowed. Any use in operational application is prohibited.

The application of the FLTed function and other functional transformation like radix-n transformation and use of n-state involution functions for n=2^k not being additions over GF(2^k) in characteristic in cryptography is protected by issued and pending US patents. No license is provided for application of the claimed subject matter in these IP cases. Contact us at info@labcyfer.com for any further information.

### How to RUN?

## Go to page 3

### Subject Matter

The Advanced Encryption Standard (AES) and Advanced Encryption Standard- Galois Counter Mode (AES-GCM) are among the most widely used encryption methods. AES-GCM is AES-CTR (Counter Mode) with authentication added.

AES-GCM is a preferred encryption standard in TLS 1.3.

AES-GCM (or CTR mode) applies AES for keystream generation. This keystream is XORed with the plain text to form the cipher text. This is different from the original AES method, which is defined in a forward (encryption) and reversing (decryption) variant. In AES-GCM the same keystream is generated in encryption and decryption and only the AES forward mode is used.

The FLT and other transformations may be applied in different aspects of AES-GCM. For the purpose of demonstration the transformation is applied only to the AddRoundKey() module and only applied for round r=9 in each block.

The program is to demonstrate the effect of functional transformation on the generated cipher text. It is not intended as a tutorial on AES or AES-GCM.

### Warning

**The effect of the functional transformation is seemingly a simple one. Don't be deceived. We are talking about factorial levels of modifications. For n=256 we are talking about a factor 10^500. Believe us, there is no way that you can remember changes you make in functions like sc256, sn256. If you do, please save the tables under their own names. We have absolutely no way to retrieve lost or forgotten parameters. You are completely on your own on this. That is another reason why we only provide a license for education, trial, testing and research.**

## The Programs

### Background

This section provides a bit of detail on the modification of the base programs.

The effect of the functional transformation is expressed in the generated cipher text. As such, the Python program is not intended to be a user friendly encryption/decryption demo. Running the programs requires user actions. The Python program as provided is a combination of the AES encryption and AES-GCM encryption programs. The AES-GCM program asks for plaintext (ASCII) input, which is converted into byte format and printed as such.

The initiation vector IV and the key KEY are preloaded/set. These are used unchanged every time the program is run. Please be aware of that. The same applies to AAD for computing the tag.

The program uses the AES_encryption step (H = aes_encryption(b'\x00' * (128 // 8), K, fun1)) to create the keystream. The variable 'fun1' is a lookup table that replaces the XORing of words of 8-bits. The 'fun1' table is loaded by:

```
  # Load the sc256.mat array
  data_fun1 = scipy.io.loadmat('sc256.mat')
  fun1 = data_fun1['sc256']
```

The table sc256.mat is a lookup table that is represented by an addition over GF(256) replacing XORing words of 8-bits.

The actual encryption of the plaintext takes place in
*GCTR(K, ICB, X)*,
which is left unchanged in this example of functional transformation.

The net effect of the replacement of XORing with table 'sc256' is that the output in ciphertext is unchanged. There will not be any errors generated in the processing of the FIPS-197 test-vectors which are included and maintained in the programs.

We created different, slightly modified programs, to test the effects of modifications. Because of the need to process in bytes, we created the encryption and decryption programs. These are essentially identical. However, the encryption program accepts the plaintext as ASCII and generates the ciphertext output as a string of hex elements. One should copy (CTRL-c) the string and then run the decryption program and do Paste (CTRL-v) when asked for input.

We provide the decrypted output in ASCII and hex format. In case of matching encryption/decryption programs the output of the decryption is identical to the plaintext. If not, one will see a hex representation.

A second set of encryption/decryption programs applies a function '*sn256*' as *fun1*, which is different from '*sc256*' and not equivalent to bitwise XORing.

### Assertion Errors

In case of running matching programs, one may get an “assertion error.” This is caused by using the ‘*sn256*’ lookup table. In that case the generated ciphertext is not the expected ciphertext. Despite the error, one may still correctly decrypt.

## How to run the programs!

**1) unzip aesgcmfltPython.zip is its own folder, for instance ‘aespython’**

**2) open CMD terminal and go to the folder**

**3) make sure you have installed Python and with PIP installed: math, scipy.io**

**4) to run standard output AES-GCM type:**

```
python aesgcm_fltfun_enc.py
```

**5) the program will print a small part of the operational lookup table as 5 by 5 array and will ask for “***Enter the plaintext as ASCII text:***”**

**6) you may enter any text, but as example enter:** *this is the required plaintext* **and hit ‘enter’**

**7) the program displays the hexadecimal format as well as the AES-GCM generated** *ciphertext*

**8) copy the displayed ciphertext (hex) string**

```
909263d4ad52b1bf8bc7e80a87b37cc2fc9f0207b8b68572477b367f5d46
```

**9) Notice that NO assertion errors are generated**

**10) Type:** *python aesgcm_fltfun_decr.py*

**11) The program will ask for:** *Enter or Paste the ciphertext as a hexadecimal string:*

**12) Paste the string copied during earlier step**

**13) The program will display:** *ciphertext1: b'this is the required plaintext'* **This is the recovered plaintext, recovered from the ciphertext. Remember: decryption in AES-GCM is re-encrypting the ciphertext.**

**The programs** *aesgcm_fltfun_enc_sn.py* **and** *aesgcm_fltfun_decr_sn.py* **work in a similar way**

**From the output you will see that the applied function sn256 is different from the earlier sc256.**

**Using the same input: this is the required plaintext , will generate an entirely different ciphertext.**

**Also assertion errors will occur as the auth_tag from test vectors will be different from expected auth_tag. You may ignore, because this is to be expected. If it annoys you, just delete the assertion statements.**

**One may enter the above copied ciphertext:**
*909263d4ad52b1bf8bc7e80a87b37cc2fc9f0207b8b5887e5e79273a565bf163e843f4f5f79c9ea2c0fd574e*
**at the request: Enter or Paste the ciphertext as a hexadecimal string:**

**The output is not the recovered plaintext as a transformed function sn256 was used.**

**Changing the Functions in** *aesgcm_fltfun_enc_sn.py* **and** *aesgcm_fltfun_decr_sn.py***.**

**The function** *sn256* **loaded from** *sn256.mat* **may be changed.**

**Do this in line 425 and 426 of both programs in statements:**

```
data_func = scipy.io.loadmat('sn256.mat')

func= data_func['sn256']
```

**Change sn256.mat to one of:** sn256a.mat **;** sn256b.mat **;** sn256c.mat**;** sn256d.mat; sin256.mat **and change [‘sn256’] to corresponding:** *[‘sn256a’]***;** *[‘sn256b’]***;** *[‘sn256c’]***;** *[‘sn256d’] or [‘sin256’].*

**It may seem that these functions are random. But they are not. They are all different involutions, just not XORings of words of 8 bits. Make sure you do identical changes in** *aesgcm_fltfun_enc_sn.py* **and** *aesgcm_fltfun_decr_sn.py.*

**The table sin256.mat is a 256-state involution that is NOT associative.**

**Date: November 9, 2024**