Sécurité des Systèmes d'Information
Mandatory TP 2 - Galois Counter Mode

October 14th, 2020

Surrender on **Moodle** your Python 3 file(s) **.py**, before **Tuesday, November 3rd, 2020 at 11:59 pm (23h59)**.

Your code needs to be **commented**.

Goal

The goal of this TP is to implement a block cipher operation mode, which is not just an encryption, but what we call an "authenticated encryption" (i.e. an encryption that provides not only confidentiality, but authentication too), called the Galois Counter Mode.

We'll need to use the AES encryption box from TP1 (to encrypt 128 bit messages with 128 bit keys). A correction of this TP is available on moodle for those who didn't do TP1 or didn't managed to make it work (see the end of this TP for more details).

Galois Counter Mode

The Galois Counter Mode combines two elements : a traditional counter mode block cipher, and an authentication system based on polynomial multiplications in a finite field (finite fields are also called "Galois fields", hence the name).
Seems familiar ? That's because we did the same polynomial multiplications in the AES TP !
There's one difference though : this time, we're using another finite field : $GF(2^{128})$, defined with polynomial $x^{128} + x^7 + x^2 + x^1 + 1$. So this time, if the multiplication gives a polynomial of degree bigger or equal to 128, then we just have to replace x^{128} by $x^7 + x^2 + x + 1$.

Galois counter mode uses 128 bits blocks (Isn't that like AES), and is often used in conjunction with... AES.

We're starting the protocol with :

- The plaintext P (any length),
- The initialisation value IV (that will be used to create the initial counter value),
- The Key K, which size needs to be suitable for the underlying encryption Box (in this case, since we're using AES, we need a 128, 192 or 256 bit key),
- What we call "Additional Authenticated Data", noted A,

Here's an overview of the whole process :

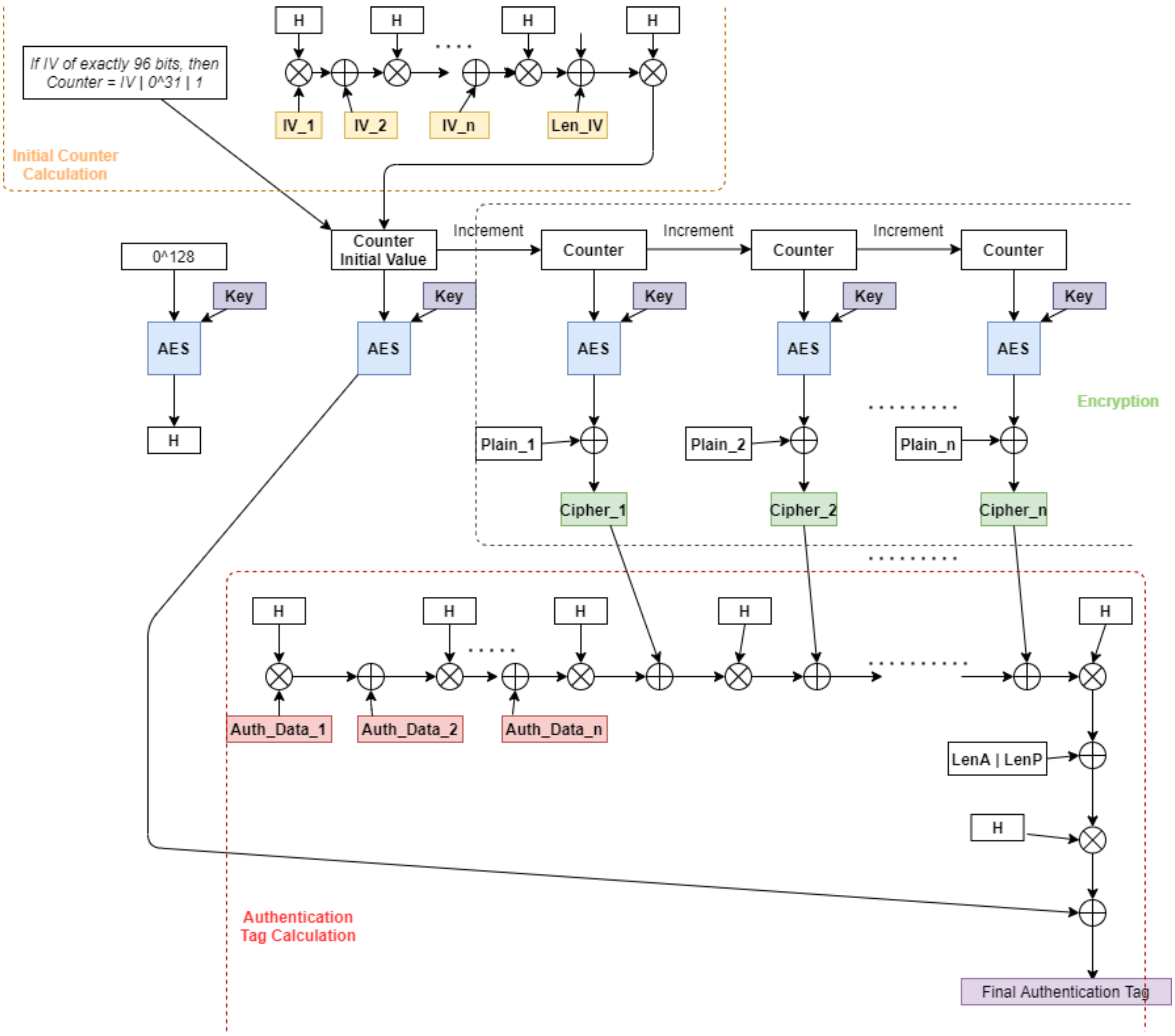

Figure 1: Galois Counter Mode Complete Diagram

The Hash Subkey

First, we need to compute something called the "hash subkey", H. H is defined as the encryption of the 128-bit message full of zeros, with the key, in the encryption box (that's completely independent of the whole counter encryption process). The resulting "cipher" is H.

Encryption with Counter Mode

We're using a simple counter mode here, which means :

- An IV which is used as a basis to compute the initial value of the counter (more on that initial value later),
- Then, this counter is encrypted with the key in the encryption box. We'll keep the result C_{start} somewhere to use it later in authentication.
- Then, the counter is incremented, and encrypted with the key, using the AES encryption box (from TP1).
- And that encrypted counter is XOR with the first 128 bit block of the plaintext to get a cipher block.
- Then we increment the counter, and we encrypt it again to XOR it with the following plaintext block, and so on.
- For the last plaintext block (which size may be anywhere from 1 bit to 128 bits), when doing the XOR with the last encrypted counter, we're only taking the most significant bits of the counter. For example, if the last block is only 40 bits long, we'll take only the 40 most significant bits of the encrypted counter to XOR them with the 40 bits of the plaintext. That means the last cipher is of the same size as the last plaintext (and the whole ciphertext is exactly as long as the whole plaintext).

Up to this point, that's just a simple counter mode (except for the initial IV calculation that is defined later).

Padding and 128 bit blocks

Before computing the authentication tag, we need to organise the Authenticated data, and the ciphertexts, and for that we may need to do some padding :

- The authenticated data is cut in 128 bit blocks. We name those A_1, A_2, ... A_{m-1}. The last block A_m^* (of size between 1 and 128 bits) is padded with zeros if needed to have 128 bits. If v is the size of A_m^*, we note $A_m = A_m^* \parallel 0^{128-v}$.

- The same thing applies to the ciphertexts : blocks C_1, C_2, ... C_{n-1} of size 128 bit, and block C_n^* (between 1 and 128 bits, same size as the last plaintext) is padded with zeros if needed. If u is the size of C_n^*, we note $C_n = C_n^* \parallel 0^{128-u}$.
- We'll need later a special block, noted L, for authentication, containing the length, i.e. the exact number of bits, both for A and C (lengths before padding). Each length will be written as a 64 bit integer (note that this limits the maximum size of both the plaintext and authenticated data). This we be a block of size 128 bit, $L = len(A) \parallel len(C)$.

Authentication with Galois field

Here is where the fun starts : we're doing the authentication as follows :

- We'll start the authentication by multiplying A_1 by H (polynomial multiplication in $GF(2^{128})$, details in next section).
- Then, we XOR that result with A_2. We multiply this new result by H again (polynomially in $GF(2^{128})$).
- This process ((result XOR A_i) * H) is repeated for each A_i...
- ... and then for each C_i. (In the end, we have ((((((A_1 * H) XOR A_2) * H)...) XOR A_m) * H) XOR C_1) * H) ...) XOR C_n) * H).
- Then we apply this process one last time with the special block L (containing the lengths of A and P, defined earlier), (result XOR L) * H.
- And finally, we will XOR that last result with the first counter encrypted C_{start} (Remember ? That's the one we said we'll keep for later). This finally gives us our authentication tag T !

The initial counter value

The initial counter value is defined in two different ways :

If the IV is exactly 96 bits long, then the initial counter (128 bits) is defined as :

$Counter_0 = IV \parallel 0^{31}1$

If it is not exactly 96 bits long, then it is defined by applying the same process as the authentication process, except we're replacing "A and C" by "the empty string and the IV" (with the same padding method as for A and C, which means you just have to pad the IV with zeros, and add one block for the original IV length) :

$Counter_0 = ((((((IV_1 * \text{H}) \text{ XOR } IV_2) * \text{H}) \text{ ... XOR } IV_N) * \text{H}) \text{ XOR } L_{IV}) * \text{H}).$

(These * are still polynomial multiplications in $GF(2^{128})$, and $L_{IV} = 0^{64} \parallel len(IV)$).

Encryption Recap : Timeline of Operations

The encryption, given P, K, A and IV, returns a pair : the ciphertexts C and the tag T.

If we recap how these parts are using each other, we need to :

1. First, compute H,
2. Then, compute the initial counter value,
3. Then do the encryption part, to obtain C_{start} and the ciphertexts C,
4. And finally, do the padding, and compute the final authentication tag T (you may anticipate the part concerning the additionnal authenticated data).

Decryption of Galois Counter Mode

The decryption is a little different : given C, T, K, A and IV, it returns either the plaintext, or a special error status "FAIL" : By computing H and the initial counter value, we can then use A and C to compute tag T'. If this tag T' is equal to the received tag T, then we're returning the decrypted plaintexts P. If the tag is not correct, we're returning just a simple special status "FAIL".

Decryption process goes as follows (mostly the same as the encryption without the plaintext encrypting part, which is replaced by the decryption of the ciphers only if tag is correct) :

1. Compute H,
2. Compute initial counter value,
3. With both these, A, and C, compute tag T',
4. If $T' \neq T$, return FAIL,
5. If $T' = T$, then compute all counters, encrypt them, and XOR them with the ciphertexts to find the plaintexts.

TP : Implementation of Galois Counter Mode

Your goal is to implement encryption and decryption with Galois counter mode.

You need an AES Encryption Box to do this TP. If you haven't succeded in implementing such a box in TP1, or haven't done TP1, you can find a correction on moodle, containing an "AES" function and all that is needed to make it work. This function takes a 128 bit message, a 128 or 192 or 256 bit key, and returns a 128 bit cipher (all three are strings, i.e. 16 ascii characters for the message and cipher, 16 or 24 or 32 characters for the key).
Feel free to use it if you want to.
Also, it is important to note that AES uses a polynomial multiplication that is the exact same concept as the polynomial multiplication of Galois, but with a different polynomial (in this case, $x^{128} + x^7 + x^2 + x + 1$). If you've done the polynomial multiplication in AES, it should be easy to adapt it to Galois.