A Note on Advanced Encryption Standard with Galois/Counter Mode Algorithm Improvements and S-Box Customization

Mădălina Chiriţă*
madalina.chirita96@stud.acs.upb.ro

Alexandru-Mihai Stroie*
alexandru.stroie@stud.acs.upb.ro

Andrei-Daniel Safta*
andrei_daniel.safta@stud.acs.upb.ro

Emil Simion†
emil.simion@upb.ro

Abstract

Advanced Encryption Standard used with Galois Counter Mode, mode of operation is one of the the most secure modes to use the AES. This paper represents an overview of the AES modes focusing the AES-GCM mode and its particularities. Moreover, after a detailed analysis of the possibility of enhancement for the encryption and authentication phase, a method of generating custom encryption schemes based on GF(2^8) irreducible polynomials different from the standard polynomial used by the AES-GCM mode is provided. Besides the polynomial customization, the solution proposed in this paper offers the possibility to determine, for each polynomial, the constants that can be used in order to keep all the security properties of the algorithm. Using this customization method, allows changing the encryption schemes over a period of time without interfering with the process, bringing a major improvement from the security point of view by avoiding pattern creation. Furthermore, this paper sets the grounds for implementing authentication enhancement using a similar method to determine the polynomials that can be used instead of the default authentication polynomial, without changing the algorithm strength at all.

Keywords: AES-GCM, Sbox, irreductible polynomials, custom encryption schemes

*Department of Computer Science, University Politehnica of Bucharest
†Mathematics Departament, University Politehnica of Bucharest

1 Introduction

High quality security represents an important concern in modern world, with solutions being continuously developed and updated in order to satisfy the current standards for protecting sensitive information. In this paper the Advanced Encryption Standard, competition winning, Rijndael Algorithm, selected by National Institute of Standards and Technology (NIST)[1] is reviewed in addition to a general presentation and definition of block ciphers and their modes of operation used to encrypt and decrypt data in section 1.1, subsections 1.1.1 and 1.1.2. The vulnerabilities identified for each mode of operation are further presented in subsection 1.1.3. In section 1.2 the Advanced Encryption Standard algorithm is presented, the four main stages are detailed using a scheme of the flow of the algorithm.

Furthermore, in section 2, the report approaches the Galois/Counter Mode, which is a mode of operation for the above mentioned Advanced Encryption Standard algorithm, focusing on the main functions involved in data encryption/decryption. The particular step of this mode of operation, represented by the authentication process and elements included that provide an additional level of security for the encryption/decryption operations is also studied in this paper. In section 3, several design considerations while analyzing the improvement possibilities are introduced and the implementation details for the proposed solution are further described in section 4. The ideas for future work in this field of research and an overview for the currently conducted work are finally presented in section 5.

In order to provide a clearer overview, in the following sections, the Rijndael Algorithm will be simply referred as AES algorithm.

1.1 Block ciphers - Overview

Block ciphers represent a type of cryptographic scheme that operates on blocks of data with previously fixed length. The initial, complete information or input is split into groups of bits(blocks) that have the same size and will be further

[1] https://www.nist.gov/

transformed at each step of the applied algorithm. As a result, from a sequence of plaintext provided as input, after applying a block cipher algorithm, a ciphertext sequence is obtained. The block cipher consists of two main functions, first of which, illustrated above, the encryption operation that applies a transformation on the text provided in clear in order to obtain a text cryptographically secure, while the second operation is represented by the decryption operation, which reverts the first one.

1.1.1 Definition

Considering the following elements: a key (K) of length k, a transformation function (E) and a plaintext P, the output of the algorithm is the result of applying the function E, defined as invertible over $[0, 1]^n$ with n representing the size of the plaintext provided, on the plaintext P using the key K (described as $E_k(P)$). The second function, the decryption operation is the inverse of the encryption function ($D_k = E_k^{-1}$), obtaining the initial text in clear from the known ciphertext.

1.1.2 Modes of operation

The limitation when using a block cipher is caused by the fact that it computes only one block of information of default size. When the plaintext size exceeds it, the partitioning of the data must be prior handled in order to execute the encryption step on each block. The method used to separate each block and manage their interaction defines a mode of operation.

The simplest and, in consequence, the weakest mode of operation is represented by ECB (Electronic Code Book) which is additionally not recommended as a standalone mode of operation when managing sensitive information.
In table 1,are presented the main modes of operation for the AES algorithm, in order to provide an overview of their limitations, usability and the added complexity to the block cipher.

Each mode of operation brings some advantages and a specific level of security, with properties reaching from fast and easy implementation, support for parallel computing at encryption or decryption stages, but additionally is prone to be the

subject of different types of attack that can temper the information, breaking the security. [2]

TABLE 1: Modes of operation

Code	Mode of op	Steps	Elements
ECB	Electronic Code Book	Step1. The plaintext is divided into fixed size blocks (64, 128 bits, data bits + padding). Step2. Same key/algorithms is used for each block	Plaintext, ciphertext, key, encryption schema
CBC	Cipher-Block Chaining	Step1. Same as ECB step1 Step2. Plaintext block is XORed with the IV (initialization vector) Step3. Next block is encrypted using the ciphertext block resulting at step 2, that is further XORed with the next plaintext block. - Each subsequent plaintext block is firstly XORed with the previous ciphertext block obtained before the encryption step.	Plaintext, ciphertext, key, encryption schema, **IV**
CFB	Cipher Feed-Back	Step1. Plaintext is divided in plaintext blocks (no padding needed) Step2. Similar to CBC but, the IV is firstly encrypted. Step3. The result is XORed with the first plaintext block obtaining the first ciphertext block. Step4. The ciphertext block from the previous step is encrypted and then XORed with the following plaintext block. - Each subsequent plaintext block is XORed with the result of applying encryption on the previous ciphertext block.	Plaintext, ciphertext, key, encryption schema, **IV**

[2]https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/

Table 1 – continued from previous page

Code	Mode of op	Steps	Elements
OFB	Output Feed-Back	Step1. Plaintext is divided in plaintext blocks (no padding needed) Step2. IV is encrypted and the result of encryption is XORed with the plaintext block to obtain the ciphertext block Step3. The result of previous encryption is further encrypted and XORed with next plaintext block – this step is continuously applied for each block	Plaintext, ciphertext, key, encryption schema, **IV**
CTR	Counter	Similar to OFB but the IV is replaced or concatenated with values of a counter (function that produces and guarantees uniqueness of a resulted sequence for a period of time)	Plaintext, ciphertext, key, encryption schema, **IV + Counter**
GCM	Galois/ Counter Mode	Developed from the Counter mode of operation - with similar steps. Step 1. The hash subkey (H) is computed, alongside a derivation of a pre-counter element from the initialization vector. Step 2. Using the pre-counter from previous step and the plaintext, a function called GCTR is applied returning a ciphertext. Step3. The ciphertext and additional authenticated data are provided as input to a function named GHASH, followed by the application of the GCTR function on the result, obtaining the necessary authentication tag	Plaintext, ciphertext, key, encryption schema, **IV + Additional authenticated data, authentication tag**

1.1.3 Vulnerabilities

In order to understand the choice of the GCM mode of operation for this paper, the above mentioned modes of operation are presented considering their vulnerabilities and their main causes. Each one of them is susceptible to several kind of attacks

of different difficulty and approach[2]:

- ECB - the encryption step is considered deterministic, meaning that same blocks in clear will result in same ciphertext blocks, that providing a recognisable pattern under any given key. Additionally, the encrypted data can be altered in term of arrangement without further warning. This is the simplest mode of operation but it is not commonly used due to its limitations and vulnerabilities when there are more than one plaintext blocks to be computed.

- CBC - the vulnerability of this mode is mainly represented by the randomness of the initialization vector. It is susceptible to attacks when the IV is not only a not repeated number but, also, a random number. When these conditions are not verified, attacks such as translation from specific plaintexts (or ciphertexts) to expected ciphertexts (or plaintexts) are possible.

- CFB - Similar to the previous detailed mode of operation, if the initialization vector is not random there are no guarantees of safety for the same attacks. Furthermore, for the translation of a specific ciphertext in an expected plaintext type of attack there are no guarantees at all.

- OFB - For this mode of operation, similar vulnerabilities are present. Furthermore, the output feedback mode, which due to its operations transforms the encryption scheme in a stream-like encryption flow, is susceptible to attacks that imply in-place modifications of bits in the encrypted sequence.

- CTR - the counter mode of operation provide better security than the previous detailed modes. The vulnerabilities are caused by the pseudo-randomness of the initial value on which the counter is applied, value that is required to be unique and random at the same time. There is a risk of collisions on the counter values after a certain period of time (as illustrated before, the counter function provides guarantees on the uniqueness of the values return for a period of time, if the number of blocks is large enough this period can be exceeded)

- GCM - due to the fact that this mode of operation originates in the CTR (counter) mode of operation, the risks of collision is inherited. Even more, this type of collisions that happen between two messages may expose an important parameter, the authentication key used (H). Revealing the key leads to the complete loss of security in terms of authentication. Additionally, another vulnerability is caused in the case in which the authentication tag size differs keeping identical key. Consequently, a vulnerability is caused by the fact that the length of the authentication tag direclty influence the level of authentication security (for x-bit tag and 2^y blocks the number of authentication security bits is represented by their difference x - y) [5]

1.2 Advanced Encryption Standard

Advanced Encryption Standard represents the substitute for Data Encryption Standard (DES) previously used, being a more mathematically efficient specification of a symmetric block cipher, a particular case or subset of Rijndael cipher algorithm that can manage several key lengths and block sizes and is considered suitable for protecting classified, sensitive information. Depending on the confidentiality level of the data, the 128-bit key length, 196 and, respectively, 256-bit key length can be chosen providing reliable resistance to attacks, especially brute-force attacks that would require a considerable (and in particular cases unreachable) number of resources. Consequently, the main concern remains the encryption key confidentiality and security[3].

1.2.1 Stages of the algorithm

The AES algorithm is based on four main functions that will be further described. Additionally, depending on the key size, 10, 12 or 14 transformation rounds are applied in each case on the initial plaintext. Moreover, for each round, a round key is obtained by applying AES key scheduling algorithm that expands starting from the initial encryption key to a series of new keys.
The algorithm processes the information in a byte-oriented approach, each block of data being partitioned and represented as an array of bytes described as the

[3]https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard

state. The operations that convert the plaintext into the final cryptographic result, the ciphertext, are [3]:

- **AddRoundKey** - the function returns the result of a bitwise XOR operation between the state and the round key (both represented as same size matrices)

- **SubBytes** - the function performs a byte substitution that assures the non-linearity in applying the cipher. Each byte from the initial state (matrix) is substituted using a substitution box (or S-box) resulted as a combination between a multiplicative inverse over GF(2^8) and affine transformation.

- **ShiftRows** – the function shifts to the left the rows of the state matrix, each by an offset of zero, one, two and, respectively three bytes for each consecutive row.

- **MixColumns** – the function provides the diffusion in the cipher and represents for each column of four elements the multiplication in the Galois Field of two four-term polynomials.

The schematic stages of the algorithm are presented in figure 1.

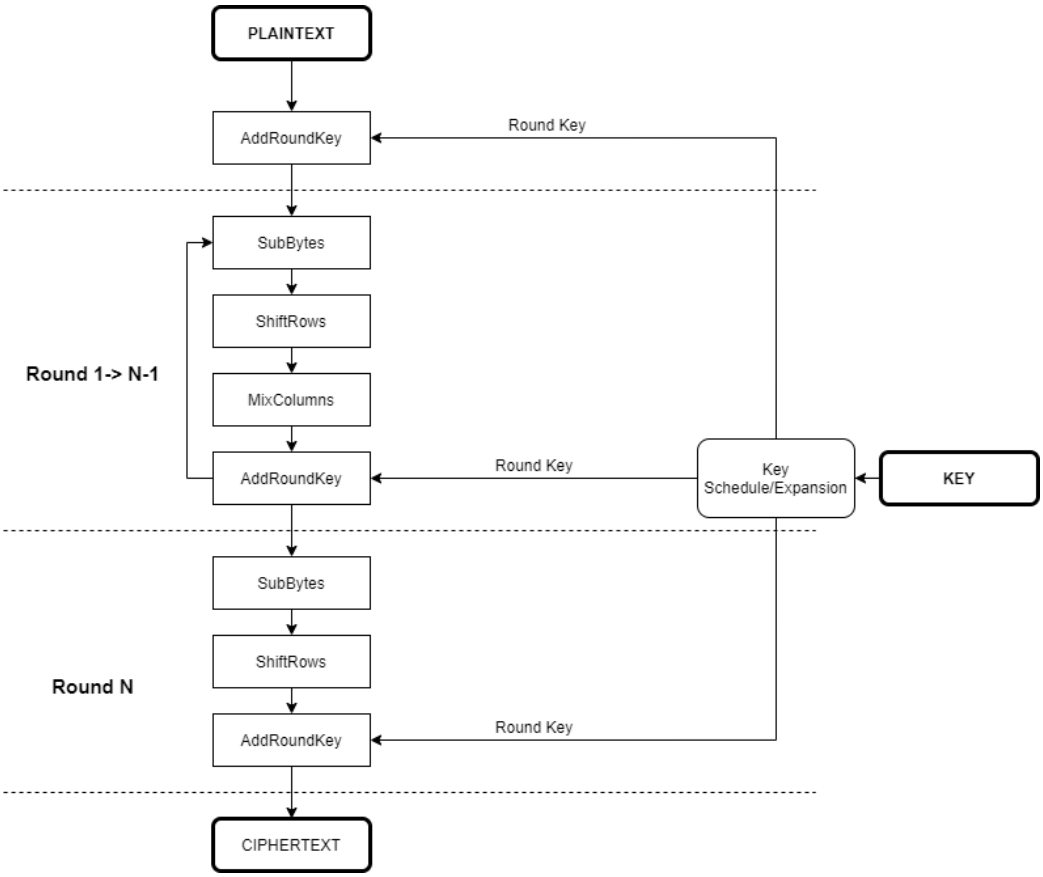

FIGURE 1: Algorithm stages

In the following sections, a new mode of operation is introduced, AES-GCM, that combines the encryption/decryption process previously described with an additional step, authentication.

2 AES-GCM

The Galois/Counter Mode represents one of the modes of operation that can be applied for the AES algorithm. Generally, it applies for block ciphers that use symmetric-key transformations with a block size predefined to 128 bits, making it suitable for the studied algorithm. This mode is widely used due to the fact that it improves the overall performance in the encryption/decryption process with

reduced number of necessary resources.

This mode of operation originates in one of the previous detailed modes of operation in table 1, Counter mode of operation. Using an alternative version of the Counter mode and a particular incrementing function which results in uniqueness for the obtained sequences during a large period of time, the GCM mode provides guarantees about the confidentiality of the information to be encrypted. An additional security layer is represented by the authentication step.

The Galois/Counter Mode provides authenticated encryption/decryption with associated data using a particular hash function defined over a binary Galois field. Consequently, the algorithm using this type of computation and mode of operation will act as a standard encryption and authentication algorithm for sensitive information (considered in this case the message to be encrypted) and, at the same time, will provide authentication for additional, associated data without applying encryption on it. The most common use is represented by the encryption of a packet, which is authenticated, but only the inside information, skipping the header, is encrypted [7].

One of the main advantages of using the Galois/Counter Mode is represented by the fact that is can detect both data alteration and modifications as a result of accidental phenomena and, respectively, deliberate and malicious, unauthorized access[4]. In the next paragraphs, a detailed description of the functions involved in the construction of the AES-GCM algorithm is provided in order to highlight the standard process and understand the main components that are further susceptible to improvements.

2.1 Authenticated Encryption/ Authenticated Decryption

Galois/Counter Mode has two main functions, authenticated encryption and authenticated decryption. For each one of them there are several input parameters needed and output elements provided listed in table 2 [7].

As illustrated in table 2, the authenticated encryption function focuses on encrypting the sensitive, confidential information and computing the correlated

	Input	**Output**
Authentication Encryption	• Plaintext P • Secret key K - customized for chosen block cipher • Initialization vector IV - arbitrary length, unique value within the context • Additional authenticated data A	• Ciphertext C - same length as P • Authentication tag T - with length t < 128
Authenticated Decryption	• Secret key K • Initialization vector IV • Additional authenticated data A • Ciphertext C • Authentication tag T	• Plaintext P or error code (FAIL) - FAIL indicates that the input is not authentic)

TABLE 2: Input/Output elements for GCM functions

authentication tag based on both the encrypted data and the additional data. On reverse, the authenticated decryption function decrypts the ciphertext only after the authenticity (authentication tag provided) is verified.

In the *Recommendation for BlockCipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC* provided by NIST the scheme inserted below (figure 2) illustrates the flow of the algorithm and the main operations applied in order to obtain the encrypted result [4].

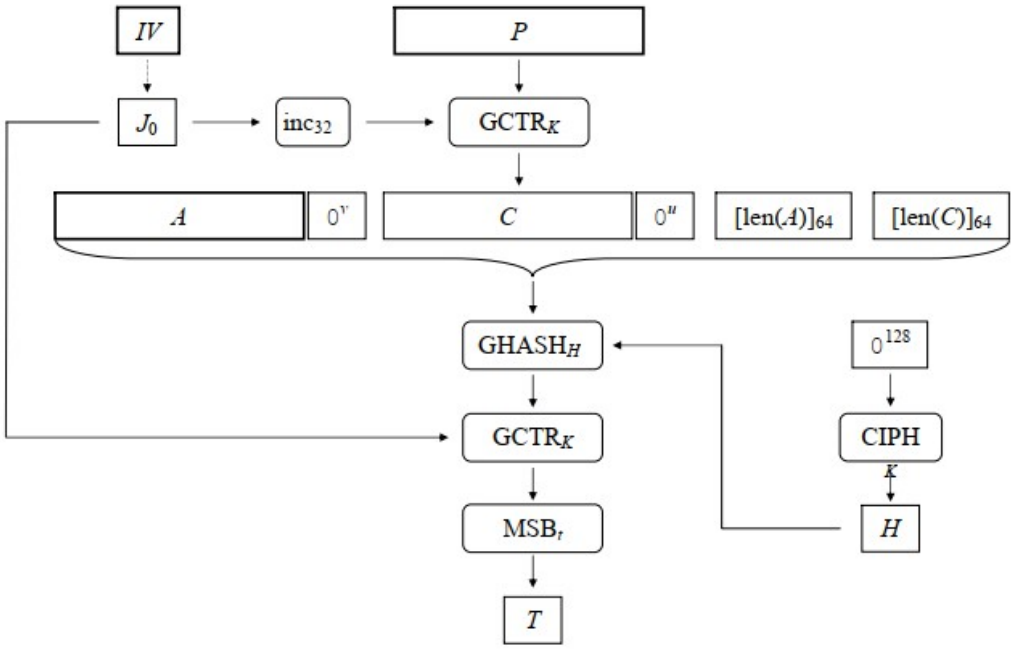

FIGURE 2: Authenticated Encryption function for AES-GCM [4]

- **Incremeting function** : the function is defined as:

$$inc_s(X) = MSB_{len(X)-s}(X)||[int(LSB_s(X)) + 1 mod 2^s]_s$$

For the above scheme, the function increments the right-most 32 (s) bits of the given bit string (defined as X in this case, a given integer represented in binary encoding, modulo 2^s)

- **GHASH function** - using this function and the encryption step the authentication tag is constructed. The input for it includes the bit string X and the hash subkey defined as block H. The hash key H is obtained using "zero block", meaning H = $E_k(0^{128})$ (string of zero bits encrypted with the standard block cipher used). Briefly, this function computes :

$$X_1 \cdot H_m \ \oplus X_2 \cdot H_{m-1} \oplus ... \oplus X_m \cdot H$$

where $X_1, X_2, ...X_m$ is the sequence of blocks that form X (X = $X_1||X_2||...||X_m$)

- **GCTR** - this function applies for a given block cipher with key K, to the bit string X of arbitrary length with an initial counter block (ICB). The input string is firstly partitioned and the initial counter block is consecutively transformed in subsequent counter blocks using the incrementing function illustrated before,the sequence starting with the initial block itself(ICB, CB_1, CB_2...CB_n). Applying the chosen block cipher on the counter blocks and the xor operation with the blocks from the bit string input(X_1, X_2, ... X_n) partial results are computed that will lead to the final output through concatenation[4].

2.2 Authenticated Encryption/Decryption Algorithms

Using the previous operations, the flow of the authenticated encryption and authenticated decryption algorithms is constructed. These algorithms can be applied using various block ciphers, they are not AES-bound, but are used with this specific cipher for this paper. The prerequisites of the authenticated encryption algorithm include the choice of the block cipher, the key and the supported lengths for the input and output. Additionally, an initialization vector, the plaintext and the authenticated additional data is provided.

As shown in figure 2 the algorithm starts with the hash subkey (H) computation (illustrated above in the definition of the GHASH function). Secondly, a pre-counter block defined as J_0 is computed using the initialization vector (nonce) with the following structure: $J_0 = IV||0^{31}||1$ when the size of the IV is 96 bits or applying GHASH function on IV padded with a custom number of "0" to obtain a result divisible with 128 and concatenated with 64 additional "0", and the length of the initialization vector represented in 64-bit format.

Once the pre-counter is calculated, it is used as input for the incrementing function. The result (initial counter block) and the plaintext are the parameters on which the GCTR function is called in order to obtain the ciphertext for this stage.

For the authentication-specific part, the remained components are included: the additional authenticated data and the ciphertext above obtained are each padded

if necessary with zeros and, alongside the representation in 64-bit format of their lengths, they are fed as input to the $GHASH_H$ function and then encrypted using the GCTR function using J_0. The output is processes using MSB function (truncates at the supported, initially defined length for the tag) resulting in the final form of the authentication tag.

The authenticated decryption algorithm follows similar reverse steps. The verification of the authentication tag may or may not precede the translation of the ciphertext back in plaintext, this step being the one that establishes if the data is tampered (signaling this with a fail message) or the result of the decryption is the expected, initial plaintext [4].

3 Improvement analysis

3.1 Design choices and considerations

Several improvements that have been identified during the analysis phase of AES-GCM algorithm will be presented in this section. The main areas that, from our perspective, leaves room for improvement without altering any of the algorithm steps but in the same time maintaining at least the same security level and same application flow, are both encryption and authentication stages.

3.1.1 Authentication process

During the authentication process, there are particular elements that can be changed in order to discourage attacks and guarantee the information protection. First of all, the authentication tag that results from applying the Galois/Counter Mode of operation has a great impact from this point of view.

It is considered that the strength of the entire authentication process is defined by the length of this particular authentication tag that is considered to be one of the most important security parameters of the algorithm. Default sizes for the length of this tag range from 128 to 96 bits, but there are cases when 64-bit or even 32-bit tags are used. The length of the authentication tag correlated with the message size are directly responsible for the number of calls on each of the encryption and

decryption subroutines; shorter tags leading to weakness from the security point of view and are not recommended to be used.Moreover due to the advance in technology, the now standard length for the authentication tag can become insufficient for the future computation power available, thus motivating the need for stronger security parameters.

Another characteristic of the authentication process is represented by the polynomial that defines the Galois field applied at this stage. The default polynomial is $x^{128} + x^7 + x^2 + x + 1$. The choice of GF($2^{128}$) is motivated by the definition for multiplication operation, defined as an operation on bit vectors, used during authentication. The properties of this field and the simplification provided (for example, translating addition of polynomials to bitwise xor operations) are further described in this paper[6], but other polynomials can be chosen, each of them having a certain impact on the application performance. However, for this paper, the focus is on the encryption stage, on the particularities and elements involved, prone to an advanced analysis and customization.

3.1.2 Encryption process

The objective of this section is to describe one of the elements that ensure a high level of security for the AES algorithm. As previously illustrated in this paper, there are several steps during encryption/decryption, each of them having specific functions attached. One of the core attached functions is *SubBytes*, a function that performs byte substitution in order to ensure non-linearity in applying the cipher. Every byte from the initial state is substituted using a substitution box.

The strength of block ciphers comes from this substitution box, usually known as S-Box, a 16X16 matrix where each row or column values are in [0, 15] ([0, F] - hexadecismal) range. Each byte of S-Box is mapped to its multiplicative inverse in GF(2^8), 00 being mapped into itself. After this step, the affine transformation over GF(2) is computed. The general ecuation for generating and S-Box matrix is:

$$Y := Ax \oplus c \; mod \; M$$

where A represents an affine transformation, x is the multiplicative inverse vector of the state matrix, c is the affine hexadecimal constant 63 and M is the default irre-

ducible polynomial used by AES, $x^8 + x^4 + x^3 + x + 1$. More information regarding S-Box generation and its properties can be found in the *"Analysis of development of dynamic S-Box Generation"* article. [1]

Taking into consideration all the above one can say that the improvements that can be brought to the encryption process of the AES-GCM algorithm, reside in the ability of changing the polynomial used and so generating a different S-Box for encrypting the plain text bytes.

4 Implementation

4.1 Overview

As stated above, the main purpose of this paper is to analyze different methods that can be used to enhance the AES-GCM. The principal idea proposed by us is replacing the default values used by the algorithm with custom values, in order to introduce a higher degree of randomness and so enhancing the algorithm security. In the previous section, two areas were identified, one is the encryption process and the other is the authentication process. However, in this paper only the customization of the encryption process will be presented.

Since the polynomial used for encrypting the plain text is placed in GF(2^8) there is a finite number of polynomials that are irreducible and can be used to generate S-Boxes. All these polynomials can be found below represented as hexadecimal numbers.

$$11B, 11D, 12B, 12D, 139, 13F, 14D, 15F, 163, 165, 169, 171, 177, 17B, 187,$$

$$18B, 18D, 19F, 1A3, 1A9, 1B1, 1BD, 1C3, 1CF, 1D7, 1DD, 1E7, 1F3, 1F5, 1F9$$

In addition to this, the algorithm uses an additive constant in order to ensure that the S-Box has no "fixed points" (S-Box[a]=a) and no “opposite fixed points” (S-Box[a]=a^{-1} - bitwise). These constants are specific to each of the polynomials mentioned above and have values that range from 00 to FF.

4.2 Detailed description

The proposal has been implemented using Python 3.6.9, the main reason being that this programming language offered us enough flexibility regarding the Galois Field specific operations (GF(2)) (such as multiplication, addition, division etc) in order to implement more freely and more naturally all the steps needed in order to generate S-boxes and Inverse S-boxes, that would later be used by the AES-GCM algorithm. Adjacent Python modules[4] have been employed which already offer support for calculating the inverse of a polynomial given another irreducible polynomial, by employing the Extended Euclidean Algorithm over GF(2). These modules are stated by the creator to be 10 times faster when it comes to the computation of the inverse than the official *scipy* default counterpart.

In this implementation, every polynomial used is represented as a *numpy* array, where the rightmost element represents the leading coefficient of the polynomial (rule imposed in order to use the adjacent Python modules needed to compute the modular inverses). In the following rows, a detailed description of each of the functions used inside the implementation of our proposal will be offered.

- *hex_to_binary_array(input)* is the function that takes a int value offered in a hexadecimal representation (for example instead of the value 15 the function should be fed 0xF) and returns the representation of the said value as an numpy array. This function is mainly used in order to convert values encoding polynomials to actual vectorial representations of the said polynomials.

- *binary_array_to_hex(input)* is the counterpart of the earlier mentioned function which does the reverse: it takes a numpy binary array value and converts it to a hexadecimal representation of the said value. One thing to be noted is that the representation that gets returned is of type *string*, in order to be on par with the representation of the int fed in it's counterpart.

[4]https://github.com/popcornell/pyGF2/tree/master/pyGF2

- *vector_xor(a, b)* is the function that as the name implies, takes two numpy binary vectors and applies the XOR operation between them (each element found in vector a on position i gets XORed with the element found in vector b on position i).

- *compute_inverse(to_invert, modulus)* if the function that receives as parameters the value to be inverted (which should be fed as an int in hexadecimal representation) and the encoding of the irreducible polynomial in GF(2^8) which should also be an int in hexadecimal representation. The function then returns the numpy binary vector of the modular inverse of to_invert, as well as a human readable hexadecimal representation of the said modular inverse.

- *multiply_rows(row1, row2)* is the function that takes two numpy binary vectors and applies the multiplication operation on them. *row1* represents the actual row of the circulant matrix used in order to compute the final value residing in an S-box and *row2* represents the numpy binary vector representation of the modular inverse. As such the multiplication rules are the ones present when it comes to multiplying a row with a column when referring to matrix operations. The only difference here is that every element of the two entities being multiplied are binary in nature, as such bit wise operations are used for addition and multiplication.

- *sbox_entrance(to_invert, modulus, constant)* is the function that takes a value to be inverted (which should be fed as an int in hexadecimal representation), an encoding of the irreducible polynomial in GF(2^8) which should also be an int in hexadecimal representation as well as a constant used in order to compute the S-box entry (which should also be fed as an int in hexadecimal representation). The function then amasses the other functions described earlier in this article and returns a hexadecimal representation of an element of the S-box, after computing the modular inverse of the index in the S-box (between 0x00 and 0xFF), multiplying said modular inverse with the circulant matrix presented below (in figure 3) and in the end adding the constant represented in numpy binary vector form.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

FIGURE 3: Circulant matrix employed

- *sbox_generator(modulus)* is the function that takes an encoding of the irreducible polynomial in GF(2^8) which should be an int in hexadecimal representation, and for each constant starting with 0x00 and ending with 0xFF computes pairs of S-boxes and Inverse S-boxes. The final result is a list of those pairs for which the constant used enforces the rule stated in the *Overview* subsection, stating that there should be no "fixed-points" in the resulting S-box and also no "opposite fixed-points" in the resulting Inverse S-box.

- *create_header_file(sboxes, modulus, f)* is the function that takes a list with the pairs of S-boxes and Inverse S-boxes generated using the *sbox_generator* function presented earlier, the encoding of the irreducible polynomial in GF(2^8) used to generate the pairs as well as a reference to a file with the name "sboxes.h" where the pairs should be written. The function then writes every S-box and Inverse S-box pair and encloses them in a *#if defined, #else if defined* preprocessor directive. As such, when testing the resulting pairs, the only aspect that must be taken care of is defining the following name: *SBOX_modulus_constant*, definition name which can also be found throughout the whole header file.

4.3 Evaluation

In order to evaluate and test the resulting S-box and Inverse S-box pairs, a modified version of the AES-GCM open source implementation provided here[5], was used. In the initial version of the AES-GCM implementation used, the S-box and Inverse S-box values were hard-coded to be 1 on 1 with the values obtained for the irreducible polynomial $11B$. Using our proposed implementation, we then generated the S-box and Inverse S-box values for the following irreducible polynomials: $11D$, $12B$. Then we replaced the S-box and Inverse S-box value with a few entries in the header file, and tested the whole encryption - decryption process. We noted that the plain text used for encryption was identical to the output of the decryption stage. In Appendix A are a few examples of entries inside the header file, which can be directly placed inside the AES-GCM implementation:

5 Conclusions and Future Work

In the previous sections, this paper introduced the idea of a more dynamic, customizable approach for the AES-GCM algorithm that is able to add an extra layer of security due to the fact that can change the default parameters at certain steps and, as a result, making it less susceptible to attacks that search for similarities between runs. We built this solution relying on the notion of increased randomness that can directly affect the overall process, more than leaving the choice of the used polynomials and constant parameters in encryption to the user. However, having the adequate possible replacements, one can continuously choose to replace these elements at a specific point in time, making it more difficult to identify parts of matching cipher texts during multiple runs.

The project can be further developed by applying the same method and studying the possibility of customization for the authentication tag related polynomials. This way, every security feature of AES-GCM algorithm would be fully controllable from the outside, reducing even more the risk of pattern identification during the encryption and authentication phases of the algorithm. In addition to this, the actual tag length might become insufficient as the computational power of the

[5]https://github.com/openluopworld/aes_gcm

machines is increasing continuously. This reason is providing another direction for future project development and remains the object of future work.

To conclude, we presented the AES-GCM algorithm, detailing its implementation steps, parameters and particularities for both encryption and authentication phases. We studied its vulnerabilities for different operation modes and provided a solution for overcoming some of the limitations identified during this process, that leaves an open path for future improvements in this area of interest.

References

[1] Mehar Chand Amandeep Singh, Praveen Agarwal. Analysis of development of dynamic s-box generation. 2017.

[2] Dobre Blazhevski, Adrijan Bozhinovski, Biljana Stojcevska, and Veno Pachovski. Modes of operation of the aes algorithm. In *The 10th Conference for Informatics and Information Technology (CIIT 2013)*, 2013.

[3] R. Doomun, J. Doma, and S. Tengur. Aes-cbc software execution optimization. In *2008 International Symposium on Information Technology*, volume 1, pages 1–8, 2008.

[4] Morris J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, Gaithersburg, MD, USA, 2007.

[5] Niels Ferguson. Authentication weaknesses in gcm, 2005.

[6] D. McGrew and J. Viega. The galois/counter mode of operation (gcm). 2005.

[7] David McGrew. *Galois Counter Mode*, pages 506–508. Springer US, Boston, MA, 2011.

A Appendix

$$Modulus : 11D, Constant : 13$$

```
#if defined(SBOX_11d_13)
static uint8_t SBOX[256] = { 0x19, 0x6, 0x2c, 0x3f, 0x83, 0x28, 0xa, 0x42, 0xee,
0xfc, 0x3b, 0x9f, 0x90, 0xb3, 0xb4, 0x25, 0x58, 0xf2, 0x51, 0xd7, 0xb2, 0xb1,
0x5a, 0x7f, 0x67, 0x1a, 0x4c, 0x96, 0x75, 0xb7, 0x7, 0x3c, 0xb9, 0x29, 0xec,
0x30, 0x3d, 0x6d, 0x7e, 0x50, 0xcc, 0xa0, 0x4d, 0x18, 0xb8, 0x4f, 0x2a, 0x87,
0x26, 0x71, 0x98, 0x40, 0x9, 0x70, 0xde, 0x95, 0x2f, 0xc4, 0xf4, 0x92, 0xac,
0x9d, 0x8b, 0xad, 0x49, 0x8a, 0xbb, 0xc3, 0x59, 0x64, 0x37, 0x65, 0xb, 0x6e,
0x23, 0x52, 0xaa, 0x60, 0xbd, 0x46, 0xf3, 0xdb, 0xc5, 0x12, 0x89, 0x2e, 0x99,
0x22, 0xc9, 0x4e, 0x88, 0x1c, 0x3a, 0xa3, 0x56, 0xc1, 0x86, 0x72, 0x97, 0xf6,
0x63, 0x2d, 0xb5, 0x45, 0x11, 0xd, 0x17, 0xa5, 0xfa, 0x36, 0x5f, 0x35, 0x2, 0xbf,
0xf7, 0x47, 0x55, 0x9a, 0x66, 0x2b, 0x79, 0x91, 0x5b, 0x5d, 0xea, 0xca, 0xf9,
0xc7, 0x31, 0x53, 0x6a, 0x73, 0x48, 0xd5, 0xce, 0x84, 0x39, 0x14, 0xa7, 0x61,
0xe, 0xda, 0x27, 0x20, 0x10, 0x5, 0xa2, 0x7b, 0xbe, 0x0, 0xbc, 0xab, 0xc0, 0x24,
0x1f, 0x1b, 0xf1, 0xfe, 0xc, 0x4b, 0x6c, 0xa8, 0xc2, 0x13, 0x77, 0x6f, 0x9c, 0x7a,
0xeb, 0xd2, 0x82, 0x8f, 0xe3, 0xe1, 0x3e, 0xd9, 0xcb, 0xd1, 0x8, 0xe4, 0x6b,
0x7d, 0x21, 0xd0, 0x32, 0x41, 0x44, 0x15, 0x4, 0xa1, 0xcf, 0xc6, 0xd6, 0xc8,
0x16, 0x68, 0x5e, 0x7c, 0x54, 0xdd, 0x9e, 0x1e, 0x3, 0xe9, 0xf5, 0x57, 0x8d,
0xcd, 0x1d, 0xd3, 0xa9, 0x1, 0xa4, 0xff, 0xfd, 0xe2, 0xe8, 0xed, 0x8e, 0x34,
0x80, 0xdf, 0xf, 0x78, 0x94, 0xae, 0xf0, 0x43, 0xd4, 0x9b, 0x8c, 0x5c, 0x85,
0xb0, 0x62, 0xf8, 0xa6, 0xe5, 0xba, 0xfb, 0x93, 0xef, 0xe7, 0x74, 0x38, 0x33,
0x81, 0xb6, 0xe0, 0xdc, 0x4a, 0xaf, 0x69, 0xe6, 0x76, 0xd8 };

static uint8_t INV_SBOX[256] = { 0x95, 0xd3, 0x70, 0xca, 0xbc, 0x91, 0x1,
0x1e, 0xb2, 0x34, 0x6, 0x48, 0x9e, 0x69, 0x8c, 0xde, 0x90, 0x68, 0x53, 0xa3,
0x89, 0xbb, 0xc2, 0x6a, 0x2b, 0x0, 0x19, 0x9b, 0x5b, 0xd0, 0xc9, 0x9a, 0x8f,
0xb6, 0x57, 0x4a, 0x99, 0xf, 0x30, 0x8e, 0x5, 0x21, 0x2e, 0x77, 0x2, 0x65, 0x55,
0x38, 0x23, 0x80, 0xb8, 0xf5, 0xdb, 0x6f, 0x6d, 0x46, 0xf4, 0x88, 0x5c, 0xa,
0x1f, 0x24, 0xae, 0x3, 0x33, 0xb9, 0x7, 0xe3, 0xba, 0x67, 0x4f, 0x73, 0x84, 0x40,
0xfa, 0x9f, 0x1a, 0x2a, 0x59, 0x2d, 0x27, 0x12, 0x4b, 0x81, 0xc6, 0x74, 0x5e,
0xcd, 0x10, 0x44, 0x16, 0x7a, 0xe7, 0x7b, 0xc4, 0x6e, 0x4d, 0x8b, 0xea, 0x64,
0x45, 0x47, 0x76, 0x18, 0xc3, 0xfc, 0x82, 0xb4, 0xa0, 0x25, 0x49, 0xa5, 0x35,
```

0x31, 0x61, 0x83, 0xf3, 0x1c, 0xfe, 0xa4, 0xdf, 0x78, 0xa7, 0x93, 0xc5, 0xb5, 0x26, 0x17, 0xdc, 0xf6, 0xaa, 0x4, 0x87, 0xe8, 0x60, 0x2f, 0x5a, 0x54, 0x41, 0x3e, 0xe6, 0xce, 0xda, 0xab, 0xc, 0x79, 0x3b, 0xf0, 0xe0, 0x37, 0x1b, 0x62, 0x32, 0x56, 0x75, 0xe5, 0xa6, 0x3d, 0xc8, 0xb, 0x29, 0xbd, 0x92, 0x5d, 0xd4, 0x6b, 0xec, 0x8a, 0xa1, 0xd2, 0x4c, 0x97, 0x3c, 0x3f, 0xe1, 0xfb, 0xe9, 0x15, 0x14, 0xd, 0xe, 0x66, 0xf7, 0x1d, 0x2c, 0x20, 0xee, 0x42, 0x96, 0x4e, 0x94, 0x71, 0x98, 0x5f, 0xa2, 0x43, 0x39, 0x52, 0xbf, 0x7f, 0xc1, 0x58, 0x7d, 0xb0, 0x28, 0xcf, 0x86, 0xbe, 0xb7, 0xb1, 0xa9, 0xd1, 0xe4, 0x85, 0xc0, 0x13, 0xff, 0xaf, 0x8d, 0x51, 0xf9, 0xc7, 0x36, 0xdd, 0xf8, 0xad, 0xd7, 0xac, 0xb3, 0xed, 0xfd, 0xf2, 0xd8, 0xcb, 0x7c, 0xa8, 0x22, 0xd9, 0x8, 0xf1, 0xe2, 0x9c, 0x11, 0x50, 0x3a, 0xcc, 0x63, 0x72, 0xeb, 0x7e, 0x6c, 0xef, 0x9, 0xd6, 0x9d, 0xd5 };

$$Modulus : 12B, Constant : 1$$

#if defined(SBOX_12b_1)
static uint8_t SBOX[256] = { 0x1, 0x1e, 0x1c, 0xe8, 0x1d, 0x45, 0xf5, 0x11, 0x9d, 0xed, 0xb1, 0x54, 0xe9, 0xf3, 0x9, 0xc2, 0x4f, 0xbf, 0xe5, 0xa3, 0x59, 0xd0, 0x39, 0x4c, 0x75, 0x37, 0x78, 0xb9, 0x5, 0x32, 0x72, 0x87, 0xb4, 0x60, 0xcc, 0x52, 0xe1, 0xb5, 0x50, 0xef, 0x2d, 0x74, 0x7b, 0x13, 0x8f, 0x96, 0x35, 0x88, 0x3b, 0x64, 0x1a, 0x6b, 0x2f, 0x83, 0x5d, 0xb6, 0x91, 0x3a, 0xa, 0xf1, 0x2a, 0x47, 0x42, 0x4e, 0x49, 0x27, 0x23, 0xba, 0xe7, 0x3c, 0xa8, 0xdf, 0x71, 0x2b, 0xc9, 0xb, 0xa9, 0x51, 0xe4, 0x7e, 0x17, 0x6a, 0xbb, 0xdd, 0xae, 0xd8, 0x8, 0x28, 0x46, 0x93, 0xca, 0x15, 0x1b, 0x22, 0x57, 0x80, 0x8e, 0xf4, 0xb3, 0x3d, 0x8c, 0xcf, 0xa6, 0x36, 0x16, 0xa4, 0xd2, 0x5f, 0xbd, 0xc5, 0x48, 0x92, 0xdb, 0x65, 0xe, 0x43, 0x84, 0xc0, 0x79, 0x90, 0x6, 0x94, 0xb0, 0x31, 0xa0, 0xf0, 0x34, 0x3e, 0x25, 0xeb, 0x12, 0x2e, 0x82, 0xe6, 0xdc, 0x4d, 0xe0, 0x77, 0x9f, 0x8d, 0xd5, 0xee, 0x6e, 0x21, 0xab, 0xfd, 0x86, 0xc1, 0xf7, 0x3f, 0x4, 0xac, 0x55, 0x6d, 0x29, 0x67, 0x61, 0x7c, 0xbe, 0xa2, 0x98, 0x4a, 0x26, 0xad, 0x5c, 0xea, 0x6f, 0x9c, 0x44, 0xf, 0x7f, 0xe2, 0x85, 0xec, 0x7, 0xa7, 0x30, 0x70, 0xda, 0xa1, 0x76, 0x68, 0x99, 0x5e, 0xc, 0x24, 0x2, 0x62, 0xb8, 0xd7, 0x53, 0xb2, 0xc6, 0x38, 0x69, 0xff, 0x58, 0x97, 0x1f, 0xd6, 0xc7, 0x4b, 0x66, 0x89, 0x40, 0xe3, 0x9a, 0xf6, 0x18, 0xc8, 0x41, 0xde, 0x7a, 0xd4, 0xbc, 0xce, 0xcd, 0xfc, 0x63, 0xfb, 0xa5, 0xf8, 0x5a, 0x7d, 0xfe, 0x81, 0x33, 0x19, 0x14, 0x0, 0x20, 0x3, 0xc3, 0xd3, 0x73, 0x6c, 0xaf, 0xd, 0x5b, 0x8a, 0x10, 0x95, 0xcb, 0xc4, 0xd9,

```
0x56, 0x8b, 0x2c, 0xd1, 0xfa, 0xf9, 0xf2, 0x9b, 0xb7, 0x9e, 0xaa };

static uint8_t INV_SBOX[256] = { 0xe5, 0x0, 0xba, 0xe7, 0x96, 0x1c, 0x78,
0xae, 0x56, 0xe, 0x3a, 0x4b, 0xb8, 0xed, 0x72, 0xa9, 0xf0, 0x7, 0x82, 0x2b,
0xe4, 0x5b, 0x68, 0x50, 0xd0, 0xe3, 0x32, 0x5c, 0x2, 0x4, 0x1, 0xc6, 0xe6,
0x8f, 0x5d, 0x42, 0xb9, 0x80, 0xa2, 0x41, 0x57, 0x9a, 0x3c, 0x49, 0xf7, 0x28,
0x83, 0x34, 0xb0, 0x7b, 0x1d, 0xe2, 0x7e, 0x2e, 0x67, 0x19, 0xc1, 0x16, 0x39,
0x30, 0x45, 0x63, 0x7f, 0x95, 0xcc, 0xd2, 0x3e, 0x73, 0xa8, 0x5, 0x58, 0x3d,
0x6e, 0x40, 0xa1, 0xc9, 0x17, 0x87, 0x3f, 0x10, 0x26, 0x4d, 0x23, 0xbe, 0xb,
0x98, 0xf5, 0x5e, 0xc4, 0x14, 0xde, 0xee, 0xa4, 0x36, 0xb7, 0x6b, 0x21, 0x9c,
0xbb, 0xda, 0x31, 0x71, 0xca, 0x9b, 0xb5, 0xc2, 0x51, 0x33, 0xeb, 0x99, 0x8e,
0xa6, 0xb1, 0x48, 0x1e, 0xea, 0x29, 0x18, 0xb4, 0x89, 0x1a, 0x76, 0xd4, 0x2a,
0x9d, 0xdf, 0x4f, 0xaa, 0x5f, 0xe1, 0x84, 0x35, 0x74, 0xac, 0x92, 0x1f, 0x2f,
0xcb, 0xef, 0xf6, 0x64, 0x8b, 0x60, 0x2c, 0x77, 0x38, 0x6f, 0x59, 0x79, 0xf1,
0x2d, 0xc5, 0xa0, 0xb6, 0xce, 0xfc, 0xa7, 0x8, 0xfe, 0x8a, 0x7c, 0xb3, 0x9f,
0x13, 0x69, 0xdc, 0x66, 0xaf, 0x46, 0x4c, 0xff, 0x90, 0x97, 0xa3, 0x54, 0xec,
0x7a, 0xa, 0xbf, 0x62, 0x20, 0x25, 0x37, 0xfd, 0xbc, 0x1b, 0x43, 0x52, 0xd6,
0x6c, 0x9e, 0x11, 0x75, 0x93, 0xf, 0xe8, 0xf3, 0x6d, 0xc0, 0xc8, 0xd1, 0x4a,
0x5a, 0xf2, 0x22, 0xd8, 0xd7, 0x65, 0x15, 0xf8, 0x6a, 0xe9, 0xd5, 0x8c, 0xc7,
0xbd, 0x55, 0xf4, 0xb2, 0x70, 0x86, 0x53, 0xd3, 0x47, 0x88, 0x24, 0xab, 0xcd,
0x4e, 0x12, 0x85, 0x44, 0x3, 0xc, 0xa5, 0x81, 0xad, 0x9, 0x8d, 0x27, 0x7d, 0x3b,
0xfb, 0xd, 0x61, 0x6, 0xcf, 0x94, 0xdd, 0xfa, 0xf9, 0xdb, 0xd9, 0x91, 0xe0, 0xc3 };
```