Criptografia AES

Matheus Oliveira, 21/1055343
Raquel Sousa, 20/0042912

October 2023

1 Introduction

A cifra AES (Advanced Encryption Standard) é um algoritmo de criptografia amplamente adotado e utilizado para proteger informações confidenciais em comunicações e armazenamento de dados. Desenvolvido para substituir o Padrão de Cifra de Dados (Data Encryption Standard - DES), o AES é uma cifra simétrica, o que significa que a mesma chave é usada tanto para criptografar quanto para descriptografar os dados. Nesse trabalho, é adotado o método de lookup tables para agilizar o código.

O AES opera em blocos de dados fixos e oferece três tamanhos de chave diferentes: AES-128, AES-192 e AES-256. Esses números representam o comprimento da chave em bits, indicando a complexidade da cifra e o nível de segurança. No desenvolvimento desse trabalho, foi utilizado o AES-128. Ele opera em uma matriz de bytes 4x4, chamada de "estado", que é manipulada em uma série de rodadas, tipicamente, o modo AES-128 adota 10 rodadas, porém, nessa implementação o número de rodadas pode ser escolhido. Cada rodada envolve 4 etapas executadas em ordem: substituição de bytes, deslocamento de linhas (shift rows), permutação de colunas (mix columns) e a combinação de chave da rodada (add round key). O processo de criptografia envolve uma rodada inicial, em que ocorre apenas a etapa de adição da chave de rodada, seguida das rodadas estabelecidas, e então uma rodada final, na qual não ocorre a etapa mix columns. Além disso, é necessário realizar a expansão de chave, já que cada rodada utiliza uma chave diferente gerada a partir da inicial.

2 Expansão de chave

A expansao de chave é responsável por gerar as chaves de rodadas, ou seja, transformar a chave inicial de 128 bits em um conjunto de chaves suficientes para que cada rodada use uma chave diferente. O núcleo dessa expansão pode ser separada em 3 processos, rotação para a esquerda, um look-up na S-box, que é a mesma utilizada na etapa de substituição de bytes, a última etapa é adicionado 2^{i-1} ao primeiro byte do conjunto, onde i é o numero da iteração atual, essa etapa tambem pode ser realizada por uma look-up table, denominada Rcon.

A expansão em si realiza a geração em grupos de 4 bytes, usando os ultimos 4 bytes gerados como referência para geração de novos bytes, por fim, é realizado um xor com os 4 bytes de mesma posição da ultima chave gerada, ou seja, na posição atual - 16, ate que o tamanho da chave seja suficiente.

```
void expand_key_core(u8 *in, u8 current_round){
    u32 *q = (u32 *) in;
    *q = (*q >> 8) | (*q  << 24);
    for (int i = 0; i < 4; ++i) {
        in[i] = s_box[in[i]];
    }
    in[0] ^= rcon[current_round];
}
void expand_key(u8 *base_key, u8 *expanded_key, int rounds){
    for (int i = 0; i < 16; ++i) {
        expanded_key[i] = base_key[i];
    }
    int bytes_gen = 16, total_bytes = 16*(rounds+1);
    int rcon_iter = 1;
    u8 temp[4];
    while (bytes_gen < total_bytes){

        for (int i = 0; i < 4; ++i) {
            temp[i] = expanded_key[i + bytes_gen - 4];
        }
        if (bytes_gen % 16 == 0 ){
            expand_key_core( in: temp,  current_round: rcon_iter++);
        }
        for (int i = 0; i < 4; ++i) {
            expanded_key[bytes_gen] = expanded_key[bytes_gen - 16] ^ temp[i];
            bytes_gen++;
        }
    }
}
```

3 Cifração

3.1 Add Round Key

A adição da chave de rodada é realizada por uma adição no campo de Galois, que no caso do $GF(2^n)$ é equivalente a operação $estado \oplus chave$

```
void add_round_key(u8 *state, u8 *round_key){
    for (int i = 0; i < 16; ++i) {
        state[i] ^= round_key[i];
    }
}
```

3.2 Substituição de bytes

A sustituicao de bytes ocorre por meio da tabela S-box, que fornece uma transformação não linear e não trivial dos bytes de entrada.

```
void sub_bytes(u8 *aes_array){
    for (int i = 0; i < 16; ++i) {
        aes_array[i] = s_box[aes_array[i]];
    }
}
```

3.3 Deslocamento de Colunas

Nessa etapa as linhas da matriz são deslocadas para a esquerda, a primeira linha não é deslocada, a segunda é deslocada 1 byte para a esquerda, a terceira 2 bytes e a quarta 3 bytes.

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} \xrightarrow{\text{Shift Rows}} \begin{bmatrix} 0 & 4 & 8 & 12 \\ 5 & 9 & 13 & 1 \\ 10 & 14 & 2 & 6 \\ 15 & 3 & 7 & 11 \end{bmatrix} \tag{1}$$

Demonstrando em código, seria:

```
void shift_rows(u8 aes_matrix[4][4]){
    u8 temp_row[4];
    for (int i = 1; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            temp_row[j] = aes_matrix[i][(j + i) % 4];
        }
        for (int j = 0; j < 4; ++j) {
            aes_matrix[i][j] = temp_row[j];
        }
    }
}
```

3.4 Permutação das colunas

A operação MixColumns realizada pelo cifrador Rijndael, juntamente com a etapa ShiftRows, é a principal fonte de difusão no Rijndael. Cada coluna é tratada como um polinômio de quatro termos, $b(x) = b_3x^3+b_2x^2+b_1x+b_0$, que são elementos dentro do campo GF(2^8). Os coeficientes dos polinômios são elementos dentro do subcampo primo GF(2).

Cada coluna é multiplicada por um polinômio fixo $a(x) = 3x^3 + x^2 + x + 2$ módulo $x^4 + 1$.

Essa operação pode ser realizada multiplicando um vetor de coordenadas de quatro números no campo de Galois de Rijndael pela seguinte matriz MDS circulante:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Assim, usamos duas lookup tables para simplificar essa equação, com valores previamente calculados, para multiplicação por 2 e por 3:

Exemplificando em código, ficaria:

```
void mix_columns(u8 aes_matrix[4][4]){
   u8 temp_matrix[4][4];
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            u8 result = 0;
            for (int k = 0; k < 4; ++k) {
                u8 val = aes_matrix[k][j];
                u8 fac = mix[i][k];
                result ^= (fac == 2) ? mul2[val] : (fac == 3) ? mul3[val] : val;
            }
            temp_matrix[i][j] = result;
        }
    }
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            aes_matrix[i][j] = temp_matrix[i][j];
        }
    }

}
```

3.5 Encryption

Juntando essas funções, já podemos realizar a criptografia:

```
void AES_encrypt(char *plaintext, u8 *expanded_key, int rounds){
    u8 aes_matrix[4][4];
    u8 aes_array[16];
    for (int i = 0; i < 16; ++i) aes_array[i] = plaintext[i];
    add_round_key( state: aes_array,  round_key: expanded_key);
    for (int i = 0; i < rounds - 1; ++i) {
        sub_bytes(aes_array);
        for (int j = 0; j < 4; ++j) {
            for (int k = 0; k < 4; ++k) {
                aes_matrix[k][j] = aes_array[(j * 4) + k];
            }
        }
        shift_rows(aes_matrix);
        mix_columns(aes_matrix);
        for (int j = 0; j < 4; ++j) {
            for (int k = 0; k < 4; ++k) {
                aes_array[(j * 4) + k] = aes_matrix[k][j];
            }
        }
        add_round_key( state: aes_array,  round_key: expanded_key + (16 * (i + 1)));
    }
    sub_bytes(aes_array);
    for (int j = 0; j < 4; ++j) {
        for (int k = 0; k < 4; ++k) {
            aes_matrix[k][j] = aes_array[(j * 4) + k];
        }
    } // array to matrix
    shift_rows(aes_matrix);
    for (int j = 0; j < 4; ++j) {
        for (int k = 0; k < 4; ++k) {
            aes_array[(j * 4) + k] = aes_matrix[k][j];
        }
    } // matrix to array
    add_round_key( state: aes_array,  round_key: expanded_key + 16*rounds);
    for (int i = 0; i < 16; ++i)  plaintext[i] = (char)aes_array[i];
}
```

4 Decifração

Para decifrar a cifra gerada pelo AES, precisamos de um método reverso para cada etapa do proceso, ou seja, uma S-box reversa, um shift para a direita, e uma função inversa a permutação.

4.1 Substituição

Assim como na cifração, essa etapa é realizada utilizando apenas uma look-up table.

```
void sub_bytes_dec(u8 *aes_array){
    for (int i = 0; i < 16; ++i) {
        aes_array[i] = reverse_s_box[aes_array[i]];
    }
}
```

4.2 Deslocamento de colunas

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} \xrightarrow{\text{Shift Rows}} \begin{bmatrix} 0 & 4 & 8 & 12 \\ 13 & 1 & 5 & 9 \\ 10 & 14 & 2 & 6 \\ 7 & 11 & 15 & 3 \end{bmatrix} \tag{2}$$

Assim como na cifração, as linhas são deslocadas para a direita.

```
void shift_rows_dec(u8 aes_matrix[4][4]){
    u8 temp_row[4];
    for (int i = 1; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            temp_row[(j + i) % 4] = aes_matrix[i][j];
        }
        for (int j = 0; j < 4; ++j) {
            aes_matrix[i][j] = temp_row[j];
        }
    }
}
```

4.3 Permutação de colunas

Durante a cifração é usado o polinomio $a(x) = 3x^3 + x^2 + x + 2$ módulo $x^4 + 1$. O inverso desse polinômio é $a^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$.

Ou seja, realizamos a multiplicacao pela matriz:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

```
void mix_columns_dec(u8 aes_matrix[4][4]){
    u8 temp_matrix[4][4];
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            u8 result = 0;
            for (int k = 0; k < 4; ++k) {
                u8 val = aes_matrix[k][j];
                u8 fac = mix_dec[i][k];
                switch (fac) {
                    case 9:
                        result ^= mul9[val];
                        break;
                    case 11:
                        result ^= mul11[val];
                        break;
                    case 13:
                        result ^= mul13[val];
                        break;
                    case 14:
                        result ^= mul14[val];
                        break;
                    default:
                        break;
                }
            }
            temp_matrix[i][j] = result;
        }
    }
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            aes_matrix[i][j] = temp_matrix[i][j];
        }
    }
}
```

4.4 Decryption

Tendo essas funções podemos decifrar a mensagem com o código:

```
void AES_decrypt(char *ciphertext, u8 *expanded_key, int rounds){
    u8 aes_matrix[4][4];
    u8 aes_array[16];
    for (int i = 0; i < 16; ++i) aes_array[i] = ciphertext[i];
    add_round_key( state: aes_array,  round_key: expanded_key + 16*rounds);
    for (int j = 0; j < 4; ++j) {
        for (int k = 0; k < 4; ++k) {
            aes_matrix[k][j] = aes_array[(j * 4) + k];
        }
    } // array to matrix

    for (int i = (rounds -1) ; i > 0; i--) {
        shift_rows_dec(aes_matrix);
        for (int j = 0; j < 4; ++j) {
            for (int k = 0; k < 4; ++k) {
                aes_array[(j * 4) + k] = aes_matrix[k][j];
            }
        } // matrix to array
        sub_bytes_dec(aes_array);
        add_round_key( state: aes_array,  round_key: expanded_key +  (16 * i));
        for (int j = 0; j < 4; ++j) {
            for (int k = 0; k < 4; ++k) {
                aes_matrix[k][j] = aes_array[(j * 4) + k];
            }
        } // array to matrix
        mix_columns_dec(aes_matrix);
    }
    shift_rows_dec(aes_matrix);
    for (int j = 0; j < 4; ++j) {
        for (int k = 0; k < 4; ++k) {
            aes_array[(j * 4) + k] = aes_matrix[k][j];
        }
    } // matrix to array
    sub_bytes_dec(aes_array);
    add_round_key( state: aes_array,  round_key: expanded_key);
    for (int i = 0; i < 16; ++i) ciphertext[i] = aes_array[i];
}
```

5 Modo CTR

O modo CTR é uma implementação de codificação baseada em contador. A ideia desse modo e evitar que um mesmo plaintext gere um mesmo ciphertext, para isso, e utilizado um vetor de inicializacao, que idealmente so e utilizado uma vez (nonce), no nosso caso, de 96 bits, que e concatenado a um contador de 32 bits, que e incrementado a cada bloco. Entao esse vetor e submetido ao AES convencional, e posteriormente e realizado a operacao $V \bigoplus P$, que efetivamente gera o ciphertext da mensagem.

5.1 Vetor de inicialização

Para implementação do algoritmo, primeiro é necessário uma forma de criar o vetor, isso é feito pela função:

```
void iv_gen(u8* iv_vec){
    srand( seed: time( timer: NULL));
    for (int i = 0; i < 12; ++i) iv_vec[i] = rand() % 256;
}
```

5.2 Encryption

Tendo isso, o algoritmo base do modo CTR já implementado, as mudanças são implementadas pela função:

```
void CTR_AES_encrypt(char *plaintext, u8 *expanded_key, int rounds, u8* init_vector, int counter){
    u8 full_vec[16];
    for (int i = 0; i < 12; ++i) full_vec[i] = init_vector[i];
    for (int i = 15; i > 11; i--) {
        full_vec[i] = (u8)(counter & 0xFF);
        counter >>= 8;
    }
    AES_encrypt( plaintext: full_vec, expanded_key, rounds);
    for (int i = 0; i < 16; ++i) {
        plaintext[i] ^=full_vec[i];
    }
}
```

5.3 Decryption

A decifração no modo CTR é mais simples que o modo convencional, já que ela é feita apenas repetindo a cifração do vetor, e ao realizar novamente a operacao $V \bigoplus C$ obtemos a mensagem inicial.

É implementado da seguinte forma:

```
void CTR_AES_decrypt(char *ciphertext, u8 *expanded_key, int rounds, u8* init_vector, int counter){
    u8 full_vec[16];
    for (int i = 0; i < 12; ++i) {
        full_vec[i] = init_vector[i];
    }
    for (int i = 15; i > 11; i--) {
        full_vec[i] = (u8)(counter & 0xFF); // Copia o byte menos significativo do contador
        counter >>= 8; // Desloca o contador para a direita em 8 bits
    }
    AES_encrypt( plaintext: full_vec, expanded_key, rounds);
    for (int i = 0; i < 16; ++i) {
        ciphertext[i] ^=full_vec[i];
    }
}
```

6 Arquivos

Com as funções do modo CTR implementadas, podemos aplicá-las na critografia e descriptografia de arquivos, usando as seguintes funções:

```
void encrypt_file(char* source_file_path, char* dest_file_path, u8* key, int rounds){
    u8 init_vector[12];
    iv_gen( iv_vec: init_vector);
    FILE *source_file = fopen( filename: source_file_path,  modes: "rb");
    FILE *dest_file = fopen( filename: dest_file_path,  modes: "wb");
    u8 content_buffer[16];
    int counter = 0;
    int quantos_bytes = 0;
    while (true){
        size_t bytes_read = fread( ptr: content_buffer,  size: 1,  n: sizeof(content_buffer),  stream: source_file);
        quantos_bytes += bytes_read;
        if (bytes_read != 16){
            for (int i = (int)bytes_read; i < 16; ++i) {
                content_buffer[i] = 0;
            }
        }
        CTR_AES_encrypt( plaintext: content_buffer,  expanded_key: key, rounds, init_vector, counter++);
        for (int i = 0; i < 16; ++i) {
            content_buffer[i] = content_buffer[i] == 0x00 ? 0x20 : content_buffer[i];
        }
        fwrite( ptr: content_buffer,  size: 1,  n: 16,  s: dest_file);
        if (feof( stream: source_file)){
            fwrite( ptr: init_vector,  size: 1,  n: 12,  s: dest_file);
            break;
        }
    }
    fclose( stream: source_file);
    fclose( stream: dest_file);
}
```

```
void decrypt_file(char* source_file_path, char* dest_file_path, u8* key, int rounds){

    FILE *source_file = fopen( filename: source_file_path,  modes: "rb");
    FILE *dest_file = fopen( filename: dest_file_path,  modes: "wb");

    u8 content_buffer[16];
    u8 init_vector[12];
    // vai para os ultimos 12 bytes do arquivo onde fica o vetor
    fseek( stream: source_file,  off: -12,  whence: SEEK_END);
    fread( ptr: init_vector,  size: 1,  n: sizeof (init_vector),  stream: source_file);
    //volta para o comeco do arquivo
    fseek( stream: source_file,  off: 0,  whence: SEEK_SET);

    int counter = 0;
    int total_bytes = 0;

    while (true){
        size_t bytes_read = fread( ptr: content_buffer,  size: 1,  n: sizeof(content_buffer),  stream: source_file);

        if (bytes_read != 16) break;


        total_bytes += bytes_read;


        CTR_AES_decrypt( ciphertext: content_buffer,  expanded_key: key, rounds, init_vector, counter++);

        for (int i = 0; i < 16; ++i) {
            content_buffer[i] = content_buffer[i] == 0x00 ? 0x20 : content_buffer[i];
        }
        fwrite( ptr: content_buffer,  size: 1,  n: 16,  s: dest_file);
        if (feof( stream: source_file)) break;
    }
    fclose( stream: source_file);
    fclose( stream: dest_file);
```

Essas funções recebem como argumento o path do arquivo de origem e de onde devem escrever o novo arquivo, seja criptografado ou o descriptografado, a chave a ser usada é o numero de rodadas durante a criptorgafia.

7 Cifração autenticada GCM

O modo de operação GCM (Galois Counter Mode) utiliza um hash em $GF(2^{128})$ para fornecer criptografia autenticada. A ideia nesta implementação é gerar uma tag de autenticação que permite verificar a alteração na mensagem. Esse modo é composto por duas etapas: a criptografia da mensagem usando o modo CTR juntamente com a criação de uma tag de autenticação, e a verificação de uma mensagem criptografada. Durante a decifração da mensagem, a tag é recalculada para verificar se houve alterações no arquivo.

7.1 Multiplicação no $GF(2^{128})$

Antes de implementar o GCM, é necessário definir como ocorre a multiplicação no $GF(2^128)$. Para descrever a multiplicação, damos o primeiro passo de como multiplicar um elemento do campo X pelo elemento P definido por:

$$P_i = \begin{cases} 1 & \text{para } i = 1 \\ 0 & \text{caso contrário} \end{cases}$$

O processo de multiplicação é explicado a seguir. O elemento P corresponde ao polinômio α. A multiplicação de um polinômio por α é simples: ela corresponde a um deslocamento de índices:

$$\alpha \cdot \left(x_0 + x_1\alpha^1 + x_2\alpha^2 + \ldots + x_{127}\alpha^{127}\right) = x_0\alpha + x_1\alpha^2 + x_2\alpha^3 + \ldots + x_{127}\alpha^{128}$$

Se $x_{127} = 0$, o produto é um polinômio de grau 127. Caso contrário, devemos dividir o resultado pelo polinômio de campo f para encontrar o resto. Para encontrar o resto de um polinômio $\alpha^{128} + a$, onde $a = a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{127}\alpha^{127}$ é um polinômio de grau 127, precisamos encontrar polinômios q e r, de modo que $\alpha^{128} + a = q \cdot f + r$, onde o resto r tem grau 127. Podemos resolver essa equação para r quando $q = 1$:

$$r = \alpha^{128} + a - f = a + 1 + \alpha + \alpha^2 + \alpha^7$$

O termo mais alto de f é cancelado (uma vez que a adição é realizada sobre $GF(2)$), e o efeito líquido é apenas adicionar os termos mais baixos de f a a. Para calcular $Y = X \cdot P$, combinamos as duas etapas de deslocamento de coeficientes e adição dos termos mais baixos de f se o termo mais alto de X for igual a um. Em operações de bits, isso pode ser expresso como:

if $X_{127} = 0$, then
$Y \leftarrow$ rightshift(X)
else
$Y \leftarrow$ rightshift$(X) \oplus R$
end if

Onde R é o elemento cujos oito bits mais à esquerda são 11100001, e os 120 bits mais à direita são todos zeros.

Para multiplicar dois elementos arbitrários do campo X e Y, podemos expressar Y em termos de P, e então usar o método descrito acima. Esse método é utilizado no seguinte algoritmo, que recebe X e Y como entradas e retorna seu produto.

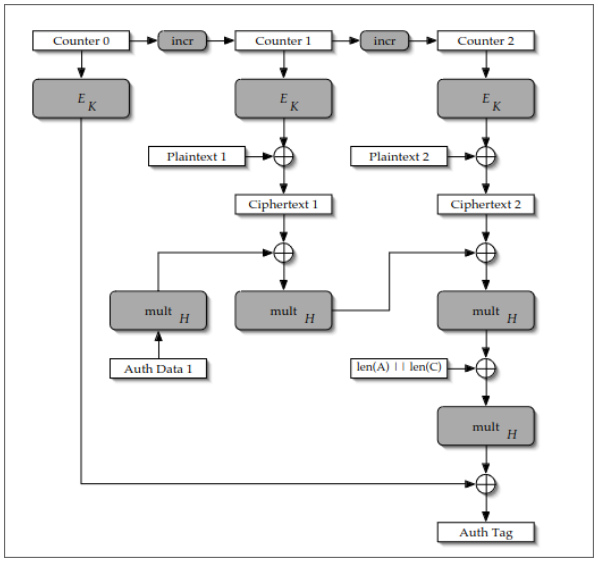
$Z \leftarrow 0$, $V \leftarrow X$ for $i = 0$ to 127 do
if $Y_i = 1$ then
$Z \leftarrow Z \oplus V$
end if
$V \leftarrow V \cdot P$
end for
retorna Z

Neste algoritmo, V percorre os valores de X, $X \cdot P$, $X \cdot P^2$, ..., e as potências de P correspondem às potências de α, modulo o polinômio de campo f. Este método é idêntico ao Algoritmo 1, mas é definido em termos de elementos do campo em vez de operações de bits.

```
void GF_128(const u8* x, const u8* Y, u8* result){
    u128 Z, V = 0,y = 0,mask, r;
    Z = 0;

    for (int i = 0; i < 16; i++) {
        V |= (u128) x[i] << (8 * (15 - i));
    }

    for (int i = 0; i < 16; i++) {
        y |= (u128) Y[i] << (8 * (15 - i));
    }

    r = 0;
    r |= (u128)0xe1<<120;

    for (int i = 127; i >=0 ; --i) {
        mask = ((u128)1)<<i;
        u8 bit_y = (y & mask)? 1 : 0;
        if (bit_y == 1) Z ^= V;

        if(V & 0x01) V = (V>>1) ^ r;
        else V >>= 1;
    }
    for (int i = 0; i < 16; i++) {
        result[i] = (u8)(Z >> (8 * (15 - i)));
    }
}
```

7.2 Encryption

Com a multiplicacao implementada, o algoritmo segue o fluxo:



sendo $len(A)$ a quantidade de bytes total do auth Data, que no caso dessa implementacao sera sempre 0, nao usamos auth data, e $len(C)$ sao os bytes totais da mensagem. Os bytes sao salvos no arquivo no formato:

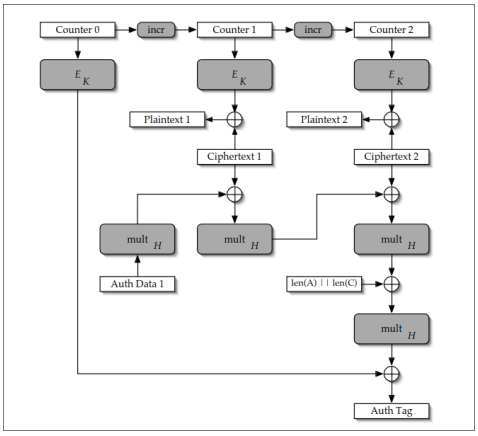
IV	Ciphertext	Tag

Isso e feito pela função:

```
void GMAC_encrypt_file(char* source_file_path, char* dest_file_path, u8* expanded_key, int rounds){
    int counter = 2;
    u8 auth_data[16] = {0};
    u8 init_vector[12];
    iv_gen( iv_vec: init_vector);
    u8 tag[16];
    u8 y0[16] = {0};
    for (int i = 0; i <12 ; ++i) y0[i] = init_vector[i];
    y0[15] = 1;
    u8 h[16] ={0}; // usado na mult
    AES_encrypt( plaintext: h, expanded_key, rounds); // inicia H = E(K,0)
    AES_encrypt( plaintext: y0, expanded_key, rounds);// y0 = E(K,Y0)
    GF_128( X: auth_data, Y: h, result: tag); // inicia tag (nesse caso hx0 = 0)
    FILE *source_file = fopen( filename: source_file_path, modes: "rb");
    FILE *dest_file = fopen( filename: dest_file_path, modes: "wb");
    u8 content_buffer[16];
    u64 len_c = 0;
    fwrite( ptr: init_vector, size: 1, n: 12, s: dest_file);
    while (true){
        size_t bytes_read = fread( ptr: content_buffer, size: 1, n: sizeof(content_buffer), stream: source_file);
        if (bytes_read != 16){
            if(bytes_read == 0) break;
            for (int i = (int)bytes_read; i < 16; ++i) {
                content_buffer[i] = 0;
            }
        }
        len_c += 128; // a cada iteracao + 16 * 8
        CTR_AES_encrypt( plaintext: content_buffer, expanded_key, rounds, init_vector, counter++);
        for (int i = 0; i < 16; ++i) {
            tag[i] ^= content_buffer[i];
        } // atualiza tag = tag xor cipher
        GF_128( X: tag, Y: h, result: tag);
        fwrite( ptr: content_buffer, size: 1, n: 16, s: dest_file);
        if (feof( stream: source_file)) break;
    }
    u8 lena_lenc[16] = {0};
    for (int i = 15; i > 11; --i) {
        lena_lenc[i] = (u8)(len_c & 0xff);
        len_c >>=8;
    }
    for (int i = 0; i < 16; ++i) tag[i] ^= lena_lenc[i]; // tag xor len A || len C
    GF_128( X: tag, Y: h, result: tag); //  ultimo Mul H
    for (int i = 0; i < 16; ++i) tag[i] ^= y0[i]; // tag = tag xor Y0
    fwrite( ptr: tag, size: 1, n: 16, s: dest_file);
    fclose( stream: source_file);
    fclose( stream: dest_file);
}
```

7.3 Verify

Nessa etapa, além de decifrar a mensagem, também é recalculada a tag, para que no final seja verificada a integridade da mensagem.



Para isso, não necessário nenhuma nova função, e é implementado pelo código:

```
void GMAC_verify(char* source_file_path, char* dest_file_path, u8* expanded_key, int rounds){
    u8 auth_data[16] = {0}; // n tem auth
    u8 init_vector[12];
    u8 tag[16] = {0};
    u8 y0[16] = {0};
    int counter = 2;
    u8 h[16] ={0};
    u8 expected_tag[16];
    GF_128( x: auth_data, Y: h, result: tag);
    FILE *source_file = fopen( filename: source_file_path, modes: "rb");
    FILE *dest_file = fopen( filename: dest_file_path, modes: "wb");
    fseek( stream: source_file, off: -16, whence: SEEK_END);
    int tamanho = ftell( stream: source_file) - 12;
    fread( ptr: expected_tag, size: 1, n: sizeof(expected_tag), stream: source_file);
    fseek( stream: source_file, off: 0, whence: SEEK_SET);
    fread( ptr: init_vector, size: 1, n: sizeof(init_vector), stream: source_file);
    for (int i = 0; i <12 ; ++i) y0[i] = init_vector[i];
    y0[15] = 1;
    AES_encrypt( plaintext: y0, expanded_key, rounds);// y0 = E(K,Y0)
    AES_encrypt( plaintext: h, expanded_key, rounds); // H = E(K,0)
    u8 content_buffer[16];
    u64 len_c = 0;
    len_c = tamanho * 8;
    while (true){
        size_t bytes_read = fread( ptr: content_buffer, size: 1, n: sizeof(content_buffer), stream: source_file);
        tamanho -=16;
        if (bytes_read != 16){
            for (int i = (int)bytes_read; i < 16; ++i) {
                content_buffer[i] = 0;
            }
        }
        for (int i = 0; i < 16; ++i) {
            tag[i] ^= content_buffer[i];
        } // atualiza tag = tag xor cipher
        GF_128( x: tag, Y: h, result: tag); //mulh
        CTR_AES_decrypt( ciphertext: content_buffer, expanded_key, rounds, init_vector, counter++);
        for (int i = 0; i < 16; ++i) content_buffer[i] = content_buffer[i] == 0 ? 0x20 : content_buffer[i];
        fwrite( ptr: content_buffer, size: 1, n: sizeof(content_buffer), s: dest_file);
        if (tamanho <= 0) break;
    }
    u8 lena_lenc[16] = {0};
    for (int i = 15; i > 11; --i) {
        lena_lenc[i] = (u8)(len_c & 0xff);
        len_c >>=8;
    }
    for (int i = 0; i < 16; ++i) tag[i] ^= lena_lenc[i]; // tag xor len A || len C
    GF_128( x: tag, Y: h, result: tag);
    for (int i = 0; i < 16; ++i) tag[i] ^= y0[i]; // tag = tag xor Y0
    int flag = 1;
    for (int i = 0; i < 16; ++i) {
        if (tag[i] != expected_tag[i]) flag = 0;
    }
    fclose( stream: source_file);
    fclose( stream: dest_file);
    //FAIL
    if (flag == 0){
        printf( format: "Falha de autenticacao\n");
        remove( filename: dest_file_path);
    }
}
```