Segurança Computacional
Trabalho 02

Eduardo Pereira de Sousa - 231018937
Luca Heringer Megiorin - 231003390
Universidade de Brasília

16 de Junho, 2025

1 Introdução

O AES (*Advanced Encryption Standard*) é uma cifra simétrica desenvolvida por Joan Daemen e Vincent Rijmen para substituir o DES. Tendo uma chave de no mínimo 128 bits, garantia uma criptografia segura e eficiente, e, em razão disso, o algoritmo foi escolhido pelo NIST dentre os outros competidores e publicado em 2001, com a intenção de eventualmente substituir o 3DES (capítulo 5 [1]). A partir de 2023, o algoritmo 3DES foi considerado obsoleto e proibido pelo NIST [2], tornando o AES o algoritmo principal recomendado pela organização.

Neste documento, serão abordados os processos presentes no algoritmo S-AES (seção 2), bem como o modo de operação *Eletronic Codebook* (ECB) aplicado a ele (seção 3) e outros modos de operação aplicados ao algoritmo AES original (seção 4). Os códigos em C e Python desenvolvidos estão disponíveis no GitHub (vide o apêndice A).

2 Simplified Advanced Encryption Standard

O S-AES (*Simplified Advanced Encryption Standard*) foi criado em 2003 pelo professor Edward Schaefer e Mohammad Musa como uma ferramenta educacional para facilitar o entendimento do AES [3]. As principais diferenças são os parâmetros reduzidos — o S-AES recebe um bloco e uma chave de 16 bits, enquanto que o AES espera um bloco de 128 bits e chave no mínimo 128 bits — e a quantidade de rodadas (o AES requer a partir de 10 rodadas, dependendo do tamanho da chave, enquanto que o S-AES tem somente 2 rodadas). De resto, a lógica e estrutura geral permanecem as mesmas, com pequenas alterações no S-AES, justamente para facilitar a implementação e aprendizado, mas mantendo a ideia, de modo que os estudantes possam entender o funcionamento do AES com mais tranquilidade.

2.1 Expansão das Chaves (KeyExpansion)

A expansão de chaves do S-AES envolve a geração de 3 subchaves (sendo a primeira a própria chave original) de 16 bits. O seu algoritmo consiste em separar a chave em duas metades ($K_1 = [w_0, w_1]$), realizando um XOR entre a parte mais significativa (w_0) e o resultado da aplicação de uma função g à parte menos significativa (w_1), para obter a parte mais significativa da segunda subchave (w_2). Usando esse resultado para fazer um XOR com a metade menos significativa da chave (w_1) produz a parte menos significativa desta subchave (w_3). Após isso, é usada a segunda chave ($K_1 = [w_2, w_3]$) nesses mesmos passos ao invés da chave original para gerar a terceira subchave ($K_2 = [w_4, w_5]$). Dessa forma:

$$w_2 = w_0 \oplus g(w_1), \quad w_3 = w_2 \oplus w_1, \quad w_4 = w_2 \oplus g(w_3) \quad \text{e} \quad w_5 = w_4 \oplus w_3$$

A função g consiste em dividir a palavra de 8 bits que recebe em duas, trocá-las de lugar (o equivalente a rotacionar de forma circular a para a esquerda 4 bits da palavra), aplicar SubNibbles a cada uma dessas metades e realizar um XOR da palavra resultante com uma constante de round (RCON, tal que $RCON(i) = x^{i+2}$, em GF(16)) concatenada a um nibble de 0's. Com isso, sendo RCON(1) = 0b10000000 e RCON(2) = 0b00110000, $w = [n_0, n_1]$ e i o índice da subchave, tem-se que:

$$g(w, i) = [SN(n_0), SN(n_1)] \oplus [RCON(i), 0b0000]$$

Abaixo consta o código da implementação, assim como no GitHub (apêndice A):

```
1  void generate_keys(byte* sub_keys, byte* key){
2      // key 1 is the same as the original key
3      sub_keys[0] = key[0];
4      sub_keys[1] = key[1];
5      // g(w) function is the same as
           ↪ [nibble_sub(lsb(w)),nibble_sub(msb(w))] XOR RCON
6      // key 2: RCON = 0b10000000
7      sub_keys[2] = sub_keys[0]^((nibble_sub((sub_keys[1] & 0x0F)) << 4)
           ↪  | nibble_sub((sub_keys[1] & 0xF0) >> 4)) ^ 0b10000000;
8      sub_keys[3] = sub_keys[1]^sub_keys[2];
9      // key 3: RCON = 0b00110000
10     sub_keys[4] = sub_keys[2]^((nibble_sub((sub_keys[3] & 0x0F)) << 4)
           ↪ | nibble_sub((sub_keys[3] & 0xF0) >> 4)) ^ 0b00110000;
11     sub_keys[5] = sub_keys[3]^sub_keys[4];
12     return;
13 }
```

Código 1: Código de geração de chaves, presente no arquivo s-aes.c.

Vale notar que o raciocínio da expansão de chaves do S-AES se assemelha ao do AES, em que a chave será separada em quatro palavras de 4 bytes, de modo que uma chave $K_0 = [w_0, w_1, w_2, w_3]$ seria expandida na primeira etapa, que gera a subchave $K_1 = [w_4, w_5, w_6, w_7]$, da forma:

$$w_4 = w_0 \oplus g(w_3), \quad w_5 = w_4 \oplus w_1, \quad w_6 = w_5 \oplus w_2 \quad \text{e} \quad w_7 = w_4 \oplus w_3$$

Esse padrão é repetido para as outras subchaves, sendo um total de 11 subchaves (contando com a original), se for considerada a chave de 16 bytes. A função g também segue o mesmo padrão do S-AES, em que é feita uma rotação circular para a esquerda

de 1 byte da palavra de 4 bytes, substituindo cada um deles usando a função equivalente SubWords e realizando um XOR do resultado com o RCON do AES ($RCON(i) = [RC(i), 0, 0, 0]$, em que cada valor é um byte e $RC(1) = 1$ e $RC(i) = 2 \cdot RC(i-1)$, com a multiplicação definida em $GF(2^8)$).

2.2 Encriptação

Como dito anteriormente, o S-AES recebe de entrada um bloco de 16 bits como mensagem e uma chave de 16 bits. Após a expansão da chave (KeyExpansion), esta é usada em uma etapa inicial de AddKey, enquanto que as subchaves restantes são usadas nas rodadas do algoritmo, compostas pelas etapas SubNibbles (SN), ShiftRows (SR), MixColumns (MC) e AddKey (A) e, assim como no AES, a última rodada não executa a etapa MixColumns. Isso pode ser representado como a função abaixo:

$$C = Enc(M, K_0, K_1, K_2) = A_{K_2} \circ SR \circ SN \circ A_{K_1} \circ MC \circ SR \circ SN \circ A_{K_0}$$

O resultado obtido da aplicação do código 2 com a chave 0xA73B e o texto em claro "ok" pode ser visto no terminal 1. O código está presente no arquivo `s-aes.c` (com mais comentários) do GitHub (apêndice A).

```
void enc_saes(byte* block, byte* return_block, byte* key){
    byte sub_keys[6];
    generate_keys(sub_keys, key);
    // Add round key (Before first round)
    return_block[0] = block[0] ^ sub_keys[0];
    return_block[1] = block[1] ^ sub_keys[1];
    // S-AES rounds
    for (int i = 0; i < ROUNDS; i++){
        // Substitute Nibbles
         return_block[0] = (nibble_sub((return_block[0] & 0xF0) >> 4)
            ↪ << 4) | nibble_sub((return_block[0] & 0x0F));
        return_block[1] = (nibble_sub((return_block[1] & 0xF0) >> 4)
           ↪ << 4) | nibble_sub((return_block[1] & 0x0F));
        shift_row(return_block); // Shift Rows
        // Mix Columns (not applied on last round)
        if (i != ROUNDS - 1){
            byte a = ((return_block[0] & 0xF0) >> 4);
            byte b = (return_block[0] & 0x0F);
            byte c = ((return_block[1] & 0xF0) >> 4);
            byte d = (return_block[1] & 0x0F);
            return_block[0] = ((a ^ gf_16_mul(4, b)) << 4) |
               ↪ (gf_16_mul(4, a) ^ b);
            return_block[1] = ((c ^ gf_16_mul(4, d)) << 4) |
               ↪ (gf_16_mul(4, c) ^ d);
        }
        // Add round key
        return_block[0] = return_block[0] ^ sub_keys[2 + 2 * i];
        return_block[1] = return_block[1] ^ sub_keys[3 + 2 * i];
    }
    return;
}
```

Código 2: Implementação do algoritmo para o S-AES, presente no arquivo s-aes.c.

```
S-AES - Encryption Mode:
| Plaintext chosen: ok | HEX: 0x6F6B | B64: b2s=
| Key chosen: 0b1010011100111011 | HEX: 0xA73B | B64: pzs=
| Subkeys Generated:
  - Subkey 1: 0xA73B | B64: pzs=
  - Subkey 2: 0x1C27 | B64: HCc=
  - Subkey 3: 0x7651 | B64: dlE=
| Iteration 0:
  - AddRoundKey: 0xC850 | B64: yFA=
| Iteration 1:
  - SubNibbles: 0xC619 | B64: xhk=
  - ShiftRows: 0xC916 | B64: yRY=
  - MixColumns: 0xECA2 | B64: 7KI=
  - AddRoundKey: 0xF085 | B64: 8IU=
| Iteration 2:
  - SubNibbles: 0x7961 | B64: eWE=
  - ShiftRows: 0x7169 | B64: cWk=
  - AddRoundKey: 0x0738 | B64: Bzg=
| Resulting Ciphertext: HEX: 0x0738 | B64: Bzg=
  - Time elapsed during encryption: 0.000300 ms
```

Terminal 1: Execução do s-aes com chave 0xA73B e bloco "ok".

2.2.1 AddRoundKey

A etapa AddRoundKey simplesmente realiza um XOR entre o texto a ser cifrado e a subchave da rodada. Antes de iniciar as rodadas, é feito um AddRoundKey com a primeira subchave (a chave original) e, posteriormente, essa é a última etapa de cada rodada, usando a subchave relativa à rodada. Isso pode ser visto no código 2. Vale notar que o mesmo ocorre no algoritmo original do AES, sendo a única mudança o tamanho da chave.

2.2.2 SubNibbles

A substituição de nibbles (SubNibbles) trata-se da primeira etapa de cada rodada do S-AES, em que cada nibble do bloco passa por uma S-BOX (descrita abaixo), de modo que os dois bits mais significativos correspondem ao índice da linha da S-BOX e os dois da direita indicam a coluna. O AES possui a mesma ideia, porém substitui cada byte a partir de uma S-BOX de dimensão 16 (SubByte). Os valores da tabela para o S-AES, em hexadecimal, correspondem ao novo valor que o nibble receberá e podem ser vistos abaixo, junto com o código que procura o valor na matriz:

$$S = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 9 & 4 & A & B \\ D & 1 & 8 & 5 \\ 6 & 2 & 0 & 3 \\ C & E & F & 7 \end{pmatrix} \end{array}$$

```
byte nibble_sub(byte nibble){
    return SBOX[(nibble & 0xC) >> 2][(nibble & 0x3)];
}
```

Código 3: Implementação do algoritmo para a substituição de nibbles, presente no arquivo s-aes.c.

2.2.3 ShiftRows

Para a etapa ShiftRows, o texto é disposto em forma de uma matriz de dimensão 2, em que cada elemento corresponde a um nibble. Esta etapa realiza uma rotação circular entre os nibbles da segunda linha da matriz apenas. Sendo $[n_0, n_1, n_2, n_3]$ um texto que chega nessa etapa, o resultado da troca de linhas será:

$$\begin{bmatrix} n_0 & n_2 \\ n_1 & n_3 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} n_0 & n_2 \\ n_3 & n_1 \end{bmatrix}$$

A lógica descrita acima pode ser vista no trecho a seguir, com melhores comentários no seu arquivo (`s-aes.c`, localizado no GitHub):

```
void shift_row(byte* block){
    byte shift_result[2];
    // Swapping bottom row
    shift_result[0] = (block[0] & 0xF0) | (block[1] & 0x0F);
    shift_result[1] = (block[1] & 0xF0) | (block[0] & 0x0F);

    block[0] = shift_result[0];
    block[1] = shift_result[1];
    return;
}
```

Código 4: Implementação do algoritmo para a troca de linhas, presente no arquivo s-aes.c.

Por tratar de mensagens de 16 bytes, o AES cria uma matriz quadrada de dimensão 4 em função de bytes, e aplica o ShiftRows para esses bytes. Da mesma forma que o S-AES, a primeira linha não é modificada, enquanto que as subsequentes sofrem deslocamentos circulares a esquerda de forma crescente, ou seja, a segunda linha sofre um deslocamento de um byte, a terceira, de 2 bytes, e a quarta, de 3 bytes.

2.2.4 MixColumns

Sendo uma das etapas mais complexas de ambos os algoritmos, o MixColumns é uma função que multiplica a matriz correspondente ao texto por uma matriz de multiplicação, tudo usando $GF(2^4)$. A sua complexidade é consequência de depender de multiplicações em Galois Field, de modo que, para melhorar a performance, tanto o AES quanto o S-AES não realizam essa etapa na sua última rodada. Enquanto que o S-AES

trabalha com uma matriz de dimensão 2 com elementos em função de 4 bits e $GF(2^4)$, o AES trabalha com uma matriz de dimensão 4, com seus elementos correspondendo a bytes, e $GF(2^8)$.

O código 2 implementa a função de MixColumns em seu loop de rodadas, checando se está na última delas, a fim de não ser executada. Dessa forma, um texto separado em nibbles de 4 bits, $[n_0, n_1, n_2, n_3]$, teria como resultado do MixColumns:

$$\begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} n_0 & n_2 \\ n_1 & n_3 \end{bmatrix} = \begin{bmatrix} n'_0 & n'_2 \\ n'_1 & n'_3 \end{bmatrix}$$

3 Electronic Codebook Operation Mode

O modo de operação Electronic Codebook (ECB) trata cada bloco recebido de forma independente [4]. Como o S-AES requer um bloco de 16 bits, surge a necessidade de preencher entradas que forem incompletas, realizando um processo chamado de padding. Para o projeto, o sistema de padding corresponde a checar se há necessidade de padding, preencher com 0's até completar o último bloco e adicionar um outro bloco ao final, contendo o número de bits 0 inseridos, mesmo que nenhum tenha sido colocado. Após o padding, basta encriptar cada bloco individualmente, com a mesma chave. A seção 4.2 discorre mais sobre este modo de operação, incluindo riscos de segurança que ele pode conter.

A seguir consta a lógica principal do algoritmo para o ECB, sendo possível ver o código na íntegra no arquivo `ecb_s-aes.c`. Além disso, o terminal 2 mostra em ação o algoritmo cada bloco do texto em claro "oka" (vale notar que foi necessário um padding de 8 bits).

```
for (int i = 0; i < n_blocks; i++){
    byte block_plaintext[2] = {return_blocks[i * 2], return_blocks[i *
        ↪ 2 + 1]};
    byte block_ciphertext[2];
    enc_saes(block_plaintext, block_ciphertext, key); // Enc(block)
    return_blocks[i * 2] = block_ciphertext[0];
    return_blocks[1 + i * 2] = block_ciphertext[1];
}
```

Código 5: Lógica principal do ECB. Algoritmo completo está no arquivo ecb_s-aes.c.

```
Eletronic Codebook with S-AES - Encryption Mode:
| Plaintext: oka | HEX: 0x6F6B 0x61 | B64: b2th
| Key chosen: 0b1010011100111011 | HEX: 0xA73B | B64: pzs=
| Plaintext (Padded): HEX: 0x6F6B 0x6100 0x0008 | B64: b2thAAAI
| ECB for each block (Block 3 is the padding info block):
 - Block 1: 0x6F6B | B64: b2s= --> Ciphertext (HEX): 0x0738 | B64: Bzg=
 - Block 2: 0x6100 | B64: YQA= --> Ciphertext (HEX): 0x3721 | B64: NyE=
 - Block 3: 0x0008 | B64: AAg= --> Ciphertext (HEX): 0x40A4 | B64: QKQ=
| Resulting Ciphertext: HEX: 0x0738 0x3721 0x40A4 | B64: Bzg3IUCk
  - Time elapsed during encryption: 0.002100 ms
```

Terminal 2: Execução do modo ecb usando s-aes com chave 0xA73B e texto em claro "oka".

4 Modos de Operação

Os modos de operação definem a forma como grandes volumes de dados são criptografados e descriptografados utilizando cifras de bloco. Uma cifra de bloco é um algoritmo de criptografia que processa os dados em blocos de tamanho fixo, como, por exemplo, o Advanced Encryption Standard (AES), que opera com blocos de 128 bits. No entanto, para proteger mensagens cujo tamanho exceda o de um único bloco, diferentes modos de operação são empregados, garantindo segurança e eficiência conforme as necessidades específicas de cada aplicação.

A seguir, serão apresentados cinco modos de operação comumente utilizados em conjunto com o AES, utilizando a biblioteca PyCryptodome, com os códigos presentes no arquivo `operation_modes.py`, presente no GitHub. Como algumas delas requerem o padding, foi desenvolvida uma função para preencher os blocos necessários:

```python
def padding(data: bytes):
    pad_length = AES.block_size - (len(data) % AES.block_size)
    padded_result = data + bytes([pad_length] * pad_length)
    return padded_result
```

Código 6: Implementação do código para padding, presente no arquivo operation_modes.py.

4.1 Electronic Codebook

O Electronic Code Book (ECB) é considerado o modo de operação mais simples entre os disponíveis para cifras de bloco. Nele, cada bloco de entrada é criptografado de forma independente, resultando em um bloco de saída de tamanho fixo [4]. Como o AES é uma cifra de bloco que opera especificamente com blocos de 128 bits, é necessário que a entrada tenha um tamanho múltiplo deste bloco. Para mensagens em que o comprimento não seja adequado, é utilizado o padding, que preenche os blocos incompletos com dados adicionais para que a cifra funcione corretamente.

Apesar de a independência tornar o algoritmo mais simples e permitir o paralelismo, o ECB corre o risco de ataques de texto escolhido (*Chosen Plaintext Attacks* — CPA), pois, com mensagens maiores, é mais fácil identificar padrões entre blocos e realizar uma criptoanálise. Abaixo consta um exemplo do seu funcionamento:

```
AES - Electronic Codebook Mode
Plaintext: I am Groot.
Key: RandomKeyAES1234
Ciphertext (HEX): 0x7156484a030e9317d8e1593c8ac78fec
Ciphertext (Base 64): cVZISgMOkxfY4Vk8iseP7A==
Elapsed time: 1.3675 ms
```

Terminal 3: Resultado do modo ECB.

4.2 Cipher Block Chaining

O Cipher Block Chaining (CBC) é um dos modos de operação mais populares para cifras de bloco. Nele, os blocos de texto claro são processados de forma encadeada, de modo que cada bloco de entrada sofra uma operação lógica XOR com o bloco de texto cifrado resultante da etapa anterior [4]. Para o primeiro bloco, como não há bloco anterior, é utilizado um vetor de inicialização (IV), que encadeia a sequência de cifragens. Assim como o ECB, este modo também necessita de padding para casos em que a mensagem seja de tamanho incompatível com os blocos da cifra.

Em razão de ser encadeado, o CBC perde a independência entre os blocos, impedindo o processamento paralelo e deixando-o mais lento. Todavia, como todos os blocos dependem do bloco cifrado anterior, e o primeiro bloco é alterado por um vetor de inicialização, que é geralmente aleatório, a aleatoriedade é propagada para o resultado da encriptação de cada bloco, tornando mais complexa a identificação de padrões e, consequentemente, os ataques CPA. Abaixo consta um exemplo do seu funcionamento:

```
AES - Cipher Block Chaining Mode
Plaintext: I am Groot.
Key: RandomKeyAES1234
Initialization Vector: 1010101010101010
Ciphertext (HEX): 0x851574bee4b5f9444f49cb33dd0dbda2
Ciphertext (Base 64): hRV0vuS1+URPScsz3Q29og==
Elapsed time: 1.6044 ms
```

Terminal 4: Resultado do modo CBC.

4.3 Cipher Feedback

O Cipher Feedback (CFB) é um modo de operação para cifras de bloco que transforma a cifra em um sistema de fluxo, permitindo a criptografia de dados em unidades menores, chamadas de s bits, o que retira a necessidade de fazer o padding de blocos. Nele, um vetor de inicialização é inicialmente encriptado, e parte do resultado (s bits mais significativos) é combinada com s bits do texto em claro por meio da operação lógica XOR, gerando parte do texto cifrado. Em seguida, esses s bits de texto cifrado entram em um registrador de deslocamento, que será utilizado para a próxima parte do processo, análoga à descrita anteriormente, sendo encriptado e realizando um XOR entre seus s bits mais significativos e outros s bits do texto em claro. Esse processo se repete até que todo o texto esteja encriptado [4].

Esse modo tem uma vantagem sobre o CBC, em que, apesar de ser sequencial, não é necessário esperar um bloco de n bits novo para realizar a encriptação, uma vez que trabalha com fluxo de bits, ou seja, é possível encriptar o texto em claro conforme os seus bits chegam à função. Todavia, mesmo oferecendo um nível de segurança relativamente maior do que os modos anteriores, qualquer alteração ou corrupção de alguma sequência de bits durante o transporte afetará a encriptação do restante dos bits, pois, do mesmo modo que propaga a entropia do vetor de inicialização gerado aleatoriamente, ele irá propagar para todo o resto da encriptação pendente. Além disso, por ser sequencial, não permite paralelização nas etapas de criptografia, o que impacta o desempenho em sistemas

que requerem alta velocidade de processamento, apesar de ser mais eficiente que o CBC, como dito anteriormente. Abaixo consta um exemplo do seu funcionamento:

```
AES - Cipher Feedback Mode
Plaintext: I am Groot.
Key: RandomKeyAES1234
Initialization Vector: 1010101010101010
Ciphertext (HEX): 0x31598aed6b5164d5d7cbe5
Ciphertext (Base 64): MVmK7WtRZNXXy+U=
Elapsed time: 1.3635 ms
```

Terminal 5: Resultado do modo CFB.

4.4 Output Feedback

O Output Feedback (OFB) é um modo de operação que também usa um fluxo de bits. Semelhante ao CFB, o processo no OFB inicia-se com a criptografia de um vetor de inicialização que, desta vez, deve ser de uso exclusivo a cada vez que for utilizado (nonce). O resultado dessa encriptação é utilizado como um bloco gerador de fluxo. Este fluxo consiste em encriptar o bloco do vetor de inicialização n vezes, tal que n é o número de blocos em que a mensagem foi dividida, sem a necessidade de padding. Com cada um dos n blocos cifrados, é então feita uma operação lógica XOR com cada bloco da mensagem, resultando no texto cifrado. No caso em que o último bloco de texto em claro não for do tamanho do bloco cifrado pelo fluxo e tiver apenas u bits, usa-se apenas os u bits mais significativos do último bloco cifrado para realizar a operação XOR [4].

O fluxo de bits do OFB é independente da encriptação de cada bloco de texto em claro, pois cada bloco n depende exclusivamente do resultado da n-ésima encriptação do vetor de inicialização. Como esse fluxo é gerado de forma determinística a partir do vetor de inicialização e da Chave, e cada bloco de texto em claro é independente do outro, erros de transmissão no texto cifrado não afetam a sincronização global, limitando o impacto de erros de bits ao bloco em que houve o erro. Todavia, a aleatoriedade do algoritmo depende fortemente da unicidade do IV: a reutilização do IV com a mesma chave para diferentes mensagens resulta na geração de fluxos de chave idênticos, tornando os textos cifrados suscetíveis a ataques de recuperação de texto claro por meio de XOR entre mensagens diferentes (Two-Time Pad Attack) — o mesmo problema ocorre caso algum bloco do texto em claro seja usado como IV. Abaixo consta um exemplo do seu funcionamento:

```
AES - Output Feedback Mode
Plaintext: I am Groot.
Key: RandomKeyAES1234
Initialization Vector: 1010101010101010
Ciphertext (HEX): 0x31af0094cf292b6c2505cc
Ciphertext (Base 64): Ma8AlM8pK2wlBcw=
Elapsed time: 1.2301 ms
```

Terminal 6: Resultado do modo OFB.

4.5 Counter

O Counter (CTR), ao invés de utilizar blocos de texto claro ou saídas anteriores como entrada para a cifra, depende de um fluxo da criptografia de um contador numérico, que aumenta a cada interação. O número inicial do contador é geralmente escolhido aleatóreamente e é incrementado de forma determinística, e o resultado da encriptação de cada um desses blocos de números é combinado com um bloco de texto em claro por meio de uma operação lógica XOR, gerando o bloco de texto cifrado correspondente [4]. Este modo não precisa de padding, então se o último bloco for incompleto, o XOR é feito apenas com os bits mais significativos do bloco do contador cifrado, do tamanho do último bloco, assim como o OFB.

A estrutura do CTR permite o paralelismo total dos processos de encriptação e decriptação, pois cada bloco de texto em claro realiza o XOR independente do outro. Como o contador é escolhido de forma aleatória e é incrementado de forma determinística a cada novo bloco, de modo que nenhum dos blocos das iterações do contador sejam iguais, é garantido que cada entrada para a cifra de bloco será única, mesmo quando a chave permanece constante. Isso elimina padrões de repetição no texto cifrado, mesmo quando blocos de texto claro idênticos são processados.

No entanto, um contador que não for gerado aleatoriamente causa um problema para a segurança do algoritmo, considerando que a reutilização do mesmo contador com a mesma chave em diferentes mensagens pode expor os dados a ataques de reutilização de fluxo de chave. O uso de um nonce, ou seja, um valor aleatório que nunca é repetido, permite alterar um contador de valor inicial fixo, o que resolve essa vulnerabilidade, mas simplesmente a geração de um número aleatório para ser o valor inicial do contador também é uma solução. Abaixo tem-se dados da execução da encriptação com esse modo:

```
AES - Counter Mode
Plaintext: I am Groot.
Key: RandomKeyAES1234
Ciphertext (HEX): 0x51130234b91e5147dba1e7
Ciphertext (Base 64): URMCNLkeUUfboec=
Elapsed time: 2.1172 ms
```

Terminal 7: Resultado do modo CTR.

4.6 Aleatoriedade

Dos cinco modos de operação apresentados anteriormente, quatro fazem uso de componentes auxiliares que dependem diretamente de fontes de aleatoriedade para garantir segurança criptográfica. Nos modos CBC, CFB e OFB, é utilizado um vetor de inicialização, cuja principal função, como comentado anteriormente, é evitar padrões repetitivos na saída cifrada, mesmo quando mensagens idênticas são criptografadas com a mesma chave, por meio da introdução e propagação da entropia ao longo da encriptação.

Os vetores devem ser imprevisíveis e únicos, sendo computacionalmente inviável prever valores futuros ou repetir vetores já utilizados com a mesma chave, o que evita vul-

nerabilidades como ataques de repetição ou correlação entre blocos. No caso da repetição ou da previsibilidade de um vetor de inicialização, o modo de operação fica suscetível a ataques CPA, pois haverá maior facilidade na identificação de padrões do texto cifrado.

No caso do modo CTR, a aleatoriedade é incorporada através de um nonce, geralmente combinado com um contador incremental para formar a entrada única de cada bloco. Assim como o IV nos outros modos, o nonce também deve ser único para cada operação de cifragem com a mesma chave, para preservar a segurança do modo de operação. O único modo de operação que não usa valores aleatórios, dos descritos neste documento, é o ECB, o que torna o seu algoritmo o mais vulnerável a ataques de reconhecimento de padrões como o CPA, pois é extremamente determinístico e sempre um mesmo bloco e uma mesma chave resultam no mesmo texto cifrado.

Todavia, não é suficiente a utilização de funções de geração de números aleatórios, caso tal função seja determinística, usando valores como horário, pois isso irá gerar um número pseudoaleatório, criando uma vulnerabilidade ao algoritmo, pois será possível replicar casos para obter o resultado desejado. Para prever isso, é necessário a utilização de funções que realmente tenham uma alta entropia, gerando números verdadeiramente aleatórios, porém isso vem a custo de eficiência.

Na implementação prática, como na biblioteca PyCryptodome, a geração dos valores aleatórios é realizada por meio da função *get_random_bytes()*, que utiliza chamadas de sistema (Windows ou Linux) que geram números pseudoaleatórios com alta entropia. As chamadas do sistema operacional utilizam da entropia coletada por dados imprevisíveis, como tempo de sistema, contadores de hardware e outras variáveis de baixo nível e lança-os em uma função de geração de números aleatórios. Em razão disso, não se pode dizer que é uma função verdadeiramente aleatória, pois a entropia fornecida serve como parâmetro para uma função pseudoaleatória. Porém, a geração de números verdadeiramente aleatórios, como aquelas que se baseiam na entropia de ruído atmosférico, é geralmente menos eficiente e mais demorada, então muitos algoritmos criptográficos se satisfazem com o meio termo que essas chamadas de sistema fornecem.

4.7 Eficiência dos Modos de Operação

Em termos de desempenho, cada modo de operação apresenta características distintas quanto à velocidade do processamento, uso de memória e capacidade de paralelização. O modo ECB destaca-se pela sua alta eficiência e facilidade de implementação, uma vez que cada bloco de texto claro é cifrado de forma independente, permitindo total paralelismo na criptografia. No entanto, suas conhecidas fraquezas de segurança muitas vezes desestimulam seu uso, apesar do desempenho satisfatório.

O modo CBC, por sua vez, impõe uma dependência sequencial entre os blocos, o que limita a capacidade de paralelização na cifragem. Cada bloco de saída depende do bloco cifrado anterior, o que pode afetar o desempenho em aplicações que demandam alta taxa de processamento. Assim como o CBC, modos de fluxo como CFB e OFB apresentam características sequenciais, porém permitem a cifragem de tamanhos de dados menores que o bloco padrão do AES, o que pode trazer vantagens em contextos onde a transmissão de pequenos pacotes é comum, mas também pode trazer custos adicionais ao desempenho.

Por fim, o modo CTR é conceitualmente considerado um dos mais eficientes em termos de desempenho, especialmente por permitir total paralelização tanto no processo de cifragem quanto na decifragem. Isso ocorre porque o contador associado a cada bloco pode ser pré-calculado de forma independente, e, posteriormente, a operação XOR pode ser feita em paralelo com cada um dos blocos de texto em claro.

Foram feitos alguns testes para a encriptação do texto "I am Groot.", chave "RandomKeyAES1234"e vetor de inicialização fixo "1010101010101010" para permitir uma comparação controlada entre os modos de operação. Ao observar os resultados da Tabela 1, percebe-se que o OCB teve a menor média. Isso ocorre, pois ele é o mais eficiente dentre os algoritmos sequenciais, enquanto que os algoritmos do ECB e CTR, que poderiam ser feitos de forma paralela e serem otimizados, foram os mais lentos, uma vez que a biblioteca PyCryptodome não realiza o paralelismo por padrão.

Vale notar que a primeira linha de tempos da Tabela 1 contém valores extremamente altos. Isso provavelmente se dá em decorrência da linguagem Python ter um overhead na sua primeira execução, e da forma que importa e inicializa o módulo PyCryptodome, que foi feita em C. A primeira linha em negrito apresenta a média dos 5 tempos obtidos (contando com os tempos de overhead), enquanto que a última linha em negrito apresenta a média dos 4 tempos obtidos sem o tempo da primeira linha que contém um overhead (execução 2 – 5). Com isso, o CTR deixa de ser o mais lento e passa a ser o segundo mais rápido, ficando atrás apenas do OFB, provavelmente por não estar otimizado com paralelismo.

	EBC	**CBC**	**CFB**	**OFB**	**CTR**
Execução 1	1.4629 ms	1.2863 ms	1.3374 ms	1.2546 ms	2.4915 ms
Execução 2	0.0532 ms	0.0327 ms	0.0218 ms	0.0298 ms	0.0372 ms
Execução 3	0.0335 ms	0.0595 ms	0.0431 ms	0.0192 ms	0.0216 ms
Execução 4	0.0279 ms	0.0195 ms	0.0240 ms	0.0134 ms	0.0163 ms
Execução 5	0.0339 ms	0.0148 ms	0.0201 ms	0.0133 ms	0.0170 ms
Média das 5 execuções	**0.32228 ms**	**0.28256 ms**	**0.28928 ms**	**0.26606 ms**	**0.51672 ms**
Media das execuções 2 – 5	**0.03712 ms**	**0.031625 ms**	**0.02725 ms**	**0.018925 ms**	**0.023025 ms**

Tabela 1: Tempo de 5 execuções de cada modo de operação, com o tempo em negrito indicando a média das execuções

5 Conclusão

Com este trabalho, foi possível entender melhor sobre o funcionamento do AES por meio da implementação do S-AES, notando as diferenças e semelhanças e como o ECB funciona com essa versão mais simples desse algoritmo simétrico. Além disso, foi possível compreender as diferenças entre os modos de operação por meio da sua aplicação no AES-128, seja em questão de segurança, seja em questão de performance. Por fim, é

possível executar tanto o código em C para o S-AES, quanto o código em Python, para os modos de operação do AES, visitando o GitHub, com o link no apêndice A.

Referências

[1] STALLINGS, W. HALL, P. *Cryptography and Network Security Principles and Practices, Fifth Edition*, 2010.

[2] BARKER, E.; ROGINSKY, A. *Transitioning the use of cryptographic algorithms and key lengths*. Mar. 2019. Disponível em: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf>. Acesso em: 12 de junho de 2025.

[3] NILS. Simplified AES (S-AES) Cipher Explained: A Dive into Cryptographic Essentials. Disponível em: <https://www.kopaldev.de/2023/09/17/simplified-aes-s-aes-cipher-explained-a-dive-into-cryptographic-essentials/>. Acesso em: 13 de junho de 2025.

[4] DWORKINS, M. Recommendation for Block Cipher Modes of Operation. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf/>. Acesso em: 14 de junho de 2025.

A Código do Exercício

O código foi desenvolvido e testado em Windows, pode haver erros inesperados em Unix, mas em tese não é para haver. O código em C está no diretório `S-AES` e, para compilar, basta usar o comando `make main` e executar o arquivo gerado.

O código desenvolvido em Python está na versão 3.13.0, testado em sistema operacional Windows. Instruções de execução e instalação de bibliotecas estão no `README.md` do GitHub e o arquivo `main.py` consta na pasta `AES Operation Modes (Python)`.

- Link para repositório: https://github.com/Luke0133/Trabalho02-SegurancaComputacional.