Análise do AES e implementação do S-AES

Maxwell Oliveira dos Reis*, Iasmim de Queiroz Freitas†

Universidade de Brasília, 16 de Junho de 2025

1 Objetivos

O objetivo deste trabalho é implementar uma versão simplificada do algoritmo de criptografia Advanced Encryption Standard (AES) e realizar uma comparação entre o algoritmo original e o simplificado. Dentre as características para a comparação estão: segurança, tempo de execução, eficiência e vulnerabilidades. Para a versão simplificada, o modo de operação Electronic Codebook (ECB) também será utilizado. Por fim, realizaremos uma comparação entre os modos de operação do AES tradicional.

2 Introdução

O Advanced Encryption Standard (AES) é um algoritmo de criptografia que foi selecionado pelo National Institute of Standards and Technology (NIST) em 2001. O objetivo do NIST era substituir o Data Encryption Standard (DES), que já estava se tornando obsoleto por algo que mais moderno que trouxesse mais segurança.

A eficiência do AES foi provada com o tempo, pois, mesmo após mais de 20 anos do seu lançamento, ele continua sendo amplamente utilizado para as mais diversas finalidades em todo o mundo.

Mesmo sendo efetivo no seu objetivo, o AES é um algoritmo complexo em que estudantes e pesquisadores iniciantes no ramo tinham dificuldade de compreender, assim o Simplified AES (S-AES) foi desenvolvido.

O S-AES não é um algoritmo de criptografia com o objetivo de ser usado em aplicações pelo mundo. Desenvolvido apenas para fins educacionais, ele opera com uma chave e um bloco menor e com menos rodadas, quando comparado ao AES tradicional. É um algoritmo criado para se introduzir ao real AES.

Todas as operações no S-AES são aplicadas no corpo finito $GF(2^4)$, composto por 16 elementos, que são representados por um polinomio de 4 bits. Nesse corpo a adição é realizada com a operação bitwise XOR, e a multiplicação é feita módulo o polinômio primitivo $x^4 + x + 1$. Essas operações são reversíveis, garantindo a possibilidade de decriptação.

3 Desenvolvimento

3.1 S-AES

Esta primeira parte do trabalho tem como objetivo a implementação do Simplified AES (S-AES), uma versão simplificada do AES, com chave e bloco de 16 bits e apenas 2 rodadas, desenvolvida para fins educacionais. Assim como o AES, o S-AES é divido em rodadas, que contém as seguintes operações:

- Add round key
- Nibbles Substitution
- Shift row
- Mix columns

A chave no S-AES também é expandida em subchaves, onde a cada rodada é gerada uma nova subchave. Esse processo de expansão de chave é chamado de $KeyExpansion$, essa será a ultima função a ser explicada nessa parte. Abaixo é apresentado o fluxo de encriptação e decriptação do algoritmo, seguido das funções desenvolvidas para aplicar as operações do S-AES:

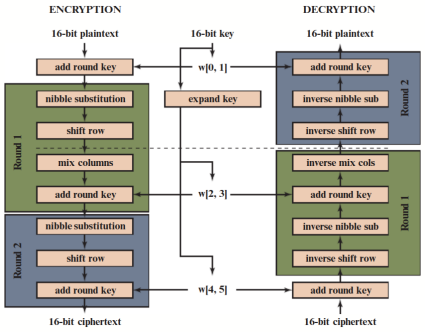

Figura 1: *Esquema de encriptação/decriptação do S-AES*
Fonte: https://www.scribd.com/document/685785464/S-AES

3.1.1 Matriz de nibbles

Antes de aplicar as operações do S-AES, o bloco de 16 bits (plaintext) é dividido em uma matriz $2X2$ de nibbles (blocos de 4 bits). Essa estrutura é similar a usada no AES e permite aplicar diretamente as transformações nas funções $SubNibbles$, $ShiftRows$ e $MixColumns$. Para isso, criamos uma função *split_to_nibbles*, apresentada a seguir:

```
vector<vector<int>> split_to_nibbles(int n){
    vector<vector<int>> nibbles(2, vector<int
        >(2));
    // Extract 4-bit chunks from least to most
        significant
    for(int i=1; i >= 0; i--){
        for(int j=1; j >= 0; j--){
            nibbles[j][i] = n & 0b1111; // Get
                the last 4 bits
            n >>= 4; // Shift right to get the
                next nibble
        }
    }
    return nibbles;
}
```

Listing 1: *Função que converte inteiro de 16 bits em matriz $2X2$ nibbles*

*maxwell.reis@aluno.unb.br
†iasmimqf@gmail.com

3.1.2 AddRoundKey

Esta função consiste apenas em aplicar o XOR em cada um dos nibbles da matriz com o da subchave da rodada. Para faciliar o código a chave também foi transformada em uma matriz $2X2$ de nibbles.

```
void add_round_key(vector<vector<int>>& nibbles
    , vector<vector<int>>& key_vector){
    for(int i=0; i < 2; i++){
        for(int j=0; j < 2; j++){
            nibbles[i][j] ^= key_vector[i][j];
        }
    }
}
```

Listing 2: *Função AddRoundKey*

3.1.3 SubNibbles

A substituição dos nibbles é feita atráves de uma simples busca na tabela(S-BOX), definida pelo S-AES. Cada um dos 4 nibbles da matriz vai ser mapeado a um novo valor. Essa função foi dividida em duas funções, a primeira (*aply_sbox*) é responsável por fazer esse mapeamento do novo valor do nibble, e a segunda (*sub_nibbles*) apenas percorre cada nibble da matriz e chama *aply_sbox*. Como implementamos a encriptação e a decriptação, a S-BOX consiste em uma matriz de dois vetores, sendo o primeiro a S-BOX de encriptação, e o segundo a S-BOX de decriptação.

```
int apply_sbox(int nibble, bool decrypt = false
    ){
    // Row 0: encryption S-box | Row 1:
        decryption S-box
    vector<vector<int>> sbox_nibble = {
        {9, 4, 10, 11, 13, 1, 8, 5, 6, 2, 0, 3,
             12, 14, 15, 7},
        {10, 5, 9, 11, 1, 7, 8, 15, 6, 0, 2, 3,
             12, 4, 13, 14}
    };
    return sbox_nibble[decrypt][nibble];
}

void sub_nibbles(vector<vector<int>>& nibbles,
    bool decrypt = false){
    for(int i=0; i < 2; i++){
        for(int j=0; j < 2; j++){
            nibbles[i][j] = apply_sbox(nibbles[
                i][j], decrypt);
        }
    }
}
```

Listing 3: *Função sub_nibbles*

3.1.4 ShiftRows

No shift de linha, a primeira linha é mantida e apenas a segunda é alterada, onde é realizado um shift circular entre o primeiro e segundo nibble, reusltando na troca de posições entre eles.

```
void shift_rows(vector<vector<int>>& nibbles){
    swap(nibbles[1][0], nibbles[1][1]);
}
```

Listing 4: *Função shift_rows*

3.1.5 MixColumns

Para entender esta operação, é importante lembrar que todas as operações são realizadas em $GF(2^4)$. Nesta etapa, a matriz de nibbles é multiplicada por uma matriz fixa (mostrada abaixo), que corresponde à matriz de encriptação. Essa matriz é equivalente à segunda (usada na forma numérica), já que, no $GF(2^4)$, o elemento 1 representa o polinômio constante 1 e o elemento 4 representa o polinômio x^2.

$$\begin{bmatrix} 1 & x^2 \\ x^2 & 1 \end{bmatrix} \Rightarrow \quad \text{(substituindo } x^2 \text{ por 4)} \Rightarrow \begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix}$$

Para a decriptação, utiliza-se a matriz inversa da anterior:

$$\begin{bmatrix} x & x^3+1 \\ x^3+1 & x \end{bmatrix} \Rightarrow \begin{bmatrix} 9 & 2 \\ 2 & 9 \end{bmatrix}$$

O resultado obtido é reduzido módulo $x^4 + x + 1$. Para isso, foi implementada uma `struct` representando o corpo finito $GF(16)$, contendo as operações de multiplicação e redução modular. A operação de adição não foi incluída, pois, nesse contexto, ela se resume à aplicação do operador XOR. Como essa parte não é o foco do trabalho, a estrutura não será detalhada aqui, mas está documentada e disponível no repositório no *GitHub*.

```
GF_16 GF; // Instance of GF(2^4) (primitive x^4
    + x + 1)
void mix_columns(vector<vector<int>>& matrix,
    bool decrypt = false){
    vector<vector<int>> ans(2, vector<int>(2,
        0));
    vector<vector<vector<int>>> mult={
        {{1, 4}, {4, 1}},  // encryption matrix
        {{9, 2}, {2, 9}}   // decryption matrix
    };
    for(int i=0; i < 2; i++){
        for(int j=0; j < 2; j++){
            for(int k=0; k < 2; k++){
                ans[i][j] ^= GF.mul(mult[
                    decrypt][i][k], matrix[k][j
                    ]);
            }
        }
    }
    swap(ans, matrix);
}
```

Listing 5: *Função mix_columns*

3.1.6 KeyExpansion

A operação de expansão da chave consiste em agrupar os 4 nibbles em duas palavras de 8 bits(1 byte), onde cada palavra representa uma coluna da matriz de nibbles. Em seguida, como é apresentado na imagem abaixo, a primeira palavra da nova subchave é obtida aplicando a função g à segunda palavra da chave original, e realizando o XOR desse resultado com a primeira palavra. Já a segunda palavra da nova subchave é gerada a partir do XOR entre a segunda palavra da chave original e a primeira palavra da nova subchave. Essa operação é realizada uma vez a cada rodada, portanto, além da chave original, são gerada duas novas subchaves.

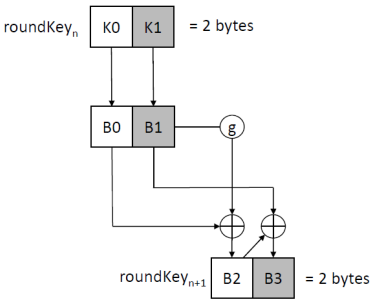

Figura 2: *Esquema da operação de expansão de chave*
Fonte: Kopaldev (2023) [link]

A função g é aplicada sobre 1 byte, que é inicialmente divido em dois nibbles. Em seguida, esses nibbles têm suas posições invertidas, e a função $SubNibbles$ é aplicada a cada um deles. Após essa etapa, realiza-se a operação de XOR no primeiro nibble com o polinômio reduzido x^{i+2}, sendo i o número da rodada. Por fim, os dois nibbles são agrupados novamente, formando um novo byte, que corresponde a palavra resultante da função g.

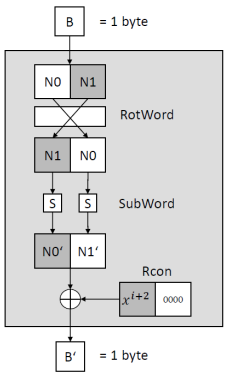

Figura 3: *Esquema da função g*
Fonte: Kopaldev (2023) [link]

```
void g_function(vector<int>& w){
    swap(w[0], w[1]);
    for(int i=0; i < 2; i++) w[i] = apply_sbox(
        w[i]);
}

void expand_key(vector<vector<int>>& key_vector
    , int round){
    vector<int> key1 = { key_vector[0][1],
        key_vector[1][1] };
    g_function(key1);

    // Add round constant: 2^(round+2) reduced
        mod the primitive polynomial.
    key1[0] ^= GF.mod(1 << (round+2));

    // First new column = previous column XOR
        g_function result
    for(int i=0; i < 2; i++) key_vector[i][0]
        ^= key1[i];

    // Second new column = first new column XOR
         previous column
    for(int i=0; i < 2; i++) key_vector[i][1]
        ^= key_vector[i][0];
}
```

Listing 6: *Função expand_key*

3.1.7 Encriptação

A encriptação é realizada em duas rodadas principais, utilizando as subchave geradas durante a expansão da chave. Assim como ilustrado no esquema 1, incialmente é aplicado o XOR entre a chave original e a matriz de nibbles, seguido pelas rodadas que aplicam, em ordem, as operações $SubNibbles$, $ShiftRows$, $MixColumns$ (apenas na primeira rodada) e $AddRoundKey$ (com a subchave da rodada correspondente). Ao final do processo, obtem-se o `ciphertext`.

Abaixo é apresentado o código da função de encriptação.

```
    vector<vector<int>> nibbles =
        split_to_nibbles(n);
    vector<vector<int>> key_nibbles =
        split_to_nibbles(this->key);

    // round 0
    add_round_key(nibbles, key_nibbles);

    // round 1
    sub_nibbles(nibbles);
    shift_rows(nibbles);
    mix_columns(nibbles);
    expand_key(key_nibbles, 1);
    add_round_key(nibbles, key_nibbles);

    // round 2
    sub_nibbles(nibbles);
    shift_rows(nibbles);
    expand_key(key_nibbles, 2);
    add_round_key(nibbles, key_nibbles);

    // Converts the nibbles into a 16-bit
        integer (ciphertext)
    int ciphertext=0;
    for(int i=0; i < 2; i++){
        for(int j=0; j < 2; j++){
            ciphertext <<= 4;
            ciphertext |= nibbles[j][i];
        }
    }

    return ciphertext;
}
```

Listing 7: *Função de encriptação do S-AES*

Para ilustrar o funcionamento da função de encriptação, a seguir é apresentado um exemplo completo da execução do algoritmo, passo a passo, uitilizando `0100 1010 1111 0101` como chave e `1101 0111 0010 1000` como plaintext, e obtendo como resultado o ciphertext `0010 0100 1110 1100`.

```
==================== S-AES - Encryption ====================

Plaintext:          1101011100101000
Initial key:        0100101011110101

------------ Round 0 --------------

Adding round key:
= 1101 0111 0010 1000   XOR  0100 1010 1111 0101
= 1001 1101 1101 1101

------------ Round 1 --------------

Substituting nibbles:
= 0010 1110 1110 1110

Shifting rows:
= 0010 1110 1110 1110

Mixing Columns:
= 1111 0110 0011 0011

Expanding Key 0 (0100 1010 1111 0101 ):
Key 1 = 1101 1101 0010 1000

Adding round key:
= 1111 0110 0011 0011   XOR  1101 1101 0010 1000
= 0010 1011 0001 1011

------------ Final Round (2) --------------

Substituting nibbles:
= 1010 0011 0100 0011

Shifting rows:
= 1010 0011 0100 0011

Expanding Key 1 (1101 1101 0010 1000 ):
Key 2 = 1000 0111 1010 1111

Adding round key:
= 1010 0011 0100 0011   XOR  1000 0111 1010 1111
= 0010 0100 1110 1100

------------------------------------------------------------

-> Ciphertext

Bin:                0010 0100 1110 1100
Hex:                24EC
Base64:             JOw=
============================================================
```

Figura 4: *Exemplo de encriptação do S-AES*

A operação de decriptação não será detalhada neste relatório, mas sua implementação completa pode ser encontrada no repositório do projeto, presente no anexo A.

3.1.8 ECB

O modo ECB (Eletronic Codebook) é a forma mais simples dentre os modos de cifras de bloco simétricos. No caso do S-AES, cada bloco tem 16 bits, então o ECB apenas divide o texto em blocos de 16 bits, cifra cada um deles de forma independente, utilizando sempre a mesma chave, e retorna estes blocos cifrados concatenados como resultado.

Na implementação do ECB, foi criada a classe `ECB`, que recebe como construtor a chave de 16 bits, como um inteiro. Na função de encriptação, a mensagem (plaintext) é recebida como uma string, que é então convertida em um vetor de bytes, onde cada caracter da string representa um byte. Como o S-AES trabalha com blocos de 16 bits (2 bytes), o código verifica se o número total de bytes é par. Caso não seja, o programa acusa erro, então é necessário que a mensagem orginal seja múltipla de 16 bits. O código percorre o vetor de dois em dois bytes, juntando os dois em um bloco de 16 bits, e aplicando a função de encriptação do S-AES apresentada na seção anterior. Este resultado é separado novamente em 2 bytes e armazenado num vetor de inteiros. Por fim, o vetor resultante é convertido para a *base*64, obtendo assim o ciphertext que é retornado pela função.

Para realizar a conversão entre o vetor de bytes e *base*64, foi criada a classe `base64`, responsável por codificar e decodificar os dados nesse formato. Essa classe foi documentada e pode ser encontrada no repositório. Além disso, também foi criada uma função para transformar a string em *base*64 em um vetor de bytes. Esta função, junto com outras funções auxiliares utilizadas ao longo do projeto, foi adicionada à pasta util no repositório.

```cpp
class ECB {
public:
    int key;
    SAES saes;

    ECB(int key_) : key(key_), saes(key_, false, true) {}

    string encrypt(string plainText) {
        auto bytes = get_bytes_from_text(plainText);
        assert((int)bytes.size() % 2 == 0);

        vector<int> result;

        for(int i = 0; i < (int)bytes.size(); i += 2) {
            auto slice = get_vector_slice(bytes, i, 2);
            int num = (slice[0] << 8) + slice[1];
            int res = saes.encrypt(num);
            result.push_back(res >> 8);
            result.push_back(res & 0xFF);
        }

        return Base64::convert_to(result);
    }
};
```

Listing 8: *Função expand_key*

Como o modo ECB apenas divide a mensagem em blocos de 16 bits e aplica o algoritmo de cifragem a cada uma delas, blocos de entrada iguais geram blocos cifrados iguais, revelando padrões no texto original. Isso é uma fraqueza desse modo, que compromete seriamente a segurança da cifragem. Esse problema será descutido com mais detalhes na seção de Análise.

Para demonstrar essa vulnerabilidade, foi criptografado um texto contendo blocos repetidos:

- **Texto original:**
  `AAABBBAAABBB`
- **Chave utilizada (em hexadecimal):**
  `3A94`
- **Texto cifrado (em base64):**
  `xYyFhopmxYyFhopm`

Percebe-se que o bloco `AAABBB` aparece duas vezes na entrada e, como resultado, o mesmo bloco cifrado `xYyFhopm` também se repete na saída. Este exemplo confirma a vulnerabilidade do modo ECB, que mantém uma correspondência direta entre blocos de entrada iguais e blocos cifrados iguais.

3.2 Análise

Nesta seção iremos comparar o Advanced Encryption Standard com sua versão simplificada. Em primeiro momento vale notar que, por ser uma versão simplificada e criada para fins didáticos, é esperado que o algoritmo original se sobressaia em aspectos como segurança e proteção contra vulnerabilidades. Por outro lado, a versão simplificada opera com

uma chave e com blocos menores, o que poderia trazer uma vantagem no desempenho.

O AES possui diferentes versões, sendo cada uma delas diferenciada pelo tamanho da chave, o AES-128, AES-192 e AES-256. Quanto maior o tamanho da chave, maior a segurança que o algoritmo consegue prover e maior a dificuldade de se efetuar ataques, principalmente de força bruta. Porém, chaves maiores exigem mais rodadas do algoritmo, o que impacta o custo computacional e seu tempo de execução.

Para as seguintes citações do AES e dos seus modos de operação, utilizamos a biblioteca Cryptodome para Python na versão 3.12.

3.2.1 Segurança computacional e vulnerabilidades

Em termos de segurança, o AES tradicional se sobressai muito sobre sua versão simplificada. Mesmo o AES-128 já possui uma chave de 128 bits, contra apenas 16 da versão simplificada. Além disso, o algoritmo tradicional opera com um bloco de tamanho fixo de 128 bits e 10 rodadas, enquanto a versão simplificada utiliza um bloco de apenas 2 bytes e usa apenas 2 rodadas.

Todos esses fatores somados tornam o algoritmo simplificado muito frágil para um ataque de força bruta, considerando que existem apenas $2^{16} = 65536$ chaves distintas. Assim, uma primeira possibilidade de ataque seria apenas testar cada uma das chaves possíveis e tentar decifrar a mensagem.

Além do tamanho da chave, outra vulnerabilidade do algoritmo simplificado vem do fato dele operar em blocos de tamanho de 16 bits. Com isso, novamente existem poucas possibilidades de resultados, então seria possível realizar um estudo de criptoanálise e explorar os padrões encontrados.

Além disso, se a versão simplificada encriptar usando o modo de operação Electronic Codebook (ECB), as análises se tornariam ainda mais simples, dado que dois blocos iguais teriam exatamente a mesma saída. Ou seja, ao explorar uma mensagem de 2^{16} bits ou mais, com certeza teríamos a repetição de ao menos um bloco. Para mensagens menores a repetição não é garantida, mas ainda aconteceria se houvessem dois blocos iguais.

Modos de operação como o Cipher Block Chaining (CBC), Ciphertext Feedback (CFB), Output Feedback (OFB) e Counter (CTR) se sobressaem quando comparados ao ECB, visto que todos eles possuem um grau de difusão/variabilidade no seu procedimento. O CBC, por exemplo, já atua de forma que dois blocos de entrada iguais não teriam o mesmo output e, para isso, usa um Initialization Vector (IV) e o encadeamento do processamento dos blocos. O CTR possui o mesmo benefício sobre blocos iguais, mas sua lógica utiliza um Nonce, que é um valor único que não deve ser reutilizado. Ou seja, dentre todos os modos de operação o ECB é o que traz menos segurança, visto a falta de difusão e aleatoriedade.

Contudo, mesmo utilizando o AES simplificado com algum dos outros modos de operação citados, ainda não obtém-se um algoritmo seguro. O modo de operação pode dificultar a criptoanálise e a gerar menos padrões, mas, mesmo assim, o AES simplificado ainda é vulnerável ao ataque de força bruta na chave.

O AES tradicional, mesmo na sua versão mais simples, já opera com uma chave e um bloco de 128 bits. Com uma chave desse tamanho, o ataque de força bruta na chave já se torna inviável, uma vez que existem $2^{128} = 340282366920938463463374607431768211456$ chaves distintas. Esse é um número tão grande que, supondo que um processador consiga testar cerca de 10^8 chaves por segundo, ainda levariam mais de 10^{23} anos para examinar todas as possibilidades.

Além disso, o AES tradicional trabalha em blocos de tamanho fixo de 128 bits, o que também nos dá mais possibilidades de blocos distintos e uma chance menor de se encontrar padrões via criptoanálise.

Por fim, o raciocínio sobre modos de operação para o AES simplificado também se aplica aqui. O ECB continua sendo o que traz uma menor segurança, visto que não utiliza nenhum tipo de aleatoriedade, enquanto os CBC, CFB, OFB e CTR já são escolhas mais seguras que utilizam algum mecanismo/técnica para gerar difusão na saída.

Assim, é possível concluir que o AES tradicional é muito mais seguro contra ataques quando comparado a sua versão simplificada, fato que decorre principalmente da diferença do tamanho das chaves e dos blocos, mas também do número de rodadas utilizadas internamente.

3.2.2 Tempo de execução e eficiência

Nesta seção iremos realizar duas comparações entre os algoritmos e seus modos de operação. A primeira comparação será composta pelo modo de operação ECB do AES tradicional, com implementação provida pelo Cryptodome, e o AES simplificado, usando nossa implementação.

Para os testes do modo de operação do AES, os seguintes códigos foram escritos com a ideia de mensurar seu tempo de execução.

As mensagens usadas para os testes estão armazenadas na pasta *messages* do repositório, todas geradas com a função *get_random_bytes*() do *Cryptodome*. Na pasta temos 3 arquivos de mensagens de diferentes tamanhos, separadas em hexadecimal e base64. Note que, também haverá o uso de uma mensagem de 268435456 bytes que não está no repositório, pois seu arquivo possui mais $350mb$.

Além disso, para todos os testes a seguir, foram definidos os seguintes valores iniciais para o AES.

Item	Valor
Chave	0xaf3ff749bbfd9a8a3dc791b2cceb193c
IV	0x7ac4b2b76533f1a702de0c1660192bfb
Nonce	0x6102500a1e90abcab67f620d

Tabela 1: *Valores de entrada para o AES*

Vale notar que, como estamos usando o AES com uma chave de 16 bytes, a versão do AES a ser utilizada será o AES-128.

Para o SAES também há a necessidade de definirmos os valores iniciais. Porém, para ele, há apenas um, que seria a própria chave.

Comparação entre AES e SAES, ambos com ECB

Para este teste, vamos realizar a mensuração de tempo usando o comando *time*, nativamente presente em sistemas operacionais baseados no *Unix*. Das informações de saída

Item	Valor
Chave	0x3A94

Tabela 2: *Valores de entrada para o SAES*

do *time*, a mensuração utilizada será a de *user*, que mede o tempo de CPU do programa.

Com tudo definido, vamos executar ambos os algoritmos com diferentes tamanhos de mensagem, mensurar seu tempo de execução e comparar os resultados. Os resultados podem ser encontrados na tabela 3.

Tamanho da mensagem	Tempo (em ms)	
	AES	SAES
2^4 bytes	59	1
2^{12} bytes	61	1
2^{20} bytes	82	3
2^{28} bytes	760	4

Tabela 3: *Tempos de execução do AES e SAES para diferentes tamanhos de mensagem*

Note que, nesses testes, o SAES se provou superior ao AES em termos de eficiência. Isso decorre diretamente de um conjunto de motivos, sendo o principal o menor tamanho da chave e menor numero de rodadas, comparado ao AES original. Porém, outros fatores podem ter influenciado essa análise, por exemplo a própria linguagem de programação utilizada nos dois códigos.

Comparação entre os modos de operação do AES

Para essa comparação, usaremos diretamente o arquivo em *python*3 criado para essa analise, que pode ser encontrado aqui.

O programa criado irá executar, para uma mesma mensagem, a encriptação e decriptação em todos os modos de operação e mensurar seu tempo em nanossegundos. Além disso, para garantir um valor mais consistente, esse processo é realizado 100 vezes para cada modo de operação (na mesma mensagem e parâmetros) e apenas a média dos valores é computada.

Os resultados podem ser encontrados na tabela 4.

Tamanho mensagem	Tempo (em ms)				
	ECB	CBC	CFB	OFB	CTR
2^4 bytes	0,053	0,044	0,041	0,037	0,048
2^{12} bytes	0,084	0,089	0,095	0,075	0,088
2^{20} bytes	1,02	5,03	5,40	4,60	3,10
2^{28} bytes	888	1904	2070	1873	1400

Tabela 4: *Tempos de execução do AES para diferentes tamanhos de mensagem e modos de operação*

Note que, para mensagens de tamanho até 2^{20} bytes, a diferença entre os tempos de execução dos algoritmos existia, mas era pequena e, nos primeiros exemplos, até irrelevante. Porém, para maior mensagem, a diferença do funcionamento de cada modo operação passa a afetar diretamente seu desempenho final.

Na 4, em termos apenas de tempo de execução, os modos ECB, OFB e CTR se sobressaem sobre os outros modos. Além disso, seria possível otimizar ainda mais os seus tempo de execução usando programação paralela para processar todos os blocos de uma vez. Essa otimização não pode ser aplicada em modos como CBC e CFB, onde um bloco depende do seu anterior.

Precisamos processar os blocos de maneira linear nos modos CBC e CFB, por isso o tempo de execução deles realmente tende a ser bem similar. Porém, o CFB realiza uma operação a mais, então ele é, na verdade, levemente mais lento.

Diferente dos outros modos, o tempo de execução do CFB é afetado por uma outra variaável, chamada $segment\_size$. Por padrão, a biblioteca usada utiliza $segment\_size = 8$ e isso faz com que a sua execução para a mensagem de $2^{28} bytes$ fique extremamente lenta, ultrapassando 16 segundos. A solução para isso foi utilizar um $segment\_size$ maior e, no nosso caso, usamos $segment\_size = 128$, o que nos trouxe um valor mais próximo da realidade.

4 Conclusão

Ao longo deste trabalho apresentamos, de forma prática e teórica, os principais conceitos que abrangem os todos os fundamentos do AES. A implementação S-AES se mostrou útil pra entender como os pequenos conceitos teóricos se unem para criar algo funcional. Aqui conseguimos passar por etapas como operações em corpos finitos, substituições, permutações e expansão da chave, conceitos fundamentais para o AES tradicional.

As análises realizadas foram fundamentais para entender as reais diferenças entre o S-AES e o AES tradicional, tanto em termos de segurança quanto em termos de desempenho. O S-AES se mostrou muito útil para o seu propósito de introduzir alunos/pesquisadores aos conceitos do AES, mas, sua simplificação o torna inutilizável para uma aplicação real. Além disso, vimos que o tamanho da chave e do bloco que o algoritmo opera afeta diretamente sua resistência contra ataques de força bruta e criptoanálise.

Além disso, a comparação entre os modos de operação exibe como a escolha do modo influencia a segurança e o desempenho do algoritmo. O ECB, modo mais simples, é o possui o processamento mais rápido, mas traz pouca segurança. Os modos como ECB, CBC, CFB, OFB e CTR podem ser mais custosos computacionalmente, mas conseguem inserir difusão e variabilidade no procedimento, garantindo uma segurança maior.

Desse modo, esse trabalho atinge seu objetivo de explorar o AES, implementar o AES, sua versão simplificada, e de entender os prós e contras entre os modos de operação. Por fim, este trabalho reforça não só a importância de entender as condições onde cada algoritmo pode ser utilizado, mas também mostra que a escolha correta dos parâmetros e modos de operação é fundamental.

Anexos

A Repositorio

O repositório do projeto contendo todo o código desenvolvido pode ser encontrado por meio deste link.