Análise e Implementação do Algoritmo S-AES (Simplified Advanced Encryption Standard)

Cauê Trindade 231019003
Vinícius Araújo 221001981

Resumo

Este relatório apresenta uma análise do algoritmo de criptografia *Simplified AES* (SAES), versão adaptada do AES utilizada para fins educacionais, destacando cada componente de sua estrutura e apresentando e comparando seus modos de operação. O objetivo é a compreensão do funcionamento do algoritmo AES por meio de sua versão simplificada.

1 Introdução

O *Advanced Encryption Standard* (AES) é o algoritmo de criptografia simétrica mais amplamente utilizado na atualidade. Adotado oficialmente em 2001 pelo governo dos Estados Unidos (FIPS PUB 197), o AES substituiu o DES como novo padrão de criptografia, oferecendo maior segurança, desempenho e flexibilidade. Seu design robusto e eficiente permitiu ampla adoção em aplicações que vão desde comunicações seguras até criptografia de discos e redes.

Inspirado na estrutura e princípios do AES, o *Simplified AES* (SAES) é uma versão reduzida proposta com fins exclusivamente educacionais. O SAES visa manter os conceitos fundamentais do AES como substituição, permutação, mistura de chaves e operações em campo finito, mas em um ambiente significativamente menos complexo, operando com blocos e chaves de apenas 16 bits. Essa simplificação torna o algoritmo impraticável para uso real, dada sua vulnerabilidade a ataques de força bruta e limitações estruturais, mas o torna uma excelente ferramenta didática para compreender a lógica interna do AES.

Ao longo deste relatório, exploraremos o funcionamento do SAES, analisando seus componentes, ciclo de encriptação e possíveis fragilidades, com o objetivo de aprofundar a compreensão de algoritmos de cifragem modernos a partir de uma versão mais acessível. Ademais, abordaremos os modos de operação do AES por meio do SAES, ressaltando suas diferenças, vantagens e desvantagens.

2 Visão Geral do SAES

O SAES opera sobre blocos de 16 bits, o que significa que cada operação de cifragem ou decifragem processa dois bytes simultaneamente. Essa estrutura permite a exploração de conceitos fundamentais do AES real, como operações em campo finito, substituições não lineares (S-boxes), permutações e transformações lineares, mas em um contexto mais simples e didático. A chave utilizada no SAES também possui 16 bits e passa por um processo de expansão que gera três subchaves de 16 bits: K_0, K_1 e K_2, que são utilizadas nos diferentes estágios do algoritmo.

Assim como no SDES, o espaço de chaves do SAES é extremamente limitado com apenas $2^{16} = 65.536$ possibilidades tornando-o completamente inseguro contra ataques de força bruta em contextos reais. No entanto, esse tamanho reduzido é proposital e serve para permitir experimentação e simulação dos ataques mais comuns, facilitando o aprendizado sobre vulnerabilidades e boas práticas criptográficas. Em contraste, o AES padrão utiliza chaves de 128, 192 ou 256 bits, resultando em um espaço de chaves exponencialmente maior e impraticável de explorar por força bruta com os recursos atuais.

O processo de cifragem no SAES segue uma estrutura inspirada no AES, dividida em etapas bem definidas:

- **AddRoundKey (Adição de chave da rodada):** A entrada é combinada com uma subchave por meio de uma operação XOR. Essa etapa ocorre no início e ao final de cada rodada.
- **SubNibbles (Substituição de Nibbles):** Cada nibble (4 bits) do bloco é substituído utilizando uma S-box fixa, introduzindo não-linearidade ao processo.
- **ShiftRows (Troca de linhas):** Uma rotação é aplicada às linhas do bloco, misturando os dados horizontalmente para aumentar a difusão.
- **MixColumns (Mistura de colunas):** Uma transformação linear é aplicada às colunas do bloco, utilizando operações em um campo finito $GF(2^4)$ (campo de Galois), promovendo maior espalhamento dos bits.
- **Rodadas:** O algoritmo executa duas rodadas completas dessas etapas (com MixColumns apenas na primeira), utilizando as subchaves derivadas.

A decifragem segue o caminho inverso, utilizando as transformações inversas de cada etapa (como a Inverse S-box, Inverse ShiftRows e Inverse MixColumns) e aplicando as subchaves na ordem inversa. Essa reversibilidade é garantida pela estrutura algébrica do algoritmo e pela forma como as operações foram projetadas para serem invertíveis.

3 Análise de Etapas do SAES

3.1 Expansão da chave

O processo de expansão da chave tem como objetivo gerar 3 chaves de 16 bits a partir de uma única chave de 16 bits que posteriormente vão ser utilizadas no processo de encriptação. Como é possível ver no código abaixo, a primeira chave é a própria chave passada por argumento. Posteriormente, para irmos criando as outras chaves, o algoritmo trabalha dividindo as chaves em duas partes de 8 bits, chamados aqui de w_n

Para gerar os primeiros 8 bits da segunda chave, o w_2 , usamos o w_0 (8 bits mais significativos da chave original) e aplicamos algumas operações nele, a primeira sendo um XOR com uma constante *RCON (round constant)* - um valor fixo para a primeira rodada de geração de chave.

Feito isso, realizaremos um outro XOR, mas agora com o resultado da função

`key_expansion_subnibble(rotate_nibble(`w_1`))` que consiste primeiro em rotacionar os 4 primeiros bits de w_1 com os últimos e aplicar a substituição com a *SBOX*, que é uma tabela do S-AES utilizada para substituir valores de 4 bits por outros pré-definidos. Tendo o w_2 em mãos, basta realizar a operação $w_2 XOR w1$ para obter o valor de w_3, e assim gerar a segunda chave. Por fim, para obter a terceira chave, basta repetir o processo realizado para gerar a segunda chave, mas ao invés de utilizar a chave original para iniciar o processo, é utilizada a chave 2; e a constante *RCON* também deve ser alterada para a devida constante da segunda rodada.

Código em C++ implementado que realiza o processo acima:

```
#define NIBBLE_MASK 0x0F
 uint8_t Sbox[16] =
 {
     0x9,0x4,0xA,0xB,
     0xD,0x1,0x8,0x5,
     0x6,0x2,0x0,0x3,
     0xC,0xE,0xF,0x7
 };
 uint8_t RCON[2] = { 0x80, 0x30 };
 uint8_t rotate_nibble(uint8_t word) {
     uint8_t upper = word << 4;
     uint8_t lower = (word >> 4) & NIBBLE_MASK;
     return upper | lower;
 }
 uint8_t key_expansion_subnibble(uint8_t word) {
     uint8_t upper = (word >> 4) & NIBBLE_MASK;
     uint8_t lower = word & NIBBLE_MASK;
```

```
    return Sbox[upper] << 4 | Sbox[lower];
}
 std::vector<uint16_t> saes_key_expansion(uint16_t key) {
    std::vector<uint16_t> keys (3, 0);
    keys[0] = key;
    uint8_t w0 = (keys[0] >> 8) & 0xFF;
    uint8_t w1 = keys[0] & 0xFF;
    uint8_t w2 = w0 ^ RCON[0] ^ key_expansion_subnibble(rotate_nibble(w1));
    uint8_t w3 = w2 ^ w1;
    keys[1] = w2 << 8 | w3;
    uint8_t w4 = w2 ^ RCON[1] ^ key_expansion_subnibble(rotate_nibble(w3));
    uint8_t w5 = w4 ^ w3;
    keys[2] = w4 << 8 | w5;
    return keys;
 }
```

3.2 Adição de chave da rodada

O processo de adição de chave da rodada é simples: consiste apenas na realização de um XOR entre o estado atual do SAES com a chave da rodada correspondente.

Esse processo é realizado 3 vezes durante o SAES: no começo do algoritmo e no fim de cada uma das duas rodadas. É importante ressaltar que o XOR é a operação escolhida por sua reversibilidade $(((AxorB)xorB) = A)$, e assim, garante-se que a segurança do algoritmo dependa do segredo da chave ao mesmo tempo que seja possível decifrar texto cifrado.

Código em C++ implementado para adição de chave de rodada:

```
std::vector<std::vector<uint8_t> > saes_add_round_key(std::vector<std::vector<
    uint8_t> > currentState, uint16_t key) {

    uint16_t currentStateNumber = convert_state_matrix_to_int(currentState);
    uint16_t newStateNumber = currentStateNumber ^ key;
    return convert_int_to_state_matrix(newStateNumber);
}
```

3.3 Substituição de *Nibbles*

O processo de substituição de *Nibbles* ocorre nas duas rodadas e funciona pegando o estado, que é uma matriz de 4 *Nibbles*, e trocando esses valores por valores tabelados pela *SBOX*.

Essa etapa introduz não linearidade ao SAES, ou seja, torna a relação entre a entrada e a saída imprevisível. Tendo papel fundamental, já que sem a substituição de *Nibbles*, o algoritmo seria apenas uma série de operações lineares, o que é facilmente reversível e inseguro.

Implementação em C++ da Substituição de *Nibbles*:

```
std::vector<std::vector<uint8_t> > saes_nibble_substitution(std::vector<std::
    vector<uint8_t> > currentState, uint8_t* Sbox){
    std::vector<std::vector<uint8_t> > result(2, std::vector<uint8_t>(2, 0));
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = Sbox[currentState[i][j]];
        }
    }
 return result;
 }
```

3.4 Troca de Linhas

O processo de troca de linhas ocorre logo após a troca de *Nibbles* em cada rodada e tem o objetivo de aumentar a dispersão do algoritmo. É um processo simples que pega o estado do S-AES e troca as duas *Nibbles* da segunda linha da matriz de lugar.

Código em C++ que implementa a Troca de Linhas:

```
std::vector<std::vector<uint8_t> > saes_shift_rows(std::vector<std::vector<
    uint8_t> > currentState) {
    std::vector<std::vector<uint8_t> > result(2, std::vector<uint8_t>(2, 0));

    // primeira linha igual
    result[0][0] = currentState[0][0];
    result[0][1] = currentState[0][1];

    // inverte a segunda linha
    result[1][0] = currentState[1][1];
    result[1][1] = currentState[1][0];
 return result;
 }
```

3.5 Mistura de Colunas

O processo de mistura de colunas ocorre na primeira rodada, depois da troca de linhas, e é o processo mais complexo do algoritmo S-AES pois consiste em multiplicar o estado por uma outra matriz:

$$\begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix}$$

Contudo, as operações são realizadas em campo de Galois e devem ser reduzidas módulo (x^4+x+1).

Devido à complexidade da etapa, foi utilizado um trecho de código copiado de outro repositório público que o implementava.

Créditos: https://github.com/mostsfamahmoud/Simplified-AES

```
#define GF_MUL_PRECOMPUTED_TERM    0x03

 uint8_t GF_MultiplyBy(uint8_t data, uint8_t mulValue) {
     uint8_t result = data;

    switch (mulValue) {
        case 2: /* Used in SAES Decryption */
            if ((result >> 3) == 1) {
                result = ((result << 1) & 0x0F) ^ GF_MUL_PRECOMPUTED_TERM;
            } else {
                 result = (result << 1) & 0x0F;
            }
        break;
    case 4: /* Used in SAES Encryption */
        for (int i = 0; i < 2; i++) {
            if ((result >> 3) == 1) {
                result = ((result << 1) & 0x0F) ^ GF_MUL_PRECOMPUTED_TERM;
            } else {
        result = (result << 1) & 0x0F;
         }
    }
    break;
     case 9: /* Used in SAES Decryption */
        for (int i = 0; i < 3; i++) {
         if ((result >> 3) == 1) {
          result = ((result << 1) & 0x0F) ^ GF_MUL_PRECOMPUTED_TERM;
         } else {
          result = (result << 1) & 0x0F;
         }
     }
        result = result ^ data;
        break;
```

```
    default:
        break;
    }
     return result;
}
std::vector<std::vector<uint8_t> > saes_mix_columns(std::vector<std::vector<
    uint8_t> > currentState) {
    std::vector<std::vector<uint8_t> > result(2, std::vector<uint8_t>(2, 0));

     for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = currentState[i][j] ^ GF_MultiplyBy(currentState[(i
                + 1) % 2][j], 4);
     }
    }
    return result;
}
```

3.6 Decriptação

Para que seja possível obter a mensagem original, é necessário que todas as operações realizadas na encriptação sejam desfeitas e para isso, a seguinte análise deve ser realizada:

As transformações realizadas durante o processo de cifragem no SAES não são comutativas entre si. Em termos matemáticos, dizemos que, para duas transformações f e g:

$$f \circ g \neq g \circ f$$

Ou seja, a ordem de aplicação das funções afeta diretamente o resultado final da cifragem.

Se modelarmos o processo de cifragem como uma composição de funções, temos:

$$C = f_4 \circ f_3 \circ f_2 \circ f_1(M)$$

Onde:

- $f_1 =$ AddRoundKey
- $f_2 =$ SubNibbles
- $f_3 =$ ShiftRows
- $f_4 =$ MixColumns

Devido à não comutatividade da composição $f_i \circ f_j$, a operação de decriptação deve seguir a ordem inversa:

$$M = f_1^{-1} \circ f_2^{-1} \circ f_3^{-1} \circ f_4^{-1}(C)$$

Contudo, é fácil perceber que $f_1^{-1} = f_1$ e $f_3^{-1} = f_3$ (Adição de Chave e Troca de Linhas são operações que, se aplicadas duas vezes, retornam o resultado original), e, portanto, os únicos processos que mudam são a substituição de *Nibbles* - que usa a *InverseSBOX* ao invés da *SBOX*;

```
uint8_t InverseSbox[16] =
    {
    0xA,0x5,0x9,0xB,
    0x1,0x7,0x8,0xF,
    0x6,0x0,0x2,0x3,
    0xC,0x4,0xD,0xE
    };
```

E a mistura de colunas que passa a realizar a multiplicação do estado com a matriz inversa da matriz (para o campo de Galois) usada na encriptação:

$$\begin{bmatrix} 9 & 2 \\ 2 & 9 \end{bmatrix}$$

Assim, temos o seguinte código C++ para decriptação:

```
std::vector<std::vector<uint8_t> > saes_inverse_mix_columns(std::vector<std::
    vector<uint8_t> > currentState) {
    std::vector<std::vector<uint8_t> > result(2, std::vector<uint8_t>(2, 0));
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = GF_MultiplyBy(currentState[i][j], 9) ^
                GF_MultiplyBy(currentState[(i + 1) % 2][j], 2);
        }
    }
    return result;
}

uint16_t saes_decrypt(uint16_t cipherText, uint16_t key) {
    std::vector<std::vector<uint8_t> > state = convert_int_to_state_matrix(
        cipherText);
    std::vector<uint16_t> keys = saes_key_expansion(key);
    state = saes_add_round_key(state, keys[2]);
    // Primeira rodada
    state = saes_shift_rows(state);
    state = saes_nibble_substitution(state, InverseSbox);
    state = saes_add_round_key(state, keys[1]);
    // Segunda rodada
    state = saes_inverse_mix_columns(state);
    state = saes_shift_rows(state);
    state = saes_nibble_substitution(state, InverseSbox);
    state = saes_add_round_key(state, keys[0]);
    uint16_t result = convert_state_matrix_to_int(state);
    return result;
}
```

4 Modos de Operação do SAES

Algoritmos de cifra de bloco operam sobre blocos de tamanho fixo. No caso do SAES, 16 bits. Para lidar com mensagens maiores que esse tamanho, são definidos modos de operação que especificam como os blocos devem ser cifrados em sequência. A seguir, descreveremos os principais modos de operação: ECB, CBC, CFB, OFB e CTR com ênfase em suas propriedades de segurança e aplicabilidade.

Modo ECB (Electronic Codebook)

O modo ECB é o mais simples entre os modos de operação. Nele, o texto claro é dividido em blocos independentes que são cifrados separadamente usando a mesma chave.

$$C_i = E_K(P_i) \tag{1}$$

Embora ofereça vantagens como paralelismo na cifragem e baixa complexidade de implementação, o ECB apresenta sérias vulnerabilidades de segurança, pois blocos de texto claro idênticos produzem blocos cifrados idênticos. Isso permite que padrões estruturais sejam preservados no texto cifrado, tornando-o inadequado, como demonstrado praticamente na análise de resultados.

Código C++ que implementa o modo ECB:

```
std::string ecb_saes_decrypt(std::string base64Input, std::string key) {
    // Decode the input
```

```
    std::vector<BYTE> decodedInput = base64_decode(base64Input);
    std::vector<uint16_t> inputWords;
    // transform the input into 16-bit words
    for (size_t i = 0; i < decodedInput.size() - 1; i += 2) {
    char firstChar = decodedInput[i];
    char secondChar = decodedInput[i + 1];
    uint16_t word = (uint8_t) firstChar << 8 | (uint8_t) secondChar;
    inputWords.push_back(word);
    }
    // Decode the key
    std::vector<BYTE> decodedKey = base64_decode(key);
    // Check if the key is 16 bits
    if (decodedKey.size() > 2) {
    std::cout << "A chave deve ter 16 bits" << std::endl;
    return "";
    }
    // Transform the key into a 16-bit word
    uint16_t keyWord = 0;
    for (size_t i = 0; i < decodedKey.size(); i++) {
    keyWord = (keyWord << 8) | decodedKey[i];
    }
    std::vector<uint16_t> outputWords(inputWords.size());
    for (size_t i = 0; i < inputWords.size(); i++) {
    outputWords[i] = saes_decrypt(inputWords[i], keyWord);
    }
    // base64_encode(outputWords);
    std::string outputStr;
    for (size_t i = 0; i < outputWords.size(); i++) {
    char firstChar = (char) (outputWords[i] >> 8) & 0xFF;
    char secondChar = (char) (outputWords[i] & 0xFF);
    outputStr += firstChar;
    outputStr += secondChar;
}
// Encode the output string to base64
    std::string output = base64_encode((unsigned char*) outputStr.c_str(),
        outputStr.size());
    return output;
}
```

Modo CBC (Cipher Block Chaining)

O modo CBC melhora a segurança ao encadear os blocos cifrados. Cada bloco de texto claro é combinado, por meio de uma operação XOR, com o bloco cifrado anterior antes de ser cifrado. O primeiro bloco utiliza um vetor de inicialização (IV) aleatório.

$$C_0 = E_K(P_0 \oplus IV), \tag{2}$$
$$C_i = E_K(P_i \oplus C_{i-1}) \tag{3}$$

Esse encadeamento garante que blocos idênticos em posições diferentes produzam blocos cifrados distintos. CBC oculta padrões do texto claro, tornando ataques por análise estatística muito mais difíceis. Contudo, ele não permite paralelismo na cifragem (somente na decifragem) e exige que o IV seja único e imprevisível para cada mensagem.

Modo CFB (Cipher Feedback Mode)

O modo CFB transforma uma cifra de bloco em uma cifra de fluxo, permitindo a cifragem de blocos menores (por exemplo, bytes ou nibbles). No caso simplificado onde o segmento processado tem o tamanho do bloco, opera da seguinte forma: O bloco cifrado anterior (ou o IV, no início) é cifrado

e seu resultado é então combinado via XOR com o bloco de texto claro para gerar o bloco de texto cifrado.

$$C_0 = P_0 \oplus E_K(IV), \tag{4}$$
$$C_i = P_i \oplus E_K(C_{i-1}) \tag{5}$$

O CFB mantém muitas das vantagens do CBC, incluindo a não repetição de blocos idênticos. No entanto, erros de transmissão se propagam: um erro em um bit afeta dois blocos consecutivos durante a decifragem.

Modo OFB (Output Feedback Mode)

O modo OFB também transforma o SAES em uma cifra de fluxo, mas difere do CFB ao não utilizar o texto cifrado no realimentador. Em vez disso, o bloco cifrado anterior (ou IV) é continuamente cifrado para gerar uma sequência pseudoaleatória de blocos, que é combinada com o texto claro via XOR.

$$R_0 = IV, \tag{6}$$
$$R_i = E_K(R_{i-1}), \tag{7}$$
$$C_i = P_i \oplus R_i \tag{8}$$

OFB é útil em ambientes com ruído, pois erros não se propagam: um erro afeta apenas o bloco correspondente. Contudo, como o mesmo IV e chave geram o mesmo fluxo, é essencial que o IV seja único para cada mensagem. Não seguir essa regra compromete totalmente a segurança.

Modo CTR (Counter Mode)

O modo CTR também converte o SAES em uma cifra de fluxo. Nele, um contador (CTR), geralmente iniciado por um nonce (número único), é cifrado a cada rodada, e seu resultado é combinado com o texto claro via XOR:

$$C_i = P_i \oplus E_K(\text{CTR} + i) \tag{9}$$

O CTR é altamente eficiente e permite cifrar e decifrar em paralelo, uma vantagem importante em implementações modernas. Também não sofre de propagação de erros. No entanto, reutilizar o mesmo nonce com a mesma chave é catastrófico, permitindo que atacantes recuperem a mensagem por simples XOR entre dois textos cifrados.

4.1 Comparativo de Segurança entre os Modos

Modo	Paralelismo	Oculta padrões?	Propagação de erro	Requer IV/nonce
ECB	Sim	Não	Não	Não
CBC	Não (na cifragem)	Sim	Sim	Sim (IV único)
CFB	Não	Sim	Sim	Sim (IV)
OFB	Parcial	Sim	Não	Sim (IV)
CTR	Sim	Sim	Não	Sim (nonce)

5 Implementação e Análise de Resultados

Para implementar o SAES e testar seus resultados com os dois modos de operação, optamos por desenvolver o código utilizando a linguagem C++ e mantendo o arquivo num repositório hospedado na plataforma *GitHub*, no seguinte link: Repositório no GitHub (https://github.com/cauetrd/Simplified-AES). O acesso ao link permite a inspeção de todo o código, onde estão disponíveis em arquivo `README`, instruções para utilização do programa.

A execução do programa exibirá no terminal o seguinte menu:

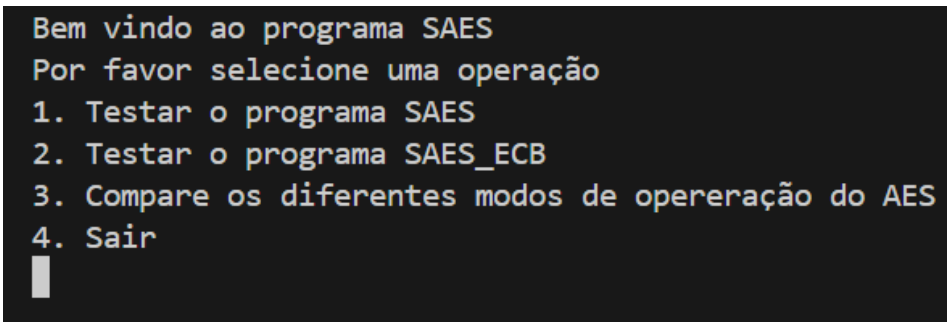

Figura 1: Menu para execução do programa

Teste de Encrypt-Decrypt

Escolhemos a mensagem "ok"(b2s= em base64) para ser cifrada sob chave "ti"(dGk= em base64). O resultado encontrado foi:

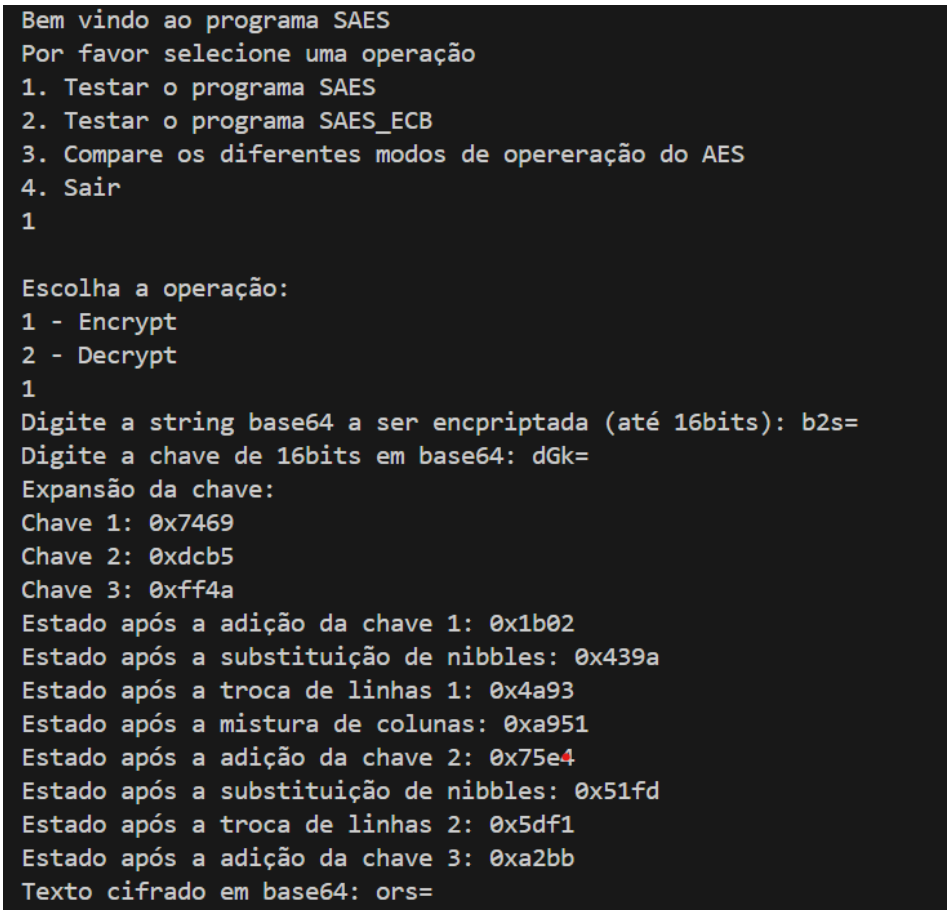

Figura 2: Encriptação

Em seguida, pegamos o texto cifrado resultante "ors="e o utilizamos na decriptação:

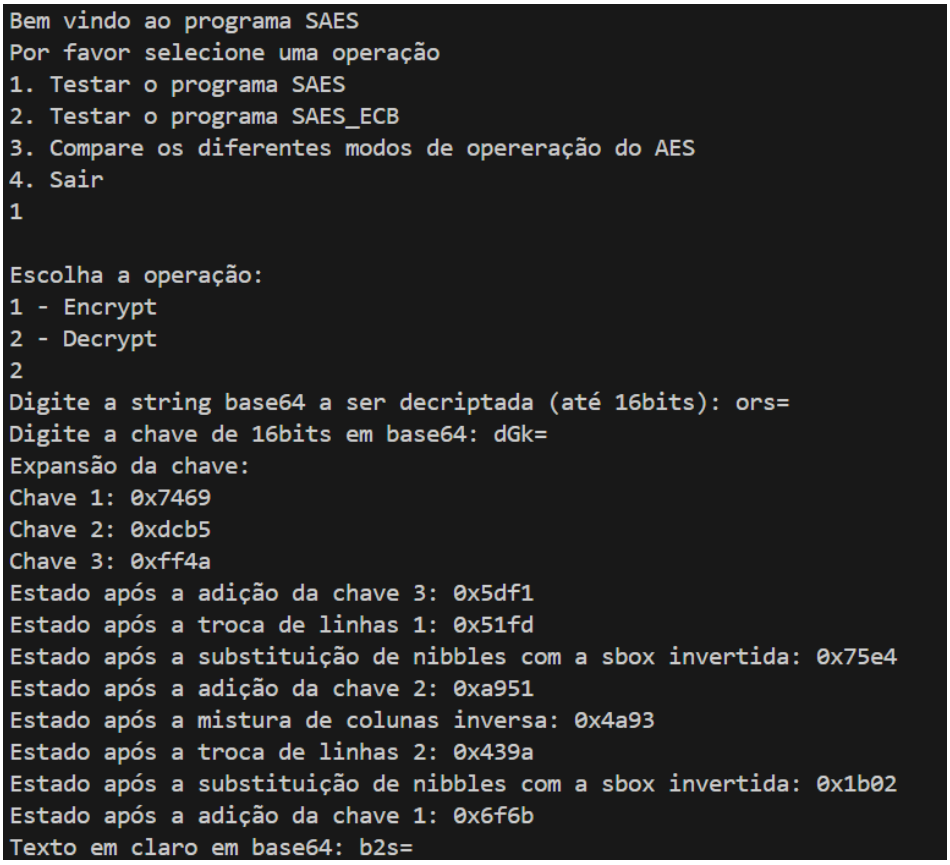

Figura 3: Decriptação

Obtendo o texto original "e comprovando, dessa forma, o funcionamento do algoritmo SAES.

Teste modo ECB

O modo ECB para o SAES foi implementado manualmente pelo código apresentado anteriormente, e para testá-lo, foi utilizado o plain text: "Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet. "e a mesma chave "ti".

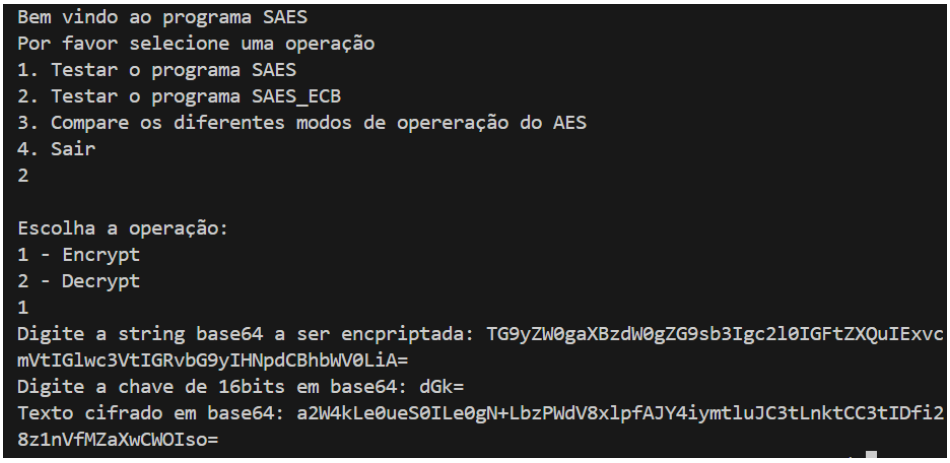

Figura 4: Encriptação ECB

O texto cifrado obtido, convertido de base64 para hexadecimal, gera:

6b65b890 b7b4b9e4 b420b7b4 80df8b6f 33d6755f 319697c0 258e22ca 6b65b890 b7b4b9e4 b420b7b4 80df8b6f 33d6755f 319697c0 258e22ca 5343

Analisando esse texto cifrado em hexadecimal, conseguimos observar o maior problema do modo ECB: blocos iguais resultam em textos cifrados iguais, o que pode levar ao reconhecimento de padrões em um texto, ou, se formos encriptar imagens, as imagens continuam com o mesmo contorno de objetos, por exemplo.

Comparação de modos do AES

Utilizamos a biblioteca Cryptopp para comparar os diferentes modos de execução do AES, ao rodar um script que encripta uma mesma mensagem nos diferentes modos. São exibidos os textos cifrados resultantes de cada modo e, posteriormente, uma tabela com o tempo de execução médio ao executar cada modo 1000 vezes. Devido ao tamanho dos textos gerados, é mais adequado que o leitor execute o programa em sua máquina caso deseje utilizar o plain text definido por nós.

Segue abaixo um exemplo de uso para string pequena:

```
Bem vindo ao programa SAES
Por favor selecione uma operação
1. Testar o programa SAES
2. Testar o programa SAES_ECB
3. Compare os diferentes modos de opereração do AES
4. Sair
3
rafia?
1. Testar com uma entrada padrão
2. Testar com uma entrada específica
2
O Vasco da Gama eh o maior time do Brasil
ECB mode:
CipherText base64: gw+6kyN1DIcQDOOuIOjNFR/QrWGeL2Sx1I2qpYf0RHYTy6ZXl5ZEPu
QFzKzAaPBC
CBC mode:
CipherText base64: nMnHyxofeWgdrQSUwc/jqUBMNPtcF2KmFfl7l3aCNP5xD8/xt04N11
u9yM/vHIHO
CFB mode:
CipherText base64: NT0z1pfITXOJDtLoEOTUpqgzFD52baRTPlarcogizqfLRsWt1CyzaX
k=
OFB mode:
CipherText base64: NT0z1pfITXOJDtLoEOTUpmLDa294UHz0XE7ZbRe4vXfYAVdQXeZXwX
U=
CTR mode:
CipherText base64: NT0z1pfITXOJDtLoEOTUph8y4Vht4zxJ6oHJ3LkNTimUsBb9ApjyD/
A=
```

Figura 5: Teste comparação de modos

É interessante ressaltar que o início do texto cifrado dos modos OFB, CTR e CFB é idêntico. Isso ocorre pois utilizamos o mesmo IV para os três modos, além de inicializar o nonce do CTR como 0. Contudo, após os bytes iniciais, o cifrado começa a se diferenciar devido ao diferente funcionamento dos métodos.

Outro ponto importante de ser observado é a média de tempo de duração após 1000 execuções de cada modo com uma string de 5600 caracteres. É possível que haja variação entre os números apresentados aqui e os números que possam ser obtidos em outra execução do programa, devido à maior variância atribuída a uma amostra pequena, mas a proporção entre resultados deve se manter similar.

Podemos ver que mesmo com modos de operação diferentes os resultados foram muito parecidos. Isso ocorre pois provavelmente a biblioteca utilizada não otimiza o modo ECB ou CTR para usarem paralelismo.

Tabela 1: Tempos de execução por modo de operação

Modo	**Tempo (ms)**
ECB	71
CBC	75
CFB	75
OFB	76
CTR	72

6 Conclusão

A implementação do SAES foi fundamental para compreender os princípios do AES em sua forma completa. Com uma estrutura mais simples, o SAES permitiu explorar, de forma prática, os processos de substituição, permutação, geração de chaves e operações no campo finito $GF(2^4)$ que são a base do funcionamento da criptografia simétrica moderna. Essa abordagem facilitou o aprendizado dos conceitos essenciais de segurança e reforçou a importância da combinação entre teoria e prática na área da criptografia.

Além disso, a análise dos modos de operação demonstrou como diferentes estratégias de aplicação do algoritmo podem impactar diretamente a segurança e a integridade dos dados cifrados. Modos como ECB e CBC, por exemplo, apresentam características e vulnerabilidades distintas que devem ser consideradas na escolha do método mais adequado para cada cenário. Dessa forma, o estudo contribuiu para uma visão mais crítica e informada sobre o uso seguro de algoritmos criptográficos em sistemas reais.

7 Referências

MUSA, Mohammad; SCHAEFER, Edward; WEDIG, Stephen. **A simplified AES algorithm and its linear and differential cryptanalyses**. Cryptologia, [S. l.], v. 27, p. 148-177, abr. 2003.

Mostafa Mahmoud, "**Função `saes_mix_columns` do repositório Simplified-AES**"GitHub, 2023. [Online]. disponível em: https://github.com/mostsfamahmoud/Simplified-AES