S-DES, ECB e CBC

Gabriel Henrique do Nascimento Neres
Arthur Diehl Barroso

June 15, 2025

Abstract

Texto detalhando o desenvolvimento de uma versão simplificada do algoritmo de criptografia AES, denominado de S-AES, detalhando a implementação do modo de operação ECB. Além disso, é abordado as diferenças entre 5 modos de operação utilizando a biblioteca pyCryptoDome do python.

Palavras-chave: Criptografia, AES, modos de operação.

Esse trabalho irá implementar e detalhar as etapas de confecção do algoritmo de criptografia simétrica S-AES, um modelo simplificado do algoritmo AES, com uso do modo de operação ECB. Em um segundo momento serão abordados 5 modos de operação que podem ser utilizados pelo algoritmo AES, sendo eles, **ECB**, **CBC**, **OFB**, **CFB** e **CTR**.

Para compreender melhor a implementação que está sendo feita, serão repassados os resultados de cada função desenvolvida baseado nos dados de entrada a baixo:

- Chave: 1010011100111011
- Mensagem: 0110 1111 0110 1011 ("ok" em ASCII)

1 S-AES

O algoritmo de encriptação que estamos denominando de **S-AES** é uma forma simplificada do modelo AES que fará uso de uma chave de tamanho 16 bits, mensagens em blocos de 16 bits e a redução do número de rodadas

para 2. A título de comparação, vale trazer que o **AES** possui blocos de 128 bits, chaves que podem ser de 128, 192 ou 256 bits e o número de rodadas para cada versão é, respectivamente, 10, 12 e 14 rodadas.

OBS: Neste trabalho a palavra "word" será utilizada para indicar um bloco de 1 byte (8 bits), o que difere da convenção de uma "word" indicar 4 bytes (32 bits). O termo foi mantido para permitir melhor correlação com a documentação do AES.

1.1 Funções auxiliares

As primeiras coisas que serão implementadas serão as funções responsáveis por fazer conversões entre um int de 16 bits e uma matriz 2x2 de nibbles, que denominaremos essa matriz de **estado**.

Para essa conversão serão utilizados o operador ">>" com um **AND** com 0xF para conseguir quebrar o valor de 16 bits em 4 partes de 4 bits e o operador "<<" com uso do **OR** para reunir os 4 nibbles novamente em um inteiro de 16 bits.

A lógica utilizada na conversão para nibbles é manter, nessa ordem, os 4, 12, 8e 16 bits mais significativos e realizar um **AND** com 0xF para obter os 4 bits menos significativos do conjunto mantido, assim somos capazes de dividir o valor de 16 bits em 4 partes.

```
def int_to_state(block: int) -> list[list[int]]:
  return [
    [(block >> 12) & 0xF, (block >> 4) & 0xF],
    [(block >> 8) & 0xF, block & 0xF]
  ]
```

Entrando com a chave de exemplo, teremos como saída a seguinte matriz:

$$\begin{bmatrix} 10 & 3 \\ 7 & 11 \end{bmatrix}$$

O processo inverso é feito seguindo a lógica de incluir zeros a direita de cada parte da matriz para obter um inteiro de 16 bits, unindo cada um dos 4 campos utilizando um **OR** após a inclusão dos zeros.

```
def state_to_int(state: list[list[int]]) -> int:
  return (
    ((state[0][0] & 0xF) << 12) |
    ((state[1][0] & 0xF) << 8)  |
    ((state[0][1] & 0xF) << 4)  |
```

```
    (state[1][1] & 0xF)
  )
```

Em comparação com o algoritmo **AES**, a implementação do estado segue uma estrutura diferente. No AES a matriz que foi denominada como estado possui tamanho 4x4 e cada parte possui comprimento de 8 bits.

1.2 Expansão da chave e round-key

Ao implementar o algoritmo, serão utilizadas 3 chaves ao longo do processo de encriptação. A primeira, utilizada antes da primeira rodada, será a própria chave passada ao algoritmo, já as outras duas serão derivadas utilizando uma função de expansão.

Essa função irá seguir a lógica apresentada na imagem 1 que poderá ser simplificada nas seguintes etapas:

- Reunir as colunas da matriz em words;
- Executar a função g com a word 1;
- Realizar o XOR entre word 0 e o resultado da g;
- Realizar o XOR entre word 1 e a word 2 (gerada pelo XOR anterior);

Esse processo resultará na chave utilizada pela primeira rodada do algoritmo, a outra cave será obtida utilizando a chave da primeira rodada na mesma função de expansão.

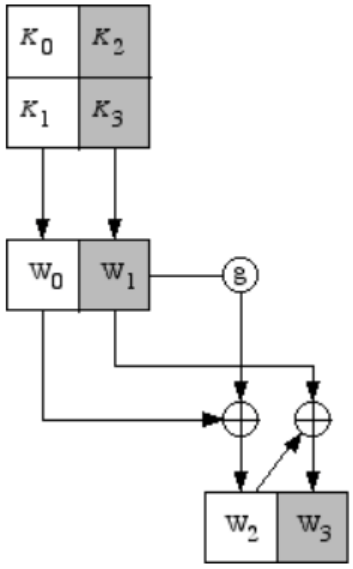
A função de expansão foi implementada da seguinte maneira:

```
def expand_key(prev_key: list[list[int]], round_num: int) ->
    list[list[int]]:
  w0 = [prev_key[0][0], prev_key[1][0]]
  w1 = [prev_key[0][1], prev_key[1][1]]

  g_w1 = g(w1, round_num)

  w2 = [w0[0] ^ g_w1[0], w0[1] ^ g_w1[1]]
  w3 = [w2[0] ^ w1[0], w2[1] ^ w1[1]]

  return [[w2[0], w3[0]], [w2[1], w3[1]]]
```

Já a função g tem um conjunto de operações no seu interior, que são:

- Divide a word nas suas duas metades;

Figure 1: Processo de expansão da chave



- Inverte a posição das metades;
- Converte cada metade utilizado o conjunto de S-box do S-AES;
- Realiza o XOR entre cada metade pela constante de round, definida por x^{j+2}.

Assim, o código utilizado ficou como:

```
def g(byte: list[int], round_num: int) -> list[int]:
  rcon = RCON[round_num]

  tmp_byte = sub_word(rot_word(byte))

  return [
    tmp_byte[0] ^ rcon[0],
    tmp_byte[1] ^ rcon[1]
  ]
```

Para exemplificar o processo de expansão da chave, utilizando a chave definida para esse trabalho teremos os resultados para as chaves 1, 2 e 3, além das etapas intermediárias do processamento apresentadas na imagem 1.2 com valores em decimal e hexadecimal (prefixada com 0x).

```
G Round 0: [11, 11]
G Round 1: [6, 10]
First key: 0xa73b
Round 1 key: 0x1c27
Round 2 key: 0x7651
```

No **AES**, o número de words utilizadas para a expansão da chave, nos modelos padrões, variam de 4 a 8 words(4 bytes cada), enquanto que no **S-AES** foi utilizado apenas 2.

1.2.1 Round Key

Ao longo da encriptação, as chaves que foram obtidas ao longo do processo de expansão serão utilizadas. Esse momento será a Round Key, onde a chave passada irá realizar um XOR com o estado (conceito definido na seção 1.1) atual da encriptação retornando uma nova versão do estado, por isso esse processo é denominado de "Adição de chave de rodada" (Add Round Key).

Na nossa versão simplificada, esse processo é realizado apenas em 3 momentos, no inicio da encriptação com o *plaintext* e a chave passada, ao fim da primeira rodada com a primeira chave gerada pela expansão e ao fim da segunda rodada com a segunda chave da expansão.

No modelo tradicional do AES esse processo se repetiria em cada rodada, da mesma forma que a expansão da chave também seria repetido sucessivas vezes para atender ao valor de rodadas necessárias.

1.3 SubNibbles, ShiftRows e MixColumns

Tendo todas as chaves em mãos, podemos partir para as principais funções ligadas a encriptação do algoritmo alvo desse estudo.

1.3.1 SubNibbles

Essa operação tem como objetivo realizar a substituição dos nibbles presentes no estado recebido por meio de uma tabela de S-boxes, que busca traduzir a operação matemática proposta pelo **AES** que envolvem a inversão do polinômio (associado com os nibbles) com um elemento de GF(16) e finalmente realizando uma multiplicação por uma matriz e adiciona um vetor.

A tabela de conversão de S-Box utilizada está na imagem 2

Figure 2: Tabela de S-boxes

nibble	S-box(nibble)	nibble	S-box(nibble)
0000	1001	1000	0110
0001	0100	1001	0010
0010	1010	1010	0000
0011	1011	1011	0011
0100	1101	1100	1100
0101	0001	1101	1110
0110	1000	1110	1111
0111	0101	1111	0111

1.3.2 ShiftRows

No nosso modelo simplificado, nessa etapa da encriptação os elementos da segunda linha da matriz de estados são trocados de posição. No **AES**, o shift das linhas é feito seguindo o número da linha menos 1, isso significa que a primeira linha não é deslocada, a segunda uma única posição a esquerda, a terceira 2 posições, e a quarta 3 posições.

1.3.3 MixColumns

Essa é a última função que precisaremos descrever para o processo de encriptação, que é feita por uma multiplicação em **GF(16)** entre a matriz de estado e a matriz $\begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix}$. Primeiramente, podemos definir a multiplicação em **GF(16)** pelo seguinte trecho de código:

```
def gf16_mul(a:int, b:int) -> int:
  result = 0
  for _ in range(4):
    if b & 1:
      result ^= a
    b >>= 1
    a <<= 1
    if a & 0b10000:
      a ^= 0x13
  return result & 0xF
```

Já o processo de misturar as colunas da matriz de estado, podemos utilizar o seguinte trecho de código, que basicamente simula a multiplicação de matrizes:

```python
def mix_columns(state: list[list[int]]) -> list[list[int]]:
  return [
    [
      gf16_mul(1, state[0][0]) ^ gf16_mul(4, state[1][0]),
      gf16_mul(1, state[0][1]) ^ gf16_mul(4, state[1][1])
    ],
    [
      gf16_mul(4, state[0][0]) ^ gf16_mul(1, state[1][0]),
      gf16_mul(4, state[0][1]) ^ gf16_mul(1, state[1][1])
    ]
  ]
```

Já no **AES**, a matriz utilizada é

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

, mantendo o mesmo processo de multiplicação em **GF(16)** para obter o resultado da operação.

1.4 Round 1 e 2

Tendo todas as funções necessárias definidas, além de dos conceitos relacionados com cada um dos mesmos, o processo de encriptação com a S-AES pode ser realizado com sucesso. Desse modo, essa parte do trabalho será dedicada a relembrar e encadear todo o processo de encriptação além de apresentar uma sequencia de resultados intermediários que levarão do *plaintext* ao *ciphertext*.

Nesse sentido, podemos definir o fluxo do algoritmo S-AES como sendo o seguinte:

1. Realizar a expansão da chave;
2. Utilizar o Add Round Key com *plaintext* e a chave de entrada;
3. Inicia a primeira rodada com o SubNibbles do estado obtido anteriormente;
4. Seguindo com o estado obtido é utilizado o ShiftRows;
5. Seguindo a mesma sequência, utiliza-se o MixColumns;

6. Finalizando a primeira rodada, realiza-se o Add Round Key com o estado obtido e a chave da primeira rodada (obtida com a expansão);

7. Inicia a segunda rodada com o SubNibbles;

8. Seguido com o ShiftRows;

9. Diferente da primeira rodada, já finaliza com o Add Round Key com a chave da segunda rodada;

A pequena diferença que ocorre entre a primeira e a segunda rodada, removendo a etapa de mix columns, se deve a intenção de simplificar o processo de decriptação. No **AES**, além das diferenças já citadas ao longo das análises, não possui essa remoção do mix columns na última rodada.

Com base nas observações feitas, podemos observar com melhor compreensão os resultados retornados pelo algoritmo resultado da pesquisa até o ponto atual, presenta na imagem 1.4.

```
First key: 0xa73b
Round 1 key: 0x1c27
Round 2 key: 0x7651
Add round key: 0xc850

Round 1
Substitute nibbles: 0xc619
Shift rows: 0xc916
Mix columns: 0xeca2
Add round key: 0xf085

Round 2
Substitute nibbles: 0x7961
Shift rows: 0x7169
Add round key: 0x0738

Final results
Hex encoded ciphertext: 0x0738
Base64 encoded ciphertext: Bzg=
```

1.5 Modo de operação: ECB

A técnica utilizada por esse modo de operação é bem simples, consistindo em dividir a entrada fornecida em blocos de tamanho fixo para serem repassados a função de criptografia desejada.

No caso da **S-AES**, a entrada foi definida como blocos de 16 bits incluindo um padding de 0s a esquerda no último bloco, em caso de não possuir os 16 bits necessários. No **AES** o valor dos blocos seria de 128 bits. O código utilizado para essa tarefa foi o seguinte:

```
def encrypt_saes_ecb(text:str, key:int):
  ciphertext = bytearray()
  for i in range(0, len(text), 2):
    block = text[i:i+2]
    if len(block) < 2:
      block = block.ljust(2, '\x00')
    ct = saes(block, key)
    ciphertext += ct.to_bytes(2, byteorder="big")
  return b64.b64encode(ciphertext).decode("utf-8")
```

Nele pode ser percebido a divisão em blocos de 2 letras(que totaliza 16 bits) e o padding de 0s a esquerda. Para exemplificar, caso a mensagem passada ao algoritmo seja **okk** a conversão resultante será o array [0110111101101011, 0000000001101011].

Vale lembrar que esse **não** é o valor retornado pela função, pois nessa função está inclusa a criptografia utilizando o algoritmo S-AES. Desse modo, o resultado final será a concatenação de todas os blocos criptografados convertidos no formato da base 64 (melhor legibilidade).

2 Modos de operação

Nessa etapa do estudo, será feita a comparação de 5 modos de operação utilizando o algoritmo **AES**. A biblioteca escolhida para a utilizar as implementações desses modos de operação foi a pyCryptoDome. Nesse primeiro momento, serão detalhados a ideia por trás de cada modo de operação e o resultado obtido em cada uma.

Para melhor comparação, foram utilizadas duas frases a serem encriptadas, para explorar possíveis diferenças no tamanho da frase na aleatoriedade gerada. A primeira foi a mesma utilizada no S-AES, a palavra "ok", e a segunda foi a frase "Uma frase mais comprida para permitir uma melhor comparação entre os diferentes modelos de modo de operação existentes, utilizando o algoritmo AES." que possui 147 caracteres de comprimento (ou 1176 bits).

2.1 ECB - Electronic CodeBook

Como foi definido anteriormente nesse trabalho, essa função consiste em receber um texto de entrada e converte-lo em blocos de texto de tamanho fixo (no caso do S-AES 16 bits) para serem criptografados e concatenados em uma única mensagem final.

Figure 3: ECB com frase curta

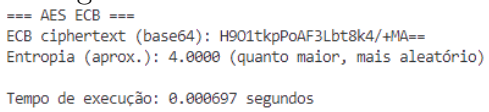
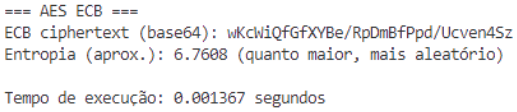


Figure 4: ECB com frase longa



Com base no que pode ser visto nas imagens 3 e 4, podemos observar que o grau de aleatoriedade teve um aumento de pouco mais de 50% da frase curta para a frase longa e o tempo de processamento foi dobrado. Visto que esse modo de operação possui padding, o tamanho da frase curta foi de 128 bits e a frase longa 1280 bits, logo é 10 vezes maior e dessa forma os aumentos relatados são bem menores do que o aumento da própria frase.

2.2 CBC - Cipher-block chaining

Nesse modo de operação, além da mensagem dividida em blocos de X bits (fixos) e da chave de X bits também será recebida um valor inicial (IV), que será utilizado para realizar o XOR com o primeiro bloco da mensagem a ser encriptada. Esse modo de operação utiliza o bloco cifrado anterior para realizar um XOR com a entrada seguinte.

Um problema presente nesse modo de operação é a propagação de erros caso algum bit seja alterado na transmissão, afetando toda a cadeia de decriptação da mensagem.

Com base no que pode ser visto nas imagens 5 e 6, podemos observar que o grau de aleatoriedade e o tempo de processamento teve aumento similar ao observado no ECB.

Figure 5: CBC com frase curta

```
=== AES CBC ===
CBC ciphertext (base64): LoBskEZIMOcH2bYJ6sKFMw==
Entropia (aprox.): 4.0000 (quanto maior, mais aleatório)

Tempo de execução: 0.000337 segundos
```

Figure 6: CBC com frase longa

```
=== AES CBC ===
CBC ciphertext (base64): ybiQUU4Yxg6hSWIuHefAbBgoTAKdtEOj
Entropia (aprox.): 6.7264 (quanto maior, mais aleatório)

Tempo de execução: 0.000770 segundos
```

2.3 CFB - Cipher feedback

Nesse modo de operação, a entrada é cifrada de forma similar a feita no CBC, realizando XOR da entrada a ser encriptada com outro valor. A principal diferença está em cifrar o valor inicial ou o bloco anterior usando o **AES** antes de realizar o XOR com o próximo bloco em texto puro. A cifragem pode ser descrita com o seguinte exemplo, sendo C o texto cifrado e P o *plaintext*:

- $C_1 = \text{AES}(IV)$ XOR P_1
- $C_2 = \text{AES}(C_1)$ XOR P_2
- $C_3 = \text{AES}(C_2)$ XOR P_3
- ...

Outra característica interessante é ser capaz de aceitar blocos de entrada de tamanhos variados, não se fazendo necessário o uso de padding. No entanto, possui o mesmo problema de

Figure 7: CFB com frase curta

```
=== AES CFB ===
CFB ciphertext (base64): cqI=
Entropia (aprox.): 1.0000 (quanto maior, mais aleatório)

Tempo de execução: 0.000316 segundos
```

Com base no que pode ser visto nas imagens 7 e 8, podemos observar que o tempo de processamento teve aumento similar ao observado nos outros modos, mas a aleatoriedade do texto foi muito menor no texto curto,

Figure 8: CFB com frase longa

```
=== AES CFB ===
CFB ciphertext (base64): GOIEYS0oESX1EY6LNbr6amoqHTGrrx76l
Entropia (aprox.): 6.6124 (quanto maior, mais aleatório)

Tempo de execução: 0.000709 segundos
```

isso se deve a ausência de padding, que manteve o texto bem curto, sem a possibilidade de grandes diferenças por parte do modo de operação no efeito criptográfico.

2.4 OFB - Output feedback

Agora no OFB vamos observar outro modo com bastante semelhança com o **CBC** e o **CFB**. A diferença nesse modelo está em qual a parte que fara o XOR com o *plaintext*, utilizando também uma parte que já foi cifrada desde o inicio, diferenciando do CBC, mas que não é o bloco anterior sendo essa a diferença do CFB. A cifragem pode ser descrita com o seguinte exemplo, sendo C o texto cifrado, P o *plaintext* e O um valor auxiliar:

- $O_1 = AES(IV)$
- $C_1 = O_1$ XOR P_2 =¿ Igual ao CFB até aqui
- $O_2 = AES(O_1)$
- $C_3 = O_2$ XOR P_3
- ...

Nesse modo de operação temos algumas vantagens, como a possibilidade de paralelizar a cifragem de um bloco com o de outros, visto que um não tem mais dependência direta com o outro, o que também retira o problema do modo anterior de propagação de erros na decriptação, em caso de falha na transmissão da mensagem. No entanto, o valor de O ainda depende do anterior, o que limita a possibilidade de paralelização.

Com base no que pode ser visto nas imagens 9 e 10, os valores seguem o que ocorreu no CFB, porém com o aumento no tempo discretamente menor.

Figure 9: OFB com frase curta

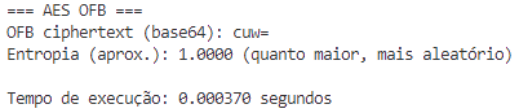
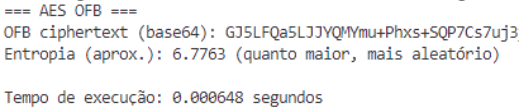


Figure 10: OFB com frase longa

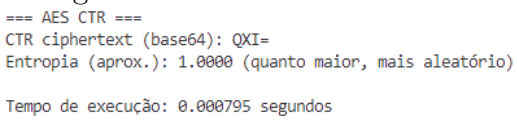


2.5 CTR - Counter

Esse modo consegue superar o problema de paralelização do **OFB**, mantendo as vantagens já apontadas. Esse processo se faz possível por meio de um contador que é acrescido a chave de inicialização utilizada, que pode ter seu valor determinado para todos os blocos a serem cifrados antes mesmo de iniciar a primeira encriptação dessa soma. Para contextualizar podemos ver a seguinte sequência:

- $K_1 = AES(IV + 0)$
- $C_1 = K_1$ XOR P_2 =¿ Igual ao CFB/OFB até aqui
- $K_2 = AES(IV + 1)$
- $C_3 = K_2$ XOR P_3
- ...

Figure 11: CTR com frase curta



Com base no que pode ser visto nas imagens 11 e 12, os valores seguem o que ocorreu no CFB, porém vale levantar que o tempo necessário foi bem maior do que em outros modos.

Figure 12: CTR com frase longa

```
=== AES CTR ===
CTR ciphertext (base64): g4WMQnJUzL3baRJXFg51uTkcapJpfjwgO
Entropia (aprox.): 6.7101 (quanto maior, mais aleatório)

Tempo de execução: 0.001734 segundos
```

2.6 Observações

Comparando todos os modos, pode ser observado que o modo "mais rápido" seria o ECB, dada a sua simplicidade, sem chaves extras, nem criptografias/XORs extras. Por outro lado, esse modo não incrementa o grau de segurança da criptografia utilizada.

Levando isso em conta, o modelo que possivelmente que se destaca é o CTR, baseado na sua possibilidade de paralelização, o que permite ganho na velocidade, mantendo um grau de segurança similar os dos outros 3 modos restantes. Porem, foi observado quem em textos curtos (menos que milhares de caracteres) essa sua vantagem de paralelização não foi muito significativa e ainda levou a uma perda de performance em comparação com os demais modos.

Por fim, o grau de aleatoriedade, segundo o método que foi estabelecida no código, que pode ser encontrado na seção **??**, se encontrou muito similar entre todos os métodos ao utilizar textos maiores, onde o padding não fez grande diferença.

Desse modo, o ganho na segurança da mensagem não é perceptível ao utilizar métodos que olham unicamente por frequência de termos, visto que o próprio AES lida bem com esse problema, camuflando bem a diferença entre os modos nos restando principalmente a análise da teoria por trás de cada método para ver suas diferenças.

3 Código

O código está disponível no github pelo link: https://github.com/Diehlgit/S-AES