

Aluno: Enzo Teles

Relatório Lista de Exercícios 02

Parte 1 - Implementação do S-AES (C++).

- Entrada e Saída:

```
189  ....string plainText = "cd"; // 16-bit plaintext
190  ....uint16_t key = 0x7144; // 16-bit key
191
192  ....// Convert hex string to unsigned short
193  ....// unsigned short int plainTextHex = static_cast<unsigned short int>(stoul(plainText, nullptr, 16));
194
195  ....// Convert hex string to unsigned short
196  ....uint16_t plainTextHex = 0;
197  ....for(char c : plainText){
198  ....    plainTextHex = (plainTextHex << 8) | static_cast<uint8_t>(c);
199  ....}
200
201  ....// Split the 16-bit plaintext into 4 nibbles and assign to the stateArray matrix
202  ....stateArray[0][0] = (plainTextHex >> 12) & 0xF; // First uint4_t (most significant)
203  ....stateArray[1][0] = (plainTextHex >> 8) & 0xF; // Second uint4_t
204  ....stateArray[0][1] = (plainTextHex >> 4) & 0xF; // Third uint4_t
205  ....stateArray[1][1] = plainTextHex & 0xF; // Fourth uint4_t (least significant)
206
```

Como parte do objetivo de inputar uma string short, resolvi fazer todo o código baseado em hexadecimal por conta de particularidades do algoritmo. O S-AES funciona fazendo operações entre 4 Nibbles (2 Bytes) que funcionam em blocos assim como na matriz da imagem, portanto sabe-se que cada hexadecimal pode ser representado com 4 bits (1 Nibble) provando a conveniência de se usar hexadecimal.

Para começar, foi definido uma chave de 16 bits “cd” e uma key em hexadecimal “0x7144” também em 16 bits, depois a chave - que é uma string - entra num laço «for» e a cada caractere é salvo o ASCII hexadecimal com operações de shift lógico 8.

Logo depois é usado essa plainTextHex para definir manualmente os nibbles, através de shift lógico e operador lógico de descarte, para armazenar no state block e ser possível realizar as operações com as demais funções em cada round.

```
107
108  Encrypted Hex: 0xA8B4
109  Base64 Encoded: CggLBA==
110
```

Saída após todas as operações e funções de cifra. No arquivo «output.txt» está toda a saída do código executado por mim e as saídas das funções visualmente amigáveis.

- Funções:

```
50
51 void ExpandKey(uint16_t masterKey, uint16_t roundKeys[]) {
52
53     ...uint8_t b[6] = {0}; // 6 bytes
54
55     ...// Key expansion
56     ...roundKeys[0] = masterKey; // Master key
57     ...b[0] = (masterKey >> 8) & 0xFF; // First uint8_t
58     ...b[1] = masterKey & 0xFF; // Second uint8_t
59
60     ...b[2] = b[0] ^ (gFunction(b[1]) ^ R_CON[0]); // Substitution of the first uint8_t
61     ...b[3] = b[2] ^ b[1];
62     ...roundKeys[1] = (b[2] << 8) | b[3]; // First round key
63     ...
64     ...b[4] = b[2] ^ (gFunction(b[3]) ^ R_CON[1]); // Substitution of the second uint8_t
65     ...b[5] = b[4] ^ b[3];
66     ...roundKeys[2] = (b[4] << 8) | b[5]; // Second round key
67
68     ...for (int i = 0; i < 3; i++) {
69     ...| ...cout << "Round Keys " << i << ": 0x" << hex << roundKeys[i] << endl;
70     ...}
71
72     ...cout << endl;
73
74 }
75
```

Uma das primeiras funções chamadas no programa é a «ExpandKey», ela quem pega a masterKey e transforma em 2 outras keys para adicionar em cada round através da próxima função «AddRoundKey».

Primeiramente defino um array de 6 bytes (3 wordKeys) e inicializo com 0 para realizar as operações. Logo defino a masterKey como a primeira chave do array de chaves de cada round e já utilizo ela para definir os bytes para as próximas operações - usando shifts lógicos e operadores de descarte «&».

Após isso, para os próximos bytes das keys, faço as operações devidas da ExpandKey que envolve a utilização de xor com o byte[0] anterior e a gFunction do byte[1] anterior xor roundConstant[0] pré definido. E assim se repete com os próximos seguindo o mesmo padrão e ao final os bytes são concatenados 1 no array de roundKeys.

```
41
42 uint8_t gFunction(uint8_t byte){
43
44     ...uint8_t rotByte = (S_BOX[byte >> 4] << 4) | S_BOX[byte & 0xF]; // RotWord functionality
45     ...uint8_t subByte = (rotByte << 4) | ((rotByte >> 4) & 0xF); // SubWord functionality
46
47     ...return subByte;
48     ...
49 }
50
```

Explicando um pouco a gFunction, pegamos cada byte e utilizamos operadores lógicos para trocar de lugar seus nibbles ao mesmo tempo que já é substituído pela Sbox fixa, depois é pego cada um novamente e armazenados na variável «subByte» e retornado.

```

76 void AddRoundKey(uint4_t stateArray[2][2], uint16_t roundKey) {
77
78     uint8_t matrix_roundKey[2][2] = {0}; // 2x2 matrix for round key
79     for (int i = 0; i < 2; i++) {
80         for (int j = 0; j < 2; j++) {
81             matrix_roundKey[i][j] = (roundKey >> ((1 - i) * 4 + (1 - j) * 8)) & 0xF; // Extracting nibbles from the round key
82         }
83     }
84     // XOR operation between the stateArray and the round key
85     for (int i = 0; i < 2; i++) {
86         for (int j = 0; j < 2; j++) {
87             stateArray[i][j] ^= matrix_roundKey[i][j]; // XOR with the round key
88         }
89     }
90 }
91
92

```

Para a «AddRoundKey», recebemos como parâmetro o stateBlock atual e a respectiva roundKey. A primeira operação realiza dois «for» para extrair cada nibble da roundKey, logo depois é utilizada para re-atribuir o stateBlock através de «xor» com cada nibble da roundKey.

```

93 void SubNibbles(uint4_t stateArray[2][2]) {
94
95     // Substitution of nibbles using S-Box
96     for (int i = 0; i < 2; i++) {
97         for (int j = 0; j < 2; j++) {
98             stateArray[i][j] = S_BOX[stateArray[i][j].to_ulong()];
99             cout << "SubNibbles stateArray[" << i << "][" << j << "]: 0x" << hex << stateArray[i][j].to_ulong() << endl;
100         }
101     }
102
103     cout << endl;
104 }
105
106

```

A operação de SubNibbles pega cada índice do stateBlock e substitui, através de dois «for», na Sbox fixa implementada no início do código.

```

107 void ShiftRows(uint4_t stateArray[2][2]) {
108
109     // Shift rows operation
110     uint4_t temp = stateArray[1][0];
111     stateArray[1][0] = stateArray[1][1];
112     stateArray[1][1] = temp;
113
114     cout << "ShiftRows stateArray[1][0]: 0x" << hex << stateArray[1][0].to_ulong() << endl;
115     cout << "ShiftRows stateArray[1][1]: 0x" << hex << stateArray[1][1].to_ulong() << endl;
116     cout << endl;
117 }
118
119

```

Na ShiftRows faço operações básicas de troca de valor de variável sem perdas através de variáveis temporárias e re-atribuo no stateBlock.

```

119
120 void MixColumns(uint4_t stateArray[2][2]) {
121     // MixColumns operation using the matrix {{1,4},{4,1}}
122     const uint8_t n00 = stateArray[0][0].to_ulong();
123     const uint8_t n01 = stateArray[0][1].to_ulong();
124     const uint8_t n10 = stateArray[1][0].to_ulong();
125     const uint8_t n11 = stateArray[1][1].to_ulong();
126
127     // Performing the matrix multiplication
128     stateArray[0][0] = uint4_t(n00 ^ GF_4[n10]); // 1*n00 + 4*n10
129     stateArray[0][1] = uint4_t(n01 ^ GF_4[n11]); // 1*n01 + 4*n11
130     stateArray[1][0] = uint4_t(GF_4[n00] ^ n10); // 4*n00 + 1*n10
131     stateArray[1][1] = uint4_t(GF_4[n01] ^ n11); // 4*n01 + 1*n11
132 }
133

```

Na imagem temos o MixColumns com Galois Field 2^4 manualmente implementado com lookup table apenas do x4, pois as operações x1 são as mesmas para cada nibble. É definido então cada nibble, realizada manualmente a re-atribuição através do «xor» com o nibble substituído com seu respectivo valor da lookup table x4 do Galois Field.

The matrix to mix the columns is is:

$$M = \begin{pmatrix} 1 & x^2 \\ x^2 & 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} 1 & 4 \\ 4 & 1 \end{pmatrix}$$

A Galois Field possui uma matriz 2x2 de operação definida acima, por isso é mais eficiente a implementação de um lookup table e substituição dos respectivos valores.

```

11  /*##### Constants #####*/
12
13  // Round constants for key expansion
14  const uint8_t R_CON[2] = {0x80, 0x30};
15
16  // Substitution box (S-Box)
17  const uint8_t S_BOX[16] = {
18      0x9, 0x4, 0xA, 0xB,
19      0xD, 0x1, 0x8, 0x5,
20      0x6, 0x2, 0x0, 0x3,
21      0xC, 0xE, 0xF, 0x7
22  };
23
24  // Lookup table for multiplication by 4
25  const uint8_t GF_4[16] = {0x0, 0x4, 0x8, 0xC, 0x3, 0x7, 0xB, 0xF, 0x6, 0x2, 0xE, 0xA, 0x5, 0x1, 0xD, 0x9};
26

```

Aí estão todas as constantes usadas e tabelas de substituição.

Parte 2 - Implementação do Modo de Operação ECB com o S-AES (C++).

Para o modo de operação ECB foi adicionado apenas vetores de armazenamento e uma função de encriptação por bloco, além de outputs mais diretos pois já foi demonstrado.

- Função `encrypt_saes_ecb`:

```
172
173 void encrypt_saes_ecb(const uint16_t roundKeys[], uint16_t plaintext){
174     ....
175     // Load the plaintext block into the state array
176     for (int i = 0; i < 2; i++){
177         for (int j = 0; j < 2; j++){
178             stateArray[i][j] = (plaintext >> ((1 - i) * 4 + (1 - j) * 8)) & 0xF;
179         }
180     }
181
182     // Round 0: AddRoundKey[0]
183     AddRoundKey(stateArray, roundKeys[0]);
184
185     // Round 1: SubNibbles, ShiftRows, MixColumns, AddRoundKey[1]
186     SubNibbles(stateArray);
187     ShiftRows(stateArray);
188     MixColumns(stateArray);
189     AddRoundKey(stateArray, roundKeys[1]);
190
191     // Round 2: SubNibbles, ShiftRows, AddRoundKey[2]
192     SubNibbles(stateArray);
193     ShiftRows(stateArray);
194     AddRoundKey(stateArray, roundKeys[2]);
195
196     // Combine the state array into a single 16-bit encrypted block
197     uint16_t encryptedBlock = 0;
198     for (int i = 0; i < 2; i++){
199         for (int j = 0; j < 2; j++){
200             encryptedBlock |= (stateArray[i][j].to_ulong() << ((1 - i) * 4 + (1 - j) * 8));
201         }
202     }
203
204     encryptedBlocks.push_back(encryptedBlock); // Store the encrypted block
205
206 }
```

Pega os nibbles e adiciona no bloco

FUNÇÕES

Pega os nibbles e adiciona na variável que armazenará no vector.

Como parâmetro da função temos as `roundKeys` e o `plaintext`, todas em 16 bits pois a função será chamada muitas vezes para diferentes blocos do texto inputado na main. O texto é separado em diversos blocos de 16 bits e armazenados em um «vector» que é passado através de um «for» para a função e assim fazendo cada operação separadamente por blocos, assim como é o ECB Mode.

Logo no início temos o `plaintext` sendo alocado numa matriz chamada `stateArray` para ser possível realizar as operações da forma como foi implementada na parte 1. Depois cada operação é chamada na ordem correta do algoritmo S-AES e ao final as partes são juntadas, convertidas em blocos e armazenadas num «vector» chamado `encryptedBlocks` para ser devidamente imprimido ao final de todo o processo.

- Entradas e Saídas:

```

214 string plainText = "Veni, Vidi, Vici"; // 16-bit plaintext ← TEXTO INPUTADO
215 uint16_t key = 0x7149; // 16-bit key
216
217 // Convert the plaintext to a vector of uint16_t plaintextBlocks
218 for (size_t i = 0; i < plainText.length(); i += 2) { ← CONVERSÃO
219     uint16_t block = (static_cast<uint8_t>(plainText[i]) << 8) |
220     (i + 1 < plainText.length() ? static_cast<uint8_t>(plainText[i + 1]) : 0);
221     plainTextBlocks.push_back(block);
222 }
223
224 // Print the plaintext blocks
225 cout << "##### Plaintext Blocks #####< endl;
226 cout << endl;
227 for (size_t i = 0; i < plainTextBlocks.size(); i++) {
228     cout << "Block " << i + 1 << ": " << hex << plainTextBlocks[i] << endl;
229 }
230 cout << endl;
231
232 cout << "##### Round Keys #####< endl;
233 cout << endl;
234
235 // Expand the key
236 ExpandKey(key, roundKeys);
237
238 // Encrypt each block
239 for (size_t i = 0; i < plainTextBlocks.size(); i++) { ← CHAMADA DA FUNÇÃO
240     encrypt_saes_ecb(roundKeys, plainTextBlocks[i]);
241 }
242

```

A imagem possui parte da «main» para mostrar a dinâmica da função. Como vemos, o texto é separado em blocos de 2 em 2 caracteres para montar o «vector» que será usado na chamada da função e nos «prints» intermediários. Logo a encriptação de cada bloco vai sendo chamada através de um «for» de cada «plainTextBlocks» juntamente com as «roundKeys» expandidas.

```

output_ecb.txt
1 ECB Demonstration!
2 The plainText: 'Veni, Vidi, Vici' ← ENTRADA
3 The key tested here is '0x7149' (16 bits) ← KEY
4
5 ##### Plaintext Blocks #####
6
7 Block 1: 0x5665 ..... Ve
8 Block 2: 0x6E69 ..... ni
9 Block 3: 0x2C20 <<1>> ,
10 Block 4: 0x5669 <<2>> Vi
11 Block 5: 0x6469 ..... di
12 Block 6: 0x2C20 <<1>> ,
13 Block 7: 0x5669 <<2>> Vi
14 Block 8: 0x6369 ..... ci
15
16 ##### Round Keys #####
17
18 Round Keys 0: 0x7149
19 Round Keys 1: 0xDC95
20 Round Keys 2: 0xFE6B
21
22 ##### Encrypted Blocks #####
23
24 Block 1: 0x51B4
25 Block 2: 0xC2A3
26 Block 3: 0x20AB <<1>>
27 Block 4: 0xE1B8 <<2>>
28 Block 5: 0xCE33
29 Block 6: 0x20AB <<1>>
30 Block 7: 0xE1B8 <<2>>
31 Block 8: 0xCA83
32
33 Base64 Encoded: UbTCoyCr4bj0MyCr4bjKgw== ← OUTPUT BASE64
34

```

Como saída temos a imagem acima. A entrada sendo o texto “Veni, Vidi, Vici” e separada em blocos com seus respectivos caracteres à direita. Vemos também todos os blocos encriptados ao final de todo o processo de criptografia e a saída em base64.

Destaco os blocos numerados «1» e «2» pois vemos que eles possuem os mesmos caracteres/hexadecimais, portanto também vão possuir a mesma cifra ao final do processo. Tudo isso evidencia o quanto o ECB Mode pode ser inseguro devido a esta natureza padronizada da cifra.

Parte 3 - Modos de Operações de Cifra com Simulação de AES Real (Python).

- Código Executáveis:

Para esta entrega, passo a estrutura do projeto de forma que seja mais facilmente utilizada e testada. Separei o programa em diferentes classes ecb, cbc, cfb, ofb e ctr representando cada modo de operação, além de separar um arquivo apenas para demais funções que foram utilizadas durante o programa. O programa é única e exclusivamente inicializado através da main.py onde é importado todas as outras funcionalidades. Mantive desta forma para melhor visualização do código, manutenção e eficiência de algoritmo.

O código fonte está bem comentado para visualização de cada funcionalidade.

```
parte 3 > output.txt
1 Digite o texto para criptografar: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
2
3 ECB Ciphertext (Base64): p1xDMpAr6EF108aVK0NeJmuKJoK6/NcRBY2Y3unfxfaYXq3bEywnza/gzHzfhc+uv+DJoFd4gNSDlMfVf5IIGWgZ1C17/4P47DfPqvYo619NoAK2A10wcG7lgcKID/yL12GDac+fpQIdW9a5Cnw==
4 ECB Decrypted: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
5 ECB Encryption Time: 0.083652 seconds
6 ECB Decryption Time: 0.000311 seconds
7
8 CBC Ciphertext (Base64): ohVkkK8rVqU3YJg/NrN50b3/36Ldd+YmvF8XbUtdgdwON+0Z45g2mDfI23J2x5zHgr13Wz5q/080s8kwI7yaAjaVA48GTWB19pZC7BUDGY18ZoeCpwHUIWqeixYgBiInD7zSZmdpKEqCmwQ4xTSSlWg==
9 CBC Decrypted: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
10 CBC Encryption Time: 0.000281 seconds
11 CBC Decryption Time: 0.000286 seconds
12
13 CFB Ciphertext (Base64): pDyv6Y/UA4wyHz+EsWtqppTcTtmGhLMjKubnySj7jgRDBUNZT8K/Us7F7fBXqawiJYD7QhJW/XrG9U24tWGJDCbD1lV9u91np71bC0zhD90btjXDeSjRvDYFMkbZ0JTlU6RgizuZnmXaAqoVZu13A==
14 CFB Decrypted: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
15 CFB Encryption Time: 0.000282 seconds
16 CFB Decryption Time: 0.000257 seconds
17
18 OFB Ciphertext (Base64): pDyv6Y/UA4wyHz+EsWtqppTcTtmGhLMjKubnySj7jgRDBUNZT8K/Us7F7fBXqawiJYD7QhJW/XrG9U24tWGJDCbD1lV9u91np71bC0zhD90btjXDeSjRvDYFMkbZ0JTlU6RgizuZnmXaAqoVZu13A==
19 OFB Decrypted: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
20 OFB Encryption Time: 0.000259 seconds
21 OFB Decryption Time: 0.000250 seconds
22
23 CTR Ciphertext (Base64): ghLXXPs6TVABHz4kMfhAYBkIfLBwFI0vStYnEqxFeYjeM9L551bd15ioyDow8bTaqxvBU8R/17cfhyjL8W7nNqd7nfm00LpSUDLdVXhvPXN+SuaMNGaIyvhJDA8JdYo1Xp/yB75G0d6kx2Sp8MA==
24 CTR Decrypted: Buscai em primeiro lugar o Reino de Deus e a Sua Justiça e todas estas coisas vos serão dadas em acréscimo.
25 CTR Encryption Time: 0.000260 seconds
26 CTR Decryption Time: 0.000263 seconds
27
```

Destaco a importância de visualizar o output.txt do texto testado na inicialização do programa, algo interessante é que a cifra do OFB e o CFB ficaram muito parecidas e coincidentemente eles foram os mais eficientes em questão de tempo, tanto de encriptação quando de deciptação.

	Tempo	Eficiência	Segurança	Aleatoriedade
ECB	0.083652 seconds	Pouca	Baixa	Péssima
CBC	0.000281 seconds	Média	Alta	Boa
CFB	0.000282 seconds	Média	Moderada	Boa
OFB	0.000259 seconds	Muito	Alta	Boa
CTR	0.000260 seconds	Muito	Alta	Excelente

Conclusão da Tarefa

Ao analisar o S-AES e o AES original, fica evidente a diferença estrutural entre eles, especialmente no que diz respeito à complexidade. Enquanto o S-AES prioriza a simplicidade didática, o AES destaca-se por sua robustez, com um maior número de rounds, operações de expansão de chave e suporte a diferentes tamanhos de chave (128, 192 e 256 bits). Essa escalabilidade e o processamento mais elaborado garantem maior segurança no AES, tornando-o adequado para aplicações do mundo real.

Além disso, o modo ECB – tanto no S-AES quanto no AES – revela-se vulnerável devido à padronização gerada pela cifragem determinística de blocos idênticos. Essa falha comprova a importância de modos de operação mais avançados (como CBC ou GCM), que introduzem aleatoriedade e mitigam riscos de ataques por análise de padrões.

Referências

<https://www.kopaldev.de/2023/09/17/simplified-aes-s-aes-cipher-explained-a-dive-into-cryptographic-essentials/>

https://youtu.be/cl7nGF0iuo0?si=cFGsCgH8jvU_dVhW

https://www.rose-hulman.edu/class/ma/holden/Archived_Courses/Math479-0304/lectures/s-aes.pdf - há um exercício de teste com o plainText “ok” e key 0xA73B.

Meu GitHub

<https://github.com/S3r4ph1el/Easy-AES> (2 Branches)

<https://github.com/S3r4ph1el/CipherOperationsModes>