View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archives of the Faculty of Veterinary Medicine UFRGS

Estudo do Padrão Avançado de Criptografia AES – Advanced Encryption Standard

Diogo Fernando Trevisan [1]
Rodrigo P. da Silva Sacchi [2]
Lino Sanabria [2]

Resumo: Este trabalho retrata a implementação do algoritmo criptográfico *Advanced Encryption Standard* (AES). A escolha do Algoritmo AES se deve ao fato de ser o atual padrão avançado de encriptação sendo selecionado após um longo concurso onde vários algoritmos foram criptoanalisados por toda a comunidade de criptologia. Além de sua eficiência, o AES também foi projetado para permitir a expansão da chave quando necessário, ser implementado tanto a nível de software quanto a nível de hardware e é disponibilizado livremente, o que permite o seu uso em aplicações diversas sem a necessidade de pagamento de *royalties*.

Abstract: This paper shows the implementation of cryptographic algorithm Advanced Encryption Standard (AES). The choice of AES algorithm considered the fact that it is the current advanced encryption standard, being selected after a long contest where various algorithms were cryptoanalized by the cryptology community. Furthermore, the AES was designed to allow expansion of the key when necessary, allow implementation in software and in hardware and is free, allowing its use in many applications without requiring the payment royalties.

1 Introdução

Criptografia, do grego *kryptós* (escondido, oculto) + *grápho* (grafia, escrita), é a arte ou a ciência de escrever em cifra ou em código; em outras palavras, é um conjunto de técnicas que permitem tornar incompreensível uma mensagem originalmente escrita com clareza, de forma a permitir, normalmente, que apenas o destinatário a decifre e compreenda. Quase sempre o deciframento requer o conhecimento de uma chave, uma informação secreta disponível ao destinatário [1],[2].

Terceiros podem ter acesso à mensagem cifrada e determinar o texto original ou mesmo a chave, "quebrando" o sistema. A criptoanálise, do grego *kryptos* + *análysis* (decomposição), é a arte ou a ciência de determinar a chave ou decifrar mensagens sem conhecer

[1]Universidade Federal do ABC, UFABC
{pancatrevisan@gmail.com}
[2]Universidade Federal da Grande Dourados, UFGD
{rodrigosacchi,linosanabria@ufgd.edu.br}

a chave. A criptologia, do grego *kryptós + lógos* (estudo, ciência), é a ciência que reúne a criptografia e a criptoanálise.

Embora existam vários métodos para esconder informações, do ponto de vista computacional predominam dois, a saber, o método de chaves simétricas e o método de chave pública ou chaves assimétricas.

No método de chaves simétricas uma mensagem ou arquivo é encriptado com uma chave e, a mesma é utilizada para fazer a decriptografia. A chave é composta por caracteres alfanuméricos – caracteres e dígitos – e também caracteres especiais (acentos, espaços), pois, o que realmente conta é seu valor em binário e, o tamanho da chave é mensurado pelo número de bits que a compõe. A encriptação por chave privada funciona muito bem quando o usuário que encripta é o mesmo que desencripta o arquivo e os arquivos não precisam ser transmitidos pela rede tampouco compartilhados. Mas quando se trata de uma mensagem que vai ser transmitida, surge um problema, pois, o receptor e o transmissor precisam antes combinar uma senha, e usar algum meio seguro para transmitir esta informação de tal modo que a informação não possa ser capturada por alheios, senão a mensagem pode ser descoberta. Um meio realmente seguro de transmissão de dados é muito caro e difícil de obter e, caso exista, a própria mensagem pode ser transmitida por ele não tendo necessidade de criptografá–la.

Na criptografia tradicional os caracteres da frase eram substituídos por outros de um alfabeto secreto. Assim, bastava conhecer o alfabeto para poder decifrar qualquer mensagem enviada. Um exemplo deste método é a muito conhecida Cifra de César, onde cada caractere é substituído por um que está três posições à frente, por exemplo, A é substituído por D [3].

Os algoritmos criptográficos baseados em blocos recebem a cada iteração um número de bits como entrada e estes bits são cifrados e armazenados em alguma mídia secundária [3]. Vários algoritmos utilizam a criptografia por blocos, como o AES, DES e RC5. A criação destes foi feita através de concursos do governo americano para ser o algoritmo padrão de criptografia [3]. Para poder concorrer os algoritmos deveria cumprir alguns requisitos, como:

- A segurança do algoritmo deve ficar restrita a chave dele e não ao algoritmo, assim, é recomendável que o algoritmo seja aberto para que pessoas do mundo todo possam testar suas fragilidades [3].
- O algoritmo também deve ser adaptável para o uso em diversas aplicações [3].
- A implementação em dispositivos eletrônicos (embarcada) deveria ser econômica [3].
- O algoritmo deveria ser eficiente [3].

O primeiro algoritmo a ser eleito como padrão americano nessas condições foi o DES. Este ficou por anos sendo o algoritmo padrão, até que em 1998 um novo concurso foi feito para a escolha e uma versão do Rijndael foi apresentada como AES [4].

O Rijndael é um algoritmo de criptografia de blocos, trabalhando com blocos de 128 bits e chaves de 128, 192 ou 256 bits. O Rijndael original foi desenvolvido para suportar tamanhos diferentes de blocos de dados e de chaves, porém, estes não são adotados na versão AES [4].

Para o funcionamento do AES são necessários alguns dados como a *S–Box* (tabela de substituição estática), o estado (que é o bloco de dados de entrada sendo modificado pelas transformações do algoritmo), a chave, e a chave de expansão (uma versão modificada da chave) [5].

O AES é dividido em dois módulos, sendo um para cifragem e um para decifragem. O módulo para cifragem conta com quatro transformações:

- *SubBytes*: Transformação que substitui os bytes do estado por bytes da *S–Box*;
- *ShiftRows*: rotaciona ciclicamente as linhas do estado, para a segunda em 1 casa, para a terceira em 2 casas e para a quarta 3 casas;
- *MixColumns*: esta operação transforma os dados das colunas do estado multiplicando por um polinômio irredutível fixado;
- *AddRoundKey*: "mistura" as colunas com uma das chaves geradas na rotina de expansão.

No módulo de decifragem existem transformações equivalentes contrárias, como a *SubBytes* inversa (que utiliza uma *S–Box* inversa), a *ShiftRows* inversa, a *MixColumns* inversa e a *AddRoundKey* inversa. Cada uma dessas faz a operação inversa de sua representante no módulo de cifragem.

2 metodologia

A primeira etapa para a implementação do AES foi compreender suas preliminares matemáticas.

Neste algoritmo, os bytes são representados por polinômios, através da correspondência óbvia

$$\begin{aligned} b &= (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) \Leftrightarrow \\ p(x) &= (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) \end{aligned} \quad (1)$$

São utilizadas as operações de soma e multiplicação de bytes. Na soma, é realizada a operação Ou–Exclusivo (XOR) entre cada bit de um byte. Dados dois polinômios, digamos

$$P1(x) = a_6x^6 + a_4x^4 + a_2x^2 + a_1x + a_0 \tag{2}$$

e

$$P2(x) = b_7x^7 + b_1x + b_0 \tag{3}$$

então

$$P1 + P2 = (a_b \oplus b_7)x^7 + (a_6 \oplus b_6)x^6 + (a_5 \oplus b_5)x^5 + (a_4 \oplus b_4)x^4 + (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0) \tag{4}$$

onde $a_i \oplus b_i$ é a operação lógica XOR entre a_i e b_i.

Em nível de byte o XOR é dado de acordo com a tabela 1.

Tabela 1. Tabela lógica da operação XOR.

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

A multiplicação entre bytes é mais complicada e não pode ser feita em nível de bit. Como os bytes suportam um valor máximo de 255 (0xFF na codificação hexadecimal), caso seja feita a multiplicação de dois bytes com valor máximo ($255 \bullet 255$) o resultado não poderia ser armazenado em um byte. Para lidar com isso é feita uma redução modular com um polinômio irredutível, para o AES, o byte irredutível é dado por

$$x^8 + x^4 + x^3 + x + 1 \tag{5}$$

ou 27 em decimal. Assim, qualquer resultado que ultrapasse o valor máximo é modularizado [5]. Por exemplo, para multiplicar o polinômio

$$x^6 + x^4 + x^2 + x + 1 \tag{6}$$

por

$$x^7 + x + 1 \tag{7}$$

obtém-se

$$x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 \quad (8)$$

resultando em

$$x^{13} + x^{11} + x^9 + x^8 + x^5 + x^5 + x^4 + x^3 + 1 \quad (9)$$

que mudularizado pelo byte irredutível (equação 5) resulta em

$$x^7 + x^6 + 1. \quad (10)$$

Esta multiplicação pode ser descomplicada através de um somatório de rotações a esquerda seguidas por uma operação (XOR) condicional com o byte 0x1B descritas na Listagem a seguir.

```
1. Xtime(byte, multiplicador)
2.   soma = 0
3.   resultado[]
4.   resultado[1] = byte
5.   para i de 2 até o numero de bits do multiplicador
6.     se bit 7 de byte = 1
7.       soma = soma << 1
8.       soma = soma XOR 1B
9.     senão
10.      soma = soma << 1
11.    fim_se
12.    resultado[i] = soma
13.    byte = soma
14.  fim_para
15.retorne resultado
```

Esta é a operação *xTime*. Para exemplificar a operação são utilizados dois bytes 0x57 = 01010111 e 0x13=10011, onde, as potências do segundo byte são x_0, x_1 e x_4 assim levando os cálculos mostrados na Tabela 2.

Após isso, comparam-se os resultados obtidos por ordem sequencial, com o multiplicando. Se o bit i do multiplicando for 1 soma-se (através da operação XOR) o resultado i, caso contrário ele é ignorado como mostrado na Tabela 3.

O próximo passo foi o projeto de um programa em Java [6], [7] que conseguisse executar tais operações. A linguagem Java foi escolhida por facilitar a futura migração para a

Tabela 2. Operações realizadas pela função *xTime*.

BYTE	Operação Executada	resultado	Hexadecimal
01010111	Nenhuma	01010111	0x57
01010111	$01010111 << 1$	10101110	0xAE
10101110	$(10101110 << 1) \oplus$ 0x1B	01000111	0x47
01000111	$01000111 << 1$	10001110	0x8E
10001110	$(10001110 << 1) \oplus$ 0x1B	0000111	0x07

Tabela 3. Resultado da multiplicação de dois bytes pelo *xTime*. Bytes com o sétimo bit com valor 1 são somados ao resultado final.

Multiplicando	**Valores**	**Resultado**
1	0x57	0x57
1	0xAE	0xF9
0	0x47	0xF9
0	0x8E	0xF9
1	0x07	0xFE

plataforma celular (J2ME). Em Java os dados do tipo byte são sinalizados podendo representar valores na faixa -128 a 127. O AES necessita bytes com valores positivos, e, como Java não possui valores sem sinal é mais interessante utilizar dados do tipo *short* que comportam valores de -32768 a 32767, tendo 16 bits cada. Todos dados internos do AES são do tipo *short*, incluindo o estado, a chave e as tabelas de substituição (*S–Box* e *S–Box* inversa).

O AES foi implementado utilizando o Java JDK versão 1.6 update 20, na IDE Netbeans 6.8 e está dividido em classes e pacotes para uma melhor organização. A Figura 1 mostra a estrutura do projeto Netbeans.

O pacote aescore tem as classes importantes para cifragem e decifragem e também uma classe para expansão de chave. Neste pacote está a classe Cipher que possui as transformações *SubBytes*, *ShiftRows, AddRoundKey* e *MixColumns* implementadas.

O método *SubBytes* recebe como entrada uma matriz 4×4 que representa o estado atual do AES. Cada byte no estado é substituído por outro byte da *S–Box* que está armazenada na classe *Sbox*. O método *ShiftRows* recebe como entrada o estado e executa as rotações cíclicas nas linhas $2, 3$ e 4 em $1, 2$ e 3 posições respectivamente.

No método *AddRoundKey* são passados como parâmetro o estado atual do algoritmo e o vetor de chaves expandidas, além do número do $round$, e, a cada $round$ uma chave expandida é misturada à matriz do estado. No método *MixColumns* as colunas do estado são misturadas a dois bytes, 0x02 e 0x03 através da multiplicação. Além destes métodos

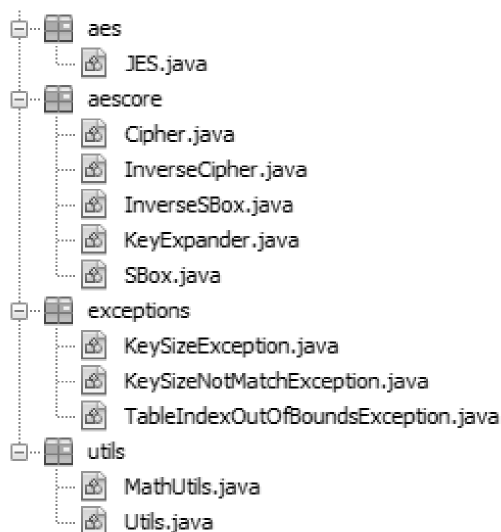

Figura 1. Estrutura do projeto da implementação do AES.

existe o método *cipherDataBlock*, que recebe como parâmetro uma matriz de 4×4 e a chave expandida, retornando uma matriz de 4×4 cifrada, executando todas operações anteriores.

A classe *InverseCipher* possui os métodos que fazem operações inversas aos da classe *Cipher* além destes possui o método *InverseCipherDataBlock* que recebe como parâmetro o estado e a chave expandida, retornando uma matriz de estado que contém os dados decifrados.

No pacote *aes* a classe *JES* cria uma camada de abstração maior aos métodos e classes de outros pacotes. Esta classe possui quatro métodos principais:

- *encryptMessage* e *decryptMessage* tendo a função de cifrar e decifrar mensagens de texto.
- *encryptFile* e *decryptFile*, tendo a função de cifrar arquivos binários.

Os métodos recebem como parâmetro a mensagem, ou, o nome do arquivo de entrada e o de saída (para os métodos que trabalham com arquivos) e a chave cifradora.

A classe *KeyExpander* possui métodos para criação da chave expandida. Esta chave é criada a partir da chave do usuário, baseada no número de rounds que serão executados.

As Classes *Sbox* e *InverseSbox* possuem a definição das tabelas de substituição e um método de acesso à elas.

No pacote *exceptions* estão as exceções de tamanho de chave (quando uma chave ultrapassa o tamanho máximo) e uma exceção de acesso à alguma posição que não existe nas tabelas de substituição *S–Box* e *S–Box* inversa.

No pacote *utils* estão algumas classes com definições de métodos úteis, como multiplicação de bytes, soma de bytes, rotação de bytes, cópia de vetores e cópia de matrizes.

A implementação começou com a construção das classes *SBox* e *InverseSBox*. A tabela armazenada nela está em forma de um arranjo de uma dimensão de 256 posições. Quando o método de substituição é invocado ele recebe como parâmetro um byte. Este byte pode ter o valor de 0 a 255, assim, o valor de substituição está na posição do valor do byte.

Após a implementação das *SBox* foram desenvolvidos os métodos da classe *MathUtils*. O principal método desta classe é o de multiplicação entre dois bytes. Também está disponível nesta classe um método que calcula a operação XOR entre duas *Words* (arranjo de quatro bytes). A multiplicação é utilizada na operação *MixColumns* apresentada adiante.

Tendo o método de multiplicação funcional a implementação pôde continuar, já que uma das operações mais difíceis estava pronta. Assim, pôde ser feita a implementação da classe *KeyExpander* que faz a rotina de expansão de chaves, recebendo como entrada a chave e gerando algumas variações desta. Cada variação será utilizada no processo de cifragem por uma rodada diferente.

Tendo a rotina de expansão de chaves pronta pôde-se fazer a classe e os métodos de cifragem. Dentro desta classe estão o método de substituição de bytes que utilizam a *SBox*, o método de rotação de linhas (*ShiftRows*), o método de mistura de colunas (*MixColumns*), método de adicionar uma chave ao estado (*AddRoundKey*) e um método principal que controla todos os outros (*cipherDataBlock*). A Listagem abaixo mostra o pseudocódigo do método de cifragem de um bloco de dados.

```
1. Short[4][4] cipherDataBlock(short[4][4] estado,
     short[] chave expandida)
2. {
3.   addRoundKey(estado);
4.   for(i= 1; i < nr; i++){
5.     SubBytes(estado);
6.     ShiftRows(estado);
7.     MixColumns(estado);
8.     AddRoundKey(estado);
9.   }
10.  SubBytes(estado);
11.  ShiftRows(estado);
12.  AddRoundKey(estado);
```

```
13.}
```

Após desenvolver o crifrador começou-se o desenvolvimento do decifrador. Os métodos usados na decifragem estão definidos dentro da classe *InverseCipher*, sendo estes *AddRoundKey* que irá adicionar uma chave expandida, *InverseMixColumns* que faz o inverso da mistura, *InverseShitRows* que faz a rotação inversa das colunas, *InverseSubBytes* que faz a substituição dos bytes pela tabela inversa (*InverseSBox*) e um método que controla todos outros *InverseCipherDataBlock*, que tem seu pseudocódigo mostrado na Listagem abaixo.

```
1. public short[][] inverseCipherDataBlock(short[][] state,
         short[][] keySchedule)
2. {
3.   addRoundKey(state,keySchedule,nr);
4.   for(int round = nr-1; round > 0; round--){
5.     inverseShiftRows(state);
6.     inverseSubBytes(state);
7.     addRoundKey(state,keySchedule,round);
8.     inverseMixColumns(state);
9.   }
10.  inverseShiftRows(state);
11.  inverseSubBytes(state);
12.  addRoundKey(state,keySchedule,0);
13.}
```

Após a implementação do crifrador e decifrador foram feitos testes para verificação de resultados baseados em [5]. Além destes testes foram verificadas questões de performance. Todas operações aqui citadas estão descritas em [5].

3 Resultados e Discussão

Os testes para verificar a eficiência da implementação foram feitos de acordo com [5]. Os resultados para o AES com chave de 128 bits pode ser visto na Tabela 4. Na Tabela 5 estão os resultados do AES com chave de 192 bits e na Tabela 6 estão os resultados obtidos para o AES com uma chave de 256 bits.

Pôde-se perceber que nos testes a saída após a cifragem foi igual à saída mostrada por [5]. Também percebe-se que a saída da decifragem é igual a entrada, ou seja, após serem cifrados os dados puderam também ser decifrados.

Tabela 4. Testes para o AES com chave de 128 bits.

CIFRAGEM AES-128 (Nk=4, Nr=10)	
Entrada	00112233445566778899aabbccddeeff
Chave	000102030405060708090a0b0c0d0e0f
Saída	69c4e0d86a7b0430d8cdb78070b4c55a
DECIFRAGEM	
Saída	00112233445566778899aabbccddeeff

Tabela 5. Testes para o AES com chave de 192 bits.

CIFRAGEM AES-192 (Nk=6, Nr=12)	
Entrada	00112233445566778899aabbccddeeff
Chave	000102030405060708090a0b0c0d0e0f1 011121314151617
Saída	dda97ca4864cdfe06eaf70a0ec0d7191
DECIFRAGEM	
Saída	00112233445566778899aabbccddeeff

Após a confirmação que a criptografia e decriptografia estavam corretas começou-se a implementação de criptografia de arquivos. A leitura dos arquivos é feita pelas classes do pacote *java.io* [6], [7]. São lidos 16 bytes do arquivo a cada vez, e, estes bytes são enviados para o *Cipher* que retorna 16 bytes criptografados. Os bytes criptografados são armazenados em um arquivo de saída, assim, são lidos e depois gravados 16 bytes a cada rodada até que o arquivo seja processado completamente.

Como os arquivos nem sempre tem o número de bytes múltiplo de 16, e, o AES trabalha com blocos de 16 bytes os bytes que faltam para completar 16 terão valor 0x00, e, na decifragem é necessário saber quantos bytes o arquivo original possuía no último bloco, pois o arquivo de saída tem que ter o mesmo tamanho do arquivo de entrada. Para isso, o primeiro byte de cada arquivo cifrado armazena quantos bytes o último bloco possui, assim, se o arquivo tiver 20 bytes o valor lido será 4, indicando que o ultimo bloco de dados possui 4 bytes e faltam 12 para completar 16. Isto evita que na decifragem seja criado um arquivo com bytes a mais.

Para testar, foi utilizado um arquivo com 20 bytes (20 caracteres). Este arquivo é criptografado gerando um arquivo de 32 bytes ilegíveis, ou seja, dois blocos de 16 bytes cada. Em seguida decriptografado, gerando um arquivo de texto legível com 32 caracteres, sendo os últimos espaços em branco. A Figura 2 mostra o arquivo de entrada, o arquivo cifrado e o arquivo decifrado.

Tabela 6. Testes para o AES com chave de 256 bits.

CIFRAGEM AES-256 (Nk=8, Nr=14)	
Entrada	00112233445566778899aabbccddeeff
Chave	000102030405060708090a0b0c0d0e0f 101112131415161718191a1b1c1d1e1f
Saída	8ea2b7ca516745bfeafc49904b496089
DECIFRAGEM	
Saída	00112233445566778899aabbccddeeff

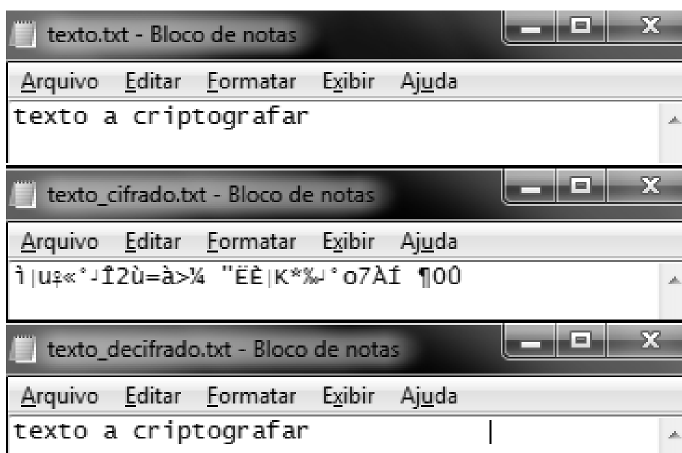

Figura 2. Arquivo de entrada com 20 bytes (topo), cifrado (meio) e decifrado com 32 bytes (baixo).

Armazenando o número de bytes que o último bloco possui os bytes excessivos (espaços no caso do texto) deixam de aparecer no arquivo decrifrado.

A velocidade de execução do AES estava comprometida pelo fato das multiplicações levarem muito tempo e, a cada bloco são feitas 32 multiplicações. Para evitar este problema os resultados possíveis das multiplicações foram armazenados em um vetor. São 4 multiplicadores utilizados na *MixColumns* e 255 possíveis valores (dos bytes) de entrada. Os resultados são armazenados em um vetor, assim, para saber o resultado de uma multiplicação de um byte basta acessar a posição do valor do byte no vetor de resultados.

Após essa implementação o tempo de execução diminuiu para um terço do anterior. Mesmo assim, o tempo de execução continuou elevado devido ao método de leitura dos arquivos, já que são lidos 16 bytes por vez, gerando mais acessos ao disco rígido.

4 Conclusões

Foi implementado o algoritmo AES em Java. Esta implementação funciona muito bem para pequenas mensagens e até mesmo para arquivos. O próximo passo é implementar o código para dispositivos portáteis como celulares e *smartphones*. Como o poder de processamento destes aumenta com o passar do tempo esta ideia fica cada vez mais aplicável, porém, a implementação ainda precisaria ser otimizada para cada plataforma.

Referências

[1] Randall K. Nichols. *Icsa Guide to Cryptography*. McGraw-Hill Professional, 1998.

[2] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[3] R. F. WEBER. Criptografia contemporânea. In *VI Simpósio de Computadores Tolerantes a Falhas*, pages 7–32, 1995.

[4] Jorge De Albuquerque LAMBERT. Cifrador simétrico de blocos: Projeto e avaliação. Dissertação de mestrado apresentada ao curso de mestrado em sistemas e computação, Instituto Militar de Engenharia, Rio de Janeiro, 2004.

[5] Announcing The Federal. Processing standards publication 197.

[6] Harvey M. Deitel and Paul J. Deitel. *Java How to Program (6th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[7] Oracle. Oracle technology network for java developers, jun 2009.