CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA

CELSO SUCKOW DA FONSECA

Geovane Pacheco da Rocha

Gerador de Números Pseudo-Aleatórios

Nova Friburgo

2012

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA

CELSO SUCKOW DA FONSECA

Geovane Pacheco da Rocha

Gerador de Números Pseudo-Aleatórios

Trabalho de conclusão de curso apresentado ao
CEFET Nova Friburgo como requisito parcial para
a conclusão do Curso Técnico de Informática

Nova Friburgo

2012

Ficha catalográfica elaborada pela Biblioteca do CEFET/RJ UnED Nova Friburgo

R672 Rocha, Geovane Pacheco da
Gerador de Números Pseudo-Aleatórios / Geovane Pacheco da Rocha. —2012.
viii, 27f. + anexos : tabs. ;enc.

Trabalho de Conclusão de Curso (Técnico) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca , 2012.
Bibliografia : f.16
Orientador : Nilson Mori Lazarin

1.Programação - computadores 2. Algoritmos 3. Gerador de números pseudo-aleatórios 4. On-time pad I.Lazarin, Nilson Mori (orient.) II.Título.

CDD 005.1

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA

CELSO SUCKOW DA FONSECA

Curso: Técnico em Informática

Ano: 2012

Aluno: Geovane Pacheco da Rocha

Título do TCC: Gerador de Números Pseudo-Aleatórios

Orientador: Nilson Mori Lazarin

Nota:

Recomendações:

_____
_____
_____
_____
_____
_____

_____

(Docente Orientador)

RESUMO

Geradores de números pseudo-aleatórios são algoritmos matemáticos que necessitam de uma *semente* para produzir uma sequência de números. Uma semente é um valor responsável por dar inicio à geração dos números pseudo-aleatórios. O determinismo de um algoritmo computacional faz com que uma *semente* idêntica produza uma sequência numérica idêntica. Este trabalho apresenta um gerador inspirado no *one-time pad, s*istema criptográfico que caracteriza-se pela não reutilização de *sementes,* possibilitando que *sementes* idênticas produzam sequências numéricas distintas. Ademais, tais sequências são aderentes ao teste de frequência Monobit do NIST Test Suite, uma bateria de testes estatísticos criada pelo NIST (National Institute of Standards Technology) com o objetivo de testar o nível de aleatoriedade de sequências binárias.

Palavras-chave: pseudo-aleatório, gerador de números pseudo-aleatórios, teste Monobit, one-time pad, aleatoriedade.

ABSTRACT

A Pseudo-random numbers generator is a mathematical algorithm that requires a *seed* to produce a sequence of numbers. A seed is a value responsible to initiate pseudo random numbers generation. The determinism of a computational algorithm makes an identical *seed* to produce an identical numerical sequence. This paper presents a generator inspired in the *one-time pad*, a cryptographic system characterized by non reutilization of seeds, making possible identical *seeds* to produce distinct numerical sequences. Besides, these sequences are adherent to NIST Test 1 - Frequency (Monobit) Test, a collection of statistic tests created by NIST (National Institute of Standards Technology) which intends to test the randomness level of binary sequences.

Keywords: Pseudo-random, Pseudo-random numbers generator, Monobit test, one-time pad, randomness.

Sumário

Lista figuras

Lista de equações

Lista de tabelas

1 Introdução

Diversas áreas do conhecimento necessitam de números aleatórios, desde a simulação de fenômenos físicos até o desenvolvimento de jogos para computador. No desenvolvimento de jogos, por exemplo, ao iniciar um determinado estágio de um jogo, pode ser exigido que os adversários estejam em posições diferentes e/ou realizem ações imprevisíveis. Outro exemplo que pode ser destacado é a geração de senhas instantâneas, por exemplo, ao acessar uma conta via banco on-line, é interessante o uso de uma senha aleatória atribuída durante um determinado tempo, onde o proprietário da conta receba essa senha através de mensagem de texto em seu celular.

Porém, a geração de números verdadeiramente aleatórios tem um alto custo financeiro. Pode-se, por exemplo, utilizar uma amostra de um elemento radioativo e submetê-la a um contador Geizer[1], considerando o tempo entre os cliques deste contador como valores aleatórios (SOUZA, G. S.; ALVES JR, 2011).

Uma das primeiras tentativas para solucionar o problema da geração de números aleatórios foi a publicação de uma tabela em 1927 com 41.600 dígitos aleatórios, posteriormente, em 1947 uma simulação eletrônica de uma roda de roleta foi utilizada para gerar um milhão de dígitos aleatórios, e desde o advento do computador, algoritmos são desenvolvidos na tentativa de gerar números aleatórios (SOUZA, G. S.; ALVES JR, 2011).

Entretanto, uma sequência aleatória é aquela que não pode ser computada por uma máquina de Turing (CAMPANI; MENEZES, 2004), pois sendo o computador uma máquina de aritmética finita e discreta, é virtualmente impossível gerar números verdadeiramente aleatórios (COSTA JR, 2006). Dessa forma, utilizar um algoritmo computacional no intuito de gerar números aleatórios parece violar a definição de aleatoriedade[2], por isso, números gerados por um computador convencionou-se chamar de pseudo-aleatórios (ROSA; PEDRO, 2002).

Para qualificar os GNPA (Geradores de Números Pseudo-Aleatórios), quanto ao nível de aleatoriedade, o NIST (National Institute of Standards and Technology) criou o NIST Test Suite. Essa Suite é composta por quinze testes estatísticos, desenvolvidos para testar a aleatoriedade de sequências binárias produzidas por hardware ou software (RUKHIN et al., 2010). Além desta, existem outros testes estatísticos, por exemplo, o (MARSAGLIA, 1995) que é considerado um dos mais rigorosos na literatura (VIEIRA et al., 2004).

Geradores de números pseudo-aleatórios são algoritmos matemáticos que dependem de um valor de inicialização denominado *semente* para produzir números, no entanto, uma mesma *semente* pode produzir sequências de números idênticas. Tal determinismo é comumente contornado através da geração de *sementes* que contenham características aleatórias, tais como posição do mouse na tela do usuário, consumo de memória RAM ou entrada e saída de rede. Porém, esta solução está atrelada à intervenção humana e seus recursos esgotam-se, tornando-se inviável em determinados casos (COSTA JR, 2006).

---

1 Medidor de radiações de partículas alfa, beta ou radiação gama e raios-X (SOUZA e ALVES, 2011).

2 Qualidade ou característica do que é aleatório, indeterminação, incerteza, causalidade (HOUAISS e VILLAR, 2009).

Segundo (ROSA; PEDRO, 2002) e (STALLINGS, 2008), um gerador de números pseudo-aleatórios deve atender aos seguintes princípios:

- Os números gerados devem seguir uma distribuição uniforme;
- Os números devem ser estatisticamente independentes entre si;
- O período de repetição deve ser suficientemente grande;
- A geração deve ser rápida.

Além das características supracitadas, é desejável que um gerador seja capaz de a partir de uma mesma *semente* de entrada, produzir sequências aleatórias distintas, ou que uma *semente* seja utilizada apenas uma única vez, evitando a repetição de sequências.

É recomendado por (VIEIRA et al., 2004) que um gerador esteja baseado em uma sólida análise matemática, além de que estes devem ser submetidos a testes estatísticos na busca de deficiências do gerador. Sendo assim, um gerador de números pseudo-aleatórios "*deve ser suficientemente simples para compreensão e implementação em hardware e software*" (LAMBERT, 2004).

Portanto, este trabalho apresenta um gerador de *palavras* binárias de 32 bits e seu respectivo pseudocódigo em anexo. Por ser inspirado no *one-time pad*, esse gerador possibilita que uma mesma *semente* produza valores aleatórios distintos e ainda, tais valores sejam aderentes ao teste de frequência Monobit do NIST Test Suite. A fim de obter-se a maior eficácia de aprovação no teste citado foram utilizadas técnicas criptográficas de *confusão*[3] e *difusão*[4].

1.1 Trabalho relacionados

Existem diversos geradores de números pseudo-aleatórios, como por exemplo, o *rand*, o proposto por (SCHRAGE, 1979) , o proposto por (COSTA JR, 2006), e o *método do quadrado do meio*, que possuem características inviáveis em determinadas aplicações.

O GNPA da linguagem de programação C (*rand*) é baseado em congruência linear mista ($X_{n+1} = (X_n . a + c) mod\ m$), para a geração de seus números com os seguintes parâmetros $X_{n+1} = (X_n . 1103515245 + 12345)\ mod\ 2^{31}$ (VIEIRA et al., 2004). No entanto, os valores das incógnitas *a* e *c* são constantes, facilitando assim a previsibilidade do gerador.

O GNPA proposto em (SCHRAGE, 1979), cujo objetivo é manter a compatibilidade do algoritmo em diversos sistemas, é baseado no método linear congruente multiplicativo, ($X_{n+1} = a\ .X_n\ mod\ m$) para a geração de seus números com os seguintes parâmetros: $X_{n+1} = 7^5\ .\ X_n\ mod\ (2^{31} - 1)$. No entanto, o valor da incógnita *a* é constante, facilitando assim a previsibilidade do gerador, e existe uma restrição referente à *semente* utilizada: se esta for igual a zero, todos os números subsequentes serão zero.

O GNPA proposto em (COSTA JR, 2006), utiliza */dev/random* do kernel GNU/Linux para produzir *sementes* com características aleatórias. Por estar fortemente atrelado ao sistema operacional Linux, inviabiliza a implementação em outras aplicações.

O *método do quadrado do meio,* proposto por John von Neumann, consiste em elevar ao quadrado uma dada *semente* e após isso, utiliza-se os algarismos centrais do resulta-

3 Tentativa do algoritmo de tornar a relação entre a entrada e a saída tão complexa quanto possível.

4 Mudança dos bits de modo que os padrões existentes sejam dispersos.

do da operação como a próxima *semente*. Deve-se ter cuidado ao utilizar este método, pois as sequências numéricas geradas podem se repetir em pouco tempo, além do fato de que se durante o processo, a *semente* se tornar zero ou um, todos os demais valores gerados serão zero ou um respectivamente (ROSA; PEDRO, 2002).

1.2 Objetivo

O objetivo deste TCC é propor um GNPA com uma estrutura de geração diferente dos geradores citados na seção 1.1, utilizando o tempo como uma das fontes de alimentação, pois assim é possível sua implementação em diversos tipos de hardware e software, além de atender ao princípio de *flexibilidade* proposto por (LAMBERT, 2004).

Para solucionar o problema do determinismo dos GNPA citados anteriormente, este trabalho tem por objetivo tornar os valores *a* e *c* na fórmula da congruência linear dinâmicos, a fim de que *sementes* repetidas gerem sequências numéricas distintas. Além das características citadas, todos os números gerados pela saída deste GNPA são submetidos ao teste de frequência Monobit do NIST Test Suite. Tais números só são efetivamente considerados se forem aprovados por tal teste.

2 Terminologia

Para compreender as considerações expostas neste TCC, é necessária a apresentação de determinados conceitos sobre programação e criptografia, são eles: Palavra, Bloco, XOR, *Permutação*, *Substituição*, Confusão e Difusão, Avalanche, One-time pad, Congruência Linear, Qui-Quadrado e Teste Monobit.

2.1 Conceitos sobre programação

2.1.1 Algoritmo

Pode-se utilizar como conceito informal de algoritmo, uma receita (de bolo, por exemplo) à qual deve-se seguir determinada sequência de instruções a fim de concluir o objetivo desejado, e formalmente como "*uma sequência de passos computacionais que transformam a entrada na saída*"(CORMEN et al., 2001).

2.1.2 Vetor

Vetor representa uma estrutura de dados de tamanho n, utilizado para armazenar valores (neste TCC um bit em cada posição do vetor) e sua enumeração está entre ($0 \leq i \leq n$-1). Para acessar determinada posição de um vetor, faz - se: NomeVetor[i].

2.1.3 Palavra

De acordo com o dicionário Priberam, *palavra* é um *"elemento de informação armazenado ou tratado sem interrupção num computador."* Neste TCC, *palavra* estará restrita a números binários de 32 bits.

2.2 Conceitos sobre criptografia

2.2.1 Bloco

Uma *palavra* binária pode ser dividida em partes, cada parte dessa *palavra* é denominada bloco, que por sua vez é definido como uma sequência finita de bits (LAMBERT, 2004). Pode-se dividir um vetor de tamanho n de bits em k blocos de tamanho m, respeitando $k*m=n$, onde k e m são números naturais. Esta técnica pode ser utilizada em diversas aplicações. Neste TCC, o bloco será usado nas etapas de *Permutação* e *Substituição*.

2.2.2 XOR

XOR (eXclusive OR) denotado pelo símbolo $\oplus$, é uma operação muito utilizada em criptografia pois acrescenta entropia sem aumentar o corpo finito da string binária. O resultado dessa operação entre dois valores A e B será 0 se os dois valores forem iguais, e 1 se forem diferentes, *c*onforme a Tabela 1.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 1 - Tabela verdade da operação XOR. Fonte: O autor.

2.2.3 Permutação

Permutação é a técnica criptográfica usada para embaralhar os bits de uma mensagem original, a fim de obter-se uma mensagem resultante completamente diferente, um exemplo dessa técnica é mostrado na Figura 1. A *permutação* tem como objetivo adicionar *difusão* à geração (SIMPLICIO JR, 2008).

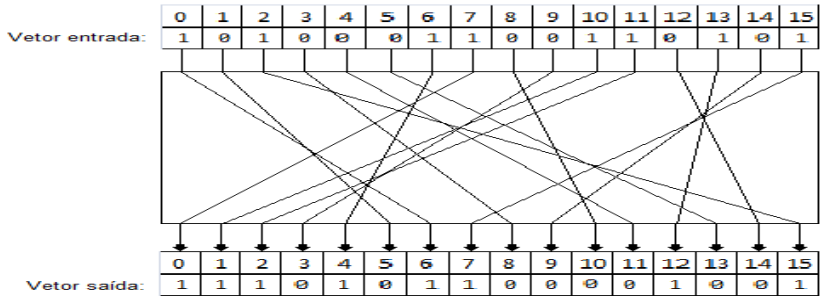


Figura 1 - Permutação de um vetor com 16 bits. Fonte: O autor.

2.2.4 Substituição

Substituição é a técnica criptográfica usada para trocar blocos binários de uma mensagem original, por outros blocos binários de uma tabela pré-determinada, como por exemplo a Tabela 2. Esta técnica tem como objetivo adicionar *confusão* à geração (SIMPLICIO JR, 2008). A Figura 2 mostra um exemplo de *substituição*.

Entrada	Saída	Entrada	Saída
0000	0111	1000	0001
0001	1111	1001	1101
0010	1010	1010	0110
0011	0011	1011	1110
0100	0010	1100	0000
0101	0100	1101	1001
0110	1000	1110	0101
0111	1011	1111	1100

Tabela 2 - Tabela de substituição. Fonte: O autor.

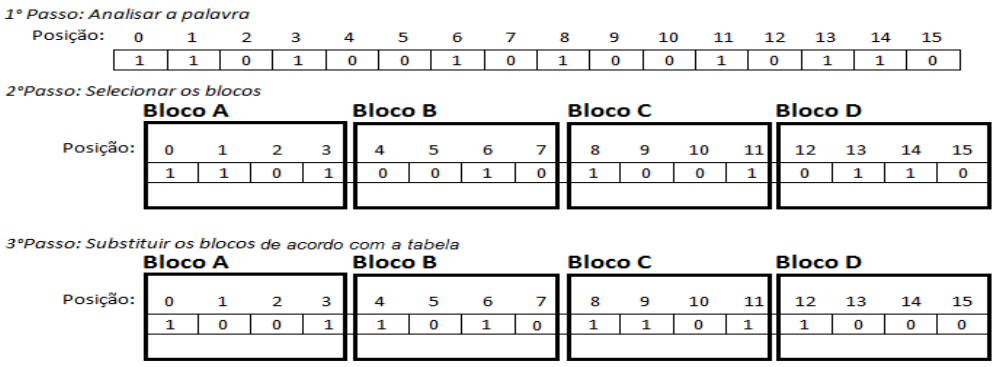


Figura 2 - Exemplo substituição. Fonte: O autor.

2.2.5 Confusão e Difusão

Confusão é um conceito associado à complexidade da relação entre a *semente* de entrada e o número aleatório gerado. *Difusão* é um conceito relacionado com a posição dos bits de forma que padrões de entrada sejam dissipados. De acordo com (LAMBERT, 2004) *"uma operação de substituição em uma etapa de transformação criptográfica adiciona confusão à cifragem, ao passo que uma operação de permutação adiciona difusão"*. Ao utilizar tais técnicas, obtém-se um ótimo efeito *avalanche*.

2.2.6 Avalanche

Fazendo uma analogia com o mundo real, avalanche é uma catástrofe natural, a qual uma enorme quantidade de neve desce montanha abaixo, afetando tudo que está em seu caminho. Na criptologia, a definição é análoga: o efeito *avalanche* é gerado ao se utilizar técnicas que produzem *confusão* e *difusão*, repetidas vezes sobre a mesma *palavra*. Por exemplo, se duas *palavras*, possuírem distancia de hamming[5] igual a um, após o uso de *confusão* e *difusão* a distância de hamming entre tais *palavras* será maior que um.

A Figura 3 mostra um exemplo do efeito *avalanche* e o quantifica através da distância de hamming. Serão utilizadas técnicas criptográficas como *Permutação* e *Substituição*. Para efetuar a *substituição*, a *palavra* será dividida em dois blocos de três bits.

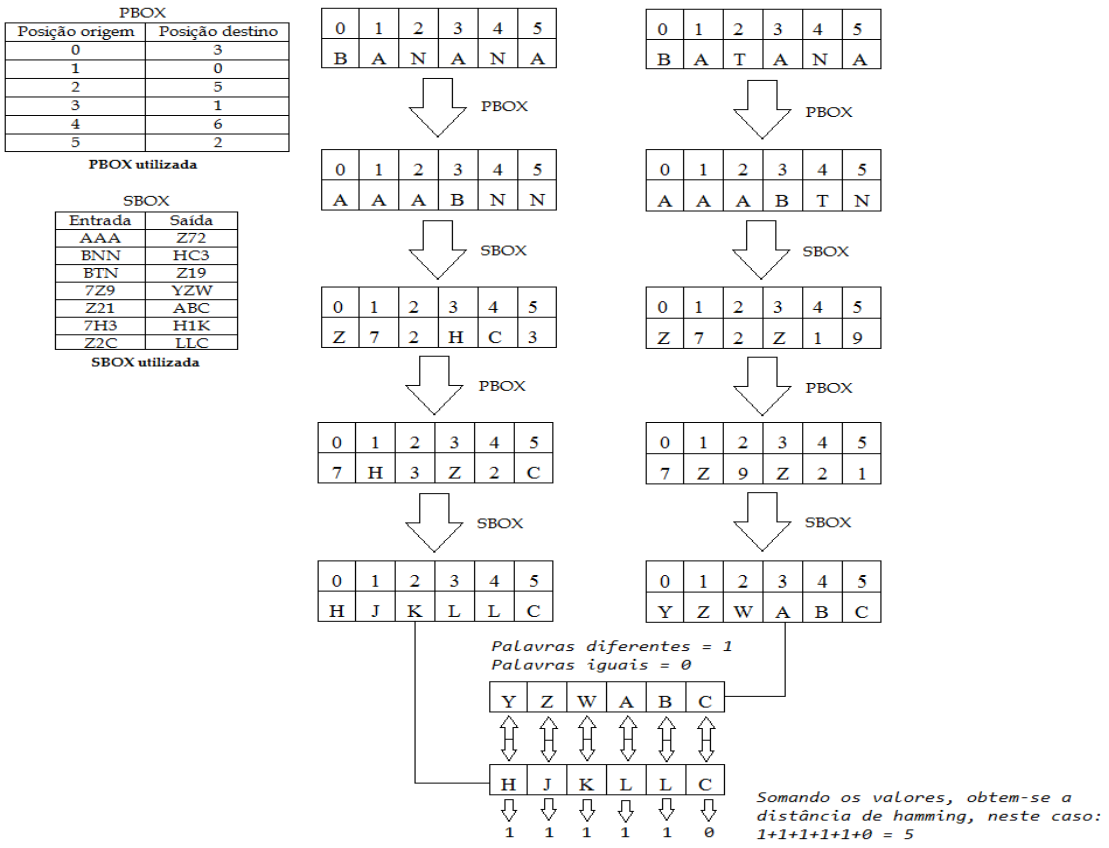


Figura 3 - Exemplo avalanche. Fonte: O autor.

As *palavras* de origem (BANANA e BATANA) possuem distância de hamming = 1, após a aplicação de *confusão* e *difusão*, pode-se observar o efeito *avalanche* entre as *palavras* finais (HJKLLC, YZWABC) que possuem distância de hamming = 5.

5 Hamming é a quantidade de bits diferentes entre duas palavras, por exemplo, mamão e mameo são diferentes em uma letra, ou seja, a distância de hamming neste caso é de um.

2.2.7 One-time pad

O *one-time pad*, foi proposto em 1917 por Mauborgne e Vernam e, em tal sistema, o remetente codifica a mensagem e, em seguida, destrói a *chave* utilizada; o receptor decifra a mensagem e também destrói a *chave* (SCHNEIER, 1996).

No *one-time pad*, uma dada mensagem cifrada é igualmente susceptível de corresponder a qualquer outra *mensagem em claro*[6] de mesmo tamanho. Pois, cada mensagem é cifrada com uma *chave* única, onde esta nunca será utilizada mais que uma vez, além de que a *chave* é tão grande quanto o texto em claro, assim não há repetição de partes da mensagem (CURI, 2009). Supondo que um intruso não tenha acesso ao *one-time pad* usado para criptografar a mensagem, este esquema é perfeitamente seguro. (SCHNEIER, 1996).

2.2.8 Congruência Linear

Congruência linear é uma fórmula muito utilizada na geração de números pseudo-aleatórios, por ser simples, eficiente, e segura (COSTA JR, 2006). Tal fórmula é dada como: $X_{n+1} = \big((X_n \,.\, a) + c\big)\, mod\, m$, onde: X_n é a *semente*; a é um multiplicador; c um incremento e m o corpo finito, ou seja, X_{n+1} será sempre menor que m, portanto, ao desejar-se o aumento do período deve-se aumentar o valor de m.

Segundo (STALLINGS, 2008), a força do algoritmo de congruência linear é que se o multiplicador e o módulo forem escolhidos corretamente, a sequência de números resultante será estatisticamente indistinguível de uma sequência retirada aleatoriamente do conjunto (1,2,...,m-1).

Por esses motivos, a congruência linear foi escolhida para ser responsável por produzir os números de base, que são utilizados em diversos estágios do GNPA em conjunto com outras técnicas criptográficas, como por exemplo: XOR, *permutação* e *substituição*. O objetivo de se utilizar números de base é aumentar a entropia da geração.

2.2.9 Qui-Quadrado (χ^2)

Qui-Quadrado é um teste estatístico muito conhecido e pode ser utilizado como teste de homogeneidade de sub populações, aderência de modelos ou independência de variáveis (ROSA; PEDRO, 2002). Diversos autores o utilizaram, como (COSTA JR, 2006), (VIEIRA et al., 2004) e (ROSA; PEDRO, 2002).

O teste χ^2 consiste em comparar frequências observadas, obtidas experimentalmente, com frequências esperadas, de acordo com a probabilidade ideal, e dessas comparações surgem diferenças (COSTA, 2005). Para serem consideradas aleatórias neste teste, as diferenças citadas anteriormente devem se aproximar de zero, ou seja, as frequências observadas tendem a ser iguais as frequências esperadas. Tal teste é calculado conforme a Equação 1:

$$\boldsymbol{\chi^2 = \sum_{i=1}^{N} \frac{(FO_i - FE_i)^2}{FE_i}}$$

Equação 1 - Fórmula do teste estatístico χ^2. Fonte: (COSTA, 2005).

Onde:

FO_i = Frequência Observada no i-ésimo termo;

6 Mensagem compreensível.

FE_i = Frequência Esperada no i-ésimo termo;

N = Espaço amostral.

Este teste estatístico será utilizado com o intuito de comparar o nível de aleatoriedade entre o GNPA proposto com outros GNPA existentes na literatura.

2.2.10 Teste Monobit

Conforme (RUKHIN et al., 2010), dada uma sequência binária qualquer, o Teste de Frequência Monobit[7], verifica se a ocorrência de 0's pode ser considerada igual à ocorrência de 1´s, ou seja, se a proporção de 0's e 1's é 1; caso seja, a sequência é considerada aleatória.

A decisão é realizada através de um teste de hipótese, onde:

- (H_0): a proporção de 0's e 1's é igual, ou seja, p = 1
- (H_1): p ≠ 1

A regra de decisão é a seguinte: rejeitar a hipótese de nulidade (H_0) se o valor p é menor que um nível de significância (α) escolhido arbitrariamente.

Conhecidos estes conceitos, pode-se interpretar os resultados do primeiro teste do NIST ao verificar se uma sequência de bits pode ser considerada aleatória. Os passos são os seguintes:

1. Calcular S_n por meio de: $S_n = \sum_{i=1}^{n} X_i$, onde: $X_i = \{-1, +1\}$ caso X_i seja 0 ou 1, respectivamente.
2. Calcular S_{obs} onde: $S_{obs} = \frac{|S_n|}{\sqrt{n}}$
3. $Z(S_{obs})$ por meio de: $Z(S_{obs}) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} {S_{obs}}^2\right)$
4. Calcular o valor p por meio de: $valor\ p\ = (1\ -\ Z(S_{obs}))$ x 2
5. Comparar o valor p com um valor de referência arbitrado (α) (nível de significância). Para o NIST, os valores usuais de α são: 0,05; 0,01; ou 0,001.

Embora a ordem de aplicação dos quinze testes disponíveis no NIST Test Suite seja arbitrária, o NIST aconselha que o teste de frequência Monobit seja executado primeiro, uma vez que este fornece a evidência mais fundamental para a existência de não aleatoriedade em uma sequência. Portanto, se esta não for considerada aleatória nesse teste, a probabilidade de também não o ser nos outros testes é elevada.

7 A "ErrorFunction" utilizada pelo teste Monobit em seus cálculos, foi implementada na linguagem de programação Java por (Sedgewick; Wayne, 2011)

3 Gerador proposto

Neste capítulo, será apresentado o projeto do gerador proposto, baseado em técnicas criptográficas aplicadas à geração de números aleatórios, atendendo aos princípios de *flexibilidade*, *simplicidade* e *credibilidade* propostos por (LAMBERT, 2004) que estão dispostos abaixo:

- ***Flexibilidade***: Um algoritmo deve ser independente de plataformas, possibilitando sua implementação em hardware ou software.
- ***Simplicidade***: Um algoritmo deve ser suficientemente simples para a compreensão e implementação.
- ***Credibilidade***: Um algoritmo deve ser aprovado em testes estatísticos de aleatoriedade.

Para atender o princípio de *flexibilidade*, em todas as etapas de geração foi utilizado o tempo como alimentação destas, pois este fornece alto grau de flexibilidade. O princípio de *simplicidade* foi atendido ao utilizar métodos simples durante a geração, como por exemplo: XOR, congruência linear, *permutação* e *substituição*. O princípio de *credibilidade* foi atendido ao acoplar durante a geração o teste estatístico reconhecido internacionalmente denominado Monobit. Dessa forma, o GNPA proposto pode ser implementado em qualquer plataforma de hardware ou software, utiliza operações de simples compreensão e é aderente ao teste estatístico Monobit, proposto pelo NIST.

O diferencial desse projeto deve-se ao fato de ter acoplado durante a geração o teste estatístico Monobit, de forma que todo número gerado seja considerado aleatório, pois normalmente um GNPA é submetido a testes estatísticos, com base em amostras de números gerados, não considerando a existência de possíveis *sementes* inapropriadas.

Outro diferencial que vale ser ressaltado é que o GNPA proposto é inspirado no *one-time pad* para solucionar o problema do determinismo (uma mesma *semente* gera uma mesma sequência numérica), de forma que uma *semente*, nesse gerador, é utilizada apenas uma única vez. Uma das soluções de menor custo, para solucionar esse problema, é o uso do domínio do tempo, por isso o GNPA proposto utiliza o instante do tempo continuum, no momento da geração, como parte da *semente*.

Ao utilizar o método supracitado, possibilita-se que uma única *semente* informada pelo usuário repetidas vezes, produzam sequências aleatórias distintas. Para resolver o problema do uso do domínio do tempo, já que o mesmo varia em uma única direção, foram utilizadas técnicas criptográficas de *confusão* e *difusão*, capazes de perturbar a linearidade temporal.

Na seção 3.1 será abordada a descrição do gerador, na seção 3.2 a caixa de *permutação*, na seção 3.3 a caixa de *substituição* e na seção 3.4 a caixa congruente.

3.1 Descrição do gerador

O gerador proposto é formado por três componentes principais: caixa de *permutação* (PBOX), caixa de *substituição* (SBOX) e caixa congruente (CBOX). O usuário informa uma *semente* inicial que sofre operações de ou-exclusivo com valores gerados pela CBOX; posteriormente o resultado da operação é submetido à PBOX e à SBOX; e em fim, o valor é submetido ao teste estatístico. Se o valor for aprovado, a geração estará completa e terá como fruto do processo um número estatisticamente aleatório. Em caso

de reprovação, será iniciada uma nova geração, tendo como *semente* aquele valor reprovado pelo teste estatístico submetido a operação de ou-exclusivo com outro valor gerado pela CBOX. O fluxograma do gerador proposto encontra-se na Figura 4.

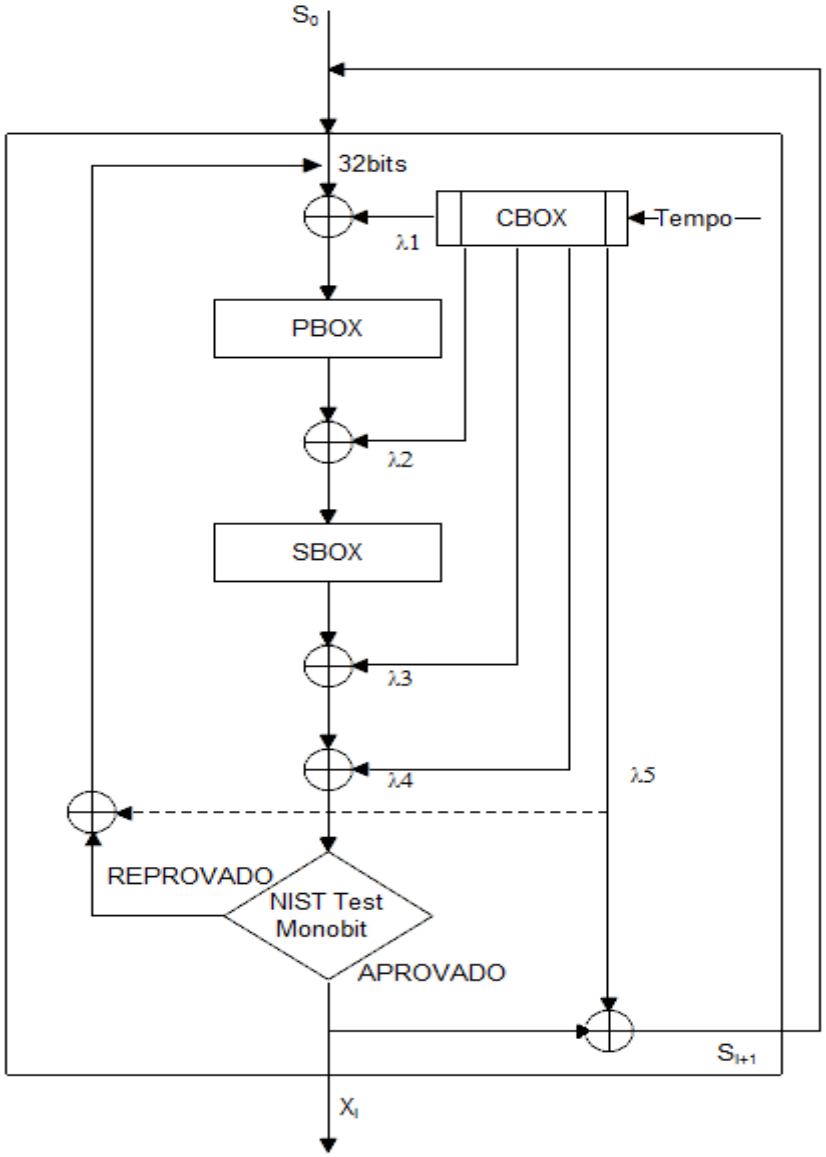


Figura 4 - Fluxograma gerador. Fonte: O autor.

Formalmente pode-se definir o gerador de acordo com a Equação 2:

$$X_i = \lambda_4 \oplus \varphi(\lambda_3 \oplus \psi((S_{i-1} \oplus \lambda_1) \oplus \lambda_2))$$

Equação 2 – Fórmula do gerador proposto. Fonte: O autor.

Onde:

- φ() é uma função de *substituição* SBOX().
- ψ() é uma função de *permutação* PBOX().
- $\Lambda = \{\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5\}$ é uma função baseada em congruência linear que produz cinco números (λ_i) utilizados no processo.
- S é uma *semente*, onde S_0 é informada pelo usuário e S_i, para qualquer $i > 0$, é obtida através de $S_i = X_i \oplus \lambda_5$.
- X_i é o número aleatório gerado.

3.2 Caixa de permutação (PBOX)

A *semente* numérica informada pelo usuário e o valor de λ_1, são convertidos em strings binárias de 32 bits, que sofrem uma operação de ou - exclusivo no primeiro passo. Posteriormente, o resultado da operação é enviado à PBOX para sofrer uma *permutação*. Os bits de entrada são embaralhados, conforme Figura 5. A string de saída é enviada para o próximo passo que é outra operação de ou-exclusivo, agora com o valor de λ_2 convertido em uma string de 32bits.

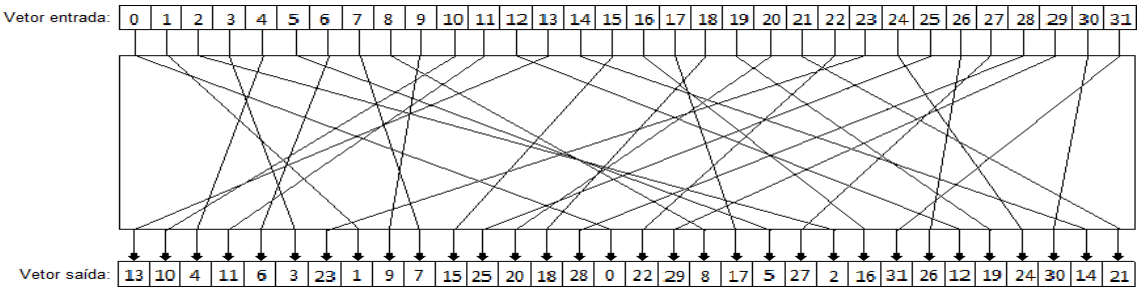


Figura 5 - Esquema permutação. Fonte: O autor.

3.3 Caixa de substituição (SBOX)

A SBOX é uma função bijetora[8] que recebe uma string de 32bits, oriunda dos passos anteriores. A string é dividida em 8 blocos de 4 bits que são substituídos, individualmente e concatenados na saída. O valor obtido da SBOX sofre uma operação de ou-exclusivo com λ_3, e o resultado desta operação é submetido a ou-exclusivo com λ_4. A tabela de *substituição* utilizada na SBOX é apresentada, em hexadecimal, na Tabela 3.

Caixa de substituição (Hexa)

Entrada	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Saída	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

Tabela 3 - Tabela de substituição. Fonte: O autor.

3.4 Caixa Congruente (CBOX)

A Caixa Congruente (CBOX) tem como objetivo produzir valores binários distintos $(\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5)$ através da congruência linear, e utilizá-los em diversas etapas da geração a fim de auxiliar no efeito *avalanche*, conforme mostra a Figura 4.

Para produzir seus valores, a CBOX faz uso do tempo pelo fato de que ao contrário da posição do mouse e outros componentes de hardware, o tempo fornece alto grau de flexibilidade, permitindo que o gerador proposto possa ser implementado em qualquer sistema operacional, em access points, servidores e outros equipamentos. Outra característica favorável do tempo é o fato de ser único (uma determinada data jamais se repetirá).

Para a CBOX produzir seus valores, o tempo foi dividido nos seguintes vetores:

- $A = \{a_0, a_1, a_2, a_3, \}$ é o vetor que recebe os algarismos do ano;
- $B = \{b_0, b_1\}$ é o vetor que recebe os algarismos do mês;
- $C = \{c_0, c_1\}$ é o vetor que recebe os algarismos do dia;

8 Cada elemento do conjunto imagem está relacionado com apenas um elemento do conjunto domínio.

- $D = \{d_0, d_1\}$ é o vetor que recebe os algarismos da hora;
- $E = \{e_0, e_1\}$ é o vetor que recebe os algarismos do minuto;
- $F = \{f_0, f_1\}$ é o vetor que recebe os algarismos do segundo;
- $G = \{g_0, g_1, g_2\}$ é o vetor que recebe os algarismos do milissegundo;
- $H = \{h_0, h_1, h_2\}$ é o vetor que recebe os algarismos do microssegundo;
- $L = \{l_0, l_1, l_2\}$ é o vetor que recebe os algarismos do nanossegundo;

A CBOX utiliza a fórmula básica de congruência linear $\lambda_{n+1} = \big((\lambda_n \,.\, \alpha) + \beta\big)\, mod\ m$, cuja as incógnitas λ_n, α, β e m são números naturais preenchidos com os elementos dos seguintes vetores concatenados entre si:

- $\lambda_0 = \{b_1, c_0, c_1, d_0, d_1, e_0, e_1, f_0\}$
- $\alpha = \{f_1, g_0, g_1, g_2, h_0, h_1, h_2, l_0\}$
- $\beta = \{l_1, l_2, a_0\ a_1, a_2, a_3, b_0\}$
- $m = mod\ 2^{32}$

A Figura 6 demonstra o funcionamento da CBOX:

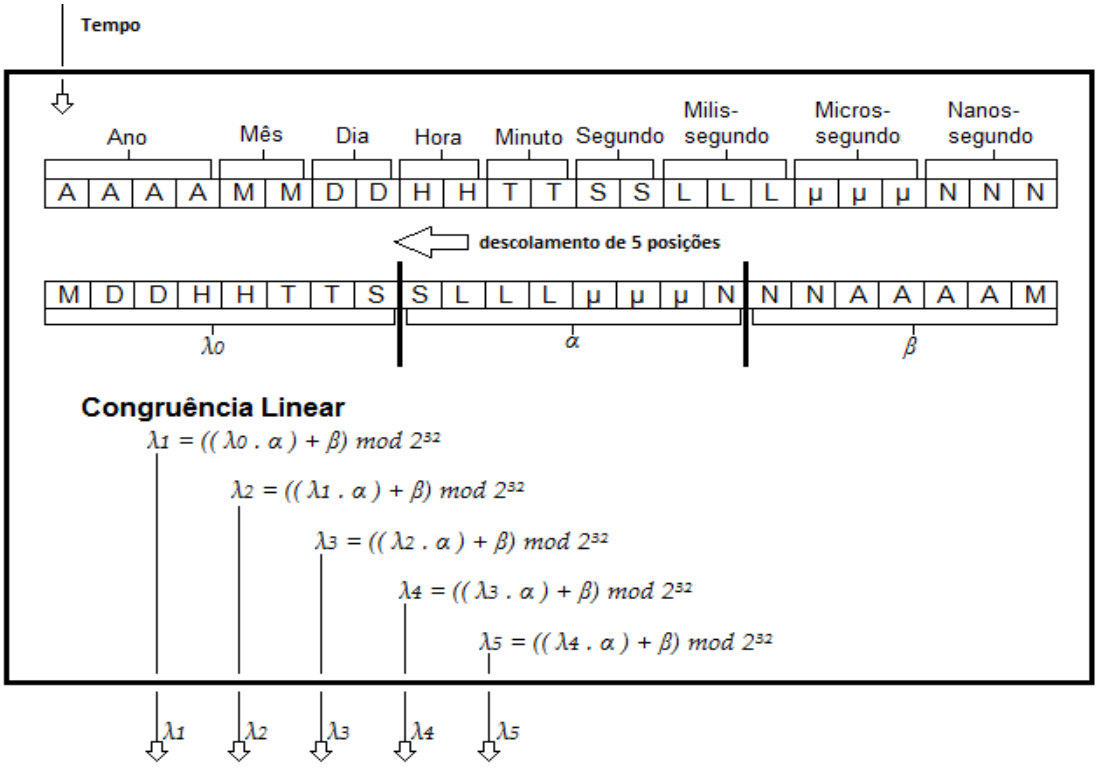


Figura 6 - Caixa Congruente. Fonte: O autor.

4 Documentação de software

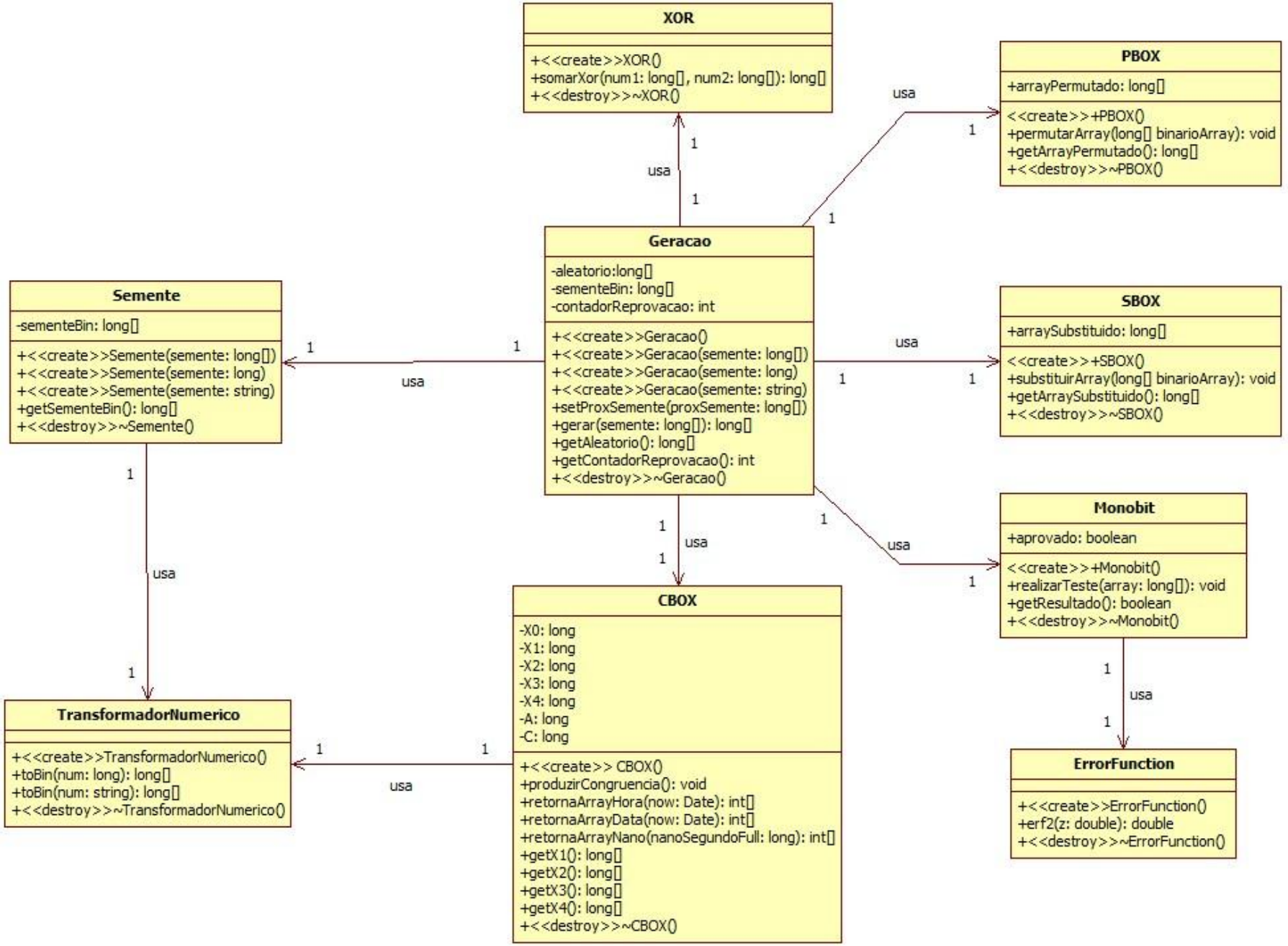
Neste capitulo será mostrada a documentação de software referente ao GNPA proposto, o qual foi projetado utilizando a metodologia da Programação Orientada a Objetos (POO); e também a utilidade de cada ferramenta utilizada, são elas: diagrama de classe, diagrama de sequência e diagrama de componente.

4.1 Diagrama de classe

Diagrama de classes tem por objetivo expor os tipos (ou classes[9]) que foram construídos para a modelagem dos objetos[10] pertencentes a um programa ou sistema, bem como as associações entre tais classes.

9 "É uma abstração das características de um grupo de coisas do mundo real" (BEZERRA, 2007)

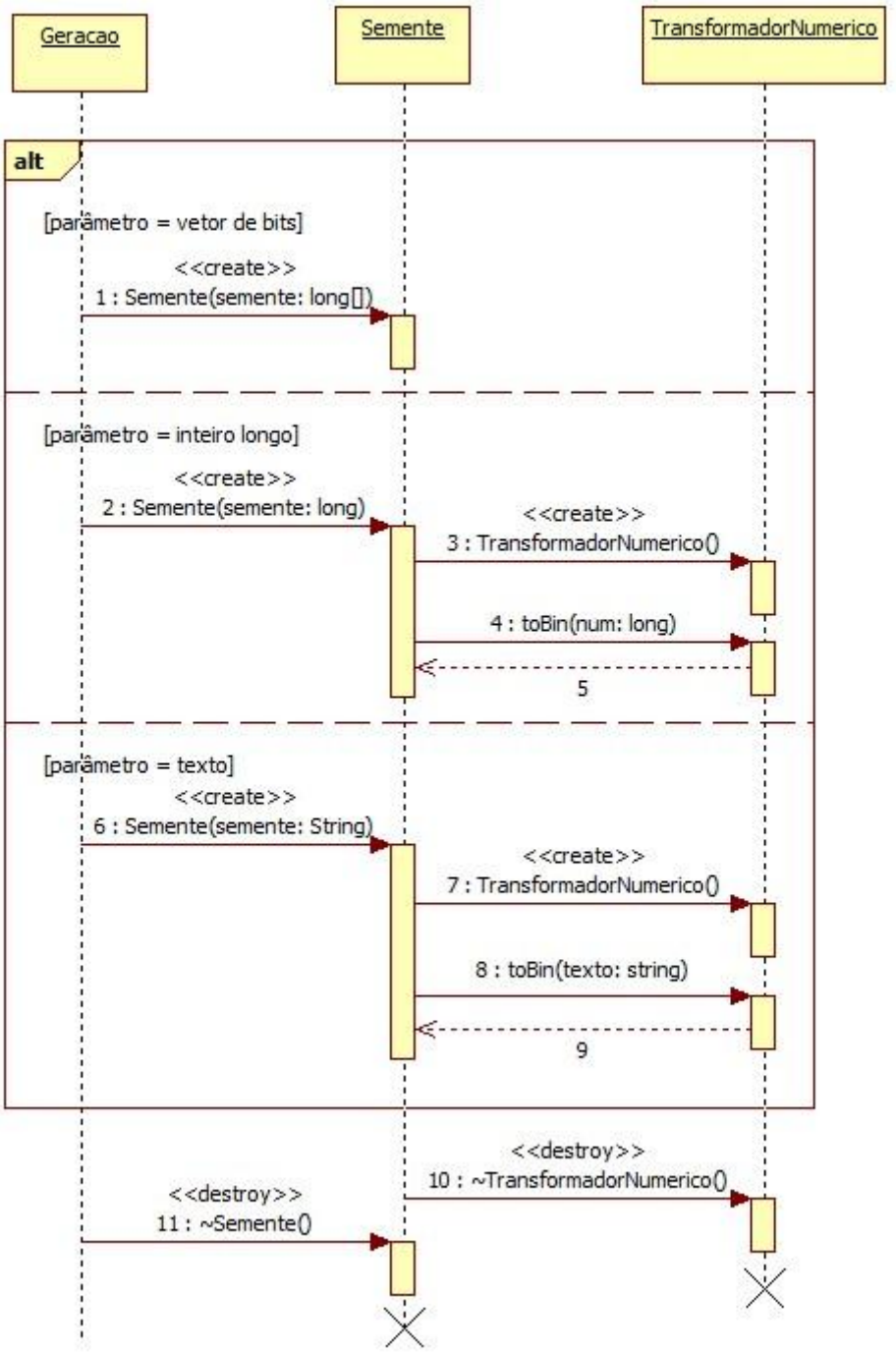
10 É uma variável de programação cujo o tipo é determinado por uma classe; também conhecido como instancia de uma classe.


XOR
+<<create>>XOR()
+somarXor(num1: long[], num2: long[]): long[]
+<<destroy>>~XOR()
PBOX
+arrayPermutado: long[]
<<create>>+PBOX()
+permutarArray(long[] binarioArray): void
+getArrayPermutado(): long[]
+<<destroy>>~PBOX()
Geracao
-aleatorio:long[]
-sementeBin: long[]
-contadorReprovacao: int
+<<create>>Geracao()
+<<create>>Geracao(semente: long[])
+<<create>>Geracao(semente: long)
+<<create>>Geracao(semente: string)
+setProxSemente(proxSemente: long[])
+gerar(semente: long[]): long[]
+getAleatorio(): long[]
+getContadorReprovacao(): int
+<<destroy>>~Geracao()
Semente
-sementeBin: long[]
+<<create>>Semente(semente: long[])
+<<create>>Semente(semente: long)
+<<create>>Semente(semente: string)
+getSementeBin(): long[]
+<<destroy>>~Semente()
SBOX
+arraySubstituido: long[]
<<create>>+SBOX()
+substituirArray(long[] binarioArray): void
+getArraySubstituido(): long[]
+<<destroy>>~SBOX()
Monobit
+aprovado: boolean
<<create>>+Monobit()
+realizarTeste(array: long[]): void
+getResultado(): boolean
+<<destroy>>~Monobit()
CBOX
-X0: long
-X1: long
-X2: long
-X3: long
-X4: long
-A: long
-C: long
+<<create>> CBOX()
+produzirCongruencia(): void
+retornaArrayHora(now: Date): int[]
+retornaArrayData(now: Date): int[]
+retornaArrayNano(nanoSegundoFull: long): int[]
+getX1(): long[]
+getX2(): long[]
+getX3(): long[]
+getX4(): long[]
+<<destroy>>~CBOX()
TransformadorNumerico
+<<create>>TransformadorNumerico()
+toBin(num: long): long[]
+toBin(num: string): long[]
+<<destroy>>~TransformadorNumerico()
ErrorFunction
+<<create>>ErrorFunction()
+erf2(z: double): double
+<<destroy>>~ErrorFunction()
usa
1

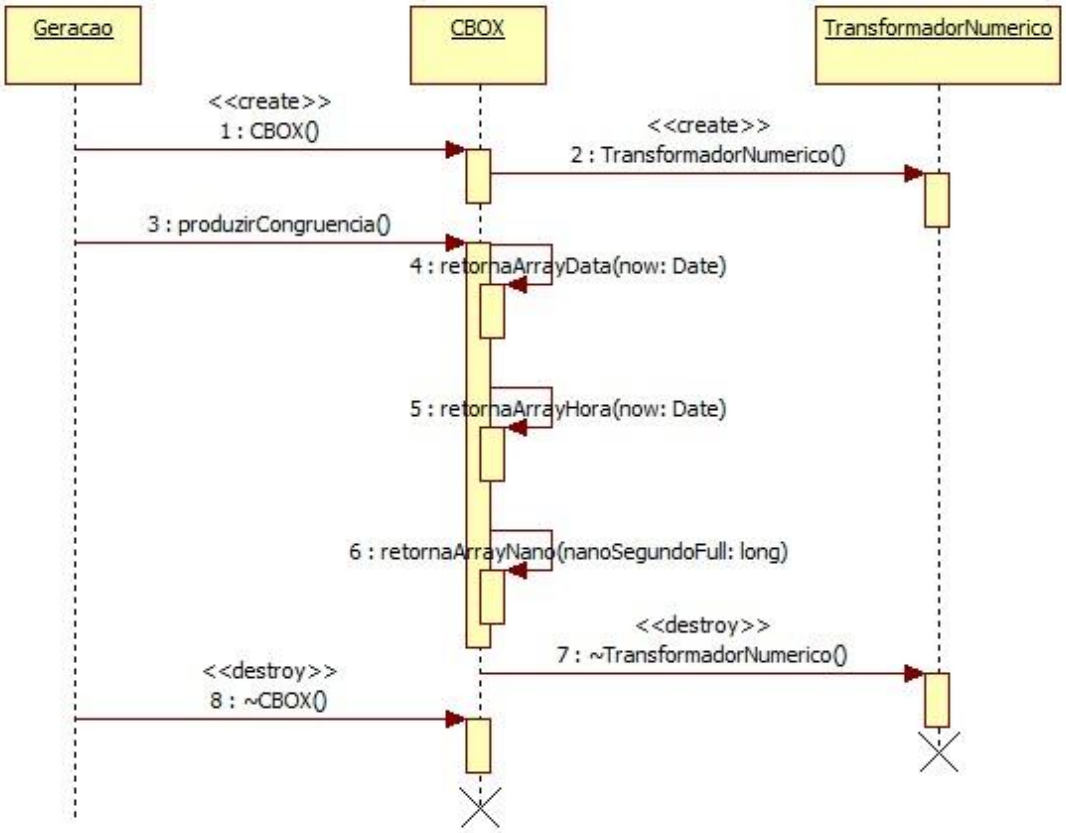
4.2 Diagrama de sequência

Diagrama de sequência tem por objetivo mostrar as interações entre objetos no momento de execução em que elas acontecem (BEZERRA, 2007).

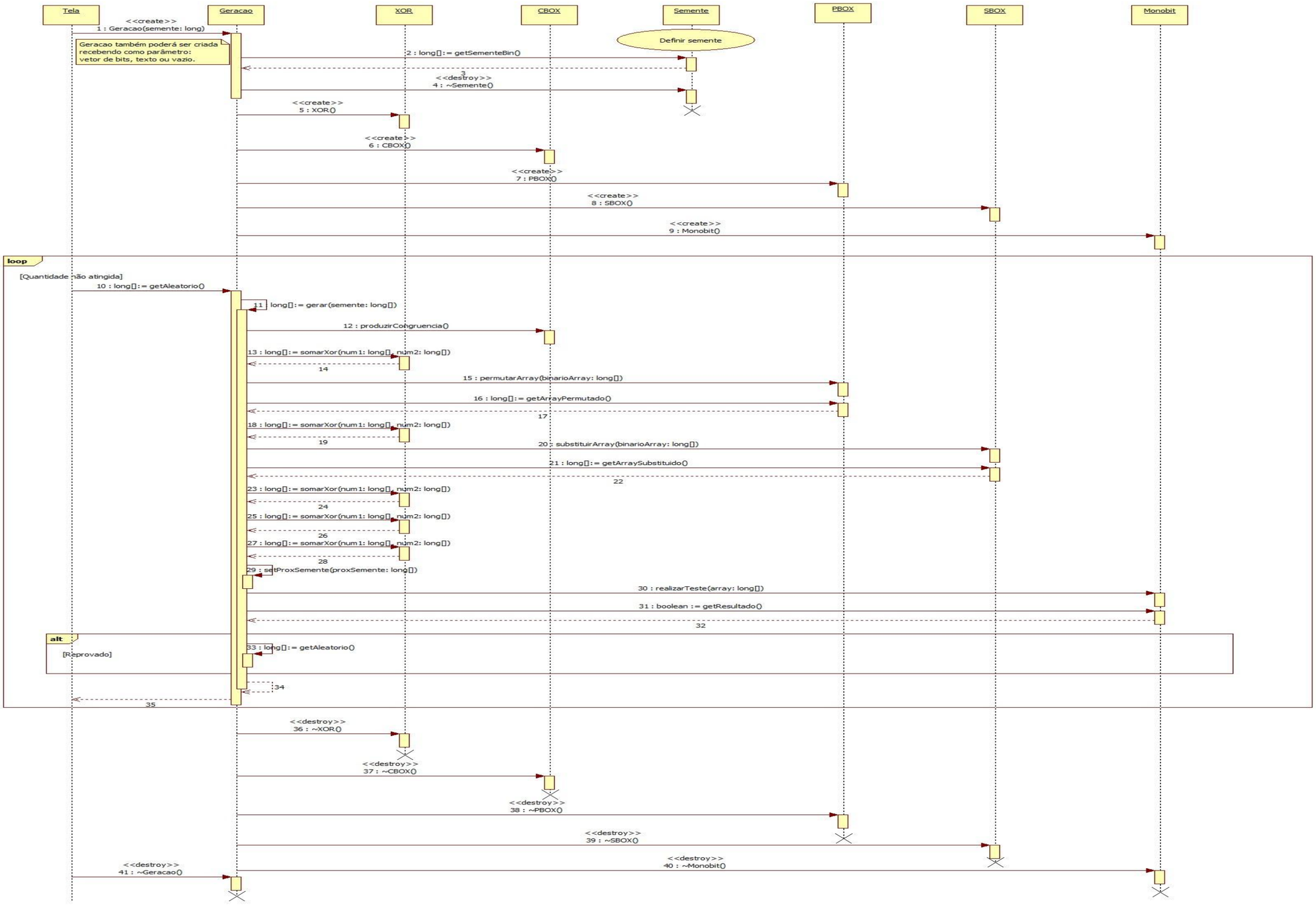
4.2.1 Diagrama de sequência "Definir Semente"



4.2.2 Diagrama de sequência "Iniciar CBOX"



4.2.3 Diagrama de sequência "Gerar números pseudo-aleatórios"



5 Demonstração do GNPA proposto

Neste capítulo serão realizadas duas demonstrações do funcionamento do GNPA proposto: a primeira irá demonstrar a geração de dois números pseudo-aleatórios passo a passo com a mesma *semente*; a segunda irá mostrar duas sequências numéricas produzidas, expondo a *semente* e o tempo utilizado em ambas. O objetivo deste capítulo é auxiliar a implementação desse gerador por outros usuários ou desenvolvedores e também servir como referência para futuras implementações, de forma que, ao implementar este algoritmo pode-se verificar se os valores obtidos equivalem aos expostos neste capítulo. Para fins de testes, a variável temporal deverá ser simulada prefixando seu valor. Em ambas demonstrações será utilizado como *semente* o valor 123.

5.1 Primeira demonstração

O objetivo desta seção é mostrar uma forma auxiliar de compreensão do funcionamento do GNPA proposto, gerando dois números pseudo-aleatórios passo a passo, e expondo-os na Tabela 4 da seguinte forma: Representação, Valor 1, Operação, Valor 2 e Resultado.

O campo "Representação" irá mostrar de forma simplificada o que será feito nos demais campos, por exemplo: Semente $\oplus$ X1 está mostrando que será feita a operação XOR entre Semente e X1 correspondentes aos campos "Valor 1" e "Valor 2" respectivamente. Algumas etapas da geração são compostas por dois valores submetidos a operação $\oplus$; esse é o motivo de se utilizar os campos denominados "Valor 1" e "Valor 2". Por outro lado, em outras etapas necessita-se apenas do "Valor 1" para realizar a operação; por isso, em alguns casos, o campo "Valor 2 " estará vazio. O campo "Resultado" será sempre utilizado como o "Valor 1" da etapa seguinte, ou seja, o i-ésimo "Resultado", será o (i-ésimo)+1 "Valor 1".

Representação	Valor 1	Operação	Valor 2	Resultado
Semente ⊕ X1	00000000000000000000000001111011	⊕	00100110001000011110000101101011	00100110001000011110000100010000
PBOX(Resultado1)	00100110001000011110000100010000	PBOX		01001010001001000001111100000000
Resultado2 ⊕ X2	01001010001001000001111100000000	⊕	01000000011111001111010100000111	00001010010110001110101000000111
SBOX(Resultado3)	00001010010110001110101000000111	SBOX		11100110111100110000011011101000
Resultado4 ⊕ X3	11100110111100110000011011101000	⊕	00100001110111101001101100011111	11000111001011011001110111110111
Resultado5 ⊕ X4	11000111001011011001110111110111	⊕	00100001111000101110110000001111	11100110110011111011000111111000

Representação	Valor 1	Operação	Valor 2	Resultado
Semente ⊕ X1	00000000000000000000000001111011	⊕	10000000010111100111100001100111	10000000010111100111100000011100
PBOX(Resultado1)	10000000010111100111100000011100	PBOX		10010000100011110101010000110010
Resultado2 ⊕ X2	10010000100011110101010000110010	⊕	11110010001000001101001111001110	01100010101011110011110111010100
SBOX(Resultado3)	01100010101011110011110111010100	SBOX		10111101011001110001100110010010
Resultado4 ⊕ X3	10111101011001110001100110010010	⊕	10010010100110101111111001011011	00101111111111011110011111001001
Resultado5 ⊕ X4	00101111111111011110011111001001	⊕	10101101111010001001101101100010	10000010000101010111110010101011

Tabela 4 - Geração passo a passo. Fonte: O autor.

Os dois resultados finais obtidos "11100110110011110111000111111000" e "10000010000101010111110010101011" convertidos de binário para decimal são 3872354808 e 2182446251, respectivamente.

5.2 Segunda demonstração

O objetivo desta seção é mostrar que é possível produzir duas sequências numéricas de dez números pseudo-aleatórios distintas, mesmo que o usuário informe a mesma *semente* nessas sequências. Para melhor visualização nessa demonstração, os números gerados serão convertidos de binário para decimal.

Ao gerar dez números pseudo-aleatórios praticamente não há alteração do tempo, pois a geração desses números leva menos de um segundo, ou seja, o ano, mês, dia, hora, minuto e segundo são constantes; ao passo que milissegundo, microssegundo e nanossegundo podem variar a cada número gerado. As variações dessas unidades de tempo que efetivamente variam estarão dispostas juntas aos números pseudo-aleatórios na Tabela 5, abreviadas da seguinte forma: M, μ, N, respectivamente.

Portanto, os valores do tempo utilizado na CBOX são os seguintes:

- Para dia, mês e ano foram tomados os valores 02, 11 e 2012, respectivamente;
- Para hora, minuto e segundo foram considerados os valores 12, 45 e 24, respectivamente.

	M : μ : N	Número gerado		M : μ : N	Número gerado
1°	766 : 908 : 581	936293498	**1°**	860 : 091 : 986	4183303171
2°	771 : 537 : 168	304395822	**2°**	864 : 652 : 324	212181665
3°	772 : 316 : 638	2056600997	**3°**	865 : 441 : 031	2657466985
4°	773 : 197 : 198	625408911	**4°**	866 : 225 : 120	3211134920
5°	774 : 592 : 958	3084451425	**5°**	867 : 006 : 643	1629943618
6°	775 : 401 : 165	2820938596	**6°**	867 : 784 : 574	2738273633
7°	776 : 182 : 688	471615226	**7°**	868 : 572 : 255	1110382961
8°	777 : 050 : 933	2804775214	**8°**	869 : 361 : 988	3857537186
9°	777 : 946 : 888	474280655	**9°**	870: 141: 459	2735013788
10°	778 : 714 : 043	3653483653	**10°**	870 : 910 : 153	1475549517

Tabela 5 - Demonstração do GNPA proposto. Fonte: O autor.

6 Experimentos

Foram realizados três experimentos que buscam analisar o desempenho, a taxa de reprovação no teste Monobit e o nível de aleatoriedade através da estatística qui-quadrado. Esses experimentos tem por objetivo comparar o gerador proposto com outros geradores de números pseudo-aleatórios.

Para a realização dos experimentos foram escolhidos quatro GNPA: *java.util.Random* que é o gerador da linguagem de programação Java, *rand* que é o gerador da linguagem de programação *C*, o gerador proposto por Schrage em (SCHRAGE, 1979) e, por último, o gerador proposto por Neumann, chamado *método do quadrado do meio*.

As *sementes* utilizadas nos testes de cada experimento estão descritas na Tabela 6, salvo as seguintes exceções: o 1° teste do gerador proposto por Linus Schrage utilizou como *semente* o número dez; o 1° e 2° teste do *método quadrado do meio* utilizou como *semente* os números dez e onze respectivamente. Tais exceções são necessárias para evitar que os GNPA produzam uma sequência de números repetidos, conforme já mencionado na seção 1.1.

Teste	Semente	Teste	Semente
1° Teste	0	6° Teste	5
2° Teste	1	7° Teste	6
3° Teste	2	8° Teste	7
4° Teste	3	9° Teste	8
5° Teste	4	10° Teste	9

Tabela 6 - Sementes utilizadas. Fonte: O autor.

Os experimentos foram realizados em um computador com a seguinte configuração: Windows 7 (64 bits), processador Intel Core i7-2630QM, 8Gb de memória RAM. Todos os geradores foram implementados na linguagem Java e seus respectivos resultados estão expostos na Tabela 7.

O primeiro experimento teve como objetivo testar a velocidade média do gerador proposto ao gerar dez milhões de números pseudo-aleatórios. Para obter a média citada acima, foi calculada a média aritmética do tempo levado para gerar em dez testes, dez milhões de números.

O segundo experimento buscou calcular a porcentagem média de reprovação dos números pseudo-aleatórios gerados submetidos ao teste Monobit, para obter a média citada acima foi calculada a média aritmética da porcentagem de reprovação ao gerar em dez testes, dez milhões de números.

O terceiro experimento buscou avaliar a média da estatística qui-quadrado de cada gerador com 63 graus de liberdade. Vale lembrar que, quanto menor o valor obtido, mais aleatório é considerado o conjunto gerado, ou seja, que a sequência produzida está mais próxima da probabilidade esperada. Foram gerados 2^{16} conjuntos com números pseudo-aleatórios entre 0 e 63.

	Desempenho	Reprovação	χ^2
Gerador proposto	104 segundos	0%	55,65
java.util.Random	14 segundos	5,27%	60,73
rand	25 segundos	5,27%	85,25
Schrage	25 segundos	5,27%	148,52
Quadrado do meio	15 segundos	100%	4128640,00

Tabela 7 - Resultado dos experimentos. Fonte: O autor.

7 Conclusão

Foi proposto neste Trabalho de Conclusão de Curso (TCC) um gerador de números pseudo-aleatórios de *palavras* de 32 bits com o teste de Frequência Monobit acoplado durante a geração. Esse GNPA foi baseado em três princípios de projeto para a construção de algoritmos criptográficos, e também baseado no sistema criptográfico *one-time pad*, o qual atribui-se características aleatórias nas *sementes* utilizadas, e a não reutilização da mesma.

O uso do tempo no decorrer do algoritmo tornou o gerador flexível, pois o tempo está presente em diversos sistemas. Ao utilizar *confusão* e *difusão* durante a geração, perturba-se a linearidade do tempo, mantendo a simplicidade do gerador. Ao adicionar o teste estatístico reconhecido internacionalmente, denominado teste Monobit, durante a geração, acrescentou-se credibilidade ao gerador.

Os resultados expostos no capítulo 4 mostram que este gerador, de acordo com o teste Monobit e χ^2, produz números que possuem alto grau de aleatoriedade. Portanto o gerador proposto está classificado para a realização dos demais testes estatísticos do "NIST Test Suite". No entanto, a velocidade durante a geração dos dez milhões de números do gerador proposto é baixa, se comparada à velocidade dos geradores da linguagem de programação Java e C, gerador proposto por Linus Schrage, e o método do quadrado do meio.

Como trabalhos futuros, podem ser realizadas pesquisas sobre como melhorar o gerador proposto, tendo como objetivo inicial o aumento da velocidade de geração, realizando a implementação de threads, ou reduzir o número de operações XOR sem afetar a qualidade dos números gerados. Além disso, deve-se realizar a implementação dos demais testes do "NIST Test Suite" a fim de aumentar a credibilidade deste gerador.

Referências Bibliográficas

BEZERRA, E. Princípios de análise e projeto de sistemas com UML. 2nd ed. Rio de Janeiro. Elsevier. 2007.

CAMPANI, C. A. P.; MENEZES, P. B. Teorias da Aleatoriedade. Revista de Informática Teórica e Aplicada, v. 11, n. 2, p. 75–98. 2004. Disponível em: <http://seer.ufrgs.br/rita/article/view/rita_v11_n2_p75-98>. Acesso em: 10 de maio 2012.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 2nd ed. The MIT Press, 2001.

COSTA JR, E. A. DA. Gerador de Números Randômicos para Criptografia. 2006. Disponível em: <http://www.icts.org.br/download/Randomicos_Edson.pdf>. Acesso em: 09 de mar. 2012.

COSTA, S. F. Introdução ilustrada à estatística. 4th ed. Harbra. Disponível em: <http://books.google.com.br/books?id=RGqjAAAACAAJ>. 2005.

CURI, F. Q. Proposta de sistema eficiente e seguro de encriptação sequencial baseado no one-time pad. 2009. Dissertação (Mestrado). Instituto Militar de Engenharia. Rio de Janeiro 2009. Disponível em: <http://pgee.ime.eb.br/pdf/felipe_curi.pdf>. Acesso em: 09 de out. 2012.

LAMBERT, J. A. Cifrador simétrico de blocos: projeto e avaliação. 2004. Dissertação (Mestrado). Instituto Militar de Engenharia. Rio de Janeiro. 2004.

MARSAGLIA, G. Diehard Battery of Tests of Randomness. The Marsaglia Random Number CDROM including the Diehard Battery of Tests. 1995. Disponível em: <http://stat.fsu.edu/pub/diehard> . Acesso em: 12 de abr. 2012.

ROSA, F. H. F. P. DA; PEDRO, V. A. Gerando Números Aleatórios. 2002. Disponível em: <http://www.feferraz.net/files/lista/random_numbers.pdf>. Acesso em: 09 de mar. 2012.

RUKHIN, A.; SOTO, J.; NECHVATAL, J. et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST - Special Publication 800-22. 2010. Disponível em: <http://goo.gl/rnv5v, 2010, April>. Acesso em: 12 de ago. 2012.

SCHNEIER, B. Applied Cryptography. 2nd ed. New York, NY, USA: John Wiley & Sons, 1996.

SCHRAGE, L. A More Portable Fortran Random Number Generator. ACM Trans. Math. Softw., v. 5, n. 2, p. 132–138. doi: 10.1145/355826.355828. 1979.

SEDGEWICK, R; KEVIN, W. ErrorFunction.java. Disponível em: <http://introcs.cs.princeton.edu/java/21function/ErrorFunction.java.html>. Acesso em: 13 de ago. 2012.

SIMPLICIO JR, M. A. Algoritmos criptográficos para redes de sensores. Dissertação (Mestrado), p. 179. 2008. Disponível em: <http://www.larc.usp.br/~mjunior/files/br/dissertacao-marcos-simplicio.pdf>. Acesso em: 02 de out. 2012.

SOUZA, G. S.; ALVES JR, N. Geradores de números aleatórios. Notas Técnicas - CBPF, v. 2, n. 1. 2011. Disponível em: <http://notastecnicas.cbpf.br/index.php/nt/issue/view/9>. Acesso em: 10 de maio 2012.

STALLINGS, W. Criptografia e segurança de redes. 4th ed. São Paulo: Pearson Prentice Hall. 2008.

VIEIRA, C. E. C.; SOUZA, R. DE C. E; RIBEIRO, C. DA C. C. Um estudo comparativo entre três geradores de números aleatórios, PUC-RioInf. p.33. Technical Report, Rio de Janeiro: PUC. 2004. Disponível em: <http://bib-di.inf.puc-rio.br/techreports/2004.htm>. Acesso em: 10 de maio 2012.

ANEXO 1

Pseudocódigo referente à função "gerar".

ENTRADA: Semente (Valor que iniciará a geração de números)

SAÍDA: Sequência binária aleatória

VARIÁVEIS

estado : natural

semente: caractere

resultado: lógico

INÍCIO

ler (semente)

estado ← semente $\oplus$ λ_1 [CBOX]

estado ← PBOX(estado)

estado ← estado $\oplus$ λ_2 [CBOX]

estado ← SBOX(estado)

estado ← estado $\oplus$ λ_3[CBOX]

estado ← estado $\oplus$ λ_4[CBOX]

resultado ← Monobit(estado)

semente = estado

SE resultado = = falso

INÍCIO

gerar(estado)

FIM

FIM SE

retorna estado

FIM.

ANEXO 2

Pseudocódigo referente à função "produzirCongruencia".

ENTRADA: tempo (tempo do sistema hospedeiro (ano, mês, dia, hora, minuto e etc))

SAÍDA: $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ (números inteiros gerados através da congruência linear)

1: VARIÁVEIS

2: α, β , $\lambda_1, \lambda_2, \lambda_3, \lambda_4$: inteiro

INÍCIO

A:inteiro = $\{a_0, a_1, a_2, a_3, \}$

B:inteiro = $\{b_0, b_1\}$

C:inteiro = $\{c_0, c_1\}$

D:inteiro = $\{d_0, d_1\}$

E:inteiro = $\{e_0, e_1\}$

F:inteiro = $\{f_0, f_1\}$

G:inteiro = $\{g_0, g_1, g_2\}$

H:inteiro = $\{h_0, h_1, h_2\}$

L: inteiro = $\{l_0, l_1, l_2\}$

$\lambda_0 = \{b_1, c_0, c_1, d_0, d_1, e_0, e_1, f_0\}$

$\alpha = \{f_1, g_0, g_1, g_2, h_0, h_1, h_2, l_0\}$

$\beta = \{l_1, l_2, a_0\ a_1, a_2, a_3, b_0\}$

$\lambda_1 \leftarrow ((\lambda_0 * \alpha) + \beta)\ mod\ 2^{32}$

$\lambda_2 \leftarrow ((\lambda_1 * \alpha) + \beta)\ mod\ 2^{32}$

$\lambda_3 \leftarrow ((\lambda_2 * \alpha) + \beta)\ mod\ 2^{32}$

$\lambda_4 \leftarrow ((\lambda_3 * \alpha) + \beta)\ mod\ 2^{32}$

FIM.

ANEXO 3

Pseudocódigo referente à função "permutarArray".

ENTRADA: estado (sequência binária de 32bits)

SAÍDA: sequência binária de 32bits permutada

VARIÁVEIS

estado = inteiro[1..32]

arrayPermutado = inteiro[1..32]

INÍCIO

arrayPermutado[0] ← estado [13]

arrayPermutado[1] ← estado [10]

arrayPermutado[2] ← estado [4]

arrayPermutado[3] ← estado [11]

arrayPermutado[4] ← estado [6]

arrayPermutado[5] ← estado [3]

arrayPermutado[6] ← estado [23]

arrayPermutado[7] ← estado [1]

arrayPermutado[8] ← estado [9]

arrayPermutado[9] ← estado [7]

arrayPermutado[10] ← estado [15]

arrayPermutado[11] ← estado [25]

arrayPermutado[12] ← estado [20]

arrayPermutado[13] ← estado [18]

arrayPermutado[14] ← estado [28]

arrayPermutado[15] ← estado [0]

arrayPermutado[16] ← estado [22]

arrayPermutado[17] ← estado [29]

arrayPermutado[18] ← estado [8]

arrayPermutado[19] ← estado [17]

arrayPermutado[20] ← estado [5]

arrayPermutado[21] ← estado [27]

arrayPermutado[22] ← estado [2]

arrayPermutado[23] ← estado [16]

arrayPermutado[24] ← estado [31]

arrayPermutado[25] ← estado [26]

arrayPermutado[26] ← estado [12]

arrayPermutado[27] ← estado [19]

arrayPermutado[28] ← estado [24]

arrayPermutado[29] ← estado [30]

arrayPermutado[30] ← estado [14]

arrayPermutado[31] ← estado [21]

retorna arrayPermutado

FIM.

ANEXO 4

Pseudocódigo referente à função “substituirArray”.

ENTRADA: estado(sequência binária de 32bits)

SAÍDA: sequência binária de 32 bits substituída com base em uma tabela pré-determinada

---

VARIÁVEIS

estado = inteiro[1..32]

arraySubstituido = inteiro[1..32]

INÍCIO

PARA i←0 até 31 FAÇA

INÍCIO

SE i+3 < 32 ENTÃO

SE binarioArray[i]==0 ^[11] binarioArray[i+1]==0 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 0;

FIM SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

---

11 Operador que representa a operação lógica "e".

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

SENÃO SE binarioArray[i]==0 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 0;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 1;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==0 ^ binarioArray[i+2]==1 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 0;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==0 ENTÃO

arraySubstituido[i] = 0;

arraySubstituido[i+1] = 1;

arraySubstituido[i+2] = 0;

arraySubstituido[i+3] = 1;

FIM SENÃO SE

SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==0 ^ binarioArray[i+3]==1 ENTÃO

arraySubstituido[i] = 1;

arraySubstituido[i+1] = 0;

```
                arraySubstituido[i+2] = 0;
                arraySubstituido[i+3] = 1;
        FIM SENÃO SE
        SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==1
^ binarioArray[i+3]==0 ENTÃO
                arraySubstituido[i] = 0;
                arraySubstituido[i+1] = 0;
                arraySubstituido[i+2] = 0;
                arraySubstituido[i+3] = 0;
        FIM SENÃO SE
        SENÃO SE binarioArray[i]==1 ^ binarioArray[i+1]==1 ^ binarioArray[i+2]==1
^ binarioArray[i+3]==1 ENTÃO
                arraySubstituido[i] = 0;
                arraySubstituido[i+1] = 1;
                arraySubstituido[i+2] = 1;
                arraySubstituido[i+3] = 1;
        FIM SENÃO SE
        }
       i←i+4
       FIM PARA
   FIM.
```