UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Aluno: Felipe Alvarenga Silva Murta (23102757)
Disciplina INE5429 – Segurança em Computação
Prof. Jean Everson Martina e Profª. Thaís Bardini Idalino

Pseudo-aleatoriedade e Primalidade: geração e verificação

Florianópolis
2025

1 Introdução

A segurança computacional moderna depende fundamentalmente da criptografia, a ciência de proteger informações por meio de codificação. No coração de muitos algoritmos criptográficos, como o RSA e o Diffie-Hellman, residem os números primos. A dificuldade de fatorar números muito grandes em seus componentes primos é o que garante a robustez desses sistemas.

O processo de encontrar um número primo tão grande geralmente envolve duas etapas principais: primeiro, a geração de um candidato, que é um número ímpar aleatório da ordem de grandeza desejada; segundo, a aplicação de testes de primalidade para verificar se o candidato é, com uma probabilidade suficientemente alta, realmente primo. Este trabalho explora ambas as etapas. A primeira seção abordará a geração de números pseudo-aleatórios, com a descrição, implementação e análise de dois algoritmos: o Linear Congruential Generator e o Blum Blum Shub. A segunda seção focará nos testes de primalidade, implementando e comparando o teste de Miller-Rabin com o teste de Fermat para validar os números gerados anteriormente.

O link para o repositório está disponível aqui e uma pasta zip com o código-fonte também será enviada pelo Moodle. As orientações de compilação e execução estão disponíveis no arquivo *README.MD*.

2 Números pseudo-aleatórios

Geradores de números aleatórios criam sequências de números aparentemente aleatórias, a partir de parâmetros iniciais quaisquer. Os números gerados não são efetivamente aleatórios, mas buscam possuir o máximo possível de propriedades que o caracterizem como tal.

Os algoritmos desenvolvidos foram o Linear Congruential Generator e o Blum Blum Shub.

2.1 Linear Congruential Generator

O Gerador Linear Congruencial (LCG) [9] é um dos algoritmos mais antigos e mais conhecidos para a geração de números pseudo-aleatórios. Sua popularidade deriva de sua simplicidade e eficiência computacional, sendo fácil de implementar e rápido para gerar sequências de números, apesar de não ser considerado criptograficamente seguro e suficientemente "aleatório"[5]. O algoritmo baseia-se em uma relação de recorrência linear definida pela fórmula:

$$X_{n+1} = (a \cdot X_n + c) \pmod{m}$$

Os parâmetros a (multiplicador) e c (incremento) na implementação foram escolhidos com base em constantes bem estabelecidas e testadas, propostas por Donald E. Knuth para o gerador de seu computador MMIX [8]. Esses valores, em conjunto com um módulo m na forma de uma potência de dois ($m = 2^k$), satisfazem as condições do Teorema de Hull-Dobell [7], garantindo que o gerador atinja seu período máximo possível. A escolha de constantes proeminentes como estas também assegura boas pro-

priedades estatísticas, evitando correlações indesejadas nos números gerados, conforme detalhado na obra de referência sobre o assunto.

A base de código apresentada foi desenvolvida em Java, utilizando a classe `BigInteger`. Esta escolha é fundamental para atender aos requisitos do trabalho, pois `BigInteger` permite a manipulação de números inteiros de precisão arbitrária, tornando possível gerar valores com centenas ou milhares de bits. A classe `LcgGenerator` é inicializada com um construtor que recebe o tamanho desejado em bits (`bitLength`). O módulo m é então definido como $2^{\texttt{bitLength}}$, o que é eficientemente calculado através da operação `BigInteger.ONE.shiftLeft(bitLength)`. A semente inicial é obtida a partir do tempo do sistema em nanossegundos (`System.nanoTime()`), uma abordagem comum para introduzir alguma imprevisibilidade no estado inicial. O método `next()` aplica diretamente a fórmula de recorrência para gerar cada novo número pseudo-aleatório da sequência.

2.2 Blum Blum Shub

O gerador Blum Blum Shub (BBS) é um algoritmo de geração de números pseudo-aleatórios que, ao contrário de geradores mais simples como o LCG, é considerado criptograficamente seguro [10]. Sua segurança teórica baseia-se na dificuldade computacional da fatoração de inteiros, um dos problemas fundamentais da teoria dos números que sustenta a criptografia de chave pública, como o RSA. O algoritmo opera a partir de uma relação de recorrência quadrática:

$$X_{n+1} = X_n^2 \pmod{M} \tag{1}$$

Onde o módulo M é o produto de dois números primos grandes, p e q, escolhidos de forma que ambos sejam congruentes a 3 módulo 4. Esses primos são chamados de "primos de Blum". A semente, X_0, é um inteiro coprimo com M. A cada iteração, o bit pseudo-aleatório gerado é tipicamente o bit menos significativo (LSB) do estado X_{n+1}. A previsibilidade da sequência de bits gerada é computacionalmente equivalente à fatoração do módulo M [1].

A implementação em Java deste gerador também faz uso de `java.math.BigInteger` para manipular os grandes números necessários. O processo de inicialização, realizado no construtor, é a parte mais complexa e computacionalmente intensiva do algoritmo. Primeiramente, ele busca por dois primos de Blum (p e q) com aproximadamente metade do tamanho de bits (`bitLength`) desejado para o número final. Para isso, o método auxiliar `findBlumPrime` gera prováveis primos e testa a condição de congruência ($p \bmod 4 = 3$) até encontrar um candidato válido. Após encontrar p e q distintos, o módulo M é calculado como $M = pq$. A semente inicial é definida usando a fonte de aleatoriedade `new Random(System.nanoTime())` para gerar um número x. O método `next()` gera um número de `bitLength` bits iterando a fórmula de recorrência, e a cada passo, o bit menos significativo do estado é extraído e concatenado ao resultado final através de operações de deslocamento de bits.

2.3 Resultados

A Tabela 1 apresenta os tempos de execução para a geração de números pseudo-aleatórios utilizando o algoritmo LCG. A observação mais imediata é a sua extrema

eficiência: mesmo para gerar 5000 números de 4096 bits, o tempo total foi de poucos milissegundos. Esse alto desempenho é um reflexo direto da simplicidade do algoritmo. A complexidade para gerar cada número de k bits é polinomial, dominada pela operação de multiplicação de BigInteger, que é aproximadamente O(k^2). Como o algoritmo consiste em uma única multiplicação, uma soma e uma operação de módulo para cada número gerado, seu custo computacional é muito baixo, tornando-o ideal para aplicações onde a velocidade é o principal requisito e a segurança criptográfica não é uma preocupação.

Tabela 1: Comparativo de Tempos de Geração para o Algoritmo LCG com 1000 e 5000 Repetições

Algoritmo	**Tamanho (bits)**	**Nº de Repetições**	**Tempo Total (ms)**
LCG	40	1000	0.4561
LCG	56	1000	0.2677
LCG	80	1000	0.2051
LCG	128	1000	0.2619
LCG	256	1000	0.2547
LCG	512	1000	0.5354
LCG	1024	1000	0.5104
LCG	2048	1000	0.6515
LCG	4096	1000	1.0519
LCG	40	5000	2.3320
LCG	56	5000	0.6668
LCG	80	5000	0.5768
LCG	128	5000	0.8223
LCG	256	5000	0.9659
LCG	512	5000	1.1404
LCG	1024	5000	1.5402
LCG	2048	5000	4.5565
LCG	4096	5000	5.0718

Em contraste com o LCG, os resultados para o gerador Blum Blum Shub (BBS) detalhados na Tabela 2 demonstram um custo computacional substancialmente mais elevado. Os tempos de execução crescem drasticamente com o aumento do número de bits, atingindo mais de 100 segundos para gerar 1000 números de 4096 bits. O tempo para 4096 bits e 5000 repetições não foi calculado pela desnecessidade de se esperar tanto tempo. A lentidão do BBS se deve a dois fatores. Primeiramente, a fase de inicialização do BBS é extremamente custosa, pois exige a busca por dois primos grandes que satisfaçam a congruência de Blum; contudo, tal fator é excluído do cálculo de tempo presente na tabela. Em segundo lugar, a geração de cada número é mais complexa. A complexidade para gerar um único número de k bits é da ordem de O(k^3), pois o laço principal executa k vezes uma operação de exponenciação modular quadrática (`modPow`), que por sua vez tem complexidade de O(k^2).

Tabela 2: Comparativo de Tempos de Geração para o Algoritmo Blum Blum Shub (BBS) com 1000 e 5000 Repetições

Algoritmo	**Tamanho (bits)**	**Nº de Repetições**	**Tempo Total (ms)**
BBS	40	1000	7.4966
BBS	56	1000	10.3712
BBS	80	1000	23.1352
BBS	128	1000	32.1783
BBS	256	1000	111.8144
BBS	512	1000	503.4510
BBS	1024	1000	2895.2676
BBS	2048	1000	21606.4588
BBS	4096	1000	119108.4206
BBS	40	5000	35.9550
BBS	56	5000	48.2268
BBS	80	5000	101.7225
BBS	128	5000	154.4213
BBS	256	5000	527.7955
BBS	512	5000	2486.8020
BBS	1024	5000	14872.5689
BBS	2048	5000	103979.5729
BBS	4096	5000	—

A comparação entre ambos algoritmos evidencia o clássico dilema do trade-off entre eficiência e segurança. O LCG se destaca pela velocidade imensa dada sua simplicidade, com complexidade de $O(K^2)$, porém é consideravelmente "previsível", e, portanto, inseguro para muitas aplicações criptográficas. Já o BBS baseia sua segurança em um problema matemático difícil, resultando em um desempenho muito inferior, de complexidade $O(K^3)$, mas garantindo a imprevisibilidade exigida em criptografia.

3 Testes de Primalidade

Testes de Primalidade são algoritmos para definir se um número inteiro qualquer é primo. Ao contrário do problema da fatoração, testes de primalidade não são computacionalmente difíceis, sendo usualmente polinomiais, como os implementados. Assim, são frequentemente utilizados em aplicações que necessitam de determinar primalidade rapidamente.

Os algoritmos implementados foram o de Fermat e o de Miller-Rabin. Na realidade, ambos algoritmos não são testes de primalidade no sentido literal; eles determinam se o número é composto, e a contrapositiva disto é utilizada para probabilisticamente dizer se um número seria primo ou não.

A escolha do Teste de Primalidade de Fermat como segundo algoritmo foi motivada por dois fatores principais. Primeiramente, por sua simplicidade conceitual e de implementação, que serve como uma excelente introdução aos testes probabilísticos. Em segundo lugar, e mais importante, sua conhecida vulnerabilidade aos números de Car-

michael oferece um contraponto ideal ao teste de Miller-Rabin, permitindo uma análise comparativa rica que destaca na prática a importância das garantias de segurança mais fortes.

3.1 Teste de Primalidade de Fermat

O Teste de Primalidade de Fermat é um dos algoritmos probabilísticos mais clássicos e se baseia no Pequeno Teorema de Fermat. O teorema afirma que, se n é um número primo, então para qualquer inteiro a que não seja um múltiplo de n, a seguinte congruência é válida [2]:

$$a^{n-1} \equiv 1 \pmod{n}$$

O teste utiliza a contrapositiva dessa afirmação: ele seleciona um número inteiro aleatório a (chamado de "testemunha") no intervalo $1 < a < n - 1$ e calcula o resultado da exponenciação modular. Se o resultado for diferente de 1, o número n é, com certeza, **composto**. Caso o resultado seja 1, o número n é considerado **provavelmente primo** [6]. A principal fragilidade deste teste é a existência de números compostos, conhecidos como números de Carmichael, que satisfazem a congruência para todas as bases a que são coprimas a eles. O menor desses números é 561. Devido a essa vulnerabilidade, o teste de Fermat é geralmente considerado menos confiável que alternativas mais modernas, como o teste de Miller-Rabin [4].

Para este trabalho, duas implementações foram desenvolvidas. A primeira, FermatTester, implementa o teste de forma direta. O método `isPrime` recebe o número `n` e a quantidade de iterações `certainty`. Dentro de um laço, ele seleciona uma testemunha `a` e realiza o cálculo principal através de uma única chamada ao método `a.modPow(n-1, n)`. Se o resultado desta operação for diferente de 1, o método retorna `false`. Caso o laço se complete, ele retorna `true`.

Para fins de análise e para demonstrar a falha teórica do algoritmo, uma segunda classe, `FermatWeakTester`, foi criada herdando da primeira. Esta versão sobrescreve o método `isPrime` para adicionar uma verificação explícita do máximo divisor comum (`a.gcd(n)`). Se a base `a` não for coprima de `n`, ela é descartada e uma nova é escolhida. Ao forçar o teste a usar apenas bases coprimas, garantimos que ele seja enganado por um número de Carmichael como 561, ilustrando de forma prática a sua principal deficiência.

3.2 Teste de Primalidade de Miller-Rabin

O Teste de Primalidade de Miller-Rabin é um refinamento sofisticado do teste de Fermat e é mais utilizado para verificação de primalidade em aplicações criptográficas. Sua robustez se baseia em uma análise mais estrita das propriedades das raízes quadradas de 1 módulo um número composto. Para um número ímpar n a ser testado, o algoritmo primeiro decompõe n-1 na forma $d \cdot 2^s$, onde d é ímpar. Uma testemunha aleatória a prova que n é composto se nenhuma das seguintes condições for satisfeita [3]:

- $a^d \equiv 1 \pmod{n}$
- $a^{d \cdot 2^r} \equiv -1 \pmod{n}$ para algum $0 \leq r < s$

A principal vantagem do Miller-Rabin é que não existem números compostos, análogos aos de Carmichael, que passem no teste para todas as bases coprimas. Para qualquer n composto, pelo menos 3/4 das bases possíveis são testemunhas de sua compostura, garantindo que a probabilidade de um falso positivo após k iterações seja inferior a $(1/4)^k$ [4].

A implementação deste algoritmo foi realizada na classe `MillerRabinTester`. O método `isPrime` inicia com as mesmas verificações de casos triviais das outras implementações. Em seguida, realiza a decomposição de `n-1` em `d` e `s` de forma eficiente, utilizando os métodos `getLowestSetBit()` para determinar `s` e `shiftRight(s)` para calcular `d`. O laço principal é executado `certainty` vezes, e a lógica de verificação foi refatorada em dois métodos auxiliares para maior clareza. O primeiro, `passesFirstCheck`, verifica a condição base, $a^d \pmod{n}$, que corresponde ao caso $r = 0$. Se esta verificação não for conclusiva, o segundo método, `passesSecondCheck`, executa um laço que eleva repetidamente o resultado ao quadrado (`modPow(TWO, n)`) para testar a segunda condição para r de 1 até $s - 1$. Este método também identifica corretamente n como composto caso uma raiz quadrada não trivial de 1 seja encontrada. Se o número n passar por todas as verificações para todas as testemunhas a sorteadas, ele é retornado como provavelmente primo.

3.3 Resultados

A metodologia para analisar os testes de primalidade difere daquela usada para os geradores. O foco desta seção é mensurar o tempo médio de uma única operação de verificação de primalidade, ao invés do tempo total para gerar múltiplos números. A razão para tal é simples: a geração de números primos é, essencialmente, dependente de sorte - o número de tentativas independe da eficiência do algoritmo de verificação, mas sim da "sorte"do número gerado ser primo (ou provavelmente primo). Os candidatos utilizados como entrada para estes testes foram obtidos através do gerador LCG previamente implementado.

Os experimentos a seguir foram conduzidos com o parâmetro de certeza fixado em $k = 100$, onde k representa o número de iterações do teste, ou seja, a quantidade de bases aleatórias distintas avaliadas para cada número candidato. Para cada cenário, a performance foi medida durante a busca por 100 números primos distintos. Um valor maior de k aumenta exponencialmente a confiança no resultado. No caso do teste de Miller-Rabin, a garantia teórica de pior caso estabelece que a probabilidade de um número composto ser falsamente identificado como primo é inferior a $(1/4)^k$. Contudo, como apontado por Cormen et al. [4], na prática de buscar primos testando números aleatórios, a maioria dos compostos é detectada com poucas iterações, sendo um valor de k baixo muitas vezes suficiente. Essa robustez é o principal diferencial do Miller-Rabin, que não possui vulnerabilidades conhecidas como a do teste de Fermat aos números de Carmichael, tornando-o o método probabilístico de escolha para aplicações criptográficas seguras.

Tabela 3: Benchmark de Tempo Médio por Verificação para o Teste de Fermat

Algoritmo	**Tamanho (bits)**	**Nº de Testes**	**Tempo Médio (ms)**
FermatTester	40	100	3.9914
FermatTester	56	100	4.4126
FermatTester	80	100	4.5224
FermatTester	128	100	5.6918
FermatTester	256	100	6.5003
FermatTester	512	100	12.8271
FermatTester	1024	100	38.3305
FermatTester	2048	100	232.2667
FermatTester	4096	100	1000.6167

A complexidade computacional do teste de Fermat é dominada pela sua operação central: a exponenciação modular $a^{n-1} \pmod{n}$, executada para cada uma das k testemunhas. Para um número n de b bits, a complexidade desta operação é aproximadamente $O(b^3)$. Isso ocorre porque o algoritmo de exponenciação por quadratura executa um número de multiplicações proporcionais ao número de bits do expoente (neste caso, b), e cada multiplicação de números de b bits tem um custo quadrático, $O(b^2)$. Esse crescimento cúbico no custo explica o aumento acentuado no tempo de execução observado na tabela para números com maior quantidade de bits.

Tabela 4: Benchmark de Tempo Médio por Verificação para o Teste de Miller-Rabin

Algoritmo	**Tamanho (bits)**	**Nº de Testes**	**Tempo Médio (ms)**
MillerRabinTester	40	100	10.4594
MillerRabinTester	56	100	3.9026
MillerRabinTester	80	100	4.5128
MillerRabinTester	128	100	2.3005
MillerRabinTester	256	100	7.0193
MillerRabinTester	512	100	22.9148
MillerRabinTester	1024	100	34.5257
MillerRabinTester	2048	100	202.6211
MillerRabinTester	4096	100	1005.6544

A complexidade do teste de Miller-Rabin é, em ordem de grandeza, similar à do teste de Fermat, sendo também dominada pela exponenciação modular. Para cada uma das k bases, o algoritmo executa uma exponenciação principal ($a^d \pmod{n}$) e, subsequentemente, uma série de até $s-1$ elevações ao quadrado. Para um número n de b bits, a exponenciação inicial tem custo de $O(b^3)$. Como o número de elevações ao quadrado, s, é no máximo b, a complexidade total para cada base testada permanece na ordem de $O(b^3)$. Embora a complexidade assintótica seja a mesma do teste de Fermat, o Miller-Rabin executa mais operações por iteração (as elevações ao quadrado adicionais), o que pode resultar em tempos de execução ligeiramente superiores, como sugerido pelos dados experimentais.

Os valores um pouco maiores para o teste de Miller-Rabin de 40 bits provavelmente

são relacionados ao custo de inicialização e otimização da Máquina Virtual Java (JVM). Java não executa código nativo diretamente; em vez disso, um compilador Just-In-Time (JIT) analisa o código em tempo de execução. Nas primeiras execuções de um método, o código é tipicamente "interpretado", um processo mais lento. Após ser identificado como um "hotspot"(trecho de código frequentemente executado), o JIT o compila para código de máquina otimizado, que é muito mais rápido.

4 Conclusão

Este trabalho demonstrou que a busca por números primos para fins criptográficos envolve um duplo *trade-off*: um entre desempenho e segurança na geração de candidatos, e outro entre simplicidade e robustez na verificação de primalidade. A análise revelou que geradores rápidos como o LCG são previsíveis e inseguros, enquanto testes simples como o de Fermat são vulneráveis a compostos específicos como os números de Carmichael. Portanto, conclui-se que a segurança em criptografia exige a escolha dos métodos mais robustos em ambas as frentes, como o gerador Blum Blum Shub e o teste de Miller-Rabin, aceitando o custo computacional significativamente maior como um requisito fundamental para garantir a imprevisibilidade e a confiabilidade do sistema.

Referências

[1] Lenore Blum, Manuel Blum e Michael Shub. "A Simple Unpredictable Pseudo-Random Number Generator". Em: *SIAM Journal on Computing* 15.2 (1986), pp. 364–383.

[2] Keith Conrad. *Fermat's Little Theorem*. https://kconrad.math.uconn.edu/blurbs/ugradnumthy/fermatlittletheorem.pdf. Accessed: YYYY-MM-DD. n.d.

[3] Keith Conrad. *The Miller–Rabin Test*. https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf. Accessed: YYYY-MM-DD. n.d.

[4] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. Cambridge, MA, USA: MIT Press, 2009.

[5] Frieze, Kannan e Lagaria. *Linear Congruential Generators — Text Notes*. Rel. téc. Carnegie Mellon University, n.d. URL: https://www.math.cmu.edu/~af1p/Texfiles/LINEARCONGRU.pdf.

[6] Mor Harchol-Balter. "Primality Testing". Em: *Introduction to Probability for Computing*. Accessed: 2025-09-21. Cambridge University Press, 2024. Cap. 23. URL: https://www.cs.cmu.edu/~harchol/Probability/chapters/chpt23.pdf.

[7] T. E. Hull e A. R. Dobell. "Random Number Generators". Em: *SIAM Review* 4.3 (jul. de 1962), pp. 230–254.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3ª ed. Reading, Massachusetts: Addison-Wesley Professional, 1997.

[9] William H. Press et al. “Numerical Recipes: The Art of Scientific Computing”. Em: 3ª ed. On Random Number Generation. Cambridge: Cambridge University Press, 2007. Cap. 7.

[10] Andrey Sidorenko e Berry Schoenmakers. “Concrete Security of the Blum-Blum-Shub Pseudorandom Generator”. Em: *Cryptography and Coding*. Vol. 3796. Lecture Notes in Computer Science. Springer, 2005, pp. 355–375. ISBN: 978-3-540-30276-6. DOI: 10.1007/11586821_24.