Trabalho sobre geração de números pseudo-aleatórios e geração de números primos

Segurança em Computação - INE5429

Universidade Federal de Santa Cataria - UFSC

Bernardo Schmidt Farias
19100519

Julho 2021

Resumo

Números primos possuem grande importância para a área de segurança, mas especificamente números primos de grandes ordens de grandeza. A segurança de métodos de criptografia assimétrica atuais se dá pela dificuldade de fatoração de um número composto como a multiplicação de dois primos, assim, o entendimento dos mecanismos de geração de números primos e verificação de primalidade se mostra como um conhecimento importante para atuantes na área de computação e, mais especificamente, em segurança. Nesse trabalho serão apresentados dois algoritmos para a geração de números pseudo-aleatórios, assim como dois algoritmos para a verificação de primalidade de números, ambos com suporte a números de alta ordem de grandeza (4096 bits).

1 Escolha de Linguagem e Considerações Iniciais

A linguagem escolhida para a implementação deste trabalho foi Python. Devido a facilidade de tratamento com números grandes (por padrão implementa números com grande número de bits) e a simplicidade do código gerado. O código fonte pode ser encontrado através do link: https://github.com/Djsouls/INE5429. Os algoritmos apresentados aqui não refletem necessariamente a implementação do trabalho, mas sim servem mais como exemplos para a análise de complexidade e implementação.

2 Números Pseudo-Aleatórios

2.1 Escolha de algoritmos

Os dois algoritmos escolhidos para realizar a geração de núemros pseudo-aleatórios foram: Xorshift e Blum-Blum-Shub. A ideia de escolher estes dois em específico foi poder fazer uma comparação entre um algoritmo performático como o xorshift e um menos performático como o blum-blum-shub e por ambos não possuírem restrição de número de bits que podem ser gerados.

2.2 Comparação e análise dos algortimos

2.2.1 Xorshift

Xorshift, descoberto por George Marsaglia, é um tipo de gerador de números pseudo-aleatórios da subclasse de LFSRs (Linear-Feedback Shift Registers), que pela sua natureza de implementação possui alta performance quando implementado em software, baseia-se em realizar sucessivas operações binárias do tipo XOR e Shift em determinado número para obter-se o próximo número da sequência. A segurança desse tipo de algoritmo se dá pela boa escolha de parâmetros de operações e da semente inicial, neste trabalho essas informações foram embasadas no trabalho original de Marsaglia [1].

Uma implementação exemplo simples do algoritmo na linguagem Python seria:

```
def xorshift(n: int) -> int:
    n ^= (n << 13)
    n ^= (n >> 7)
    n ^= (n << 17)

    return n
```

Como podemos perceber o algortimo é relativamente simples, sendo composto de apenas 3 instruções para este exemplo em específico. `n` é o valor de entrada no qual serão feitas as operações e mais tarde retornado. Possuindo apenas 3 operações que se aproximam de opeações nativas de processadores, a complexidade utilizando a notação Big-O seria $\mathcal{O}(3)$, que se reduz para $\mathcal{O}(1)$, tendo assim complexidade constante.

2.2.2 Blum-Blum-Shub

Já no caso do Blum-Blum-Shub, por ter a natureza de ser um gerador de bits pseudo-aleatórios sua complexidade naturalmente tendo a ser maior, já que a cada número gerado, apenas parte dos seus bits serão utilizados a fim de construir um outro número. É um subconjunto dos LCG (Linear Congruential Generator), um conjunto de geradores de números pseudo-aleatórios que vem ganhando cada vez mais destaque na área. A fórmula geral para esse gerador em específico é $x_{n_{+1}} = x_n^2 \mod M$, onde os requerimentos para uma boa geração

são que M seja a multiplicação de dois números primos que são congruentes a 3 mod 4 e que a semente, ou seja o valor de x_0, seja coprimo de M.

```
p = 30000000091
p = 40000000003
m = p * q

def bbs(n: int) -> int:
    return n = n ** 2 % m
```

A cada número gerado, extraímos certa quantidade de bits de informação para a geração de um novo número. Mais especificamente, no caso implementado neste trabalho retiramos o bit menos significativo do número gerado e usamos esse bit para formar outro número:

```
p = 30000000091
p = 40000000003
m = p * q

def bbsNBits(nBits: int, n: int) -> int:
    b = 0

    x = n ** 2 % m

    # Pega o bit menos significativo do numero gerado
    # e coloca no bit menos significativo de b
    b = b | (x & 1)

    for i in range(1, nBits):
        x = x ** 2 % m

        # Pega o bit menos significativo do numero
        # gerado e o coloca na devida posicao do
        # numero b
        b = b | ((x & 1) << i)

    return b
```

Como podemos perceber, não só o algoritmo implementa operações mais custosas como multiplicação de números que podem ter até 4096 bits e resto da divisão, mas também possui um laço interno que varia linearmente com o número de bits que se deseja gerar. Mais formalmente $\mathcal{O}(3 + 2n)$, evidenciando ainda mais o caráter linear do algoritmo e indicando que é possível um aumento no tempo de geração quanto maior o número de bits, diferente do caso do xorshift.

2.3 Benchmark

O tempo foi medido através do pacote `time` oferecido pela linguagem Python.

2.3.1 Xorshift

Xorshift	
Número de bits	Tempo (s)
40	0.000027
56	0.000019
80	0.000011
128	0.000017
168	0.000025
224	0.000024
256	0.000024
512	0.000011
1024	0.000011
2048	0.000012
4096	0.000012

Tabela 1: Benchmark do algoritmo Xorshift

Como pode-se perceber na tabela 1, o aumento no número de bits do número que se deseja gerar não causa grande impacto aparente na performance do algoritmo, levando a crer que a diferença entre os tempos de geração se deve muito mais a trocas de contexto e escalonamento de processos por parte do Sistema Operacional, do que pelo algoritmo em si.

2.3.2 Blum-Blum-Shub

Blum-Blum-Shub	
Número de bits	Tempo (s)
40	0.000631
56	0.000857
80	0.001244
128	0.001589
168	0.001969
224	0.002425
256	0.002369
512	0.004363
1024	0.009249
2048	0.017957
4096	0.035720

Tabela 2: Benchmark do algoritmo Blum-Blum-Shub

Já no caso do algoritmo Blum-Blum-Shub, como apresentado pela tabela 2 pode-se notar uma clara tendência do aumento de tempo pra geração proporcional ao tamanho de bits do número que se deseja gerar.

Ambos resultados são coerentes com a análise de complexidade e explicação do funcionamento dos algoritmos feitos anteriormente.

3 Números Primos

Agora, serão explicados os algoritmos e técnicas utilizados para verificação e geração de números primos com determinados número de bits. Além do algoritmo de Miller-Rabin, foi implementado o teste de Fermat, pela sua simplicidade de desenvolvimento. Ambos algoritmos baseiam-se fortemente em teoria de números e utilizam de operações custosas ao processador, como exponenciação e operação modular.

3.1 Miller-Rabin

Uma implementação exemplo do teste de Miller-Rabin poderia ser como a que segue:

```
def millerRabin(p: int, rounds: int = 8) -> bool:
    d = p - 1

    # Checking how many times d can be divided by 2
    while d % 2 == 0:
        d >>= 1

    for _ in range(rounds):
        if isComposite(p, d):
            return False

    return True

def isComposite(p: int, d: int) -> bool:
    a = randint(2, p - 1)

    x = pow(a, d, p)

    if x == 1 or x == p - 1:
        return False

    for _ in range(d):
        x = x ** 2 % p

        if x == 1:
            return True
        if x == p - 1:
            return False
```

```
    return True
```

Analisando as operações, uma análise de complexidade simples poderia ser perceber a execução de `k` vezes da função `isComposite()`, a qual por sua vez também possui um laço interno variando `d` vezes com operações de exponenciação e modular, então, poderia-se assumir a complexidade inicial como sendo $\mathcal{O}(k * d)$. Entretanto, fazendo uma análise menos ingênua [2], sabe-se que é possível computar a operação $x^t \mod n$ em tempo $\mathcal{O}(log^2 n)$, além disso, sabemos que nossa quantidade de variação `d` não será maior do que $log_2 n$, já que realizamos sucessivas divisões do número por 2. Fazendo as simplifações, chegamos no resultado que a complexidade do algoritmo de Miller-Rabin é de $\mathcal{O}(k * log^3 n)$, onde `k` é o número de rounds que o algoritmo será executado e `n` o número a ser testado.

3.2 Teste de Fermat

Agora, analisando o Teste de Fermat, vemos que os algoritmos possuem grandes semelhanças, mas este sendo de mais fácil implementação:

```
def fermat(p: int, rounds: int = 8):
    for _ in range(rounds):
        a = randint(2, p - 1)

        if pow(a, p - 1, p) != 1:
            return False

    return True
```

Temos novamente uma variação linear em função do número de rounds desejado na verificação. Similar com o teste de Miller-Rabin, possuímos operações de exponenciação modular, entretanto um laço interno a menos. Levando em conta novamente informações do custo de operações de exponenciação modular, chega-se na complexidade $\mathcal{O}(k * log^2 n)$ [3], onde `k` é o número de rounds para realizar a verificação e `n` o número no qual se deseja aplicar a verificação de primalidade.

3.3 Benchmark

Adicional a quantidade de bits do número e do tempo levado para ser gerado, outra métrica chamada "Números testados"foi adicionada. Já que grande parte da demora na geração de números primos se deve por ter "sorte"ao gerar um número que é primo, achou-se importante adicionar essa informação na tabela. Por questões de formatação, a tabela foi colocada no respositório do github, sendo acessável através do repositório do github: https://github.com/Djsouls/INE5429. A tabela possui informações da demora para a geração de cada número primo, assim como sua quantidade de bits e quantidade de números testados.

4 Considerações Finais

Analisando as informações colhidas até aqui, foi verificado que a maior demora na geração de números primos se dá por dois fatores: a sorte de gerar pseudo-aleatoriamente um número primo e a demora de verificação de todos os casos em que se gera um número não primo. Poderia-se considerar o atraso da geração dos números também, mas, como foi apresentado na tabela 1, a geração dos números em si é o menor gargalo em todo esse processo. Por utilizar a linguagem de programação Python, não foram encontrados grandes problemas para a geração de números com variadas quantidades de bits. O tempo para a geração de primos com cada número de bits se encontram também na tabela posta no repositório do github.

Referências

[1] MARSAGLIA, George. Xorshift RNGs. Journal of Statistical Software, [S.l.], v. 8, Issue 14, p. 1 - 6, july 2003. ISSN 1548-7660.

[2] https://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf

[3] https://en.wikipedia.org/wiki/Fermat_primality_test