Relatório de Estudo Orientado: Uma visão geral sobre criptografia de chave pública

Raphael Bernardino F. Lima [1]

[1] Instituto de Computação – Universidade Federal Fluminense (UFF)
Rio de Janeiro – RJ – Brazil

raphaell@id.uff.br

1. Introdução

Foram estudados alguns tópicos de teoria dos números, algoritmos de combinação de chaves em grupos, problema do logaritmo discreto e de fatoração, assim como, os teoremas de Lagrange, Fermat e Euler.

Os tópicos estudados em teoria dos números, dentre os quais alguns serão apresentados na seção 2, compreendem: princípio da indução, teorema binomial, números triangulares e diferença de dois quadrados, máximo divisor comum de dois números, números relativamente primos, algoritmo Euclidiano, mínimo múltiplo comum, equações Diofantinas, teorema fundamental da aritmética, quantidade dos divisores de um número n, decomposição primária de n!, estimativas sobre quantidades de primos, função π dos números primos, decomposição de números e o crivo de Eratóstenes, conjetura de Goldbach, progressões aritméticas e primos. Grande parte da base teórica é proveniente do estudo do texto de aula do professor Rudolf Maier da UNB ([Maier 2005]).

Uma proposta de combinação de chaves, que tinha como objetivo substituir o Diffie-Hellman, foi estudada e implementada, apesar do algoritmo ter se provado inseguro e frágil a alguns ataques. A ideia era reduzir a quantidade de mensagens trocadas, e assim, minimizar o tempo no caso de algum participante deixar ou se juntar ao grupo.

As implementações dos algoritmos de fatoração e algumas análises serão feitas na seção 3.1 e serão apresentadas soluções como: Pollard Rho e Pollard Smooth. A base teórica para entendimento desses algoritmos foi feita através do livro de criptografia aplicada do Menezes [Menezes et al. 1996].

Os algoritmos de testes de primalidade como Solovay-Strassen, Miller-Rabin e Baillie-PSW serão apresentados na seção 3.2. Exceto no caso do Baillie-PSW, os demais algoritmos também foram estudados através do livro supracitado.

2. Teoria dos números

Essa seção tem como objetivo dar a base teórica para o entendimento dos algoritmos que serão apresentados neste documento. Se faz necessário um estudo sobre teoria dos números, visto que, grande parte dos algoritmos de chave pública, e combinação de chaves, são baseados em problemas matemáticos complexos.

Além disso, a teoria dos números é dita a mais pura disciplina dentro da mais pura das Ciências (da matemática) e tem sua origem nas antigas civilizações da humanidade, passando por Pitágoras, Euclides, Eratóstenes, Diofantos e chegando até Charles de la Vallée Poussin, matemático belga que descreveu a distribuição assintótica dos números primos.

2.1. Princípio da indução

Este princípio consiste em que todo conjunto não-vazio S de números naturais possui um elemento mínimo. Ou seja,

$$\forall\, S \subseteq \mathbb{N}, S \neq \emptyset, \exists\, m \in S \; tal \; que \; m \leqslant n \; \forall\, n \in S.$$

Daí surge a proposição de que dado um conjunto T de números naturais que satisfaça as seguintes propriedades:

1. $1 \in T$
2. Sempre se $n \in T$, então também $n + 1 \in T$.

Então $T = \mathbb{N}$ é o conjunto de todos os números naturais. Isso é possível devido a $1 \in \mathbb{N}$ e os números posteriores, $n + 1$, também estarem contidos em $\mathbb{N}$.

O princípio da indução é muito útil para elaboração de provas matemáticas devido a utilizar casos bases conhecidos e avançar em pequenos passos até que um determinado limiar seja atingido e a hipótese possa ser provada. Problemas que utilizam generalização também podem ser expressos dessa forma, por exemplo, a soma dos n primeiros números naturais ímpares é o n-ésimo quadrado perfeito:

$$1 + 3 + 5 + 7 + ... + (2n - 5) + (2n - 3) + (2n - 1) = n^2$$

Podemos provar que isso é válido da seguinte forma: Seja $T = \{n \in \mathbb{N} \mid \sum_{k=1}^{n}(2k - 1) = n^2\}$ o conjunto dos números naturais para os quais a fórmula acima é válida. Para mostrar que $T = \mathbb{N}$, só é preciso verificar que $1 \in T$ e que, se $n \in T$, então $n + 1 \in T$. Para $n = 1$, que será a nossa base, temos que $1 = 1^2$, o que é verdade, ou seja, $1 \in T$. Suponhamos para qualquer $n \in T$,

$$1 + 3 + ... + (2n - 1) = n^2$$

Somando-se $2n + 1$ dos dois lados, obtemos:

$$1 + 3 + ... + (2n - 1) + (2n + 1) = n^2 + (2n + 1)$$

da onde segue que,

$$1 + 3 + ... + (2n - 1) + (2(n + 1) - 1) = (n + 1)^2$$

ou seja, temos que $n+1 \in T$. Pela proposição concluímos que o conjunto verdade da fórmula supracitada é o conjunto $T = \mathbb{N}$ de todos os números naturais. ■

2.2. Números triangulares

Dizemos que um número $m \in \mathbb{N}$ é triangular caso obedeça à seguinte equação:

$$t_m = \frac{m(m + 1)}{2}$$

onde t_m é m-ésimo número triangular.

Uma observação válida a se fazer é que para todo $m \in \mathbb{N}$ temos:

$$t_{m+1} = t_m + (m+1)$$

Portanto, a sequência dos números triangulares é:

$$(t_m)_{m\in\mathbb{N}} = (1, 3, 6, 10, 15, ..., \frac{m(m+1)}{2}, ...)$$

A expressão "número triangular" pode ser explicada por causa da quantidade de pontos t_m que um triângulo equilátero de m lados possui.

2.3. Diferença de dois quadrados

Outro aspecto fundamental é o entendimento da diferença de dois quadrados, visto que, esse raciocínio será utilizado em diversos algoritmos que veremos mais a frente. Um ponto a se ressaltar é que 75% dos números naturais são obtidos a partir da diferença de dois quadrados. Os números que são a diferença de dois quadrados podem ser facilmente identificados verificando se n é impar ou divisível por 4.

Podemos expressar essa ideia da seguinte forma: Suponha $n = x^2 - y^2$ e n é par. Isto quer dizer que x e y ambos são pares ou ambos são ímpares: Se $x = 2k$ e $y = 2l$, temos que $n = x^2 - y^2 = (2k)^2 - (2l)^2 = 4(k^2 - l^2) \notin \{2, 6, 10, ...\}$. Se $x = 2k - 1$ e $y = 2l - 1$ temos também que $n = x^2 - y^2 = (2k-1)^2 - (2l-1)^2 = 4(k^2 - l^2 - k + l) \notin \{2, 6, 10, ...\}$. Suponhamos, reciprocamente, que $n \notin \{2, 6, 10, ...\}$, isto significa que n é impar ou divisível por 4. Se n é ímpar, $n \pm 1$ é par e portanto $\frac{n\pm1}{2} \in \mathbb{N}_0$. Como ainda $(\frac{n+1}{2})^2 - (\frac{n-1}{2}^2) = \frac{(n+1)^2-(n-1)^2}{4} = \frac{4n}{4} = n$, concluimos que

$$n = (\frac{n+1}{2})^2 - (\frac{n-1}{2})^2$$

é uma possível decomposição de n como diferença de dois quadrados. Se $n = 4k$, decompomos $n = (k+1)^2 - (k-1)^2$, ou seja,

$$n = (\frac{n}{4} + 1)^2 - (\frac{n}{4} - 1)^2$$

2.4. Máximo divisor comum (MDC)

O máximo divisor comum $d \in \mathbb{N}$ entre dois números $a, b \in \mathbb{Z}$, com pelo menos um diferente de zero, é definido como $d = mdc(a, b)$ e segue as seguintes propriedades:

1. $d \mid a$ e $d \mid b$, ou seja, d é o divisor comum de a e b.
2. Se algum $c \in \mathbb{N}$ dividir ambos a e b então temos que $c \mid d$.

Podemos expressar d através de $d = ax_1 + by_1$ para $x_1, y_1 \in \mathbb{Z}$. A demonstração pode ser visualizada abaixo:

Consideremos o conjunto $S = \{ax + by \mid x, y \in \mathbb{Z}, ax + by > 0\}$. Seja primeiro $a \neq 0$. Fazendo $y = 0$ e $x = 1$ se $a > 0$ $x = -1$, se $a < 0$ vemos que $ax + by =$

$a(\pm 1) + b.0 = \mid a \mid > 0$ o que mostra que $S \neq \emptyset$. Se $a = 0$, então $\mid b \mid > 0$ e uma escolha análoga de x e y mostra que $S \neq \emptyset$ também neste caso.

Pelo princípio da indução, existe um $d \in S$ mínimo. Como $d \in S$ temos que $d > 0$ e existem $x_1, y_1 \in \mathbb{Z}$ tais que $d = ax_1 + by_1$.

Afirmemos que d é o mdc(a, b) e dividamos a por d com resto $\exists\, q, r \in \mathbb{Z}$ tais que $a = qd + r$ e $0 \leq r \leq d - 1$. Então, podemos escrever $r = a - qd = a - q(ax_1 + by_1) = a(1 - qx_1) + b(-qy_1)$. Se fosse $r > 0$, poderiamos concluir que $r \in S$, o que é um absurdo, já que $r < d$ e d é o elemento mínimo de S. Logo, $r = 0$ e $a = qd$, o que significa que $d \mid a$.

Da mesma forma, mostra-se que $d \mid b$. Logo, d é divisor comum de a e b. Seja $c \in \mathbb{N}$ tal que $c \mid a$ e $c \mid b$. Pela regra da comutatividade podemos concluir que $c|ax_1+by_1 = d$. Logo, $d = mdc(a, b)$. ■

2.5. Algoritmo Euclidiano

O algoritmo euclidiano busca obter o máximo divisor comum utilizando os restos das divisões a partir de a e b. Podemos representar qualquer número como $a = bq + r$, onde $q, r \in \mathbb{N}$. O máximo divisor comum entre a e b será obtido quando $r = 0$. Por exemplo, podemos calcular o mdc(102, 38):

$$\begin{aligned} 102 &= 2 * 38 + 26 \\ 38 &= 1 * 26 + 12 \\ 26 &= 2 * 12 + 2 \\ 12 &= 6 * 2 + 0 \end{aligned}$$

Daí, temos que o mdc(102, 38) = 2. O algoritmo mais simples pode ser compreendido da seguinte forma:

Entrada: Números a e b
Saída : máximo divisor comum de a e b
$r = b$;
while $r \neq 0$ **do**
  $r = RESTO(a, b)$;
  $a = b$;
  $b = r$;
end
RETORNE a;

Algorithm 1: Algoritmo de Euclides simples.

O algoritmo de Euclides pode ser estendido, de forma que não apenas nos dê o maior divisor comum d de dois inteiros a e b, mas sim também, os inteiros que satisfazem

a equação $ax + by = d$. O algoritmo de Euclides estendido pode ser visualizado abaixo.

Entrada: Números a e b
Saída : máximo divisor comum de a e b e x, y que satisfazem $ax + by = d$
if $b = 0$ **then**
  $d = a$;
  $x = 1$;
  $y = 0$;
  RETORNE d, x, y;
end
$x_2 = 1$;
$x_1 = 0$;
$y_2 = 0$;
$y_1 = 1$;
while $b > 0$ **do**
  $q = PISO(a/b)$;
  $r = a - q * b$;
  $x = x_2 - q * x_1$;
  $y = y_2 - q * y_1$;
  $a = b$;
  $b = r$;
  $x_2 = x_1$;
  $x_1 = x$;
  $y_2 = y_1$;
  $y_1 = y$;
end
RETORNE a, x_2, y_2;

Algorithm 2: Algoritmo de Euclides estendido.

Vale ressaltar que ambos os algoritmos de Euclides possuem a mesma ordem de complexidade $O((lgn)^2)$.

2.6. Mínimo múltiplo comum (MMC)

O MMC entre a e b, com $a, b \in \mathbb{Z}$ e ambos não nulos, é definida através de $MMC(a, b)$ e possui as seguintes propriedades:

1. $a \mid m$ e $b \mid m$, ou seja, m é um múltiplo de a e b.
2. Se $a \mid c$ e $b \mid c$ para algum $c \in \mathbb{N}$, então $m \mid c$.

Os múltiplos de $a = 7$ e $b = 12$ são $\{\pm 84, \pm 168, \pm 336, \pm 672, ...\}$, porém o menor entre estes é

$$m = mmc(7, 12) = 84$$

Podemos utilizar o MDC para calcular o MMC fazendo, por exemplo:

$$mmc(7, 12) = \frac{7 * 12}{mdc(7, 12)} = \frac{7 * 12}{1} = 84$$

Sejam $0 \neq a, b \in \mathbb{Z}$, $d = mdc(a, b)$ e $m = mmc(a, b)$. Então, podemos utilizar a seguinte relação:

$$md = \mid ab \mid$$

Façamos $m' = \frac{|ab|}{d}$. Existem $r, t \in \mathbb{Z}$ tais que $dr = a$ e $dt = b$. Temos que $m' = \frac{|a|}{d} \mid b \mid = \pm rb$ e também que $m' = \mid a \mid \frac{|b|}{d} = \pm at$. Isto mostra que m' é múltiplo comum de a e b.

Seja $c \in \mathbb{N}$ tal que $a \mid c$ e $b \mid c$. Existem então $u, v \in \mathbb{Z}$ tais que $au = c = bv$. E portanto, existem $x_1, y_1 \in \mathbb{Z}$ com $ax_1 + by_1 = d$. Segue que,

$$\frac{c}{m'} = \frac{cd}{\mid ab \mid} = \frac{c}{\mid ab \mid}(ax_1 + by_1) = \frac{c}{\mid b \mid}\frac{ax_1}{\mid a \mid} + \frac{c}{\mid a \mid}\frac{by_1}{\mid b \mid} = \pm \frac{c}{b} x_1 \pm \frac{c}{a} y_1 = \pm vx_1 \pm uy_1 \in \mathbb{Z}.$$

Mostramos assim que $\frac{c}{m'} \in \mathbb{Z}$, o que significa que $m' \mid c$. Assim $m' = m$. ■

2.7. Teorema fundamental da aritmética

Esse teorema diz que qualquer número $p \in \mathbb{N}$ maior que 1 é denominado primo se seus divisores únicos são p e 1. O conjunto dos números primos é representado por

$$\mathbb{P} = \{p \in \mathbb{N} \mid p \text{ é } primo\}$$

e todos os demais números podem ser decompostos em um produto de primos. Ou seja, podemos dizer então que

$$p \in \mathbb{P} \iff (\forall a, b \in \mathbb{N} : p = ab \implies a = p \text{ e } b = 1 \text{ ou } a = 1 \text{ e } b = p)$$

Assim, n é composto, se existem $r, s \in \mathbb{N}, 1 < s \leq r < n$ com $n = rs$.

Se $p \in \mathbb{P}$ então $\forall a, b \in \mathbb{N} : p \mid ab \implies p \mid a \text{ ou } p \mid b$. Vamos supor então que $p \mid ab$ e $p \nmid a$. Isso significa que $mdc(p, a) = 1$. Daí, podemos concluir que $p \mid b$.

Podemos observar também que essa propriedade necessária também é suficiente para que um $n \in \mathbb{N}$ seja primo. Se $n = rs$ é composto, temos que $n \mid rs$ porém $n \nmid r$ e $n \nmid s$. Por exemplo, se $5 \mid ab$ então temos que um dos fatores é múltiplo de 5. Mas, temos também que $6 \mid 12 = 3 * 4$, porém tanto $6 \nmid 3$ quanto $6 \nmid 4$.

2.8. Estimativas sobre quantidades de primos

O teorema de Euclides diz que o conjunto dos números primos $\mathbb{P}$ é infinito. Vamos demonstrá-lo.

Suponhamos que $\mathbb{P} = \{p_1, p_2, p_3, ..., p_r\}$ fosse um conjunto finito. Consideremos o número natural $n = p_1 * p_2 * ... * p_r + 1$. Pelo Teorema fundamental da aritmética, temos que $n > 1$ é divisível por algum primo q. Pela suposição, $q = p_k$ para algum

$k \in \{1, 2, 3, ..., r\}$ segue o absurdo de que $q \mid 1$. Logo, nenhum conjunto finito pode abranger todos os primos.

Para o n-ésimo número primo p_n vale a estimativa

$$p_n \leq 2^{2^{n-1}}$$

Podemos provar esse fato partindo do caso base $n = 1$, pode-se afirmar que $2 = p_1 \leq 2^{2^{1-1}} = 2^1 = 2$, o que é verdade. Suponhamos provadas as desigualdades de p_1 até p_n, onde $p_1 \leq 2^{2^0}, p_2 \leq 2^{2^1}, ..., p_n \leq 2^{2^{n-1}}$.

Se $\mathbb{P} \ni q \mid p_1 * p_2 * ... * p_n + 1$, então $q > p_n$, particularmente, $p_{n+1} \leq q$. Segue que

$$p_{n+1} \leq p_1 * p_2 * ... * p_n + 1 \leq 2^{1+2+2^2+...+2^{n-1}} + 1 = 2^{2^n-1} + 1 \leq 2^{2^n-1} + 2^{2^n-1} = 2^{2^n}$$

■

Uma outra estimativa que pode ser utilizada segue do teorema de Tchebychef, que diz que sempre existe um primo p com $m < p < 2m$, onde $2 \leq m \in \mathbb{N}$. A consequência desse teorema consegue nos dar a estimativa

$$p_n \leq 2^n$$

Pois, considerando que temos $2 = p_1 \leq 2^1$ e $\forall n = 1, 2, 3, ...$ tem-se $p_n < p_{n+1} < 2 * p_n$. De $p_n \leq 2^n$ segue então que $p_{n+1} \leq 2 * 2^n = 2^{n+1}$. ■

2.9. Função PI dos números primos

Para todo $0 \leq x \in \mathbb{R}$ define-se a função $\pi(x)$ por

$$\pi(x) = \mid \{p \in \mathbb{P} \mid p \leq x\} \mid$$

onde $\pi(x)$ representa a quantidade de números primos menores ou iguais a x.

Por exemplo, temos $\pi(x) = 0$ se $0 \leq x \leq 2$, $\pi(x) = 1$ se $2 \leq x < 3 \pi(x) = 2$, se $3 \leq x < 5$. Em geral, se $p_1 = 2, p_2 = 3, p_3 = 5, ..., p_{25} = 97, ...$ é a sequência dos números primos em ordem natural, então

$$\pi(x) = r \; se \; p_r \leq x < p_{r+1} \;\; (r = 1, 2, 3, ...)$$

O comportamento assintótico da função $\pi(x)$ foi descrita em 1896 e trouxe uma grande luz a cerca dos números primos. Essa descoberta foi conhecida como o *Teorema dos números primos* e feita pelos matemáticos Cauchy, Hadamard e De La Vallée Poussin. O comportamento pode ser descrito como

$$lim_{n \to \infty} = \frac{\pi(x)}{\frac{x}{ln(x)}} = 1$$

Ou seja, quanto maior x, melhor a aproximação gerada da quantidade dos números primos menores ou iguais a x.

2.10. Crivo de Eratóstenes

Como vimos anteriormente, todos os números naturais compostos $n \in \mathbb{N}$ podem ser representados por uma multiplicação de primos tal como $n = rs$, onde $r, s \in \mathbb{N}$. O crivo irá se aproveitar disso para encontrar todos os números primos menores ou iguais a n, dado $2 \leq n \in \mathbb{N}$.

Sejam $n, r, s \in \mathbb{N}$ tais que $n = rs$ com $1 \leq s \leq r \leq n$. Os dois casos que temos são

1. Temos $s \leq \sqrt{n} \leq r$.
2. Se n for composto, então existem $r, s \in \mathbb{N}$ tais que $1 < s \leq \sqrt{n} \leq r < n$ e $n = rs$.

Se $s \leq r < \sqrt{n}$, temos a contradição $n = rs < \sqrt{n}\sqrt{n} = n$. De forma análoga, se $\sqrt{n} < s \leq r$, temos que $n = \sqrt{n}\sqrt{n} < rs = n$. Portanto, devemos ter $s \leq \sqrt{n} \leq r$. E por essa conclusão, não precisamos testar todos os números até n, apenas até $\sqrt{n}$, o que otimizará o cálculo do crivo.

O crivo iniciará com todos os números de 2 até n marcados como primos e riscará a cada iteração i todos os múltiplos de i. Os números que sobrarem são, de forma garantida, números primos, visto que não há múltiplos nesse intervalo.

Por exemplo, para $n = 81$, iremos remover todos os números pares, ou seja, os múltiplos de 2. Depois iremos remover todos os múltiplos de 3 e assim sucessivamente. Note que não iremos riscar o 4 novamente, já que a iteração com $i = 2$ o removeu, então após a iteração de $i = 3$, iríamos para $i = 5$. A iteração será limitada por $\sqrt{n}$ e os números que restarem são efetivamente os números primos menores que n. Vale ressaltar que essa é uma abordagem que utilizará muito espaço de armazenamento em memória, já que precisará alocar todos os números para assim ir eliminando-os pouco a pouco.

3. Segurança em chaves públicas

Diversos problemas matemáticos complexos como fatoração de inteiros muito grandes (maiores que 256 bits), problema do logaritmo discreto e outros são utilizados para garantir a segurança de grande parte dos algoritmos de chaves públicas. De maneira informal, um problema é dito complexo quando não existe um método polinomial de resolução para um determinado problema.

Os problemas apresentados nessa seção não possuem algoritmos eficientes para resolução, porém serão apresentados algoritmos que tentam resolver uma boa parte dos problemas em um tempo considerado satisfatório. Todos os algoritmos, de sua respectiva categoria, rodarão com a mesma entrada e serão analisados o tempo gasto em relação a quantidade de bits. As entradas escolhidas para os testes de fatoração diferem das entradas dos testes de primalidade.

3.1. Problema da fatoração

Descobrir os números primos que compõe um determinado número composto é considerado um problema difícil devido a quantidade de testes que se é necessário realizar. Um

problema análogo é descobrir se um número é primo ou não, porém que se é de muito fácil resolução. Portanto, o primeiro passo que se deve fazer é testar para o caso de n, o número que estamos tentando fatorar, ser primo. Os testes de primalidade serão abordados mais a frente na seção 3.2.

3.1.1. Pollard Rho

Esse algoritmo tem como objetivo encontrar um fator não trivial pequeno de n. Mas isso não impede que o algoritmo seja executado recursivamente e todos os demais fatores sejam encontrados. A abordagem segue o mesmo raciocínio da busca de ciclos de *Floyd* e calcula iterativamente pares (a, b) para encontrar os fatores que compõe o número composto. O algoritmo pode ser compreendido da seguinte forma:

Entrada: Números n a ser fatorado
Saída : Um fator não trivial de n
$a = b = 2$;
while $d == 1$ **do**
  $a = a^2 + 1 \; mod \; n$;
  $b = (b^2 + 1 \; mod \; n)^2 + 1 \; mod \; n$;
  $d = MDC(a - b, n)$;
  if $1 < d < n$ **then**
    RETORNE d;
  end
  if $d == n$ **then**
    Finalize com erro;
  end
end
RETORNE a;

Algorithm 3: Algoritmo de Pollard Rho.

Em alguns casos, como o número de 128 bits 161720844233529689499498448342098256937, o algoritmo apresenta dificuldades para ser fatorado. Pode-se notar na figura 1 que grande parte do gargalo acontece devido ao cálculo do máximo divisor comum (MDC).

Foram feitas análises do tempo de execução para números, somente ímpares, gerados de forma aleatória com as quantidades de bits de 128 e 256. A figura 2 mostra o diagrama de chamadas realizado pelos 100 números, assim como o tempo gasto em cada função. Percebe-se que mesmo para 256 bits o gargalo continua sendo na função de MDC, porém, o tempo relativo gasto é maior do que antes. Podemos então concluir que conforme a quantidade de bits cresce, mais tempo será gasto para determinar o MDC entre os dois números.

A distribuição de tempo pode ser visualizada nas figuras 3 e 4 abaixo para os casos de 128 e 256 bits, respectivamente. Em ambos os casos houveram testes que demoraram mais do que outros.

Na figura 3 percebemos que grande parte conseguiu ser calculada em menos de 1 segundo, um tempo satisfatório, visto que o algoritmo não é tão complexo assim.

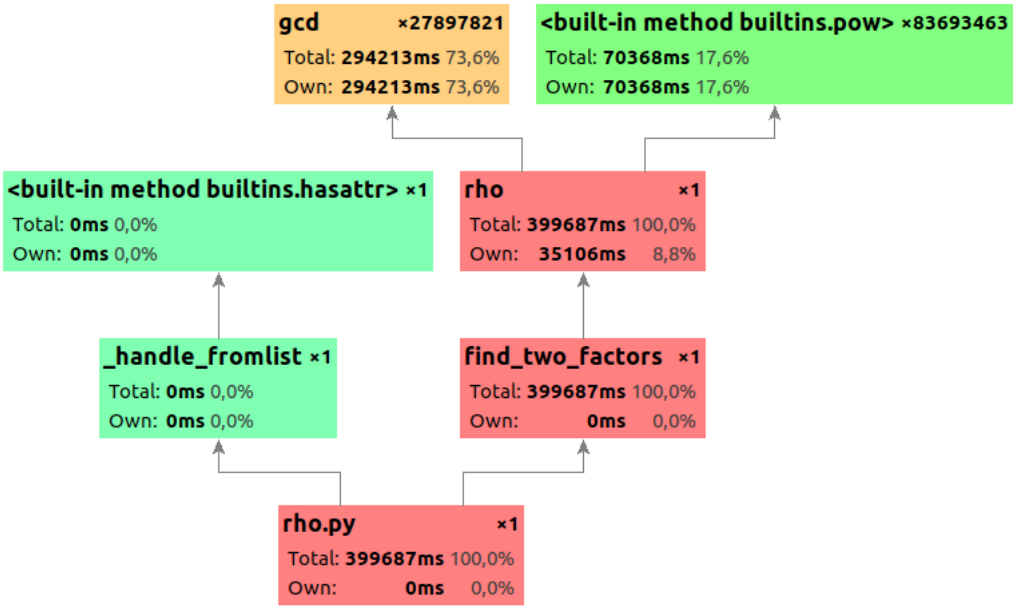

Figure 1. Diagrama de chamadas na execução do algoritmo Rho.

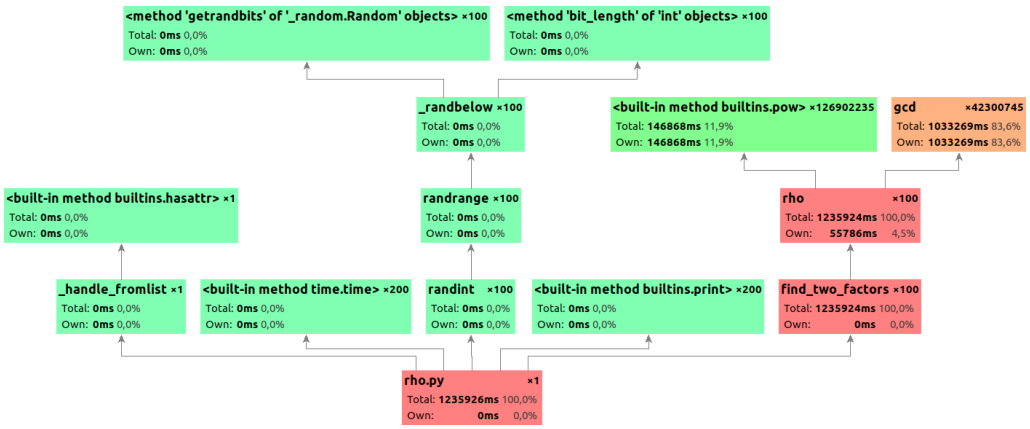

Figure 2. Diagrama de chamadas na execução do algoritmo Rho para 256 bits.

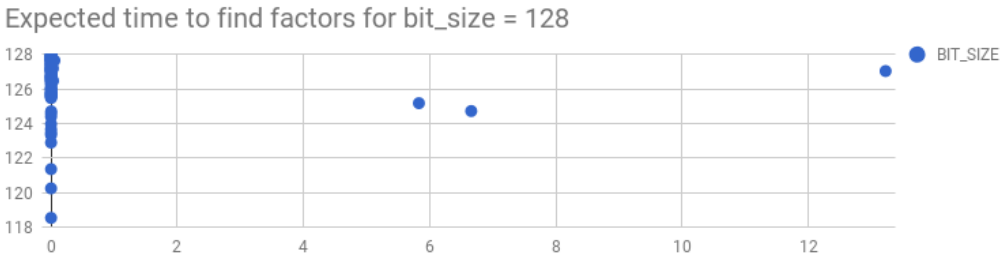

Figure 3. Distribuição de tempo na execução do algoritmo Rho para 128 bits.

Os casos que levaram mais tempo são: 175120399228547846888456991422137905271 com os fatores 323585755543 e 541186984373534352000155648, executando em, aproximadamente, 13 segundos, 352441748691385957668916185284726176 27 com os fatores 675418735891 e 521812218055294631563755 52, executando em, aproximadamente, 7

segundos, 48333053950913730667611297492172966883 com os fatores 445481210753 e 108496279493395540680376320, executando em, aproximadamente, 6 segundos.

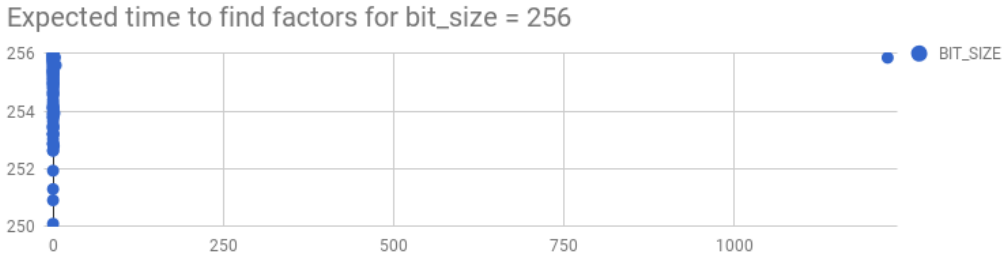

Figure 4. Distribuição de tempo na execução do algoritmo Rho para 256 bits.

Enquanto na figura 4 percebemos que exceto um teste não pode ser calculado em menos de 1 segundo. O teste que gerou mais trabalho foi o caso para $n = 104719802762381912039208921868056844202718606698699371733963822856819421644967$ que possui os fatores 360132402793429051942437300590382479079141983407244857734856704 e 290781395814719, levando em torno de 1225 segundos, ou, 20 minutos aproximadamente.

3.1.2. Pollard Smooth

O algoritmo de Pollard Smooth pode ser utilizado para encontrar os fatores primos p de um número composto n no qual $p - 1$ é um limiar suave (*smooth bound*). A definição de *smooth bound* segue de n é dito um limiar suave de B se todos os fatores primos são menores ou iguais a B. Ou seja, a idéia deste algoritmo é encontrar um limiar B tal que n seja suave em relação a B.

Seja Q é menor múltiplo comum (MMC) de todas as potências de primos menores ou iguais a B. Se $q^l \leq n$, então $llnq \leq lnn$, e então $l \leq \left\lfloor \frac{lnn}{lnq} \right\rfloor$. Daí,

$$Q = \prod_{q \leq B} q^{\lfloor lnn/lnq \rfloor}$$

onde q são todos os primos distintos menores ou iguais ao limiar B. Se p é um fator primo de n tal que $p - 1$ é suave em relação a B, então $p - 1 \mid Q$, e portanto, $MDC(a, p) = 1$. Do teorema de Fermat, temos que $a^Q \equiv 1 \ (mod\ p)$. Então se $d = MDC(a^Q - 1, n)$, então $p \mid q$. É possível que $d = n$, onde o algoritmo irá falhar. Mas isso é pouco provável de acontecer se n possuir pelo menos dois fatores, distintos,

grandes. O algoritmo pode ser visualizado abaixo:

Entrada: Números n a ser fatorado e que não é uma potência de primos
Saída : Um fator não trivial de n
Escolha um limiar suave B;
Gere um número aleatório a entre 2 e $n-1$;
$d = MDC(a, n)$;
if $d \geq 2$ **then**
  RETORNE d;
end
foreach $q \leq Q$ **do**
  $l = \left\lfloor \frac{lnn}{lnq} \right\rfloor$;
  $a = a^{q^l} \ mod \ n$;
end
$d = MDC(a-1, n)$;
if $d == 1 \ \ OU \ \ d == n$ **then**
  Finalize com erro;
end
RETORNE d;

Algorithm 4: Algoritmo de Pollard p-1.

Ao contrário do Pollard Rho, agora temos que o gargalo acontece devido às exponenciações feitas. A geração dos números primos em memória e a escolha do limiar suave também são um *tradeoff* a ser considerado. Um fato importante a se considerar é que com o limiar utilizado não foi possível encontrar todos os fatores. A estratégia utilizada foi iniciar o limiar em $log_2(n)$ e dobrá-lo a cada insucesso obtido, ou até atingir um total de 15 insucessos. O diagrama de execução dos 100 casos de testes de 128 bits pode ser visualizada na figura 5, onde 3 casos falharam. Utilizando 256 bits mais três casos também falharam.

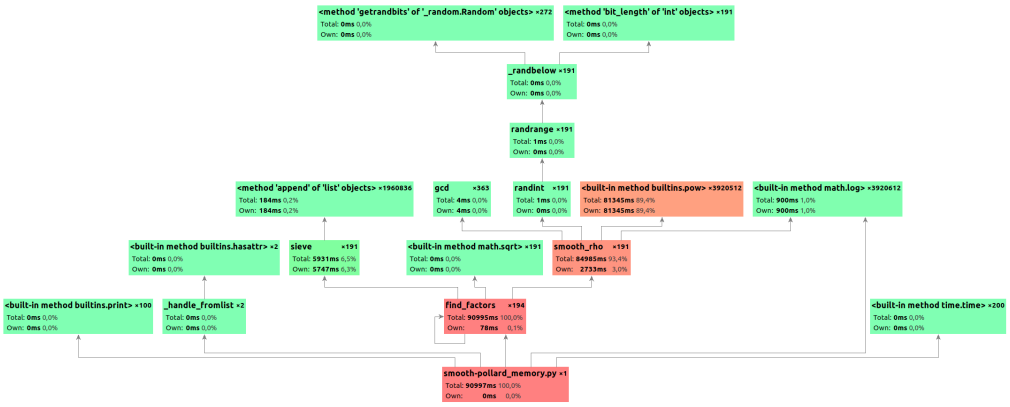

Figure 5. Diagrama de chamadas na execução do algoritmo smooth de Pollard.

A distribuição dos tempos de 128 bits pode ser visualizada na figura 6 e a de 256 bits na figura 7. O fator limitante para os casos não conseguirem encontrar os fatores foi a questão de ter atingido o limite máximo de tentativas.

Antes apenas 3 casos levavam mais de 1 segundo, agora 4 casos levam mais de 1

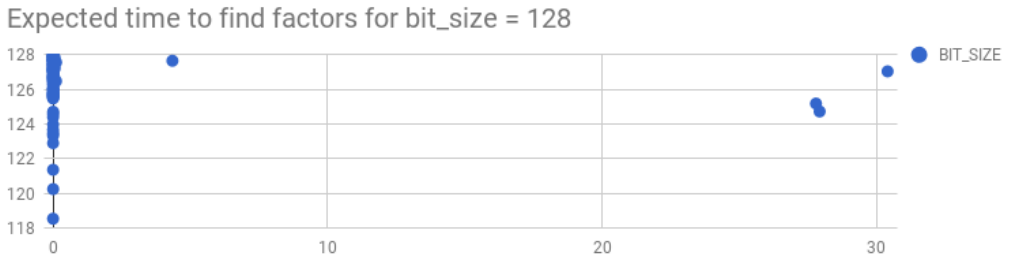

Figure 6. Distribuição do tempo para fatoração dos números de 128 bits usando smooth Pollard.

segundo. Isso pode ser entendido como uma piora no caso médio, já que a distribuição dos números foi realizada de forma equiprovável.

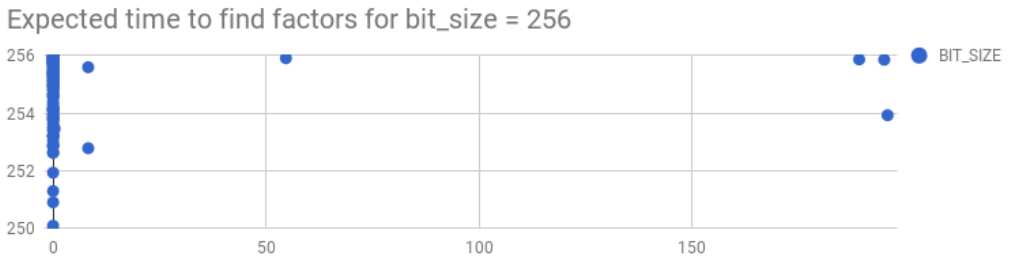

Figure 7. Distribuição do tempo para fatoração dos números de 256 bits usando smooth Pollard.

Na figura 7 também podemos perceber uma piora na quantidade de testes que levaram mais de 1 segundo, porém isso representou uma melhora no tempo médio de execução, visto que os piores casos são eliminados pelo fator limitante de tempo.

3.2. Testes de primalidade

Outro problema análogo ao anterior é verificar se um número é primo ou não. Esses testes de primalidade são muito importantes para a criptografia de chave pública devido a grande maioria utilizar grupos de primos ([Diffie and Hellman 1976]), números primos para geração das chaves públicas ([Rivest et al. 1978]) e escolhas similares. A grande parte destes algoritmos também utilizam números com uma grande quantidade de bits, 300 ou mais, o que torna impraticável qualquer ataque *naïve* nesses criptossistemas, com o objetivo de reduzir a probabilidade do número escolhido ser descoberto por um atacante com um grande poder computacional.

Essa seção tem então como objetivo demonstrar alguns dos algoritmos existentes e realizar uma análise a cerca do tempo necessário para determinar se um número é primo ou composto. Os algoritmos podem ser determinísticos e probabilísticos, sendo a última classe mais eficiente e com uma pequena taxa de erro. Os algoritmos determinísticos são menos eficientes pois testam de fato todos os números para concluir a qual tipo o número se encaixa, ou seja, um grande poder computacional precisa ser utilizado a fim de testar números muito grandes. Enquanto que os algoritmos probabilísticos não determinam de fato se o número é primo ou não, ao invés disso, são retornados números como “prováveis primos”.

Apenas para efeitos de curiosidade a geração de números primos utilizados nos diversos criptossistemas acontece de maneira aleatória através de uma geração de números ímpares. Ou seja, um número ímpar é gerado de forma aleatória e testado pro caso do número ser primo, e caso seja, esse será utilizado no criptosistema. Em termos de programação, ou código, pode-se utilizar o operador binário **OU** para gerar números ímpares fazendo $n \mid 1$, onde n é número aleatório gerado.

3.2.1. Solovay-Strassen

O teste probabilístico de Solovay-Strassen não é mais utilizado hoje em dia, porém será apresentado para efeitos de comparação com os demais. O teste baseia-se no fato de que se n é um primo ímpar, então $a^{\frac{(n-1)}{2}} \equiv (\frac{a}{n})\ (mod\ n)$ para todos os inteiros a que possuem $MDC(a, n) = 1$. O inteiro a será denominado como testemunha (*witness*) de Euler caso o $MDC(a, n) > 1$ ou $a^{(n-1)/2} \not\equiv (\frac{a}{n})\ (mod\ n)$, senão se, $MDC(a, n) = 1$ e $a^{\frac{(n-1)}{2}} \equiv (\frac{a}{n})\ (mod\ n)$, n será dito como um pseudoprimo de Euler para a base a. O algoritmo pode ser visualizado abaixo:

Entrada: um inteiro ímpar n maior ou igual a 3 e t maior ou igual a 1
Saída : classificação do número em primo ou composto
for $i = 1$ *até* t **do**
  Escolha um número aleatório a, $2 \leq a \leq n - 2$;
  $r = a^{(n-1)/2}\ mod\ n$;
  if $r \neq 1\ E\ r \neq n - 1$ **then**
    RETORNE composto;
  end
  Calcule o símbolo de Jacobi $s = (\frac{a}{n})$;
  if $r \not\equiv s\ (mod\ n)$ **then**
    RETORNE composto;
  end
end
RETORNE primo;

Algorithm 5: Descrição algorítmica do Solovay-Strassen.

Se o $MDC(a, n) = d$, então d é um divisor de $r = a^{(n-1)/2}\ mod\ n$. Portanto, ao testarmos se $r \neq 1$, já estaremos eliminando a necessidade de testar se $MDC(a, n) \neq 1$. A probabilidade de um número n ser classificado incorretamente como primo é menor que 2^{-t}.

O diagrama de chamadas, que pode ser visualizado na figura 8, mostrou que apesar do algoritmo se mostrar eficiente até mesmo para entradas de 512 bits, existe um gargalo pela quantidade de operações de exponenciação feitas. O cálculo do símbolo de Jacobi também influencia com 10% no tempo da execução. O que talvez poderia ser melhorado na implementação é a geração dos números aleatórios a necessários descritos no algoritmo, que também representa em torno de 8% do tempo total.

Os resultados obtidos já eram esperados, visto que esse algoritmo não é mais utilizado devido a sua baixa eficiência e performance, porém, demonstra ser ainda que razoável para execuções pequenas.

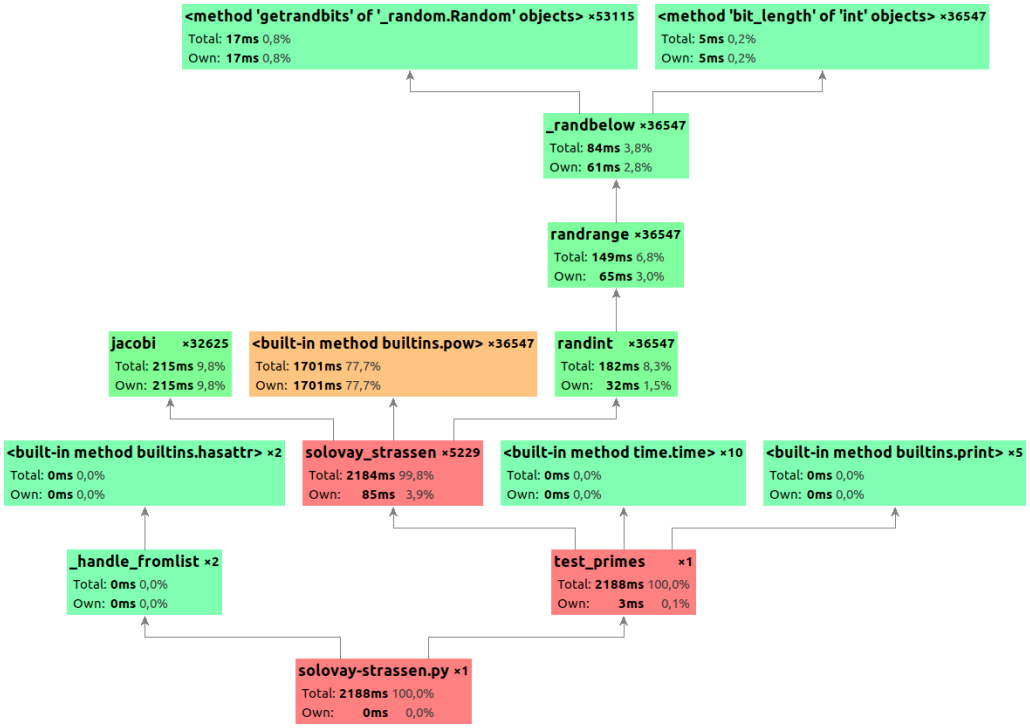

Figure 8. Gráfico de chamadas para a execução do algoritmo Solovay-Strassen.

Outra observação a ser feita a respeito desse algoritmo é sua instabilidade, ou variação, de tempo de execução. Para números pequenos, menores que 10000, o tempo de execução superou os testes para números de 64 bits. Mais detalhes são apresentados na seção 3.2.4.

3.2.2. Miller-Rabin

Outro teste probabilístico conhecido como Miller-Rabin vem cada vez mais sendo utilizado devido a sua facilidade de implementação e performance. O algoritmo consegue obter melhor desempenho que o teste de Fermat, que falha para os números de Carmichael, e é tão correto quanto o teste de Solovay-Strassen.

A base desse teste de primalidade vem dos fatos de que se n é um primo ímpar, $n - 1 = 2^s r$, onde r é ímpar, e $MDC(a, n) = 1$, então $a^r \equiv 1 \ (mod\ n)$ ou $a^{2^j r} \equiv -1 \ (mod\ n)$, para algum j, $0 \leq j \leq s - 1$. O algoritmo pode ser compreendido da

seguinte forma,

Entrada: um inteiro ímpar n maior ou igual a 3 e t maior ou igual a 1
Saída : classificação do número em primo ou composto
Escreva $n-1 = 2^s r$ tal que r seja ímpar.;
for $i = 1$ *até* t **do**
  Escolha um número aleatório a, $2 \leq a \leq n-2$;
  $y = a^r \ mod \ n$;
  if $y \neq 1 \ E \ y \neq n-1$ **then**
    $j = 1$;
    while $j \leq s-1 \ E \ y \neq n-1$ **do**
      $y = y^2 \ mod \ n$;
      if $y == 1$ **then**
        RETORNE composto;
      end
      $j = j + 1$;
    end
    if $y \neq n-1$ **then**
      RETORNE composto;
    end
  end
end
RETORNE primo;

Algorithm 6: Descrição algorítmica do Miller-Rabin.

A ideia é testar cada base a que satisfaz as seguintes condições, considerando n um primo ímpar, $n-1 = 2^s r$ e $a \in [1, n-1]$:

1. Se $a^r \not\equiv 1 \ (mod \ n)$ e $a^{2^j r} \not\equiv -1 \ (mod \ n)$ para todo j, $0 \leq j \leq s-1$, então a é chamado de *strong witness* de n.
2. Senão se, $a^r \equiv 1 \ (mod \ n)$ ou $a^{2^j r} \equiv -1 \ (mod \ n)$ para algum j, $0 \leq j \leq s-1$, então n é dito *strong pseudoprime* para a base a.

A implementação difere um pouco da apresentada acima de forma que o maior proveito seja tirado através da linguagem de programação Python, porém a complexidade do algoritmo permanece a mesma. A escolha do parâmetro t foi baseada no $paper$ do *NIST FIPS PUB 186-4*([FIPS 2013]), que recomenda a utilização de 3 testes para números entre 1024 e 1536 bits, 4 para números 512 e 1024, 7 testes para maiores de 512 e 10 para os demais casos.

Como vemos na figura 9, o tempo relativo utilizado nas exponenciações aumentou, porém o tempo total de execução caiu de, aproximadamente, 2.2 para 1.6 segundos. Mais a frente, na seção 3.2.4, veremos a comparação dos algoritmos e iremos perceber que o método de Miller-Rabin se aproxima bastante do método de Solovay-Strassen conforme o número de bits de n cresce.

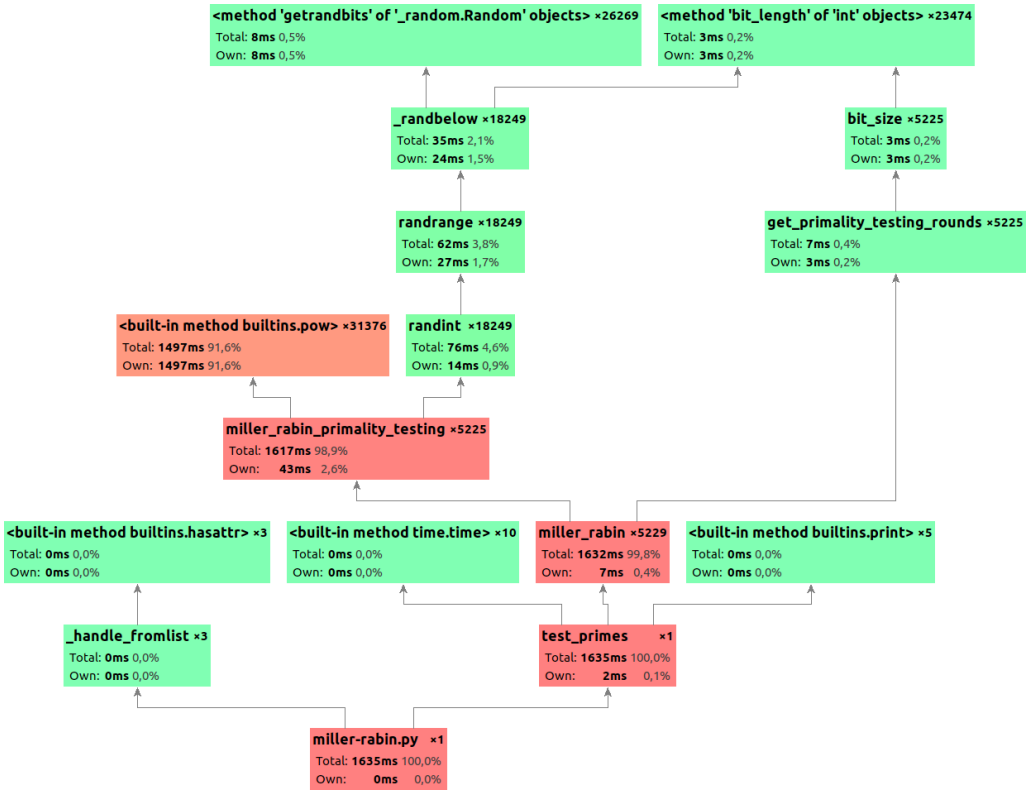

Figure 9. Gráfico de chamadas para a execução do algoritmo Miller-Rabin.

3.2.3. Baillie-PSW

O teste de primalidade desenvolvido por Robert Baillie, Carl Pomerance, John Selfridge e Samuel S. Wagstaff se mostra muito eficiente devido a se basear em testes de *strong pseudoprimes* e no de Lucas *pseudoprimes* [Baillie and Wagstaff 1980]. Diversas variações existem [Martin b], mas de maneira geral, o algoritmo segue a seguinte ideia:

1. Faça um teste de *strong pseudoprimes* base 2 em n. Se o teste falhar, retorne que n é composto. Caso contrário, o número n é um provável primo. Vá para o passo 2.
2. Na sequência $5, -7, 9, -11, 13, ...$ encontre o primeiro número D no qual o símbolo de Jacobi $D/n = -1$. Então, faça o teste de $pseudoprimes$ de Lucas com o discriminante D em n. Se o teste falhar, retorne que n é composto. Senão, n é muito provável de ser primo.

O teste de Baillie-PSW é determinístico para números com menos de 64 bits e não foi encontrado até hoje nenhum número composto acima desse limite que passe no teste [Pomerance 1984]. Em 1984, Pomerance colocou um prêmio de 30 dólares para quem encontrasse um número que passasse no teste. Em 1994 esse valor aumentou para 620 doláres. O que significa que, muito provavelmente, o algoritmo não falha em nenhum dos casos. Além disso, o programa *PRIMO* [Martin a], que testa a primalidade de curvas elípticas, o utiliza e caso falhasse em algum teste isso já teria sido percebido.

Pelas razões do teste se basear em outros algoritmos, esse se torna de fácil entendimento, caso o leitor conheça os demais métodos. As escolhas adotadas na implementação

foram:

1. Verifique se n é 2. Caso seja, retorne PRIMO.
2. Caso n seja par, retorne COMPOSTO.
3. Verifique se o número é uma potência quadrática. Caso seja, retorne COMPOSTO.
4. Realize *trial division* de 3 até a raiz quadrada de n. Caso a divisão por tentativa retorne COMPOSTO, pare e retorne COMPOSTO, senão prossiga.
5. Faça o teste de *strong prime*, no caso Miller-Rabin base 2, e verifique se n é composto. Caso seja, retorne COMPOSTO.
6. Se todos os demais casos passarem, faça o teste de *strong prime* de Lucas.

O teste possui uma performance muito boa pois os números serão filtrados a cada passo, impedindo que as partes que exigem um alto poder computacional, como o teste de Lucas, sejam alcançadas. A figura 10 demonstra esse fato, onde apenas cerca de 1300 de 5200 execuções (em torno de 25%) foram utilizadas.

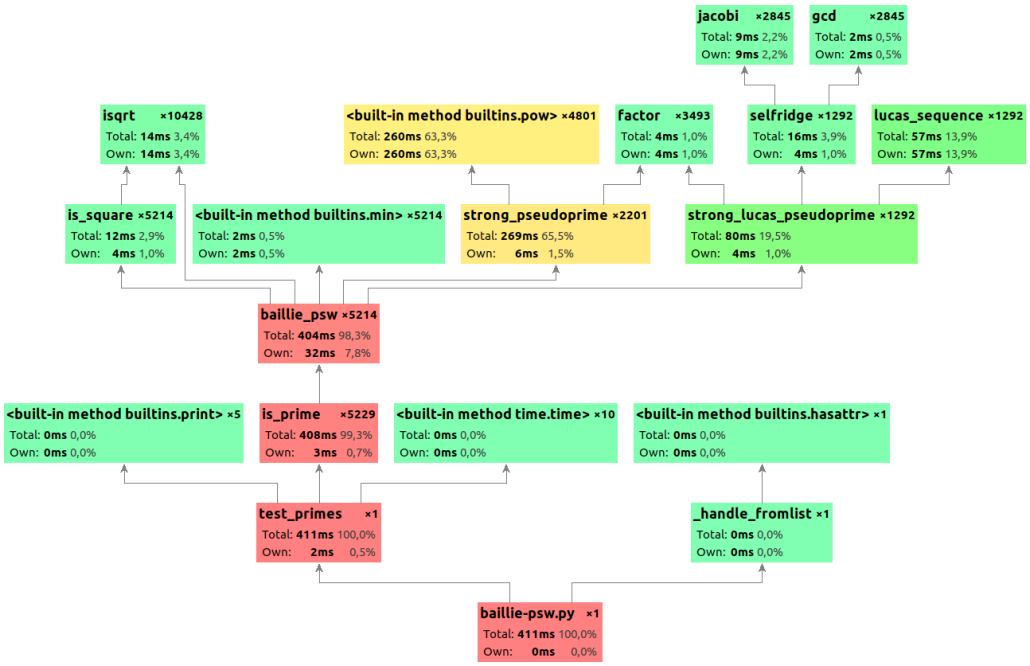

Figure 10. Gráfico de chamadas para a execução do algoritmo Baillie-PSW.

O algoritmo, assim como os demais, apresenta um gargalo na operação de exponenciação, mas isso não é possível de ser resolvido. Outro ponto que podemos notar na figura 10 é que a sequência de Lucas representa, aproximadamente, 14% da execução, sendo executada apenas 1292 vezes. A divisão por tentativa, que levou $32ms$ também eliminou 3013 execuções do *strong prime (miller-rabin base 2)*, que seriam testadas de forma desnecessária.

3.2.4. Comparação dos testes de primalidade

A análise dos testes de primalidade mostram que são muito mais eficientes do que os testes de fatoração apresentados na seção 3.1. Por este motivo, deve-se primeiro checar se o

número é primo para logo após tentar fatorá-lo. Na figura 11 é mostrada uma comparação dos algoritmos Solovay-Strassen, Miller-Rabin e Baillie-PSW, todos mostrados ao decorrer deste relatório.

Note que algumas otimizações nas implementações ou escolha de linguagens de programação melhores, como C, poderiam melhorar ainda mais o desempenho obtido, porém o tempo relativo deve permanecer próximo.

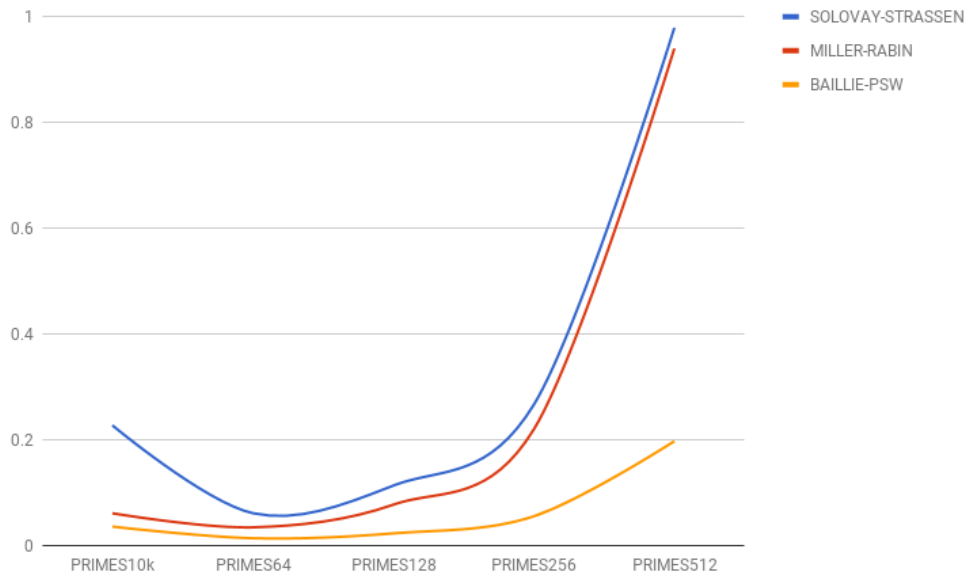

Figure 11. Gráfico comparativo dos testes de primalidade Solovay-Strassen, Miller-Rabin e Baillie-PSW.

A partir da análise da figura 11 pode-se perceber que o método de Solovay-Strassen se apresenta muito ruim para números menores que 64 bits e depois sofre uma melhora considerável. O crescimento tanto do método de Solovay-Strassen quanto de Miller-Rabin podem ser aproximados por uma curva semelhante, sendo que Miller-Rabin possui um crescimento maior. Em todos os casos o método Baillie-PSW se mostrou o melhor e nunca ultrapassou 200 milissegundos, o que é algo muito bom. A curva mostrada no gráfico também é quase linear.

Um ponto a se ressaltar é que os valores apresentados estão em segundos e representam o tempo total de execução necessário para testar todos os primos de cada conjunto de dados. Por exemplo, no caso do Baillie-PSW com 512 bits, foram realizados 1000 testes, logo, o caso médio é de 0.2 milisegundos para cada número.

4. Conclusão

O estudo dos algoritmos de fatoração e testes de primalidade foram enriquecedores do ponto de vista de implementação e análise. O conhecimento agregado através do testes, no qual foi possível identificar onde existem os gargalos de cada algoritmo e onde cada algoritmo obtém um desempenho melhor é algo insubstituível. Nas implementações futuras, deve-se atentar ao fato de que a exponenciação foi um gargalo em grande parte dos casos, logo algoritmos de exponenciação melhores precisam ser estudados, assim como,

algoritmos de multiplicação, gerência de memória e operações quando números grandes são utilizados.

As análises feitas comprovam que os algoritmos de teste de primalidade são muito mais eficientes do que os algoritmos de fatoração, portanto, estes devem ser feitos em último caso. A implementação de outros algoritmos para resolução de problemas como o logaritmo discreto também ser analisados, já que não foram abordados neste trabalho.

Algumas matérias de estudo foram aprendidas, porém, por falta de tempo não foram abordadas, como o algoritmo de trocas de chaves de Diffie-Hellman, ou o algoritmo que tinha como objetivo substituí-lo para grandes grupos sem haver a necessidade de recombinar as chaves.

Todas as implementações apresentadas ao longo deste relatório podem ser visualizadas no repositório do Github: https://github.com/raphaelbernardino/estudo-orientado.

References

[Baillie and Wagstaff 1980] Baillie, R. and Wagstaff, S. S. (1980). Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417.

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.

[FIPS 2013] FIPS, P. (2013). 186-4. *Digital Signature Standard (DSS)*.

[Maier 2005] Maier, R. R. (2005). Teoria dos números. *Notas de aula, Departamento de Matemática-IE/UnB*.

[Martin a] Martin, M. Primo-primality proving. *single-processor ECPP software available fro m http://www. ellipsa. net/public/primo/primo. html*.

[Martin b] Martin, M. Re: Baillie-psw-which variant is correct?, 2004.

[Menezes et al. 1996] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC press.

[Pomerance 1984] Pomerance, C. (1984). Are there counter-examples to the baillie-psw primality test. *Dopo Le Parole aangeboden aan Dr. AK Lenstra. Privately published Amsterdam*.

[Rivest et al. 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.