Universidade Federal do Paraná

Giane Lima de Almeida

Análise dos Testes de Primalidade do RSA

Curitiba PR
2016

Giane Lima de Almeida

Análise dos Testes de Primalidade do RSA

Trabalho apresentado como requisito para conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Curitiba PR
2016

"O estudo, a busca da verdade e da beleza são domínios em que nos é consentido sermos crianças por toda a vida."

Albert Einstein

Agradecimentos

Por tantas vezes quase fui vencida seja pelo cansaço ou por frustração, sabe Deus de onde veio forças para prosseguir. Então finalmente cheguei aqui. E como se superar é tudo o que sempre fazemos, seguimos em frente para a conclusão deste trabalho.

Muitos depositaram em mim sua confiança e contribuíram para que este trabalho adquirisse forma e se materializasse a contento. Sou grata a todos, profundamente.

Agradeço o apoio e ajuda de todos os envolvidos durante esta jornada exaustiva. Agradeço primeiramente a Deus, pela força e saúde que tive para superar todos os obstáculos encontrados. Ele sempre me dá a força que preciso.

Ao meu orientador Luiz Carlos Pessoa Albini, por sua infinita paciência, pelo exemplar professor e incentivador do apreço ao conhecimento. Seu exemplo de profissionalismo sem a qual este trabalho não se realizaria. E a todos os meus professores, que de uma forma um tanto incomum, me fizeram olhar o impossível como possível. A minha Mãe que por tantas vezes me incentivou, ajudou e me aguentou durante a jornada deste trabalho.

A todos meus amigos e familiares que me incentivaram nesta etapa do trabalho.

Resumo

Atualmente existem mais de uma centena de testes de primalidade, cada um deles com uma fundamentação teórica diferente, iremos estabelecer uma análise comparativa. Para uma análise precisa será utilizado o algoritmo de criptografia RSA, que por definição necessita de dois números primos "muito grandes" na fase de geração das chaves, o ponto exato onde os testes são de vital importância. Os testes de primalidade são divididos em 2 categorias: os testes determinísticos apresentam geralmente um custo de processamento maior do que os testes probabilísticos, então é frequente que antes de se aplicar um teste determinístico, seja avaliado em uma análise o comportamento de testes probabilísticos. Veremos alguns testes de ambas as categorias, tópicos de teoria dos números serão responsáveis pela definição precisa e coesa de cada teste, e ao final uma análise completa de cada módulo do processamento é apresentada para fixar as características gerais e avaliar o desempenho do algoritmo desenvolvido.

Palavras-chave: Criptografia, Números Primos, RSA, Chave Privada, Chave Pública, Decriptografia, Teste de Primalidade.

Abstract

Currently there are more than a hundred primality tests, each with a different theoretical foundation, we will establish a comparative analysis. For an accurate analysis we will use the RSA encryption algorithm, which by definition needs two "very large" prime numbers in the key generation phase, the exact point where the tests are of vital importance. Primality tests are divided into two categories: deterministic tests usually have a higher processing cost than probabilistic tests, so it is often necessary to evaluate the probabilistic test behavior before a deterministic test is applied. We will see some tests of both categories, number theory topics will be responsible for the precise and cohesive definition of each test, and at the end a complete analysis of each processing module is presented to fix the general characteristics and evaluate the performance of the algorithm developed.

Keywords: Cryptography, Prime Numbers, RSA, Private Key, Public Key, Decryption, Primality Test.

Sumário

Lista de Figuras

Lista de Algoritmos

Lista de Acrônimos

AKS	Manindra **A**grawal, Neeraj **K**ayal e Nitin **S**axena
APR	Leonard **A**dleman, Carl **P**omerance e Robert Scott **R**umely
ASCII	American Standard Code for Information Interchange
Baillie-PSW	Robert Baillie, Carl **P**omerance, John **S**elfridge e Samuel **W**agstaff
BIT	Simplificação para dígito binário, binary digit
BYTE	Binary Term, uma sequência de 8 bits, ou um octeto
CRT	Chinese Remainder Theorem
DINF	Departamento de Informática da UFPR
MDC	Máximo Divisor Comum
MIT	Massachussetts Institute of Technology
MMC	Mínimo Múltiplo Comum
RSA	Ronald Linn **R**ivest, Adi **S**hamir, Leonard **A**dleman
UFPR	Universidade Federal do Paraná

Lista de Símbolos

φ	função de Euler
ϕ	Totiente de Euler
α	alfa, primeira letra do alfabeto grego
β	beta, segunda letra do alfabeto grego
θ	theta, nona letra do alfabeto grego
γ	gama, terceira letra do alfabeto grego
$\mathbb{Z}$	conjunto dos números inteiros
$\mathbb{N}$	conjunto dos números naturais
$\mathbb{Z}_{+}^{*}$.	conjunto dos números inteiros positivos
$a\ (mod\ b)$	o resto da divisão inteira de a por b
$a \equiv b\ mod\ n$	a e b pertencem a mesma classe de equivalência módulo n
$a \not\equiv b\ mod\ n$	a e b não pertencem a mesma classe de equivalência módulo n
$\exists$	existe
$\forall$	para todo
$A \subset B$	o conjunto A está contido propriamente no conjunto B
$A \subseteq B$	o conjunto A está contido ou é igual ao conjunto B
$B \supset A$	o conjunto B contém propriamente no conjunto A
$B \supseteq A$	o conjunto B contém ou é igual ao conjunto A
$a \in A$	o elemento a pertence ao conjunto A
$a \notin A$	o elemento a não pertence ao conjunto A
$A \cup B$	união dos conjuntos A e B
$A \cap B$	intersecção dos conjuntos A e B
$\emptyset$	conjunto vazio
/	tal que
$A \backslash B$	A exceto B
∞	infinito
$a := b$	preencha a com o valor de b
$a \mid b$	a divide b
$a \nmid b$	a não divide b
$\lceil x \rceil$	menor inteiro maior ou igual a x
$\lfloor x \rfloor$	maior inteiro menor ou igual a x
$max\{a, b\}$	máximo valor entre a e b
$min\{a, b\}$	mínimo valor entre a e b
$mdc\ (a, b)$	máximo divisor comum entre a e b
$mmc\ (a, b)$	mínimo múltiplo comum entre a e b
$\sum_{i=x}^{k}$	somatório com i variando de x até k
$\prod_{i=x}^{k}$	produtório com i variando de x até k
$a \Rightarrow b$	a implica em b. Se a é verdadeiro então b é verdadeiro,

	se a é falso então b é falso
$a \Leftrightarrow b$	a é verdadeiro se somente se b é verdadeiro
$a^{-1}\ (mod\ n)$	inverso multiplicativo de a (mod n)
$f : X \rightarrow Y$	função f que mapeia o conjunto X no conjunto Y
f^{-1}	função inversa da função f
$log_a\ b$	logaritmo de b na base a
$ln\ b$	logaritmo de b na base e, sendo $e \approx 2{,}71828183$
$\log\ b$	logaritmo de b na base 10
$\log_2\ b$	logaritmo de b na base 2
$L(a, b)$	símbolo de Legendre
$J(a, b)$	símbolo de Jacobi
$\binom{n}{p}$	número de combinações de n elementos agrupados n a p
$O(f(n))$	notação de complexidade computacional de pior caso no comportamento assintótico da função $f(n)$
$\pi(n)$	número de primos menores que n
$\oplus$	ou exclusivo
$n!$	fatorial de n

Capítulo 1

Introdução

Atualmente quando mandamos mensagens tudo o que queremos é que o sigilo seja mantido e que somente o destinatário possa receber a mensagem. Portanto para proteger todas as informações enviadas, um assunto entra em pauta: a criptografia. Criptografia é o processo de codificação de mensagem (ou informações), de tal maneira que intrusos ou hackers não possam ler, mas as partes autorizadas sim [35]. Em um esquema de criptografia, a mensagem é criptografada usando um algoritmo de criptografia, transformando-o em um texto ilegível cifrado.

Criptografia Simétrica: por definição [35], é uma forma de criptossistema em que a cifragem e a decifragem são realizadas usando-se a mesma chave. Também conhecida como criptografia convencional [Figura 1.1].

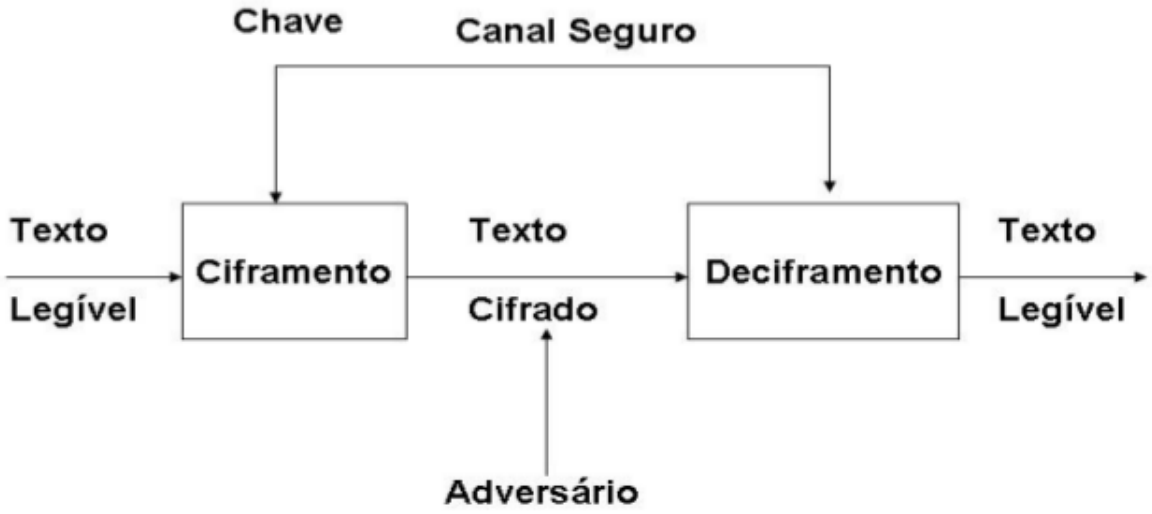


Figura 1.1: Exemplo de Criptografia Simétrica

Criptografia Assimétrica: por definição [35], é uma forma de criptossistema em que a criptografia e a decriptografia são realizadas usando-se duas chaves diferentes, uma conhecida como chave pública e outra conhecida como chave privada. Também conhecida como codificação de chave pública [Figura 1.2].

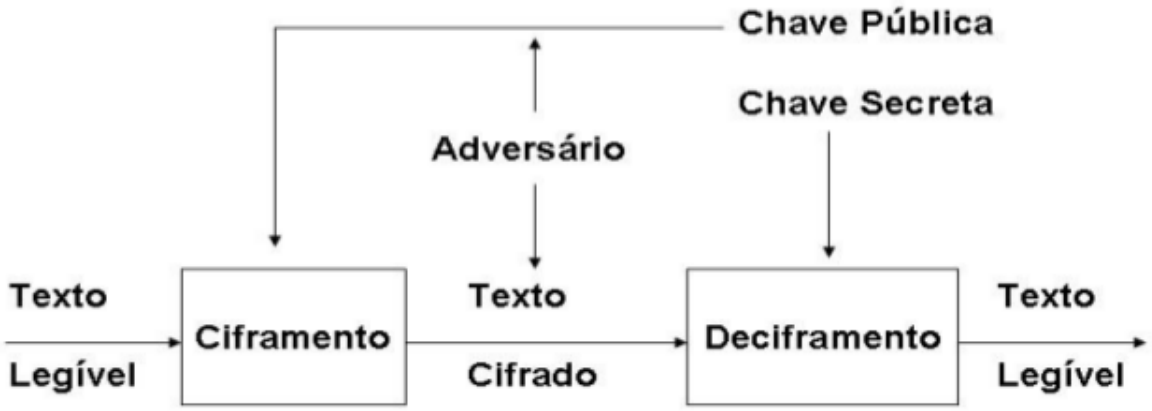


Figura 1.2: Exemplo de Criptografia Assimétrica

A figura 1.3 apresenta uma breve comparação entre as principais características das criptografias simétricas e assimétricas.

Algoritmo	Simétrico	Assimétrico
Origem	Cerca de 5000 anos	Cerca de 30 anos
Chave	Única	Par de chaves, sendo uma pública e uma privada
Sigilo da chave	Manter em sigilo	Manter em sigilo chaves privadas, chaves públicas podem ser divulgadas
Compartilhamento de chave	Emissor e receptor compartilham a mesma chave	Emissor e receptor possuem pares de chaves distintas
Distribuição de chaves	Crescimento quadrático	Crescimento linear
Velocidade de processamento	Rápida (Gb/seg.)	Lenta (Mb/seg.)
Ciframento	Permite	Permite
Assinatura Digital	Não permite	Permite
Protocolo/Algoritmo criptográfico	Compartilhados entre emissor e receptor	Compartilhados entre emissor e receptor

Figura 1.3: Comparação entre as Criptografias Simétrica e Assimétrica [35]

Existem diferentes algoritmos assimétricos, um dos mais conhecidos é o algoritmo de criptografia RSA [Capítulo 3], este é amplamente utilizado nos navegadores, para sites seguros e para cifrar e-mails. A característica principal do RSA está na busca por números primos grandes, os responsáveis por mais da metade do tempo de geração das chaves.

Os números primos nos levam a estudar o assunto central deste trabalho: os testes de primalidade, e para que seja possível fazer uma análise qualitativa e quantitativa dos algoritmos é necessário conhecer um pouco sobre o algoritmo de criptografia RSA, o qual utiliza dois números primos na fase de geração de chaves em que os testes serão aplicados.

Provar a primalidade é um campo complicado, no qual existem muitos algoritmos, todos com diferentes vantagens e desvantagens. Os algoritmos para determinação de primalidade são classificados em duas classes de métodos: os determinísticos e os probabilísticos. Testes determinísticos são testes que asseguram certeza matemática para todas as respostas. Testes probabilísticos são testes que contém probabilidade de erro, ou de incerteza. Escolhe-se um

número aleatório e aplica-se a ele critérios que possam refutar ou confirmar certa propriedade. Escolhemos nos concentrar em versões de algoritmos probabilísticos como: Solovay-Strassen [Definição 4.1.1], Miller-Rabin [Definição 4.3.1], Fermat [Teorema 13], Lucas (com suas variações usando: número de Jacobi [Definição 4.0.3], número de Mersenne [Definição 4.0.4] e número de Riesel [Definição 4.0.5]). Apresentaremos também os algoritmos não-probabilísticos como: Agrawal-Kayana-Saxena (AKS) [Definição 4.7.1], Adleman-Pomerance-Rumely (APR) [Definição 4.8.1], Pocklington [Definição 4.10.1], Baillie-PSW [Definição 4.9.1], Lenstra-Pomerance [Definição 4.11.1]. O algoritmo de criptografia RSA mantém sua implementação habitual, apenas modificamos os teste de primalidade durante a fase de geração das chaves pública e privada. Durante a análise de cada teste de primalidade ideias diferentes foram testadas. Primeiro analisamos de forma aleatória, ou seja, geração de número aleatório, seguindo da geração aleatória por intervalos sugeridos, e por último uma análise de todos os teste de primalidade usando um vetor de números aleatórios contendo apenas um número primo fixado na posição 10, usou-se valores fixos para este.

Dentre os testes realizados o que obteve o melhor resultado dentre o probabilístico foi o teste de primalidade de Solovay-Strassem 4.1.1, e dentre os determinísticos foi o teste de primalidade AKS 4.7.1. Estas análises serão apresentadas nos capítulos 4 e 5.

Neste capítulo veremos também a descrição de conceitos (ou noções básicas) para o entendimento do Algoritmo de Criptografia RSA e noções mais precisas de como o teste de primalidade é importante. Este documento vem discutir em especial um método criptográfico (realmente uma das grandes famílias de métodos criptográficos) que permite ser usado para transferir informação firmemente e convenientemente entre dois meios na rede. Nossa apresentação envolve a teoria e prática de formatos variados de operações matemáticas, mas o RSA usa primordialmente a fatoração de números primos grandes.

1.1 Motivação

Hoje, toda a segurança de compras e transações bancárias feitas pela internet baseia-se na dificuldade de os computadores fatorarem números muito grandes. A crescente expansão da rede nos coloca diante de um problema bem definido a segurança da própria rede e dos dados. Garantir a transmissão de dados de forma sigilosa e segura, através dos diversos dispositivos em funcionamento. Além do desafio de colocar em prática os conceitos de criptografia utilizada atualmente, é necessário que a velocidade de distribuição das chaves criptográficas seja bastante elevada. O algoritmo de criptografia RSA foi escolhido por ser o mais usado atualmente, isto implica num estudo mais aprofundado sobre este. Os parâmetros para a avaliação demonstram a variabilidade de opções, que dependem do contexto. Assim o presente trabalho visa avaliar três visões: teoria dos Números, RSA e testes de primalidade [4, 3].

Objetivos Gerais: Avaliar as etapas do algoritmo de criptografia RSA, e os vários tipos de testes de primalidade.

Objetivos Específicos:

1. Analisar a qualidade e limitações do algoritmo de criptografia - RSA.
2. Aprofundar o conhecimento da Teoria dos Números (aritmética modular, números primos, algoritmo de Euclides, testes de primalidade).
3. Aplicar os conceitos na implementação do RSA.
4. Descrição das diversas técnicas de Primalidade, apropriadas para geração das chave do RSA.

5. Analisar ferramentas que melhor se adaptem na resolução dos testes de primalidade e aplicação dos conceitos na Teria dos Números.

Este trabalho está dividido em capítulos que apresentam o tema, motivação e objetivo assim como a estrutura na qual foi elaborado o texto. Para cada definição, conceito ou problema, é apresentado uma informação adicional ao escopo deste documento. Após é feita uma avaliação preliminar sobre o trabalho de implementação do algoritmo de criptografia do RSA, com uma análise e sugestões de continuidade do trabalho de graduação. O trabalho está estruturado da seguinte forma:

Capítulo 1 – Introdução: É apresentado o formato geral da morfologia deste trabalho, os objetivos gerais e específicos. Descrevemos os fundamentos da criptologia, apresentamos os conceitos e as noções básicas de criptografia e o motivo central para estudarmos os testes de primalidade, e descrevemos brevemente o projeto.

Capítulo 2 – Fundamentos Matemáticos: Apresentamos as definições e fundamentos matemáticos necessários para o bom entendimento do tema, e que serão usados ao longo do trabalho. [18].

Capítulo 3 – Algoritmo de Criptografia RSA : É descrito o algoritmo de Criptografia RSA e a sua estrutura morfológica. Apresentamos brevemente algumas possíveis otimizações e alguns tipos de ataques.

Capítulo 4 – Testes de Primalidade: Apresentamos e descrevemos cada um dos testes de primalidade. Mostramos as dificuldades que encontramos ao tentar obter uma implementação dentre as fundamentações teóricas diferentes. Além disso, é feita a descrição dos principais algoritmos utilizados na implementação do processo de desenvolvimento dos testes.

Capítulo 5 – Análise Comparativa e Resultados dos Testes de Primalidade: Apresentamos os resultados computacionais da implementação e a análise comparativa dos testes de primalidade.

Capítulo 6 – Conclusão: Apresenta as considerações finais da monografia, e as conclusões obtidas no desenvolvimento, a análise do algoritmo implementado, e à relação direta destes com a segurança do RSA.

Referências Bibliográficas: Por fim, as Referências Bibliográficas as quais foram consultadas e que enriqueceram a execução deste trabalho.

Anexos: Apresentação do código de criptografia e decriptografia.

Capítulo 2

Fundamentos Matemáticos

A força dos algoritmos criptográficos reside na dificuldade em fatorar grandes números inteiros. Portanto a fundamentação matemática que será apresentada é essencial para a compreensão do algoritmo RSA e de vital importância para os testes de primalidade.

Descreveremos os conceitos que irão fundamentar os algoritmos de primalidade fornecendo uma base matemática para compreensão do funcionamento de cada método. Manteremos a exposição como auto-contido quanto possível, iremos destacar o necessário para nosso objetivo e de maneira sucinta, para melhor entendimento e aproveitamento. O conteúdo apresentará estes conceitos que são importantes para o entendimento do texto.

2.1 Conceitos Básicos da Teoria dos Números

Definição 2.1.1. (Conjunto Numérico) [36, 13] Conjunto Numérico é um conjunto cujos elementos são formados exclusivamente por números e estes guardam entre si algumas características comuns. Um conjunto numérico possui seus elementos perfeitamente caracterizados.

Definição 2.1.2. (Conjunto $\mathbb{N}$) [36, 13]
Conjunto dos números naturais: $\mathbb{N} = \{0, 1, 2, 3, ...\}$. $\mathbb{N}_*$ Conjunto dos números naturais exceto 0.

Definição 2.1.3. (Conjunto $\mathbb{Z}$) [36, 13]
O conjunto dos inteiros $\{..., -2, -1, 0, 1, 2, ...\}$ é denotado pelo símbolo $\mathbb{Z}$. O conjunto dos inteiros não nulos positivos $\{1, 2, ..., \infty\}$ é denotado pelo símbolo $\mathbb{Z}_+^*$.

Definição 2.1.4. (Números Primos) [36, 13]
Um número primo é um inteiro $n > 1$ tal que n não é divisível por nenhum inteiro além de 1 e n. Para dois números x e y, $x < y$ significa $y - x > 0$.

Definição 2.1.5. (Número Pseudo-Primos) [36, 13]
Dois números a e b são ditos co-primos, relativamente primos ou primos entre si, se o maior divisor de a e b é 1.

Teorema 1. [36, 13, 12, 4]
Existem infinitos números primos.

Teorema 2. (Teorema Fundamental da Aritmética) [36, 13, 12, 4]
Todo número inteiro positivo n maior do que 1 ou é primo ou se escreve de modo único (exceptuando a ordem dos fatores) como um produto de números primos.

Teorema 3. (Teorema Fundamental dos Primos) [36, 13, 12, 4]
Seja p um número primo e a,b inteiros positivos. Se p divide o produto ab então p divide a ou p divide b.

Definição 2.1.6. (Propriedade Fundamental dos Primos) [13]
Sejam a, b e c inteiros positivos e suponhamos que a, b são primos entre si.
(1) Se b divide o produto ac então b divide c.
(2) Se a e b dividem c então o produto ab divide c.

Teorema 4. (Teorema da Divisão) [36, 13]
Sejam a e b inteiros positivos. Existem números inteiros q e r tais que $a = bq + r$ e $0 \leq r < b$. Os valores de q e r são únicos. Se $r = 0$, então diz-se que b é divisor de a, ou que b divide a, denotado por $b \mid a$.

Teorema 5. (Teorema da Inversão) [36, 13]
A classe $\bar{a}$ tem inverso em $\mathbb{Z}_m$ se, e somente se, a e n são primos entre si [Definição 2.1.5]. Seja n um número primo. O inteiro positivo de a é o seu próprio inverso módulo n se, somente se, $a \equiv 1(\mathrm{mod} n)$ ou $a \equiv -1(\mathrm{mod} n)$. Portanto a solução $\bar{a}$ de $ax \equiv 1(\mathrm{mod} n)$ é chamada de um inverso de a módulo n.

Teorema 6. (Teorema da Fatoração Única) [13]
Dado um inteiro positivo $n \geq 2$ podemos sempre escrevê-lo, de modo único, na forma $n = p_1^{e_1} \cdots p_k^{e_k}$ onde $1 < p_1 < p_2 < \cdots < p_k$ são números primos e $e_1, \cdots, e_k$ são inteiros positivos.

Definição 2.1.7. (Máximo Divisor Comum) [36, 13]
O MDC de dois inteiros a e b (a ou b diferente de zero), denotado por (a,b), é o maior inteiro que divide a e b.

Definição 2.1.8. (Mínimo Múltiplo Comum) [36, 13]
Sejam $a, b \in \mathbb{Z}$ dois números, ambos não nulos. O mínimo múltiplo comum entre a e b é o número natural $m = \mathrm{mmc}(a,b)$ definido pelas duas propriedades:
a) $a \mid m = b \mid m$ (isto é, m é múltiplo comum de a e b.)
b) Se $a \mid c$ e $b \mid c$ para algum $c \in \mathbb{N}$ então temos também $m \mid c$.

Definição 2.1.9. [36, 13, 34]
Os inteiros a e b são relativamente primos se seu maior divisor comum for 1.

Teorema 7. Seja $n \in \mathbb{N}$ par. n é um número perfeito se, somente se, $n = 2^{k-1}(2^1 - 1)$ para algum $k \in \mathbb{N}$ é primo.

Teorema 8. (Teorema de Euler) [36, 13]
Sejam $a, n \in \mathbb{Z}$, com $n > 0$, tais que $(a,n) = 1$. Então $a^{\varphi(n)} \equiv 1 \pmod{n}$ para algum $k \in \mathbb{N}$ é primo.

Teorema 9. (Função de Euler) [36, 13]
Se n é um inteiro positivo, a função φ de Euler, denotada por $\varphi(n)$, é definida como sendo o número de inteiros positivos menores ou iguais a n que são relativamente primos com n.

Definição 2.1.10. (Algoritmo Euclidiano) [36, 13]
Sejam a, b números inteiros positivos. Suponhamos que existam inteiros g e s tais que $a = bg + s$. Então $\mathrm{mdc}(a,b) = \mathrm{mdc}(b,s)$.

Definição 2.1.11. (Algoritmo Euclidiano Estendido) [36, 13]
Sejam a, b inteiros positivos e seja d o máximo divisor comum entre a e b. Existem inteiros α e β tais que $\alpha \cdot a + \beta \cdot b = d$.

Definição 2.1.12. (Hipótese de Riemann) [30]
Um número n é primo se, e somente se, passa por um teste probabilístico de primalidade para todas as bases a, com $1 < a < 2\, log_2\, n$.

Definição 2.1.13. (Crivo de Erastóstenes) [34, 30, 36]
Se um número natural $a > 1$ é composto, então ele é múltiplo de algum número primo p tal que $p^2 \leq a$. Equivalentemente, é primo todo número a que não é múltiplo de nenhum número primo p tal que $p^2 < a$. [1]

Definição 2.1.14. (Símbolo de Legendre) [30, 13]
Seja p um número primo ímpar))) e $a \in \mathbb{Z}$. O símbolo de Legendre para a em relação a p é definido por:

$$\mathrm{L}(a,p) = \begin{cases} 0, & \text{se} \quad p \mid a, \\ 1, & \text{se} \quad a \text{ é resíduo quadrático (mod } p), \\ -1, & \text{se} \quad a \text{ não é resíduo quadrático (mod } p). \end{cases}$$

Dado um número primo ímpar p e um número $a \in Z$, o símbolo de Legendre é uma ferramenta útil para se rastrear se a é resíduo quadrático módulo p.

2.2 Aritmética Modular

Definição 2.2.1. Se a e b são inteiros e $a \neq 0$, então a divide b se existe um inteiro c tal que $b = a \cdot c$ ou equivalentemente: se $a \mid b$ é um inteiro. Quando a divide b, dizemos que a é um fator b e que b é um múltiplo de a. A notação $a \mid b$ denota que a divide b. E $a \nmid b$ então a não divide b.

Definição 2.2.2. Um inteiro n é dito um pseudoprimo na base b se n é composto, mas $\exists\, b \in \mathbb{Z}$ tal que $b^{n-1} \equiv 1 \mod n$.

Lema 10. Se n é um pseudoprimo nas bases b_1 e b_2 então n é pseudoprimo nas bases b_1, b_2 e b_1^{-1}.

Teorema 11. Seja n um número composto. Se existe ao menos um $b \in \mathbb{Z}_n$ tal que $b^{n1} \equiv 1 \mod n$, então n não será um pseudo-primo em pelo menos metade das possíveis bases.

Definição 2.2.3. (Números Compostos) [13]
Se $n > 0$ e $1 < b < n - 1$ são números inteiros e $b^{n-1}1 (\mathrm{mod}\ n)$, então n é um número composto. O número b é conhecido como testemunha do fato de n ser composto.

Definição 2.2.4. Se $n - 1 = q^k R$ onde q é primo e existe um inteiro a tal que $a^{n-1} \equiv 1 (\mathrm{mod}\ n)$ e $\mathrm{mdc}(a^{\frac{(n-1)}{q}} - 1, n) = 1$ então qualquer fator primo de n é côngruo a 1 módulo q^k.

[1] O método consiste em peneirar os números naturais em um intervalo $[2, n]$, jogando fora os números que não são primos.

Teorema 12. (Teorema Chinês do Resto) [13]
Sejam m e n inteiros positivos, primos entre si. O sistema

$$x \equiv a (\text{mod } m)$$

$$x \equiv b (\text{mod } n)$$

sempre tem uma única solução em $\mathbb{Z}_{mn}$.

Capítulo 3

Algoritmo de Criptografia RSA

O impacto da matemática e da Teoria dos Números no algoritmo criptográfico RSA são fortes, e em contrapartida, o desenvolvimento da cifra só foi permitido por conta de grandes avanços na teoria aritmética dos números. O importante marco da criação da criptografia assimétrica é o artigo publicado em 1976 por Whitfield Diffie e Martin Hellman [15], sugeria que com o desenvolvimento das redes de computadores, algumas informações deveriam ser cifradas antes de serem enviadas. Os dois cientistas da computação propuseram um novo método para que a chave fosse enviada de forma segura, em que todas as informações necessárias para a troca fossem disponibilizadas publicamente. A ideia consiste em usar uma função que seja fácil de se calcular mas computacionalmente difícil de inverter, a menos que se conheça o segredo.

Partindo desta definição surge o algoritmo de criptografia RSA [12, 18, 35] que foi desenvolvido no Instituto Tecnológico de Massachussets (MIT) em 1977 por Ronald Linn Rivest, Adi Shamir, Leonard Adleman. É matematicamente baseado na Teoria dos Números, principalmente nas áreas: de Aritmética Modular e Primalidade [36].

O RSA é um dos principais algoritmos de criptografia utilizado na atualidade. A ideia do algoritmo RSA concentra-se no fato de que, embora seja fácil encontrar dois números primos de grandes dimensões (maiores do que 100 dígitos), o tempo estimado para fatorar estes números, por exemplo, de 308 dígitos, com os algoritmos clássicos é de aproximadamente 100 mil anos [2, 33]. De fato, ele mostra-se computacionalmente inquebrável com números de tais dimensões, e a sua força é geralmente quantificada com o número de bits utilizados para descrever tais números. Para um número de 100 dígitos são necessários cerca de 350 bits, e as implementações atuais superam os 512 e mesmo os 1024 bits. A análise morfológica inicia-se com o pré-processamento do texto, seguido dos testes de primalidade que são protagonistas para geração das chaves. Após este processo, utiliza-se a chave pública para cifrar o texto, e a chave privada para decifrar o texto cifrado.

O algoritmo criptográfico RSA envolve três etapas fundamentais:

- Geração aleatória de dois números inteiros que devem passar pelo teste de primalidade, o qual resultará na geração do par de chaves pública (como o nome propriamente diz, pública e conhecida de todos)[12, 18, 35]; e privada (secreta, sendo de conhecimento apenas do seu proprietário o qual se responsabiliza em mantê-la sob sigilo)[18, 35, 12];
- A criptografia utilizando a chave pública;
- A decriptografia utilizando a chave privada.

A primeira etapa do método RSA é a escolha de dois números inteiros p e q gerados aleatoriamente. Cada número gerado passa pelo teste de primalidade (motivo central do presente

trabalho), ou seja, é verificado se é um número par ou um número composto [definição 2.2.3]. Caso seja verdadeiro o número é descartado e outro será testado até que se obtenha dois números primos. Computa-se o produto dos dois primos selecionados $n = p \cdot q$. Com o valor de n calculado podemos gerar o tamanho dos blocos [12, 18, 35] para cifrar o texto. A mensagem é dividida em blocos de tamanho fixo (exemplo: 64 bits, 128 bits). Cada bloco [1] [Figura 3.1] é cifrado como uma unidade, a mensagem é cifrada bloco por bloco. Para o cálculo do tamanho dos blocos é necessário garantir a corretude dos algoritmos de criptografia e decriptografia, mantendo a condição de $M < n$ e o tamanho dos blocos devem ter no máximo o tamanho $log_2 n$. Sempre que o tamanho da codificação do texto puro não proporcionar uma divisão exata pelo tamanho do bloco, haverá a necessidade de se completar o tamanho do último bloco, conforme um padrão a ser adotado entre as partes envolvidas, por exemplo, preenchendo-o com bytes nulos ao final.

Portanto ao final desta etapa temos o produto dos primos n e o texto pré-formatado em blocos M. Para a geração das chaves é necessário ainda alguns passos [12, 18, 35].

Calculamos a função totiente de Euler [Teorema 8] $\varphi(n) = (p-1)(q-1)$. Selecionar e um número inteiro positivo gerado aleatoriamente tal que e seja relativamente primo e que seja inversível módulo $\varphi(n)$, tal que $MDC(e, \varphi(n)) = 1$. Determinar d um número inteiro, calculado pelo algoritmo estendido de Euclides [Teorema 2.1.11] tal que $d \cdot e \equiv 1 (\text{mod } \varphi(n))$. Obtemos ao final desta etapa o par de chaves: a chave pública (n, e) e a chave privada (n, d).

A segunda etapa do método RSA é o passo da criptografia [figura 3.2 e figura 3.3] ou codificação do texto (algumas pessoas chamam de encriptação E). Com o par de números (n, e) temos a chave pública escolhida, por definição do RSA a criptografia é feita da seguinte forma: $E_{n,e}(M) = M^e \mod n$, que é garantido matematicamente pela função de Euler [Teorema 8 e 9]. Ao final deste processo temos o texto cifrado ou encriptado.

A terceira e última etapa é o passo da decriptografia [figura 3.2 e figura 3.3] ou decodificação do texto cifrado (algumas pessoas chamam de decifração D). Com o par de números (n, d) temos a chave privada escolhida, por definição do RSA a decriptografia fica da seguinte forma: $D_{n,d}(M) = M^e \mod n$. Ao final deste processo temos o texto puro novamente [36].

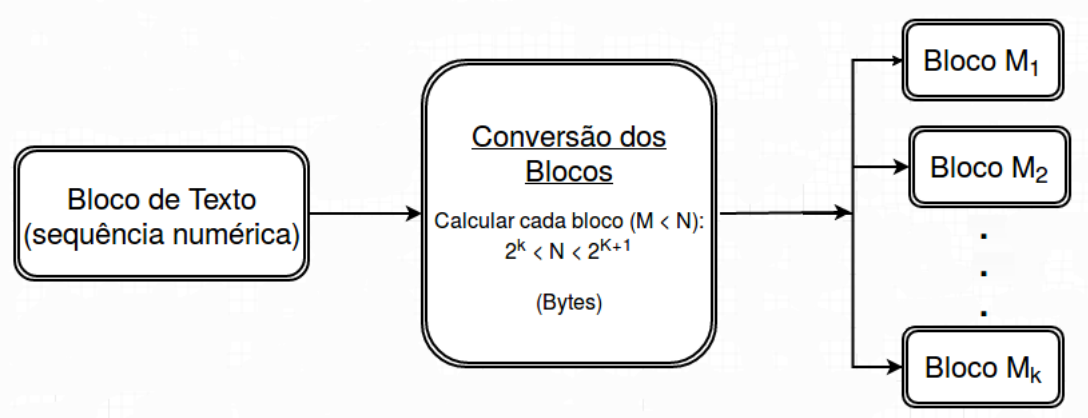


Figura 3.1: Exemplo de Diagrama de Bloco

[1] Modelo de cifra em bloco: Electronic CodeBook (ECB) é uma mensagem particionada em blocos de comprimento n-bytes e estes são cifrados separadamente.

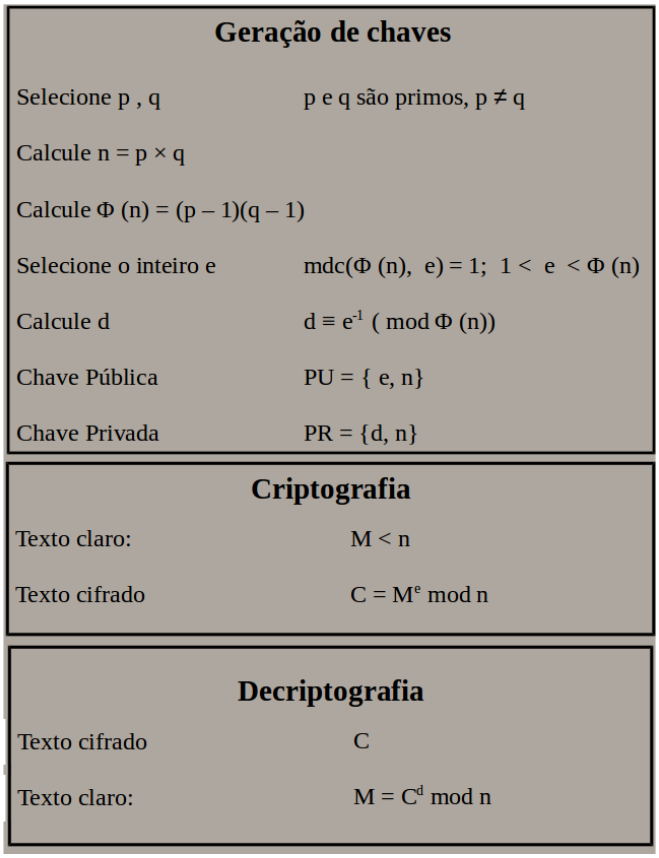


Figura 3.2: Algoritmo RSA

PRÉ-PROCESSAMENTO:

1. Criptografar a mensagem: “HELLO”.
2. Utilizando a tabela ASCII converter cada carácter em valor numérico e colocá-lo lado a lado:

M = 7269767679

3. Gerar aleatoriamente dois números inteiros.
4. Testar a primalidade dos números gerados.
5. Os dois números primos aleatórios são:

p = 29, q = 37.

6. Calcular n = p·q = 29 * 37 = 1073.

7. φ(n) = (p – 1)(q – 1) = (29 – 1)(37 – 1)= 1008.

8. Gerar aleatoriamente: e = 71, sendo :
   mdc (φ(n) , e) = 1 e 1 < e < φ(n).
9. Calcular d (d e e são inversos multiplicativos):
71*d mod 1008 = 1, d = 1079.

Portanto o par de chaves é:
   Chave Pública : (1073, 71);
   Chave Privada: (1073, 1079);
Transformar a mensagem em blocos (M < n).

Com blocos de tamanho = 3 , então a mensagem fica dividida da seguinte forma 726 976 767 900 (completamos com zeros).

CIFRAGEM :

Para cifrar cada bloco:
   726^{71} mod 1073 = 436
   976^{71} mod 1073 = 822
   767^{71} mod 1073 = 825
   900^{71} mod 1073 = 552
A mensagem cifrada é 436 822 825 552.

DECIFRAGEM:

Para decifrar a mensagem criptografada : 436 822 825 552.
   436^{1079} mod 1073 = 726
   976^{1079} mod 1073 = 976
   767^{1079} mod 1073 = 767
   900^{1079} mod 1073 = 900
Ou seja, a sequência de números 726976767900.
A nossa mensagem em texto simples 72 69 76 76 79: “HELLO”.

Figura 3.3: Exemplo Numérico do RSA

Capítulo 4

Testes de Primalidade

Neste capítulo iremos apresentar os conceitos de primalidade e definições correlatas aos testes de primalidade estudados.

Provar a primalidade é um campo complicado, no qual existem muitos algoritmos, todos com diferentes vantagens e desvantagens. O problema de determinar se um número é primo está dividido em duas classes: Algoritmos rápidos porém probabilísticos (podem dar resultado falsos), e a classe dos algoritmos determinísticos cujo desempenho para números muito grandes deixa a desejar. Veremos versões de algoritmos probabilísticos e algoritmos determinísticos.

O primeiro teste de primalidade conhecido é o Crivo de Erastóstenes [definição 2.1.13], proposto aproximadamente em 240 a.c., a definição de número composto deste é utilizada em todos os algoritmos testados. Algumas definições a seguir serão importantes para o bom entendimento dos algoritmos de primalidade.

Definição 4.0.1. (Teste de Primalidade) [13]
Seja $n > 0$ um inteiro tal que $n - 1 = p_1^{e_1} \cdots p_r^{e_r}$ onde $p_1 < \cdots < p_r$ são primos. Se para cada $i = 1, \cdots, r$ existirem inteiros positivos $b_i (2 \leq b_i \leq n - 1)$ que satisfaçam

$$b^{n-1} \equiv 1 (\text{mod } n)$$

$$b^{\frac{n-1}{p}} 1 (\text{mod } n)$$

, então n é primo.

Definição 4.0.2. (Número de Charmichael)[13]
Um número de Charmichael é um inteiro que é um pseudoprimo para toda base $b \in \mathbb{Z}$.

Definição 4.0.3. (Número de Jacobi) [13]
Sejam a e m inteiros. Denominamos $(\frac{a}{m}) = (\frac{a}{p_1}) \cdot (\frac{a}{p_2}) \cdots (\frac{a}{p_r})$ como Número de Jacobi, onde $p_1, p_2, \cdots, p_r$ é a decomposição em primo de m. O método tem o nome do matemático alemão Carl Gustav Jakob Jacobi.

Definição 4.0.4. (Número de Mersenne) [13]
Esse conjunto de números foi proposto no século XVII, recebeu o nome do seu descobridor o frade e matemático Marin Mersenne. Um número de Mersenne é todo número $a \in \mathbb{N}$ tal que $a = 2^a - 1 , a \in \mathbb{N}$.

Definição 4.0.5. (Número de Riesel [1]) [19]
Em 1956, Hans Riesel mostrou que existem infinitos números inteiros k de tal modo que $k \cdot 2^n - 1$.

[1] https://it.wikipedia.org/wiki/Numero_di_Riesel

Um número de Riesel é um número natural ímpar positivo k tal que cada número inteiro n da forma $k \cdot 2^n - 1$ é composto, isto é, não é um número primo. Em outras palavras, quando k é um número de Riesel, todos os números do conjunto são compostos do seguinte:

$$\{k \cdot 2^n - 1 : n \in \mathbb{N}\}$$

4.1 Teste de Primalidade de Solovay-Strassen

Desenvolvido por Robert Martim Solovay e Volker Strassen em 1977, foi recomendado junto com o artigo do RSA [33]. É um teste muito simples com passos rápidos e bastante preciso. É probabilístico e serve para determinar se um número é composto ou primo. O processo envolve computar o símbolo de Jacobi [definição 4.0.3] (a, n), ou seja, tomar um número inteiro a com $1 < a$ e $a \neq n$, que consiste em operações modulares sucessivas a, $a == n$ até que $MDC(a, n)! = 1$, e também verificar se o Critério de Euler [Teorema 8] para n é obedecido ($a^{(n-1)/2} \mod n$).

Definição 4.1.1. (Solovay-Strassen) [13]
O teste de Solovay-Strassen [Algoritmo 1] é baseado no critério de Euler [Teorema 8]. Seja n um primo ímpar. Então $a^{\frac{(n-1)}{2}} \equiv (\mod n)$ para todos inteiros a que satisfaçam $mdc(\frac{n}{a}) = 1$. O que motiva as seguintes definições:

(I) Seja n um inteiro composto ímpar e seja a um inteiro, $1 \leq a \leq n - 1$.

(II) Se $MDC(a, n) > 1$ ou $a^{\frac{(n-1)}{2}} \equiv (\frac{n}{a})(\mod n)$, então a é chamada testemunha de Euler (número composto) para n. Caso contrário, isto é, se $MDC(a, n) = 1$ e $a^{\frac{(n-1)}{2}} \equiv (\frac{n}{a})(\mod n)$, então n é dito pseudoprimo de Euler para a base a. (Ou seja, n age como um primo na medida em que satisfaz o critério de Euler para a base particular a.) O inteiro a é chamado um número composto de Euler para n. Seja n um inteiro composto ímpar. Então, no máximo $\frac{\Phi(n)}{2}$ de todos os números a, $1 \leq a \leq n - 1$, são números compostos de Euler para n.

Algoritmo 1 Algoritmo Solovay-Strassen

1: **Entrada:** Um inteiro ímpar n e um número inteiro $a \in \mathbb{Z}_n$
2: **Saída:** "Primo" ou "Composto"
3: **if** $mdc(a, n) \neq 1$ **then**
4: **return** Composto
5: **end if**
6: **if J** $(a, n) \not\equiv a^{\frac{n-1}{2}} (\mod n)$ **then**
7: **return** Composto
8: **end if**
9: **return** Primo

4.2 Teste de Primalidade de Fermat

O algoritmo foi proposto por Pierre de Fermat no século XVII e propõe uma forma de provar que todo número na forma $2^{2^n} + 1$, com $n \geq 0$ é primo.

Teorema 13. (Pequeno Teorema de Fermat) [36, 13]
Dado um número primo p, então $\forall a \in \mathbb{Z}_p$, $p \nmid a$, $a^{p-1} \equiv 1 \mod p$. No entanto, este teste falha se p for um número de Charmichael [Definição 4.0.2] (falsos primos).

Teorema 14. (Teorema de Fermat I) [13]
Se p é primo então, para todo inteiro a, $a^p \equiv a \pmod{p}$.

Teorema 15. (Teorema de Fermat II) [13]
Seja p um número primo e a um inteiro que não é divisível por p. Então $a^{p-1} \equiv 1 (\mathrm{mod} p)$.

Algoritmo 2 Algoritmo Fermat

1: **Entrada:** Um número inteiro ímpar n
2: **Saída:** "Primo" ou "Composto"
3: Comece com $x = \sqrt{n}$, se $n = x^2$ então x é o fator de n; caso contrário vá para a etapa 2.
4: Incremente x de uma unidade e calcule $y = \sqrt{x^2 - n}$
5: Repita a etapa 2 até encontrar um valor inteiro para y, ou até que x seja igual a $\frac{(n+1)}{2}$; no primeiro caso n tem fatores $x + y$ e $x - y$, no segundo n é primo.

4.3 Teste de Primalidade de Miller-Rabin

Este teste foi desenvolvido por Gary Lee Miller e Michael Oser Rabin. Esse algoritmo testa se um número inteiro p satisfaz a propriedade $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$ para uma base a qualquer. Sua versão original, devido a G. L. Miller , é determinística , mas o determinismo se baseia na não provada hipótese de Riemann estendido [Definição 2.1.12]. Foi modificado por M. O. Rabin para obter um incondicional algoritmo probabilístico.

Definição 4.3.1. (Teste de Primalidade de Miller-Rabin) [13]
Dado um inteiro ímpar n, se $n = 2^r s + 1$ com s ímpar. Em seguida, escolha um inteiro aleatório a com $1 \leq a \leq n - 1$. Se $a^s \equiv 1 (\mathrm{mod}\ n)$ ou $a^{2^j s} \equiv -1 (\mathrm{mod}\ n)$ para algum $0 \leq j - 1$, então n passa no teste. Um primo passará no teste para todo a.

Algoritmo 3 Algoritmo Miller-Rabin

1: **Entrada:** Um número inteiro ímpar n e um número inteiro $a \in \mathbb{Z}_n$
2: **Saída:** "Primo" ou "Composto"
3: Escreva $n - 1 = 2^t \cdot q$, com q ímpar.
4: Calcule sucessivamente $a_0 = a^q \mod n$, $a_1 = a_0^2 \mod n$, ..., $a_K = a_{K-1}^2 \mod n$; até que $k = t$ ou $a_k \equiv 1 \mod n$.
5: Se $k = t$ e $a_k \not\equiv 1 \mod n$ retorne Composto.
6: Se $k = 0$, retorne Primo.
7: Se $a_{k-1} \not\equiv -1 \mod n$ retorne Composto.
8: Retorne Primo.

4.4 Teste de Primalidade de Lucas Para o Número de Jacobi

Édouard Lucas, um matemático francês, provou que os fatores primos de um número de Fermat [Teorema 13] são da forma $k2^n + 1$, e também é um algoritmo probabilístico. Em resumo, utiliza o símbolo de Jacobi [definição 4.0.3] e uma sequência de inteiros S para verificar a primalidade de um dado número n.

Definição 4.4.1. (Teste de Primalidade de Lucas) [13]
Seja n um inteiro ímpar e b um inteiro tal que $2 \leq b \leq n-1$. Se $b^{n-1} \equiv 1 (\text{mod } n)$ e $b^{\frac{n-1}{p}} \not\equiv 1 (\text{mod } n)$, para cada fator p de $n-1$, então n é primo.

Algoritmo 4 Algoritmo Lucas

Entrada: Um número inteiro ímpar n, $n > 1$
Saída: "Primo" ou "Composto"
$D \leftarrow n$
$P \leftarrow 1$
$Q \leftarrow \frac{(1-D)}{4}$
$U \leftarrow 0$
$U_2 \leftarrow 1$
$V \leftarrow 2$
$V_2 \leftarrow P$
$k \leftarrow \frac{(n+1)}{2}$
while $(k \neq 0)$ **do**
    $U_2 \leftarrow U_2 \times V_2$
    $V_2 \leftarrow V_2 \times V_2 - 2 \times Q$
    if $(k \ bmod \ 2 \neq 0)$ **then**
        $U_{AUX} \leftarrow U$
        $U \leftarrow U_2 \times V + U \times V_2$
        if $(U \ bmod \ 2 \neq 0)$ **then**
            $U \leftarrow U + n$
        end if
        $V \leftarrow V_2 \times V + U_2 + U_{AUX} \times D$
        if $(V \ bmod \ 2 \neq 0)$ **then**
            $V \leftarrow V + n$
        end if
        $U \leftarrow U \ / \ 2$
        $V \leftarrow \frac{V}{2}$
        $U \leftarrow U \mod n$
        $V \leftarrow v \mod n$
    end if
    $Q \leftarrow Q \times Q$
    $k \leftarrow \frac{k}{2}$
end while
if $(U \neq 0)$ **then**
    return Composto
end if
return Primo

4.5 Teste de Primalidade de Lucas-Lehmer para Número de Mersenne

Concebido originalmente por Édouard Lucas [13] em 1878 foi melhorado por Derrick Henry Lehmer em 1932. Basicamente é um algoritmo probabilístico que seleciona um número inteiro positivo de uma sequência de inteiros definido recursivamente por $S_0 = 4$ e $S_{k+1} = S_k^2 - 2$, se este número M(p) [Definição do Número de Mersenne 4.0.4] divide S_{p-2} então $M(p)$ é primo, caso contrário é composto.

Definição 4.5.1. (Teste de Primalidade de Lucas-Lehmer) [13]
Seja p um primo positivo. O número de Mersenne M(p) é primo se, e somente se, $S_{p-2} \equiv 0 (\text{mod } M(p))$.

Algoritmo 5 Algoritmo Lucas-Lehmer para Número de Mersenne

1: **Entrada:** Um número inteiro ímpar n, $n > 1$
2: **Saída:** "Primo" ou "Composto"
3: $a \leftarrow 4$
4: **for** i = 3 até $s = s2 - 2 \bmod 2P - 1$ **do**
5: **if** $s == 0$ **then**
6: **return** Primo
7: **end if**
8: **end for**
9: **return** Composto

4.6 Teste de Primalidade de Lucas-Lehmer-Riesel Para o Número de Riesel

O teste foi desenvolvido por Hans Riesel e baseia-se no teste de primalidade Lucas-Lehmer [Definição 4.5.1]. É um teste probabilístico, em resumo, utiliza os números de Riesel [definição 4.0.5] para verificar a primalidade de um dado número n.

Definição 4.6.1. (Teste de Primalidade de Lucas-Lehmer) [13]
Seja p um primo positivo e K um número inteiro ímpar. O número de Riesel R(p) é primo se, e somente se, $K \cdot S_{p-2} \equiv 0 (\text{mod } M(p))$.

Algoritmo 6 Algoritmo Lucas-Lehmer-Riesel para o Número de Riesel

1: **Entrada:** Dois números inteiros ímpar n e k, $n,\ k > 1$
2: **Saída:** "Primo" ou "Composto"
3: $a \leftarrow 4$
4: **for** i = 3 até $s = (K \cdot s2) - 2 \bmod 2P - 1$ **do**
5: **if** $s == 0$ **then**
6: **return** Primo
7: **end if**
8: **end for**
9: **return** Composto

4.7 Teste de Primalidade AKS

O algoritmo AKS foi desenvolvido em 2002 por Manindra Agrawal - Neeraj Kayal - Nitin Saxena. É um teste determinístico que executa somente em tempo polinomial. Está fundamentado no pequeno Teorema de Fermat [Teorema 13] e em uma das suas generalizações.

Definição 4.7.1. (Teste de Primalidade AKS) [30]
Para $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$, (a,n) = 1, n é primo se, e somente se:

$$(x + a)^n = x^n + a (\text{mod } n)$$

Outra forma:

$$(x+a)^p = \sum_{j=0}^{p} \binom{p}{j} x^{p-j} a^j$$

Teorema de Manindra Agrawal – Neeraj Kayal - Nitin Saxena: Para um dado inteiro $n \geq 2$, seja r um inteiro positivo menor que n, para o qual n tem ordem menor que $(log\ n)^2$ módulo r. Então n é primo se, e somente se:
• n não é uma potência perfeita de algum número inteiro;
• n não tem algum fator primo menor ou igual a r;
• $(x+a)^2 \equiv x^2 + a (\text{mod } n, x^r - 1)$ para cada inteiro a, $1 \leq a \leq \sqrt{r} log\ n$.

Algoritmo 7 Algoritmo AKS

Entrada: Um número inteiro ímpar n, $n > 1$
Saída: $Se\ N = a^b\ com\ b > 1,\ retorna\ Composto.$
Encontrar o menor r tal que $ord_r N > \frac{1}{2}(\log N)^2$.
Se mdc(a, N) > 1 para algum primo $a < r$, retorna Composto.
Se N < r, retorna Primo.
for a = 1 até $\lfloor \sqrt{\frac{\varphi(r)}{2}} / \log_2 N \rfloor$ **do**
    if $(x+a)^N x^N + a (\text{mod } x^r 1, N)$ **then**
        return Composto
    end if
end for
return Primo

4.8 Teste de Primalidade APR

O algoritmo APR foi desenvolvido por Leonard Adleman, Carl Pomerance e R. S. Rumely. É um teste determinístico, é resultado da melhoria de teste de Lucas-Lehmer [Definição 4.5.1].

Definição 4.8.1. (Teste de Primalidade APR) [6]
Pomerance dá uma descrição compreensível, do teste: Seja n número inteiro que se quer testar a primalidade, então são calculados os menores quadrados inteiros $f(n)$, de tal modo que $\prod_{(q-1)|f(n)} q > \sqrt{n}$ onde q é primo. Os fatores primos de $f(n)$ são ditos primos iniciais, e os primos q, com $q - 1|f(n)$, são chamados de primos Euclidianos. É então verificado se n é dividido por primos não iniciais ou Euclidianos. Se n é composto então ele tem um fator primo $r \leq \sqrt{n}$. A dificuldade está em encontrar r.

Se $p > 2$, existem $a, b \in \mathbb{Z}$ tal que $ab(a+b) \not\equiv 0 \text{ mod } p$ e

$$\hat{\theta}_{a,b} \stackrel{\text{def}}{=} \sum_{p-1}^{u=1} \not\equiv 0 \text{ mod } p$$

onde u^{-1} denota um inverso de u mod p.

Algoritmo 8 Algoritmo APR

Entrada: Um número inteiro ímpar n, $n > 2$

Saída: "Primo" ou "Composto"

A. Passo de Preparação

A.1 Calcule o número inteiro $f(n)$ livre de quadrado positivo

$$\prod_{q-1|f(n),qprimo} q > n^{\frac{1}{2}}.$$

Definir os primos iniciais para serem os fatores primos de $f(n)$. Define os primos euclidianos como sendo os primos q com $q-1 \mid f(n)$.

A.2 Testar qualquer divisão do primo inicial ou Euclidiana de n; Se não é igual a n, declare n composto e pare. Calcule a menor raiz primitiva positiva t_q para cada primo euclidiano q.

A.3 Para cada primo inicial $p > 2$, encontre os inteiros a, b com $0 < a$, $b < p$, $a + b \equiv 0 \bmod p$, e $\hat{\theta}_{a,b} = \sum_{u=1}^{p-1} \theta_{a,b}(u) \cdot u_{-1} \equiv 0 \bmod p$ como garantido pela Proposição 4.8. Para $p = 2$, coloque $a = b = \hat{\theta}_{a,b} = 1$.

A.4 Para cada primo p e cada primo euclidiano q com $p\, mid q-1$, fixa o primo ideal $\mathcal{Q}_q = (q, \zeta_p - t_q^{\frac{Q-1}{p}})$ Se sobre q em $\mathbf{Z}[\zeta_p]$, onde $\zeta_p = e^{\frac{2\pi i}{p}}$. Calcule a soma de Jacobi $J_p(q) \in \mathbf{Q}(\zeta_p)$: Se $p = 2$, coloque $J_p(q) = -q$, Se $p > 2$, coloque $J_p(q) = -J_{a,b}(\mathcal{Q}_q) = -\sum_{x=2} \mathcal{Q}_q)_p^{-a}(\frac{1-x}{\mathcal{Q}_q})$ Onde a, b são os inteiros (dependendo de p) calculados em A.3 acima.

B. Passo de Extração

B.1 Para cada primo p inicial, execute o processo MDC em $\mathbf{Q}(\zeta_p)$ em relação a n e o conjunto de $\sigma J_p(q)$ onde q varia sobre todos os primos euclidianos com $P \mid q-1$ e σ intervalos sobre $Gal(\mathbf{Q}(\zeta_p)/\mathbf{Q})$. Assim que declarar n composto ou construir um ideal $\mathcal{A}$ em $Z[\zeta_p]$, um inteiro $s \geq 1$, e inteiros $j(\sigma, q)$ com $1 \leq J(\sigma, q) \leq 0$, tal que (i) cada $(\sigma J_p(q))^{\frac{n^f-1}{p^s}} \equiv \zeta_p^{j(\sigma,q)} \bmod \mathcal{A}$, (ii) ou $p \nmid (n^f-1)/p^s$ ou algum $\zeta_p^{j(\sigma,q)} \neq 1$, Onde f denota a ordem de $n \bmod p$.

B.2 Para cada primo p inicial, faça o seguinte. Se algum $j(\sigma_0, q_0) \neq p$, deixe $\gamma = \sigma_0 J_p(q_0)$. Neste caso, construa inteiros $m(\sigma, q)$ para todo σ, q tal que $0 \leq m(\sigma, q) \leq p-1$ e

$$(\gamma^{(n^f-1)/p^s})^{m(\sigma,q)} \equiv ({}_p(q))^{(n^f-1)/p^s} \bmod \mathcal{A}$$

. Se todo $j(\sigma, q) = p$, defina todos os $m(\sigma, q) = 0$.

C. Passo de Consolidação

Para cada inteiro k, $1 \leq k \leq f(n)$, faça C.1 a C.4

C.1 Para cada $q > 2$ use o Teorema Chinês do Resto para calcular os inteiros $I(k, q)$ tal que $I(k, q) \equiv k\hat{\theta}_{a,b}^{-1} \sum_{j=1}^{p-1} j \cdot m(\sigma_j^{-1}, q) \bmod p$ para cada $p \mid q-1$. Também deixe $I(k, 2) = 1$.

C.2 Para cada q, calcule o menor inteiro positivo $r(k, q) \equiv t_q I(k, q) \bmod q$.

C.3 Use o Teorema Chinês do Resto para calcular o menor inteiro positivo $r(k, q)$ tal que $r(k, q) \equiv r(k, q) \bmod q$ para cada q.

C.4 Verifique se $r(k) \mid n$. Se ele faz e $r(k) \neq 1$, declare n composto e pare. Caso contrário, continue com o próximo valor de k.

C.5 Declare n primo.

4.9 Teste de Primalidade de Baillie-PSW

É um teste de primalidade determinístico construído na década de 1980 por Robert Baillie, Carl Pomerance, John Selfridge e Samuel Wagstaff. Utiliza dois outros testes de

primalidade [Lucas 4.4.1 e Miller-Rabin 4.3.1] para definir se um número inteiro ímpar é primo ou composto.

Definição 4.9.1. (Teste de Primalidade de Baillie-PSW) [29]
Este teste utiliza o teste de Miller-Rabin [Definição 4.3.1], com base em $a = 2$. Caso o número que está sendo testado passe neste teste, é realizado um teste de Lucas [Definição 4.4.1], que em resumo, utiliza o símbolo de Jacobi [Definição 4.0.3] e sequências de Lucas para verificar a primalidade de um dado número. Se o número n passar por este teste também, declara-se que n é primo.

Algoritmo 9 Algoritmo Baillie-PSW

1: **Entrada:** Um número inteiro ímpar n, $n > 2$
2: **Saída:** "Primo" ou "Composto"
3: **if** ((miller_Rabin(n,1) $\neq$ 1) **then**
4: **return** Composto
5: **end if**
6: **if** ((lucas(n) $\neq$ 1) **then**
7: **return** Composto
8: **end if**
9: **return** Primo

4.10 Teste de Primalidade de Pocklington

É um teste de primalidade determinístico idealizado por Henry Cabourn Pocklington e Derrick Henry Lehmer para decidir se um determinado número inteiro n é primo, que é determinado a partir de fatores primos de $n - 1$, operação que pode não ser tão fácil de se realizar.

Definição 4.10.1. (Teste de Primalidade de Pocklington) [27]
Sejam $N - 1 = q^n R$, q primo, $n \geq 1$ e $q \nmid R$. Se existe $a > 1$ tal que

$$a^{N-1} \equiv 1 (\text{mod } n)$$

$$MDC(a^{\frac{N-1}{q}} - 1, N) = 1.$$

Então os fatores primos de N são todos da forma $mq_n + 1$, $m \geq 1$.
Em outras palavras, temos que $a^t \equiv 1 \pmod{n}$ se e somente se, $ord_n\ a \mid t$. Em particular, para todo a com mdc(a,n) = 1 temos $ord_n\ a \mid \varphi(n)$.
Critério de Pocklington : Seja n um inteiro positivo. Se houver um primo p, $p > \sqrt{n} - q$, tal que $p|(n-1)$; e um inteiro a tal que $a^{n-1} \equiv 1 \bmod n$ e $mdc(a^{\frac{(n-1)}{p}}, n) = 1$. Então n é primo.

Algoritmo 10 Algoritmo Pocklington

Entrada: Um número inteiro ímpar n, $n > 2$
Saída: "Primo" ou "Composto" // p é primo
for p até $p|(n-1)$ **do**
  if $p|(n-1)$ e $q > \sqrt{n} - 1$. **then**
    return Primo
  end if
  if $mdc(a^{\frac{(n-1)}{p}} - 1, n) = 1$ *e* $a^{(n-1)} \equiv 1 (\text{mod } n)$ **then**
    return Primo
  end if
end for
return Composto

4.11 Teste de Primalidade de Lenstra-Pomerance

Desenvolvido por Hendrik Willem Lenstra e Carl Pomerance como uma melhoria do algoritmo AKS [7], e como o AKS ele também é um algoritmo determinístico.

Definição 4.11.1. (Teste de Primalidade de Lenstra-Pomerance) [13, 25]
Baseia-se na troca do polinômio $(x_r - 1)$ utilizado para as operações de módulo. Tendo esse polinômio $f(x)$, segue da forma:

$$(x + a)^n \not\equiv x^n + a (\text{mod } f(x),\ n)$$

Algoritmo 11 Algoritmo Lenstra-Pomerance

Entrada: Um número inteiro ímpar n, $n > 1$
Saída: Se $N = a^b$ com $b > 1$, retorna Composto.
Encontrar o menor r tal que $ord_r n > \frac{1}{2}(\log n)^2$.
Se mdc(a, n) > 1 para algum primo $a < r$, retorna Composto.
Se N < r, retorna Primo.
for a = 1 até $\lfloor \sqrt{\frac{\varphi(r)}{2}} / \log_2 n \rfloor$ **do**
  if $(x + a)^n \not\equiv x^n + a (\text{mod } f(x),\ n)$ **then**
    return Composto
  end if
end for
return Primo

Capítulo 5

Análise Comparativa e Resultados dos Testes de Primalidade

Neste capítulo apresentaremos a análise e os resultados obtidos da implementação dos testes de primalidade [Apresentados no capítulo 4]. O formato geral das análises dos dados apresentados pelos testes aferem a utilização do número de operações em módulo n, número de operações matemáticas realizadas, e os dados de entrada fornecidos ao algoritmo. Dentre todas as operações matemáticas, o custo maior está na fatoração e nas sucessivas operações modulares realizadas pelos algoritmos. A análise consiste na escolha aleatória do primo, formando um conjunto de testes para todos os algoritmos testados.

Os resultados foram obtidos utilizando um conjunto de dados relevantes para o processo de desenvolvimento das avaliações:

1. Para os algoritmos probabilísticos a execução dos testes foi com acurácia entre 50% (Teste de Solovay-Strassen [4.1.1]) e 70% (para os demais testes) sobre um mesmo número composto n.

2. Os arquivos de testes tem tamanhos diversos (8 B, 18 B, 144 B, 2246 B, 2900 B).

3. Diversos testes foram feitos para cada um deles, ou seja, geração aleatória de números e geração aleatória de números por intervalo fixos.

4. Um teste especial com um vetor de tamanho fixo, no qual o número primo está na última posição do vetor (utilizado na comparação de todos os testes com um valor fixo e único).

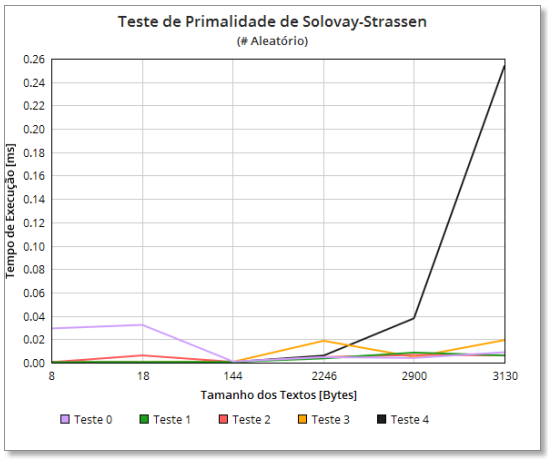
5. Mediu-se o tempo em segundos para o cálculo do desempenho. Note-se que o objetivo aqui é somente ter uma noção do desempenho, pois utilizou-se entradas muito pequenas quando comparadas as entradas utilizadas na prática.

6. Para a realização dos testes foi utilizado um computador com processador AMD FX 8350 de 4.00GHZ, 16GB de memória de RAM. O sistema operacional utilizado foi Ubuntu 16.04LTS, com programas escritos em linguagem C e o compilador gcc (Ubuntu 5.4.0-6ubuntu1 16.04.4). Toda a implementação dos algoritmos foi desenvolvida a partir das definições que foram apresentadas.

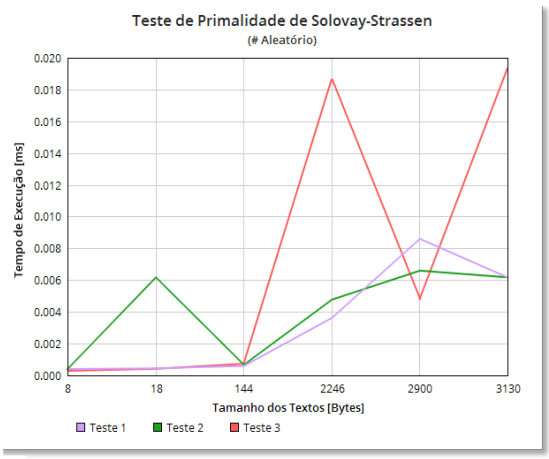
Teste de Solovay-Strassen: O algoritmo consiste em escolher um inteiro n aleatoriamente e testar se este atende ao critério de Euler [Definição 8]. Caso este teste falhe, temos uma prova de que n é composto. Caso contrário, o próximo passo é obter o cálculo do número

de Jacobi [Definição 4.0.3], de maneira intuitiva podemos definir a primalidade de n com uma probabilidade inversamente proporcional a 50% de erro.

Dentre os resultados obtidos durante a geração aleatória dos números [Figura 5.1], os melhores desempenhos dentre os testes realizados são os teste 1 e 2 [Figura 5.1(b)] com um tempo de execução quase linear e contínuo (varia muito pouco) para todos os tamanhos de textos. O pior desempenho (Teste 4) [Figura 5.1(a)] apresenta uma variação crescente, contínua e progressiva no tempo de execução. Note que apesar da entrada (número aleatório) variar, nem sempre a execução do algoritmo nos fornece primos rapidamente. Vale salientar que o tamanho da entrada é muitas vezes irrelevante (para estes testes, o tamanho do número variou entre 2 e 4 dígitos), pois um texto grande pode ser criptografado mais rápido (Teste 2) do que um texto pequeno e vice-versa (Teste 0). Tudo depende da escolha do tamanho do texto e não da escolha do número primo ou do número de operações realizadas para obtê-lo.



(a) Conjunto Completo de Testes



(b) Conjunto Reduzido de Testes

Figura 5.1: Testes de Primalidade de Solovay-Strassen - Aleatório

Dentre os resultados obtidos durante a geração aleatória dos números por intervalos fixos [Figura 5.2], o melhor desempenho dentre todos os testes realizados foi com o intervalo [10 - 100] e conseguiu-se um tempo de execução pequeno para todos os tamanhos de textos. O pior desempenho está no intervalo [10000 - 100000] que oscilou muito com o tamanho dos textos. Portanto o tamanho do número (número de dígitos) a ser fornecido ao algoritmo é de fundamental importância, quanto maior for o número a ser testado, mais lento o algoritmo fica.

Note que os resultados foram mantidos o mesmo padrão dos dados apresentados na [Figura 5.1] pode-se variar o tamanho do intervalo e se obter com isso uma oscilação do tempo de execução. Ou seja, aumentar o tamanho da chave faz o algoritmo variar mais rapidamente e torna o algoritmo mais moroso.

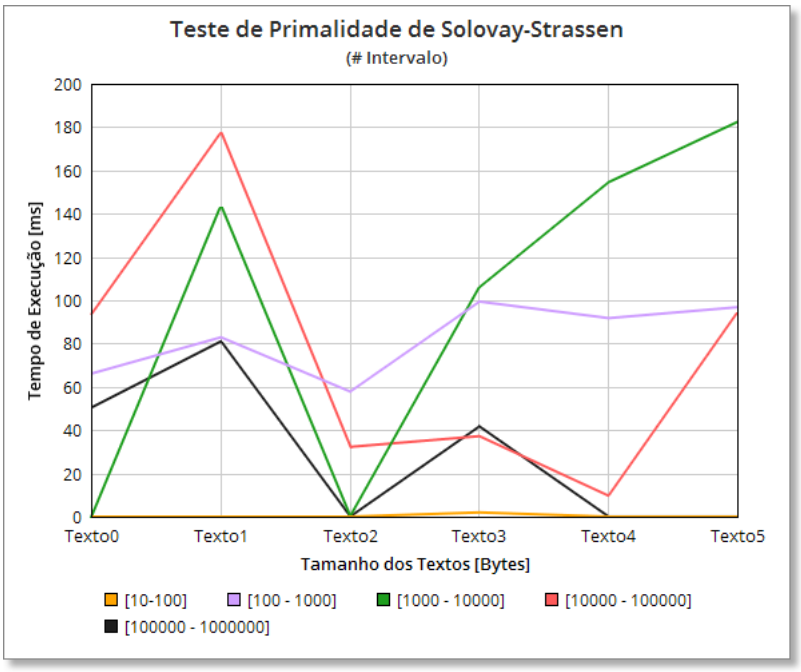


Figura 5.2: Teste de Primalidade de Solovay-Strassen - por Intervalos Fixos

Observando o tamanho da entrada pode-se calcular o custo do algoritmo simplesmente por $O(log_2^3\ n)$ [30] operações binárias. Enfim, o algoritmo tem custo polinomial, e em muitos casos sua execução terminará com a análise de cada número, devido à segunda condição que contempla $MDC(a, n) = 1$, oferecendo então uma execução realmente rápida.

Teste de Miller-Rabin: não usa o símbolo de Jacobi. É semelhante com o teste de Fermat [Definição 2] porque este também depende de um conjunto de desigualdades que são verdadeiros se o número que está sendo testado for primo. O teste é executado com bases distintas e informa se o valor de n fornecido é um pseudoprimo ou primo. A probabilidade de ser primo é de 70%. Selecionamos duas testemunhas para a análise modular, $s = n - 1$ e um valor aleatório a , em um laço testamos n através de uma condicional simples de congruência e cálculos modulares aferem o resultado.

O diferencial deste algoritmo está nas operações modulares, que em certas condições causa uma lentidão na sua execução (Teste 3 e 4) [Figura 5.3], a dificuldade de se analisar este algoritmo está na sua variabilidade de tempos.

Dados estabelecidos por intervalos [Figura 5.4] nos permite verificar que para números com mais de 6 dígitos e menores que 3 o algoritmo é lento [Figura 5.4(a)] e de custo alto. Para intervalos entre 3 e 5 dígitos [Figura 5.4(b)] o algoritmo tem comportamento assimétrico e variado.

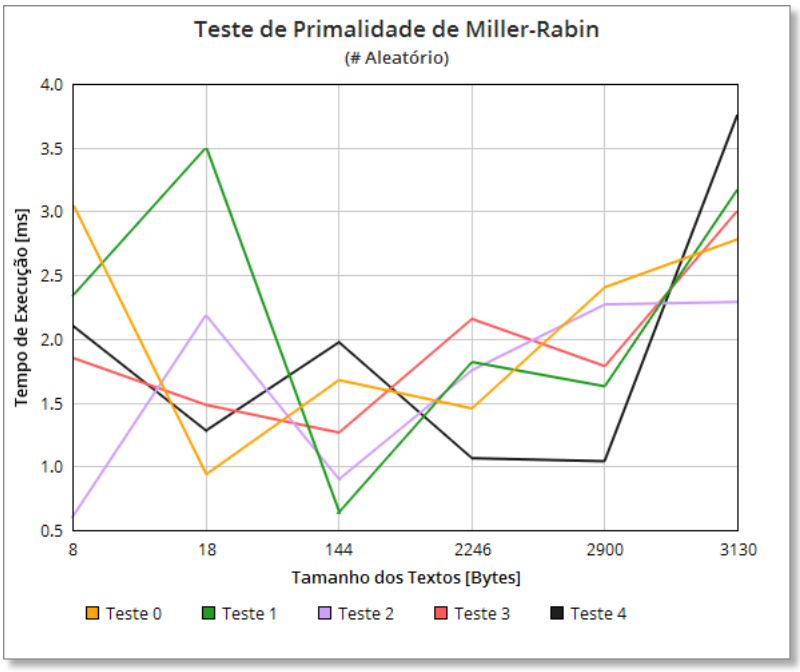
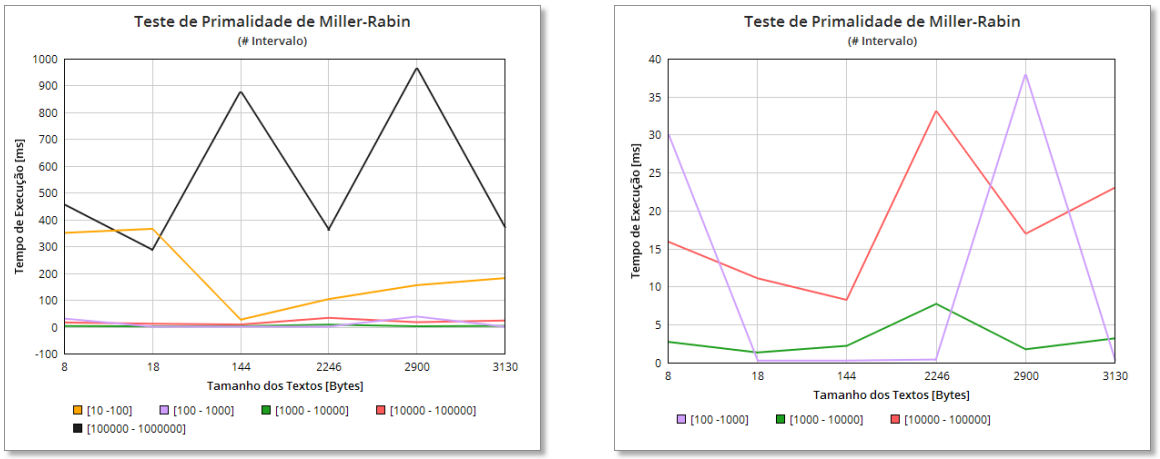


Figura 5.3: Teste de Primalidade de Miller-Rabin - Aleatórios



(a) Conjunto Completo de Testes

(b) Conjunto Reduzido de Testes

Figura 5.4: Teste de Primalidade de Miller-Rabin - por Intervalos Fixos

Teste de Fermat: usa o pequeno teorema de Fermat [Teorema 13]. O teste realiza diversos cálculo modulares e comparações de MDC em uma base numérica e testa uma sequência de números (ou testemunhas de Fermat, gerados também aleatoriamente); obtendo como saída um pseudoprimo ou primo. A probabilidade de sucesso do algoritmo é de 70%.

O algoritmo [Figura 5.5] apresenta dados relevantes para uma análise expressiva, pois cada teste realizado oscila em tempo diferente. Ao testar n (número aleatório de entrada) é possível verificar que o teste 1 é mais rápido exibindo comportamento quase contínuo. O mais lento (Teste 4) revela uma grande dependência entre a entrada e o número de testemunhas testadas pelo algoritmo.

Para números restritos ao intervalo [Figura 5.6] o algoritmo de Fermat é extremamente lento, com maior custo dos métodos clássicos (700 segundos para criptografar um texto

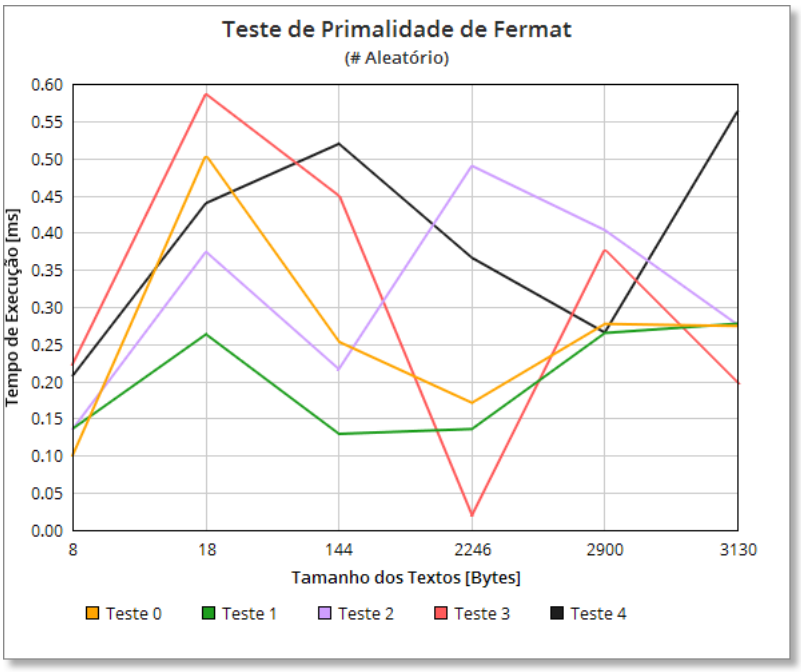


Figura 5.5: Teste de Primalidade de Fermat - Aleatórios

com 3130B). Para números grandes acima de 3 dígitos se torna inviável a execução do algoritmo.

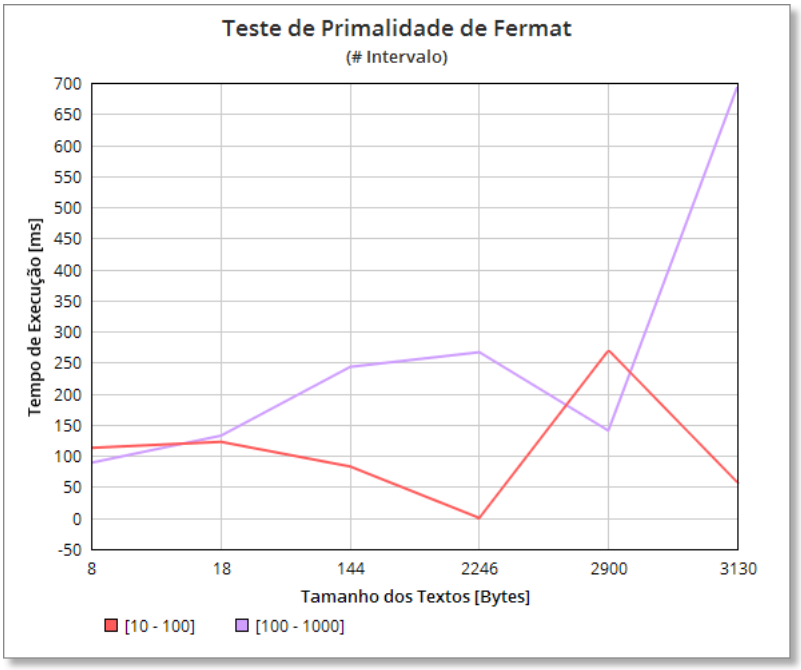


Figura 5.6: Teste de Primalidade de Fermat - por Intervalos Fixos

Teste de Lucas: A implementação usa o número de Jacobi [Definição 4.0.3]. Em um laço testa se cada número é primo ou composto. Este algoritmo requer que fatores primos de $(N-1)$ seja conhecido e testado (sucessivas divisões e multiplicações modulares, e um comparativo com o MDC do número são realizadas).

Os testes 1,2 e 3 [Figura 5.7] propiciam uma análise estática com regularidade crescente e contínua (ou seja, se aumento o tamanho dos textos o custo cresce proporcionalmente). Exceto os testes 0 e 4, que conforme os textos variam de tamanho o custo varia de forma irregular e inconstante (ou seja, oscilam concomitantemente).

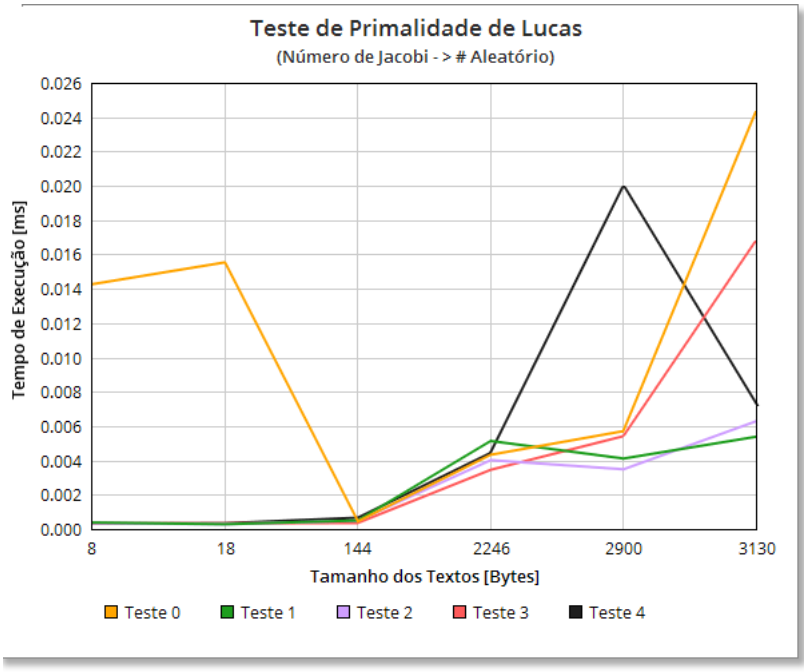


Figura 5.7: Teste de Primalidade de Lucas - Aleatórios

Para os dados fornecidos por intervalos fixos [Figura 5.8], apenas o intervalo com 2 dígitos tem comportamento constante e linear. Se o valor do número de entrada varia fica difícil determinar a sua primalidade em tempo constante.

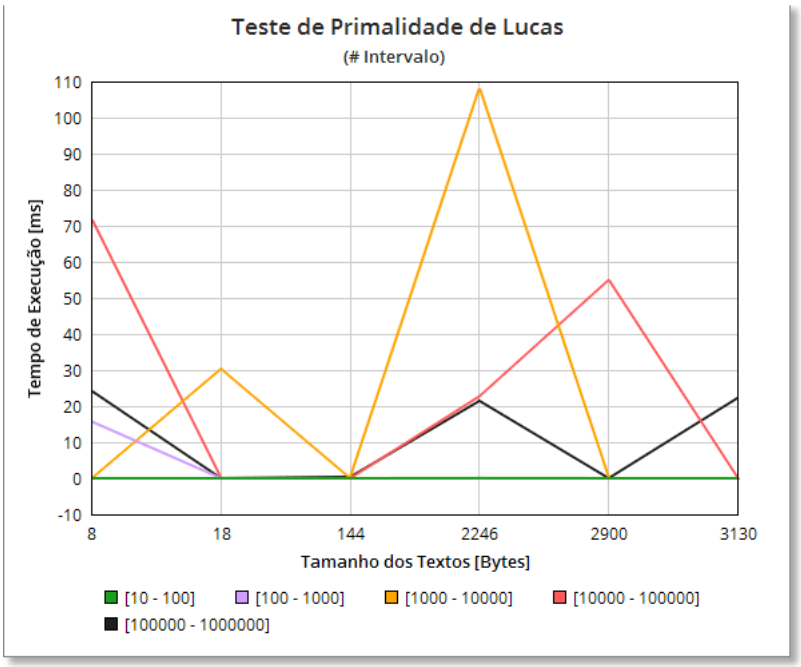


Figura 5.8: Teste de Primalidade de Lucas - por Intervalos Fixos

Teste de Lucas-Lehmer: usa o número de Mersenne [Definição 4.0.4]. A função testa se o número n fornecido como entrada é primo, e junto com o algoritmo de Lucas[definição 4.4.1] (basicamente cálculos modulares, divisões e deslocamento de bits) gera o número de Mersenne.
Tem custo de processamento igual a $(p \log p \, 2^{O(\log^* p)})$ para cálculo de 2 números primos por operações binárias.

É necessário ressaltar que diferentemente do algoritmo de primalidade de Lucas, este algoritmo é muito inconstante, visto que além dele testar o número aleatório este deve pertencer a sequência de números de Mersenne. Este fato aliado ao tamanho da entrada inviabiliza os testes feitos por intervalos fixos, visto que é bem difícil achar uma variedade grande de números de Mersenne nos intervalos {10-100, 100-1000, 1000-10000, 10000-100000, 100000-1000000}, por este fato apresentamos apenas os testes por intervalo aleatório [Figura 5.9].

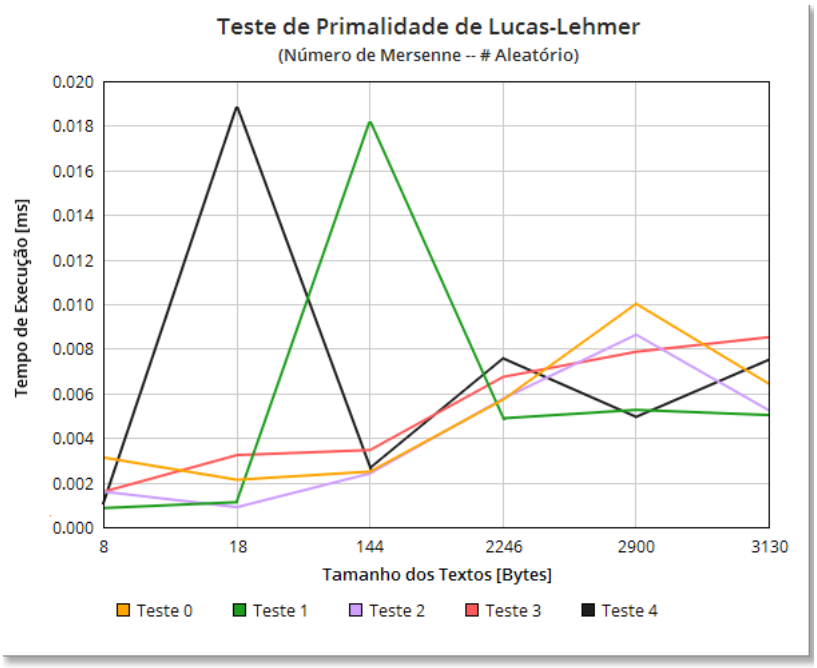


Figura 5.9: Teste de Primalidade de Lucas-Lehmer

Teste de Lucas-Lehmer-Riesel: usa o número de Riesel [Definição 4.0.5]. A função testa se o número $k \cdot n$ (sendo k um número ímpar) fornecido como entrada é primo, e junto com o algoritmo de Lucas[definição 4.4.1] (basicamente cálculos modulares, divisões e deslocamento de bits) gera o número de Riesel.

Assim como o algoritmo de Lucas-Lehmer e por definição este teste envolve ainda o problema de gerar a sequência de números de Riesel (ou seja, "técnica de revestimento em conjunto": um conjunto de pequenos números primos tais que cada extremidade de uma sucessão é divisível por um deles), o que torna inviável e moroso os testes por intervalos {10-100, 100-1000, 1000-10000, 10000-100000, 100000-1000000}. Portanto apresentamos apenas os testes de números gerados aleatoriamente [Figura 5.10].

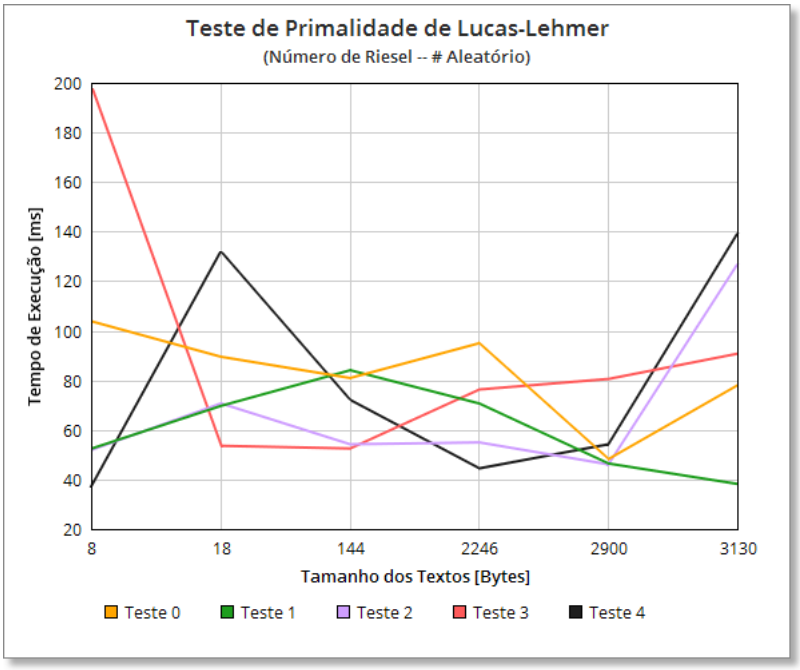


Figura 5.10: Teste de Primalidade de Lucas-Lehmer-Riesel

Uma das peculiaridades deste teste é o cálculo que gera a sequência numérica e a operação matemática subsequente, quando o número selecionado é pequeno temos um custo pequeno, descrescente (Teste 1) e pouco variável. Quando esse número é grande (Teste 3 e 4) existe uma variação irregular [198s, 37s respectivamente] para o mesmo tamanho de texto. Portanto a técnica matemática utilizada facilmente evidência o custo final deste algoritmo.

Teste de Primalidade AKS: Entender este teste requer que você saiba como o Triângulo de Pascal (binômios) funciona em relação a "Escolha"da operação. A ideia básica é você tomar a seguinte equação: $(x-1)^p - (x^p - 1)$ (usando p como o número que você deseja testar), se esse número puder dividir todos os coeficientes resultantes uniformemente (significa que ele não tem resto), então esse número é primo. Como um exemplo básico disso, podemos usar o número 3. Se $(x-1)^3 - (x^3 - 1)$ permite expandir isso para: $x^3 - 3x^2 + 3x - 1 - x^3 + 1$; aqui você nota que o primeiro termo será sempre igual a x^p e sempre irá cancelar, assim como os outros. Neste caso, isso nos deixará com: $3x^2 + 3x$. Novamente, para a AKS, só nos interessa testar os coeficientes no nosso caso este é [3, 3]. No entanto, queremos eliminar a redundância ao notar que o triângulo de Pascal repete-se em cada linha. Portanto, só precisamos testar metade da linha. No nosso caso, apenas um

3. Sendo que 3 divide 3, a saída é a resposta que o número selecionado é primo. Custo estimado do tempo de Processamento [30]: $O(log_2^{19}\ (n))$ operações binárias.

O comportamento dos dados estão distribuídos de forma razoavelmente regular em termos da entrada, o número de processamento da função promovem uma distribuição com pouca variação.

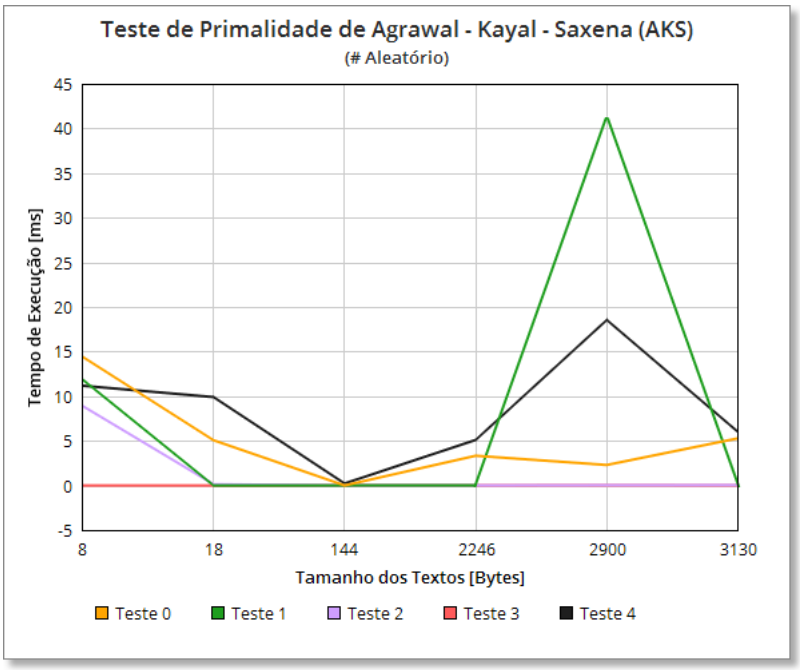


Figura 5.11: Teste de Primalidade de Agrawal-Kayana-Saxena (AKS)

Para números restritos a intervalo [Figura 5.12] o algoritmo AKS é extremamente rápido, com menor custo dos métodos modernos (teste de primalidade determinísticos). Para números grandes acima de 5 dígitos a execução do algoritmo torna viável a proteção das informações a qual ele deverá manter.

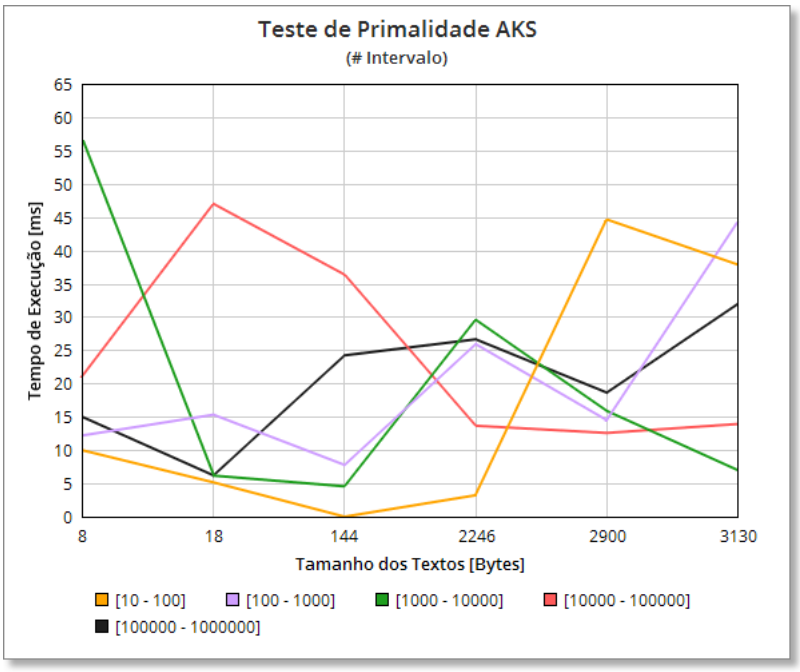


Figura 5.12: Teste de Primalidade de Agrawal-Kayana-Saxena (AKS)- por Intervalo Fixo

Teste de Primalidade Baillie-PSW: Custo de tempo de Processamento : $O(log_2^3(n))$ operações binárias.

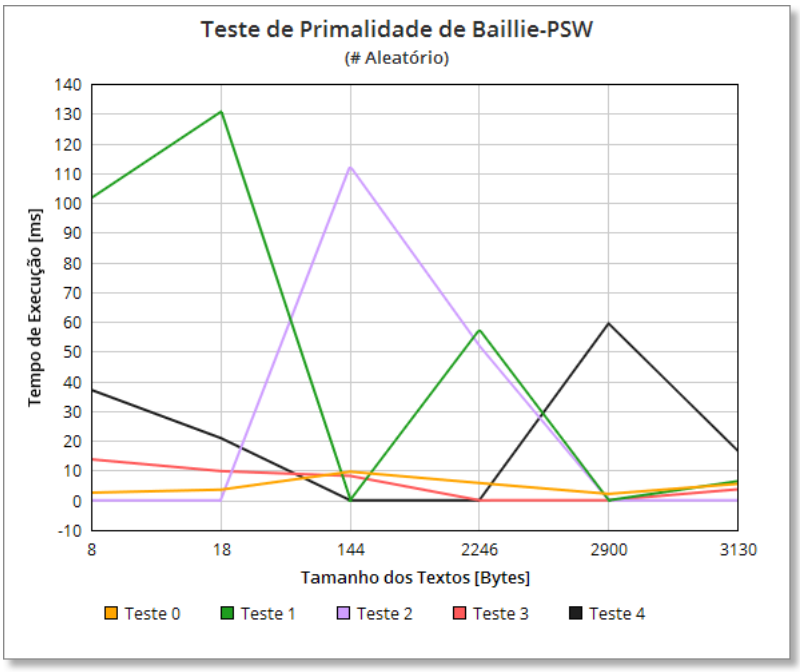


Figura 5.13: Teste de Primalidade de Baillie-PSW - Aleatório

Para números restritos ao intervalo [Figura 5.14] o algoritmo de Fermat é extremamente lento, com maior custo dos métodos clássicos. Para números grandes acima de 3 dígitos se torna inviável a execução do algoritmo.

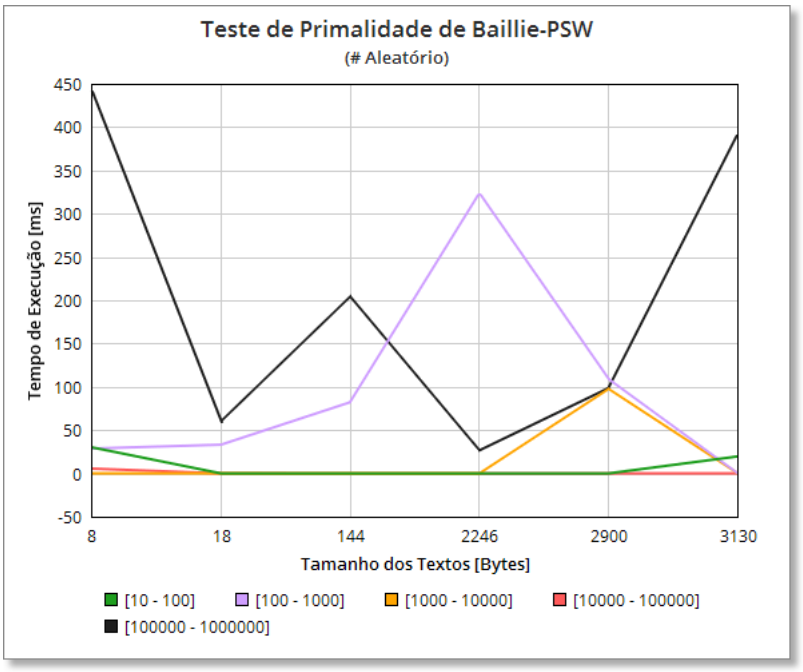


Figura 5.14: Teste de Primalidade de Baillie-PSW - por Intervalos Fixos

Teste de Pocklington: O algoritmo foi implementado para testar um inteiro n como entrada, utiliza um fator primo q testado através de cálculos modulares $q|n$; pela proposição, todo fator primo de n deve ser côngruo a 1 módulo q. Como isto vale para qualquer fator primo, de n deve ser côngruo a 1 módulo F. Como $F > \sqrt{n}$, isto implica que n é primo. É utilizado um outro primo para testemunhar através de fatorações, se o número de entrada n é primo ou composto.

Diferentemente de outros testes modernos de primalidade, este teste é muito semelhante ao custo dos algoritmos determinísticos em geral. Se não houver restrição do tamanho do número de entrada [Figura 5.15], o comportamento do algoritmo é estável e tem seu custo progressivo se aumentarmos o número de dígitos da entrada n.

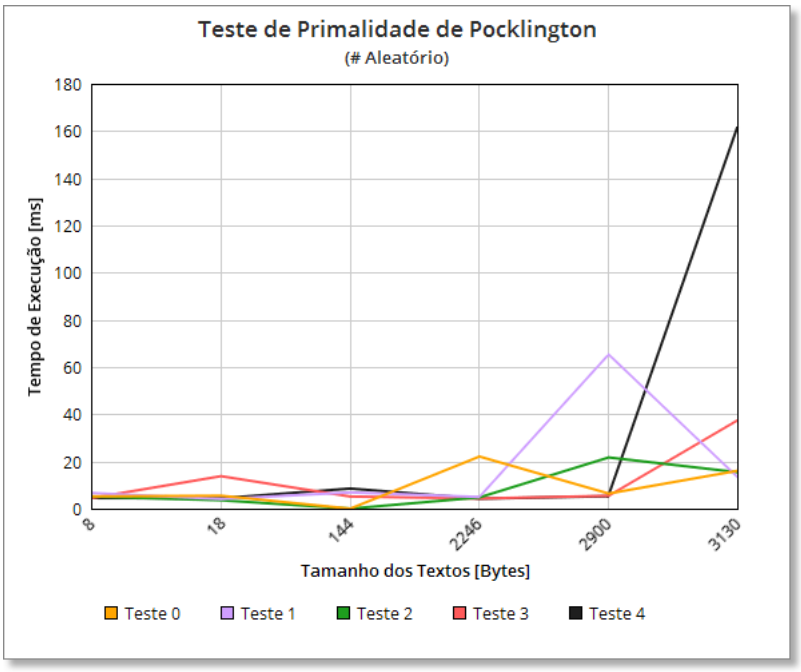


Figura 5.15: Teste de Primalidade de Pocklington - Aleatório

Assim como o algoritmo de Fermat ao se aplicar testes em intervalos fixos [Figura 5.16] o algoritmo é extremamente moroso, com maior custo dos métodos de primalidade modernos. Para números grandes acima de 4 dígitos se torna inviável a execução do algoritmo, haja visto que cada vez que aumentamos o intervalo seu custo aumenta em igual proporção.

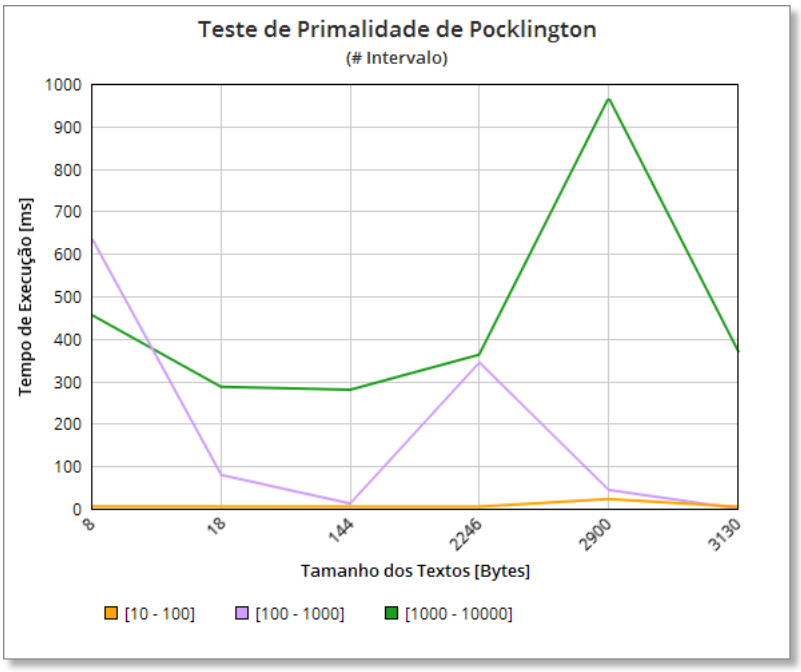


Figura 5.16: Teste de Primalidade de Pocklington - por Intervalos Fixos

Teste de Lenstra-Pomerance: por ser uma melhoria do teste de primalidade do AKS, este também utiliza um polinômio (para esta implementação um polinômio de segundo grau). Em um laço usando o polinômio testamos o coeficiente do número em questão, se for divisível por n, ele retorna composto; caso contrário é primo. Observação importante: o polinômio a ser utilizado pode ser qualquer um, deste que mantenha a condição fundamental do algoritmo trocar o polinômio $(x^r - 1)$ a ser utilizado.
Custo de tempo de Processamento : $O(log_2^6(n))$ operações binárias.

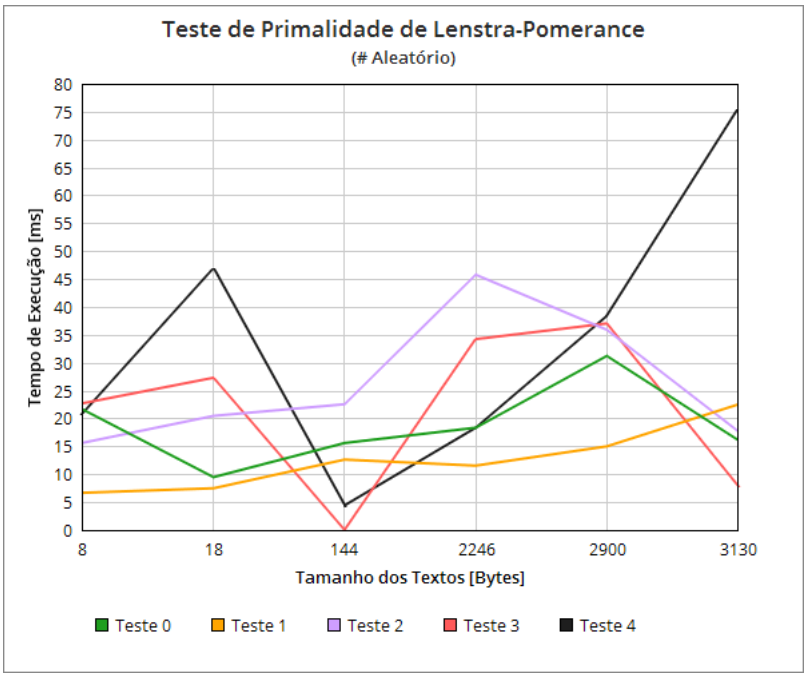


Figura 5.17: Teste de Primalidade de Lenstra-Pomerance - Aleatório

Analisando o comportamento sob intervalos fixos [Figura 5.18] o algoritmo é extremamente disperso (com 3 e 6 dígitos), com custo alto. Para números pequenos com 2 dígitos é um algoritmo com custo médio, tornando-o inviável e de fácil fatoração.

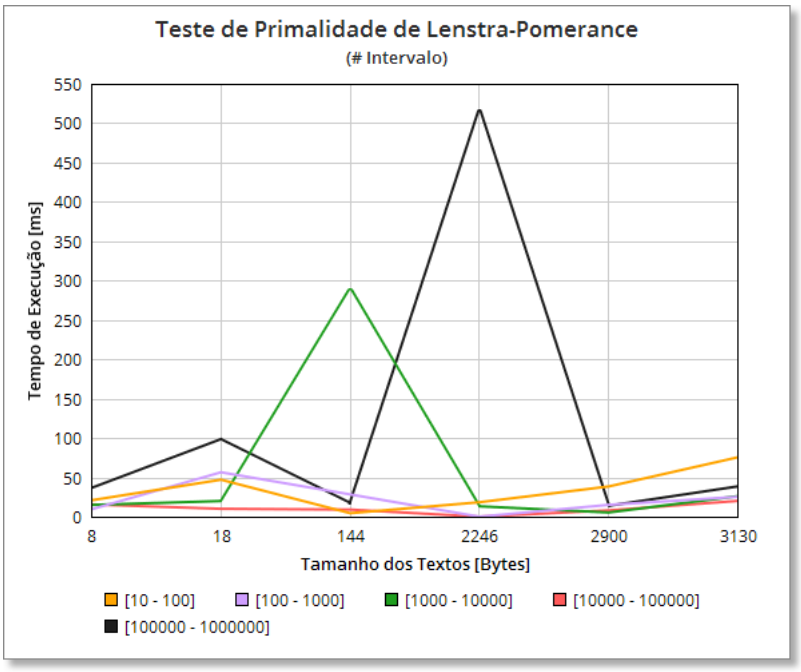


Figura 5.18: Teste de Primalidade de Lenstra-Pomerance - por Intervalos Fixos

5.1 Comparativo de Todos os Teste de Primalidade

Para estes testes foram utilizados dois vetores de números fixos

$$V_p = \{4, 38, 120, 45, 66, 28, 77, 354, 765, 599\}$$

e

$$V_q = \{4, 38, 120, 45, 66, 28, 77, 354, 765, 857\}$$

, ou seja, um para escolha do valor de p e outro para a escolha do valor de q (lembrando que estes são os dois números de entrada do algoritmo de criptografia RSA). Ao analisarmos os algoritmos de primalidade [Figura 5.19] é possível notar os tempos de execução bem variados para cada tamanho de texto. Os métodos clássicos de primalidade tendem a ser mais lentos (Fermat) em alguns casos e rápidos (SolovayStrassen) em outros. Já os métodos modernos de primalidade são rápidos na maioria das vezes (AKS), independente do tamanho da entrada.

Para um análise diversificada [Figura 5.20], utilizou-se uma outra máquina para realização dos testes, um computador do departamento de informática da UFPR, com processador Intel(R) Xeon(R) CPU E5-2690 v2 3.00GHz. O sistema operacional utilizado foi o Linux Mint 17.3 Rosa (GNU/Linux 4.8.8terms x86_64), com programas escritos em linguagem C e o compilador gcc (Ubuntu 5.4.1-2ubuntu1 16.04). Toda a implementação dos algoritmos foi desenvolvida a partir das definições que foram apresentadas.

Nesta análise há diferenças discrepantes na execução das máquinas apresentadas, exceto em alguns casos em que o teste de primalidade de Baillie-PSW e Lenstra-Pomerance são melhores que o teste de primalidade AKS. Os algoritmos mostram-se, em geral, satisfatoriamente rápidos. Já os testes probabilísticos apresentam comportamento oposto, conforme aumento do tamanho das mensagens, o custo aumenta proporcionalmente. O melhor é o Teste de Lucas que mantém seu custo progressivo e constante, variando muito pouco conforme o tamanho dos arquivos. O teste de Solovay-Strassen é o que mais oscila durante sua execução (pior desempenho com arquivos de 8B e 3130B). Miller-Rabin e Fermat são os piores dentre os clássicos (um ponto

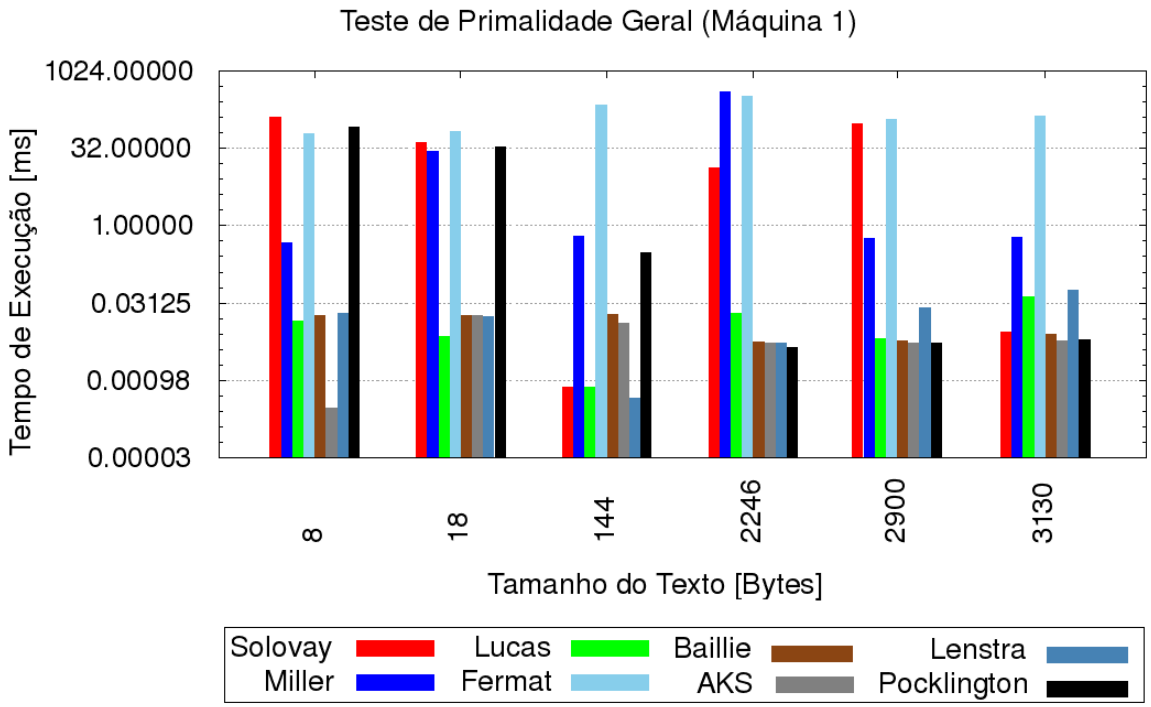


Figura 5.19: Teste de Primalidade - por Vetor

similar entre as duas máquinas testadas), estes são extremamente moroso e possuem custo alto independente do tamanho do texto ou do tamanho da entrada (que neste caso especificamente é fixo). Em outras palavras em um processador mais rápido os métodos de primalidade clássicos sempre tem custo maior que os métodos modernos de primalidade.

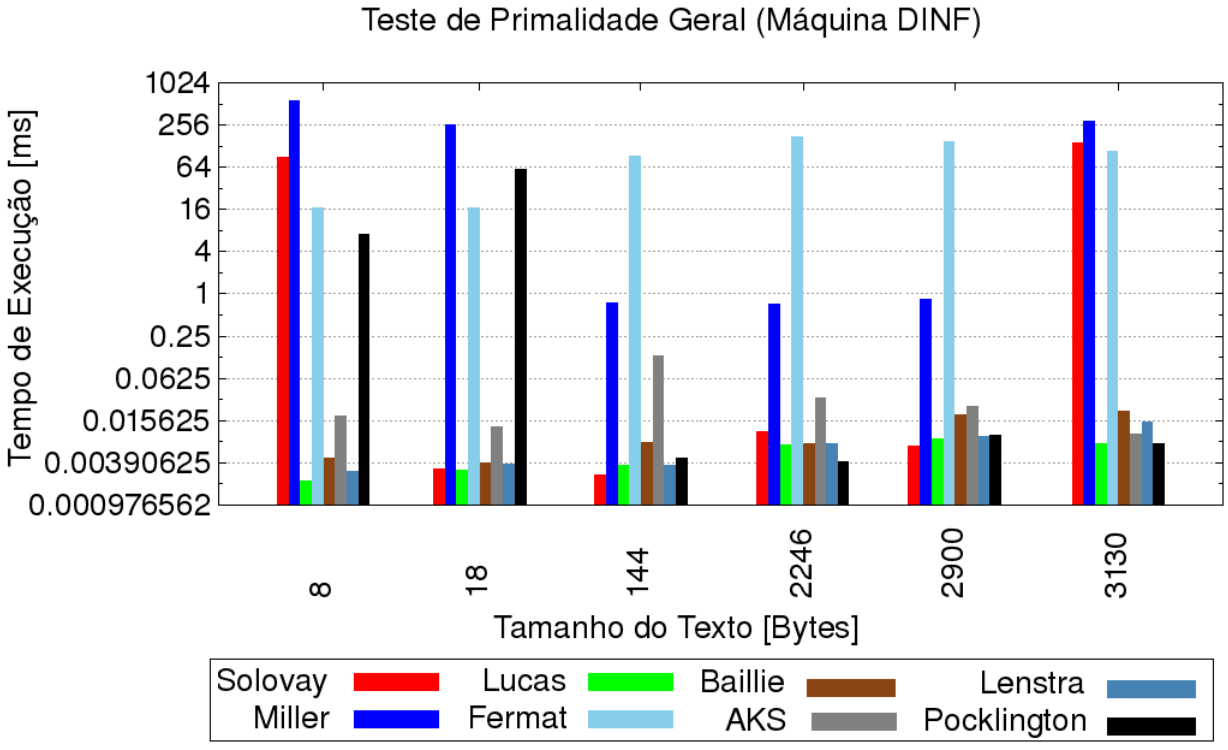


Figura 5.20: Teste de Primalidade - por Vetor

Capítulo 6

Conclusão

O processo de escolha dos testes de primalidade é bastante complexo, visto que, cada algoritmo avaliado demonstrou uma riqueza de detalhes. Durante a implementação dos algoritmos de primalidade utilizou-se um número gerado aleatoriamente a princípio com 2 dígitos até o máximo de 6 dígitos, o que exige cuidados especiais para evitar a explosão da capacidade de memória das máquinas. Como pode ser observado, não existe uma diferença perceptível no tempo de execução quando testamos primeiro os testes de primalidade clássicos (Testes Probabilísticos).

Exceto os testes de primalidade de Fermat e Miller-Rabin, o resto dos algoritmos probabilísticos são muito eficientes, sendo capazes de determinar a primalidade de um número primo em menos de um segundo. Para Fermat e Miller-Rabin é justificado por ter eficiência reduzida (para um número de 3 dígitos utiliza mais ou menos 350 segundos para determinar sua probabilidade) porque os testes requerem que fatores primos de $(N-1)$ sejam conhecidos (fazer uma fatoração antes de verificar a primalidade dos melhores resultados).

Para os testes de primalidade modernos (teste de primalidade determinístico), a principal desvantagem foi o aspecto do tempo de execução variar conforme o tamanho da entrada, interferindo na escolha do teste a ser utilizado. O algoritmo AKS foi o mais rápido dentre os métodos modernos determinando um número primo com mais agilidade em menos de um segundo, enquanto o Pocklington tem custo elevado dentre os algoritmos determinísticos. As técnicas matemáticas utilizadas podem variar, como por exemplo do AKS um polinômio, ou uma testemunha no caso do Miller-Rabin, essas variam conforme à necessidade da aplicação, seja utilizando os métodos clássicos (teste de primalidade probabilístico) ou modernos (teste de primalidade determinístico).

Uma questão importante observada, é a representação da informação e conhecimento da estrutura dos algoritmos de primalidade. Encontrar provas mais simples dos resultados e estudar problemas podem ser examinados visualizados neste contexto em maior profundidade levando em consideração o tamanho da entrada, o tamanho do texto, cálculos e funções matemáticos envolvidos. Estas informações moldam a diversidade dos resultados que são um diferencial, e realmente transforma a análise do tempo de execução dos algoritmos tornando-os muito interessante.

Na prática, é fácil decidir qual método de prova de primalidade usar:

- Para encontrar qualquer número grande o suficiente para fazer a lista dos maiores números primos conhecidos, deve ser usado uma versão do teste de primalidade de Solovay-Strassen, pois é mais rápido e simples de ser implementado.

- Para apontar para a segurança de dados e ou informações, deve ser usado os teste de primalidade determinístico como AKS, por ter um bom tempo de execução ou procurar

um teste mais apropriado a sua aplicação. Neste ponto algoritmos mais complexos como o AKS são mais úteis pois conferem maior precisão e proteção aos dados, inviabilizando ataques por fatoração (Recordando: se o número tiver 1024 Bytes, não existe uma forma de quebrar a criptografia, pelo menos não conhecida).

- Se em vez de olhar para n, você fornecer n, cheque primeiro os pequenos fatores. Se não tiver nenhum, tente um teste clássico para comprovar que é um provável primo. Se assim for, tente brevemente o fator $n + 1$, $n - 1$, ..., caso contrário em seguida checar um método moderno (teste de primalidade determinístico).

Este trabalho apresenta uma reflexão sobre alguns dos principais aspectos dos testes de primalidade, e como mencionado acima recomenda-se avaliar cautelosamente o tipo de projeto ou aplicação ao qual se destina trabalhar, antes de se utilizar qualquer uma da técnicas apresentadas. Aconselha-se que sempre comece com uma pequena avaliação utilizando os métodos clássicos e somente a partir da análise dos resultados tente implementar os métodos modernos.

6.1 Trabalhos Futuros

Durante o trabalho uma descoberta interessante foi o fato de não haver uma forma específica para aplicações do algoritmo RSA [12, 18] e os testes de primalidade, inclusive é uma sugestão de continuidade. Outras possíveis extensões a pesquisa é atingir mais aspectos e ferramentas, incluindo outras plataformas de desenvolvimento. Trazendo aspectos como desempenho, melhorias, cobertura de testes e análise da arquitetura das mesmas. Diante dos avanços tecnológicos na atualidade, o desenvolvimento de criação de chaves criptográficas forense, é um dos pontos para uma nova abordagem e foco de novos estudos trazendo sinais de continuidade. Outros pontos merecem uma atenção maior:

- Fazer um novo algoritmo de criptografia RSA, utilizando uma outra linguagem de programação que possa representar melhor números primos extremamente grandes. Existem várias grandes bibliotecas gratuitas para aritmética com inteiros grandes (Exemplo: “GNU Multiple-Precision Library” ou “GMPL”), bem como para provar a primalidade de inteiros grandes.

- Um estudo preliminar sobre: o comportamento de outros tipos de testes de primalidade ainda não estudados, e estudar o algoritmo de curvas elípticas que podem mudar o âmbito da segurança das informações na Internet.

- Sintetizar novos estudos do comportamento do RSA, envolvendo transmissão de informação através da rede (ou seja, conexão máquina a máquina e transmissão segura de informações).

- Produzir testes de ataques e defesas do RSA.

Referências Bibliográficas

[1] Security inc. laboratórios rsa. Disponível em : http://www.rsasecurity.com/rsalabs/, Acessado em 12/03/2016.

[2] Rsa - wikipedia, 2010. Disponível em : http://pt.wikipedia.org/wiki/rsa, Acessado em agosto de 2013.

[3] Algoritmo de euclides estendido, 2013. Disponível em : http://pt.wikipedia.org/wiki/Algoritmo_de_Euclides_estendido, Acessado em agosto de 2013.

[4] Teorema fundamental da aritmética, 2013. Disponível em :http://pt.wikipedia.org/wiki/Teorema_Fundamental_da_Aritm%C3%A9tica, Acessado em agosto de 2013.

[5] Uma revisão comentada das abordagens do problema quadrático de alocação, 2013. Disponível em : www.scielo.br/scielo.php?pid=50101-74382004000100005&script=sci_artext, Acessado em agosto de 2013.

[6] Leonard M Adleman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 387–406. IEEE, 1980. Disponível em : http://alpha.math.uga.edu/~rr/APR.pdf, Acessado em 23/09/2016.

[7] Richard A. Gonçalves Alessandro Dias Batista. Estudo e implementacão de criptografia avancada utilizando programaÇÃo paralela.

[8] Richard A. Gonçalves Alessandro Dias Batista. Estudo e implementacão de criptografia utilizando programacão paralela.

[9] J-C Bajard and Laurent Imbert. A full rns implementation of rsa. *IEEE Transactions on Computers*, 53(6):769–774, 2004. Disponível em : https://hal.archives-ouvertes.fr/lirmm-00090366/document , Acessado em 23/09/2016.

[10] Daniel J Bernstein. Circuits for integer factorization: a proposal. *At the time of writing available electronically at http://cr. yp. to/papers/nfscircuit. pdf*, 2001.

[11] Dan Boneh, Glenn Durfee, and Yair Frankel. Exposing an rsa private key given a small fraction of the private key bits. 1514:25–34, 1998. Disponível em : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.1461&rep=rep1&$type$=pdf, Acessado em 12/02/2016.

[12] Steve Burnett and Stephen Paine. *The RSA Security's Official Guide to Cryptography*. RSA press - McGraw Hill, 2001. Disponível em : http://www.it-ebooks.info,Acessado em agosto de 2013.

[13] Severino Collier Coutinho. *Números inteiros e criptografia RSA*. IMPA - Instituto Nacional de Matemática Aplicada, Rio de Janeiro, 2 edition, 2013. Série de Computação e Matemática.

[14] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006. Disponível em : https://books.google.com.br/books?hl=pt-BR&lr=&id=ZXjHKPS1LEAC&oi=fnd&pg=PR7&dq=Prime+Numbers:+A+Computational+Perspective+.&ots=m9XufBP5Yi&sig=UW4o9XCCWlye2ik0kGFQ6ACi2oE#v=onepage&q=Prime%20Numbers%3A%20A%20Computational%20Perspective%20.&f=false, Acessado em setembro de 2016.

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

[16] G Doroeviae, T Unkasevia, and M Markoviae. Optimization of modular reduction procedure in rsa algorithm implementation on assembler of tms320c54x signal processors. In *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*, volume 2, pages 811–814. IEEE, 2002. Disponível em : http://ieeexplore.ieee.org/document/1028214/, Acessado em 23/09/2016.

[17] Wenjun Fan, Xudong Chen, and Xuefeng Li. Parallelization of rsa algorithm based on compute unified device architecture. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 174–178. IEEE, 2010. Disponível em : http://ieeexplore.ieee.org/document/5662508/, Acessado em 23/09/2016.

[18] Bruce Ferguson, Niels e Schneier. *Practical Cryptography*. Wiley publishing, Inc, 1 edition, 2003.

[19] Michael Filaseta, Carrie Finch, and Mark Kozek. On powers associated with sierpiński numbers, riesel numbers and polignac's conjecture. *Journal of Number Theory*, 128(7):1916–1940, 2008. Disponível em : http://ac.els-cdn.com/S0022314X08000462/1-s2.0-S0022314X08000462-main.pdf?_tid=c0876360-aec6-11e6-acb4-00000aab0f26&acdnat=1479608113_328bfc2a97fac06505181adc8e6b51a5, Acessado em 23/09/2016.

[20] Apostolos P Fournaris and O Koufopavlou. A new rsa encryption architecture and hardware implementation based on optimized montgomery multiplication. In *2005 IEEE International Symposium on Circuits and Systems*, pages 4645–4648. IEEE, 2005. Disponível em : http://ieeexplore.ieee.org/document/1465668/ , Acessado em 23/09/2016.

[21] Willi Geiselmann and Rainer Steinwandt. Yet another sieving device. In *Cryptographers' Track at the RSA Conference*, pages 278–291. Springer, 2004.

[22] Gabriel Vasile Iana, Petre Anghelescu, and Gheorghe Serban. Rsa encryption algorithm implemented on fpga. In *Applied Electronics (AE), 2011 International Conference on*, pages 1–4. IEEE, 2011.

[23] Young Sae Kim, Woo Seok Kang, and Jun Rim Choi. Asynchronous implementation of 1024-bit modular processor for rsa cryptosystem. In *ASICs, 2000. AP-ASIC 2000. Proceedings of the Second IEEE Asia Pacific Conference on*, pages 187–190. IEEE,

2000. Disponível em : http://ieeexplore.ieee.org/document/896940/, Acessado em 23/09/2016.

[24] Taek-Won Kwon, Chang-Seok You, Won-Seok Heo, Yong-Kyu Kang, and Jun-Rim Choi. Two implementation methods of a 1024-bit rsa cryptoprocessor based on modified montgomery algorithm. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 650–653. IEEE, 2001. Disponível em : http://ieeexplore.ieee.org/document/922321/, Acessado em 23/09/2016.

[25] Arjen K. Lenstra. Memo on rsa signature generation in the presence of faults. October 1996. Disponível em : https://infoscience.epfl.ch/record/164524/files/nscan20.PDF, Acesso em: 21/11/2015.

[26] Arjen K Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of bernstein's factorization circuit. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–26. Springer, 2002.

[27] Fabio Brochero Martinez, Carlos Gustavo Moreira, Nicolau Saldanha, and Eduardo Tengan. Teoria dos números: um passeio com primos e outros números familiares pelo mundo inteiro. *Rio de Janeiro: IMPA*, 2010. Disponível em : http://www.tsouanas.org/teaching/fmc1/2016.1/docs/Teoria_dos_numeros_Um_passeio_com_primos_3ed.pdf, Acessado em 23/09/2016.

[28] Mohit Mishraa, Utkarsh Chaturvedib, and KK Shukla. A new heuristic algorithm based on molecular geometry optimization and its application to the integer factorization problem. In *Soft Computing and Machine Intelligence (ISCMI), 2014 International Conference on*, pages 115–120. IEEE, 2014. Disponível em : http://ieeexplore.ieee.org/document/7079366/ , Acessado em 23/09/2016.

[29] Anderson Zudio de Moraes and Victor Cracel Messner. O algoritmo aks e a evolução dos testes de primalidade. Master's thesis, Instituto de MatemÁtica E Estatística da Universidade Estadual do Rio de Janeiro, Rio de Janeiro - RJ, Junho 2016.

[30] Samáris Ramiro Pereira. *O Sistema Criptográfico de Chave Pública RSA*. PhD thesis, February 2008. Disponível em : http://docplayer.com.br/4051100-O-sistema-criptografico-de-chave-publica-rsa.html, Acessado em 12/03/2016.

[31] Carl Pomerance. A tale of two sieves. *Biscuits of Number Theory*, 85:175, 2008.

[32] Sukumar S Raghuram and Chaitali Chakrabarti. A programmable processor for cryptography. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 5, pages 685–688. IEEE, 2000. Disponível em : http://ieeexplore.ieee.org/document/857574/, Acessado em 23/09/2016.

[33] A. Shamir R.L. Rivest and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Computer*, 25:120–126, March 1978. Disponível em : http://ieeeExplore.org.br/10.1109/2.121503, Acessado em agosto 2013.

[34] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. MacGraw Hill, 7 edition, 2012.

[35] William Sallings. *Cryptography and Network Security, Principles and Practice*. Pearson Prentice Hall, 5 edition, 2011. Disponível em : `http://williamstallings.com/Cryptography/` ,Acessado em agosto de 2013.

[36] José Plínio de Oliveira Santos. *Introdução á Teoria dos Números, Terceira Edição*. IMPA - Instituto Nacional de Matemática Aplicada, 2011.

[37] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[38] Adi Shamir and Eran Tromer. Factoring large numbers with the twirl device. In *Annual International Cryptology Conference*, pages 1–26. Springer, 2003.

[39] Pitcha Tyoviriyakul and Surin Kittitornkun. Optimizing rsa encryption for arm microprocessor. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008. ECTI-CON 2008. 5th International Conference on*, volume 1, pages 117–120. IEEE, 2008.

[40] Zhongbo Wang, Zhiping Jia, Lei Ju, and Renhai Chen. Asip-based design and implementation of rsa for embedded systems, 2012. Disponível em : `http://ieeexplore.ieee.org/document/6332338/`, Acessado em 23/09/2016.

Apêndice A

Código da Implementação do RSA

Apresentação do código utilizado para a avaliação dos testes de Primalidade do RSA. Nesta seção vamos mostrar uma implementação na linguagem C. A codificação e a decodificação está sendo feita no mesmo algoritmo, em ordem subsequente. Os comentários sobre a estrutura estão nas próprias linhas de comando.

A.1 Criptografia e Decriptografia

Criptografia -> transforma um texto puro em um texto cifrado.
Decriptografia -> transforma um texto cifrado em um texto puro.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <time.h>
5  #include <string.h>
6  #include <assert.h>
7  #include <sys/stat.h>
8  #include <sys/types.h>
9  #include <sys/time.h>
10 #include <fcntl.h>
11 #include <unistd.h>
12 #include <math.h>
13 #include <inttypes.h>
14
15 /*---------------------------------------------------------------------*/
16 #define BUFFER 1024 // Buffer inicial;
17 #define _GNU_SOURCE
18 #define ACCURACY 5 // (margem de erro máxima de 50%)
19 #define SINGLE_MAX 10000
20 #define EXPONENT_MAX 1000
21 #define TRUE 1
22 #define FALSE 0
23 /* O argumento agora deve ser um double (não um ponteiro para um double) */
24 #define GET_TIME(now) { \
25   struct timeval t; \
26   gettimeofday(&t, NULL); \
27   now = t.tv_sec + t.tv_usec/1000000.0; \
28 }
29 /*---------------------------------------------------------------------*/
30
```

```
31 /**
32  * Compute a^b mod c
33  */
34 unsigned int modpow(long long a, long long b, unsigned int c) {
35       unsigned int res = 1;
36       while(b > 0) {
37             /* Precisa de multiplicação de um "long long",
38             tratar para evitar overflow... */
39             if(b & 1) {
40                   res = (res * a) % c;
41             }
42             b = b >> 1;
43             a = (a * a) % c; /* Mesma coisa aqui */
44       }
45       return res;
46 }
47
48 /**
49  * Compute o simbolo de Jacobi, (a, n); Éuma generalizacao do simbolo de
50  * Legendre.
51  */
52 unsigned int jacobi(unsigned int a, unsigned int n) {
53       unsigned int twos, temp;
54       unsigned int mult = 1;
55       while(a > 1 && a != n) {
56             a = a % n;
57             if(a <= 1 || a == n) break;
58             twos = 0;
59             while(a % 2 == 0 && ++twos) a /= 2;
60             /* Fator multiplicativo de saida 2 */
61             if(twos > 0 && twos % 2 == 1)
62                   mult *= (n % 8 == 1 || n % 8 == 7) * 2 - 1;
63             if(a <= 1 || a == n) break;
64             if(n % 4 != 1 && a % 4 != 1)
65                   mult *= -1; /* Coeficiente para transformacao */
66             temp = a;
67             a = n;
68             n = temp;
69       }
70       if(a == 0) return 0;
71       else if(a == 1) return mult;
72       else return 0; /* a == n => gcd(a, n) != 1 */
73 }
74
75 /**
76  * Verificar se um éCritério de Euler, para n
77  */
78 unsigned int SolovayPrime(unsigned int a, unsigned int n) {
79       unsigned int x = jacobi(a, n);
80       if(x == -1) x = n - 1;
81       return x != 0 && modpow(a, (n - 1)/2, n) == x;
82 }
83
84 /**
85  * Testa se n éprovavelmente primo, usando a acuracia k
86  * (k testes de Solovay)
87  */
88 unsigned int probablePrime(unsigned int n, unsigned int k) {
```

```
89        if(n == 2) return 1;
90        else if(n % 2 == 0 || n == 1) return 0;
91        while(k-- > 0) {
92              if(!SolovayPrime(rand() % (n - 2) + 2, n)) return 0;
93        }
94        return 1;
95  }
96
97  /**
98   * Busca um primo randomico (provavel) entre 3 e n - 1, esta distribuição é
99   * nem de longe uniforme
100  */
101 unsigned int randPrime(unsigned int n)
102 {
103       unsigned int prime = random_number(10,100) % n;
104       n += n % 2; /* n precisa ser mesmo assim modulo, preserva */
105       prime += 1 - prime % 2;
106       while(1) {
107             if(probablePrime(prime, ACCURACY)) return prime;
108             prime = (prime + 2) % n;
109       }
110 }
111
112 /**
113  * Compute gcd(a, b)
114  */
115 unsigned int gcd(unsigned int a, unsigned int b) {
116       unsigned int temp;
117       while(b != 0) {
118             temp = b;
119             b = a % b;
120             a = temp;
121       }
122       return a;
123 }
124
125 /**
126  * Busca um expoente x entre 3 e n - 1 de tal modo que gcd(x, phi) = 1,
127  * essa distribuição ésemelhante em nenhuma parte uniforme
128  */
129 unsigned int randExponent(unsigned int phi, unsigned int n)
130 {
131       unsigned int e = rand() % n;
132       while(1) {
133             if(gcd(e, phi) == 1) return e;
134             e = (e + 1) % n;
135             if(e <= 2) e = 3;
136       }
137 }
138
139 /**
140  * Compute n^-1 mod m para metodo Euclides extendido
141  */
142 unsigned int inversoMult(unsigned int n, unsigned int modulus) {
143       unsigned int a = n, b = modulus;
144       unsigned int x = 0, y = 1, x0 = 1, y0 = 0, q, temp;
145       while(b != 0) {
146             q = a / b;
```

```
147               temp = a % b;
148               a = b;
149               b = temp;
150               temp   = x; x = x0 - q * x; x0 = temp;
151               temp = y; y = y0 - q * y; y0 = temp;
152         }
153         if(x0 < 0) x0 += modulus;
154         return x0;
155   }
156
157   /**
158     * Leia o arquivo fd em um array de bytes prontos para criptografia.
159     * A matriz será preenchido com zeros até que divide o número de
160     * Bytes criptografados por bloco. Retorna o número de bytes lidos.
161   */
162   unsigned int transformaArquivo(FILE* fd, char** buffer, unsigned int bytes)
163   {
164         unsigned int len = 0, cap = BUFFER, r;
165         char buf[BUFFER];
166         *buffer = malloc(BUFFER * sizeof(char));
167         while((r = fread(buf, sizeof(char), BUFFER, fd)) > 0) {
168               if(len + r >= cap) {
169                     cap *= 2;
170                     *buffer = realloc(*buffer, cap);
171               }
172               memcpy(&(*buffer)[len], buf, r);
173               len += r;
174         }
175         /* Pad o último bloco com zeros para sinalizar final de
176         criptograma. Um bloco adicional éadicionado, se não há espaço */
177         if(len + bytes - len % bytes > cap)
178               *buffer = realloc(*buffer, len + bytes - len % bytes);
179         do {
180               (*buffer)[len] = '\0';
181               len++;
182         }
183         while(len % bytes != 0);
184         return len;
185   }
186
187
188   /**
189     * Codificar a mensagem de determinando comprimento, usando a chave
190     * pública (expoente, módulo)
191     * A matriz resultante será do tamanho len/bytes, cada índice da
192     * criptografia
193     * Os "bytes" de caracteres consecutivos, dados por
194     * m = (m1 + m2 * 128 + m3 * 128^2 + ...),
195     * Codificado = m^expoente módulo mod
196    */
197   unsigned int* criptografa(unsigned int len, unsigned int bytes,
198   char* message, unsigned int exponent, unsigned int modulus, FILE *Arq,
199   char *buffer, FILE *arquivolog)
200   {
201         unsigned int *cod = malloc((len/bytes) * sizeof(unsigned int));
202         unsigned int x, i, j ;
203         printf("\t\t\t");
204         escrever_cripto("\t\t\t",arquivolog);
```

```
205       for(i = 0; i < len; i += bytes) {
206             x = 0;
207             for(j = 0; j < bytes; j++) x += message[i + j] * (1 << (7 * j));
208             /* Codificar a mensagem m usando expoente público e módulo,
209               c = m^e mod n */
210             cod[i/bytes] = modpow(x, exponent, modulus);
211 #ifndef MEASURE
212             printf("%d ", cod[i/bytes]);
213             fprintf(Arq,"%d ", cod[i/bytes]);
214             sprintf(buffer,"%d ", cod[i/bytes]);
215             escrever_cripto(buffer,arquivolog);
216 #endif
217       }
218       return cod;
219 }
220
221 /**
222  * Decodificar o criptograma de determinado comprimento,
223  * usando a chave privada (expoente, módulo)
224  * Cada bloco criptografado deve representar em "bytes" blocos codificados
225  * da mensagem.
226  * Mensagem retornada tem tamanho len * bytes.
227  */
228 unsigned int* decriptografa(unsigned int len, unsigned int bytes, unsigned
229 int* cryptogram, unsigned int exponent, unsigned int modulus,char *buffer,
230 FILE *arquivolog)
231 {
232       unsigned int *decod = malloc(len * bytes * sizeof(unsigned int));
233       unsigned int x, i, j;
234       printf("\t\t\t");
235       escrever_cripto("\t\t\t",arquivolog);
236       for(i = 0; i < len; i++) {
237       /* Decodificar criptograma c usando expoente privado
238         e módulo público, m = c^d mod n */
239             x = modpow(cryptogram[i], exponent, modulus);
240             for(j = 0; j < bytes; j++) {
241                   decod[i*bytes + j] = (x >> (7 * j)) % 128;
242 #ifndef MEASURE
243                   if(decod[i*bytes + j] != '\0'){
244                         if (decod[i*bytes + j] != '\n'){
245                           printf("%c", decod[i*bytes + j]);
246                           sprintf(buffer,"%c", decod[i*bytes + j]);
247                           escrever_cripto(buffer,arquivolog);
248                         }
249                         else{
250                             printf("\n\t");
251                             escrever_cripto("\n\t",arquivolog);
252                         }
253                   }
254 #endif
255             }
256       }
257       return decod;
258 }
259
260 unsigned int random_number (unsigned int min, unsigned int max)
261 {
262   /* Armazenar um descritor de arquivo aberto para /dev/random em uma
```

```
263     variavel estatica. Dessa forma, não precisa abrir o arquivo de cada
264     vez e esta função échamada. */
265   static unsigned int dev_random_fd = -1;
266
267   char* next_random_byte;
268   unsigned int bytes_to_read;
269   unsigned random_value;
270
271   /* Certifique-se de MAX émaior do que MIN.*/
272   assert (max > min);
273
274   /* Se esta éa primeira vez que esta função échamada, abrir um arquivo
275      descritor para /dev/random.*/
276   if (dev_random_fd == -1) {
277     dev_random_fd = open ("/dev/random", O_RDONLY);
278     assert (dev_random_fd != -1);
279   }
280
281   /* Leia bytes aleatórios suficientes para preencher
282        uma variável unsigned inteira. */
283   next_random_byte = (char*) &random_value;
284   bytes_to_read = sizeof (random_value);
285   /* Loop lido com o suficiente bytes. Desde /dev/random épreenchido
286      a partir de ações gerado pelo usuário, a leitura pode bloquear,
287      e só pode retornar um único byte aleatório de cada vez. */
288   do {
289     unsigned int bytes_read;
290     bytes_read = read (dev_random_fd, next_random_byte, bytes_to_read);
291     bytes_to_read -= bytes_read;
292     next_random_byte += bytes_read;
293   } while (bytes_to_read > 0);
294
295   /* Calcular um número aleatório na faixa correta. */
296   return min + (random_value % (max - min + 1));
297 }
298
299
300 /*---------------------------------------------------------------------*/
301 unsigned int sizeBlock(unsigned int n, char *buffer,FILE *arq)
302 {
303       unsigned int cont = 0, num = 0;//, original = n;
304       while (n/2 > 0){ //testa o resto da divisao por 2
305
306             if ((n % 2) == 0){
307                   num ++;
308             }
309             else { num ++; }
310             n = n/2;
311       }
312       cont = num / 8; // transformando bits em bytes
313       printf("\n\t\t\tTamanho do Bloco = %d \n",cont);
314       sprintf(buffer,"\n\t\t\tTamanho do Bloco = %d \n",cont);
315       escrever_log(buffer,arq);
316       return cont;
317 }
318 /*---------------------------------------------------------------------*/
319 int main(int argc, char **argv)
320 {
```

```
321     unsigned int p, q, n, phi, e, d, bytes, controle;
322     unsigned int *cod, *decod;
323     double startTime, endTime, tempo;
324     FILE    *entrada = NULL;
325     FILE    *saida =  NULL;             /*arquivo de saida */
326     size_t  tam   = (sizeof(char)*BUFFER);
327     char    *buffer = (char*)malloc(tam);
328     char    *buf;
329
330     /* testa caso o numero de parametros seja menor e finaliza o programa*/
331     if (argc < 2)
332     {
333      printf("Falha na entrada dos dados.........\n");
334      printf("EXEMPLO DE EXECUCAO : \n");
335      printf("RSA_SolovayStrassen [arquivo de entrada][arquivo de saida]");
336      exit(EXIT_FAILURE);
337     }
338     else
339        {
340              printf("-----IMPLEMENTATION OF R.S.A ALGORITHM-----\n");
341              printf("Teste de Primalidade de Solovay-Strassen\n");
342              srand(time(NULL));
343              GET_TIME(startTime);
344              if (argc == 2)
345              {     /* se igual a 2; Acrescentar arquivo de saida
346                       e executar programa */
347                    entrada = fopen(argv[1], "r");
348                    saida = fopen("saida.txt","w+");
349                    if(entrada == NULL && saida == NULL)
350                    {
351                          printf("Erro ao abrir arquivo.\n");
352                          printf("Fim execucao do programa. \n");
353                          exit(EXIT_FAILURE);
354                    }
355                    p = randPrime (SINGLE_MAX);
356                    printf("Primeiro fator primo, p = %d \n", p);
357                    q = randPrime(SINGLE_MAX);
358                    printf("Segundo fator primo, q = %d \n", q);
359                    n = p * q;
360                    printf("Modulus, n = pq = %d ", n);
361                    if(n < 128) {
362                          printf("Módulo éinferior a 128,
363                          não épossível codificar bytes simples.
364                          Tentando novamente \n");
365                    }
366                    bytes = sizeBlock(n,buffer,arquivolog);
367                    phi = (p - 1) * (q - 1);
368                    printf("Totiente, phi = %d \n", phi);
369                    e = randExponent(phi, EXPONENT_MAX);
370                    printf("Expoente Publico, e = %d,
371                          Chave Publica eh (%d, %d) \n", e, e, n);
372
373                    d = inversoMult(e, phi);
374                    printf("Calcule expoente privado, d = %d
375                          Chave Privada eh (%d, %d)\n", d, d, n);
376                    printf("Abrindo arquivo %s para
377                          leitura\n",argv[1]);
378                    controle = transformaArquivo(entrada, &buf, bytes);
```

```
379                     /* controle será um múltiplo de bytes*/
380
381                     printf("Arquivo %s lido com sucesso, %d bytes
382                     lidos. Codificacao do bloco em byte de %d bytes",
383                     argv[1], controle, bytes);
384                     printf("Codificando mensagem com sucesso");
385                     cod = criptografa(controle, bytes, buf, e, n,saida,
386                           buffer,arquivolog);
387                     printf("Decodificando a mensagem com sucesso");
388                     decod = decriptografa(controle/bytes, bytes, cod,
389                            d, n,buffer,arquivolog);
390               }else {
391                     entrada = fopen(argv[1],"r");
392                     saida = fopen(argv[2],"w+");
393                     if(entrada == NULL && saida == NULL)
394                     {
395                           printf("Erro ao abrir arquivo.\n");
396                           printf("Fim execucao do programa.\n");
397                           exit(EXIT_FAILURE);
398                     }else{
399                     p = randPrime (SINGLE_MAX);
400                     printf("Primeiro fator primo, p = %d \n", p);
401                     q = randPrime(SINGLE_MAX);
402                     printf("Segundo fator primo, q = %d \n", q);
403                     n = p * q;
404                     printf("Modulus, n = pq = %d ", n);
405                     if(n < 128) {
406                           printf("Módulo éinferior a 128,
407                           não épossível codificar bytes simples.
408                           Tentando novamente \n");
409                     }
410                     bytes = sizeBlock(n,buffer,arquivolog);
411                     phi = (p - 1) * (q - 1);
412                     printf("Totiente, phi = %d \n", phi);
413                     e = randExponent(phi, EXPONENT_MAX);
414                     printf("Expoente Publico, e = %d,
415                           Chave Publica eh (%d, %d) \n", e, e, n);
416
417                     d = inversoMult(e, phi);
418                     printf("Calcule expoente privado, d = %d
419                           Chave Privada eh (%d, %d)\n", d, d, n);
420                     printf("Abrindo arquivo %s para
421                           leitura\n",argv[1]);
422                     controle = transformaArquivo(entrada, &buf, bytes);
423                     /* controle será um múltiplo de bytes*/
424
425                     printf("Arquivo %s lido com sucesso, %d bytes
426                     lidos. Codificacao do bloco em byte de %d bytes",
427                     argv[1], controle, bytes);
428                     printf("Codificando mensagem com sucesso");
429                     cod = criptografa(controle, bytes, buf, e, n,saida,
430                           buffer,arquivolog);
431                     printf("Decodificando a mensagem com sucesso");
432                     decod = decriptografa(controle/bytes, bytes, cod,
433                            d, n,buffer,arquivolog);
434                     }
435                   }
436         }
```

```
437         GET_TIME(endTime);
438         tempo = endTime - startTime;
439         printf("Tempo de Execução do Algoritmo RSA = %.6lf ----\n",tempo);
440         printf("\n\n\t\t\t ---- Fim demonstracao do RSA! ----\n\n");
441         free(cod);
442         free(decod);
443         fclose(entrada);
444         fclose(saida);
445         return EXIT_SUCCESS;
446 } /*fim do programa */
447
448 /*--------------------------------------------------------------------------*/
```

Apêndice B

Código da Implementação do RSA

O RSA é um desafio que se tornou mais premente após o seu desenvolvimento pois requer uma abordagem teórica e matemática bastante especial que instiga um estudo mais aprofundado, e como não é foco central desta monografia pretende-se apenas induzir novas descobertas sobre sua morfologia. Segue um breve estudo sobre otimizações e ataques.

B.1 Otimizações do RSA

Existem várias visões e diferentes métodos de otimização do algoritmo criptográfico RSA. Mas a partir de diferentes visões encontradas pretendem-se instigar os desenvolvedores a repensar as diferentes possibilidades de solução de um problema. Suscitando a sensibilidade do pensar e de buscar maneiras diversas de resolver os problemas propostos pelo algoritmo, para que o ato perpasse as circunstâncias complexas que se apresentam na atualidade. Encontra-se bons artigos, dissertações, e ou publicações sobre otimização do algoritmo criptográfico RSA, alguns utilizam: Programação Paralela [8, 7, 17, 9]; Processadores e Melhoria em Hardware [40, 32, 39]; Programação de baixo nível "Assembler"[22, 20, 16]; linguagem VHDL [23, 24]; e até mesmo utilizando Química Computacional [28], com inspiração em um algoritmo heurístico e a minimização de energia de geometria molecular de um cristal.

B.2 Ataques de fatoração do RSA

Existem diversas formas de se fatorar um número, mas para um número com quase dezesseis trilhões e meio de casas decimais um certo cuidado é essencial. Então uma tentativa de se quebrar a chave privada é usar a chave pública que é conhecida de todos, mas tentar fatorá-la por força bruta já se provou ser inviável. Portanto ao longo dos anos a comunidade científica tem tomado para si a árdua tarefa de estudar novos métodos para se fatorar grandes números. O tamanho da chave a utilizar no RSA depende do grau de segurança que se pretende. Quanto maior for a chave, maior será a segurança, mas também mais lento será efetuar o algoritmo do RSA. O RSA *Laboratories* [1] recomenda que as empresas utilizem chaves de 1024 bits para proteção dos dados, pois esta criptografia é extremamente forte e de difícil quebra. Na figura [B.1] iremos apresentar um pequeno exemplo destes processos de fatoração.

Progresso na fatoração

Número de dígitos decimais	Número aproximado de bits	Data em que foi alcançado	MIPS-anos	Algoritmo
100	332	abril de 1991	7	Crivo quadrático
110	365	abril de 1992	75	Crivo quadrático
120	398	junho de 1993	830	Crivo quadrático
129	428	abril de 1994	5000	Crivo quadrático
130	431	abril de 1996	1000	Crivo de corpo numérico generalizado
140	465	fevereiro de 1999	2000	Crivo de corpo numérico generalizado
155	512	agosto de 1999	8000	Crivo de corpo numérico generalizado
160	530	abril de 2003	—	Crivo de malha (Lattice sieve)
174	576	dezembro de 2003	—	Crivo de malha (Lattice sieve)
200	663	maio de 2005	—	Crivo de malha (Lattice sieve)

Figura B.1: Exemplo de Processo de Fatoração [35]

B.2.1 Crivo Quadrático

O crivo quadrático [31, 14, 5, 11] é um dos mais importantes métodos de fatoração de números inteiros (chaves de 100 a 129 dígitos na forma decimal). Sua implementação não é uma tarefa trivial, não é o objetivo deste trabalho apresentar uma descrição detalhada. Em resumo é necessário:

1. Encontrar uma base, um fator inteiro. Temos um número a ser fatorado e um valor limite como entrada.

2. Determinar um conjunto de números que podem ser completamente fatorados sobre a base de fatos. Determinar a base do fator e calcular o símbolo de Legendre 2.1.14.

3. Usar a Eliminação Gaussiana para encontrar um produto dos números determinados no passo 2 que seja um quadrado perfeito. Obtendo como resultado dois números fatorados (ou dois fatores).

B.2.2 Crivo de Corpo Numérico

O crivo de corpo numérico [31] produz uma fatoração de um número algébrico caracterizando até que estejam próximos da potência do número, além da manter as propriedades desejáveis. É um dos métodos mais usados na atualidade. O algoritmo apresenta três variantes: um especial crivo de corpo numérico, o geral crivo de corpo numérico (quadrado perfeito), e uma variante criada por Coppersmith, onde utiliza vários polinômios. Descreveremos apenas a ideia principal do algoritmo:

- Definir os elementos suaves sobre uma base algébrica de fatores em $\mathbb{Z}[\alpha]$ e em $\mathbb{Z}$.

- Encontrar números suaves (método semelhante ao item acima, buscando um quadrado perfeito): crivo.

- Verificar se existem elementos em $\mathbb{Z}[\alpha]$ e em $\mathbb{Z}$ que são quadrados perfeitos.

- Dos números suaves ao quadrado perfeito.

B.2.3 Crivo de Malha

O crivo de malha [37, 10, 38, 21, 26] são processos locais de crivo em grandes pedaços, baseado em uma malha sistólica 2D de nós idênticos. Cada nó executa três funções:

- Fazer parte de uma rede genérica de roteamento de pacotes de malha.
- Atribuir uma parte das progressões.
- Incumbir-se de certas localizações do crivo em cada intervalo de locais de peneiramento.

Para cada intervalo de peneiramento (operações básicas):

1. Cada processador inspeciona as progressões armazenadas dentro e emite todas as contribuições relevantes como pacotes: $(a, log\ p)$.
2. Cada pacote $(a, log\ p)$ é encaminhado, através de roteamento de malha, para a célula de malha responsável pela localização do crivo a.
3. Quando uma célula responsável pela localização do crivo a recebe um pacote $(a, log\ p)$, ele o consome e adiciona $log\ p$ a um acumulador correspondente a (inicialmente 0).
4. Uma vez que todos os pacotes chegaram, os acumuladores são comparados com o limiar.

No crivo baseado em malha[1], roteamos e somamos as contribuições de progressão para peneirar locais. Na álgebra linear baseada em malhas, rotear e somar entradas da matriz multiplicado por entradas de vetores antigos para novas entradas de vetor, em ambos os casos, equilibrar o custo da memória e da lógica.

[1]Autores: Adi Shamir and Eran Tromer `https://www.cs.tau.ac.il/~tromer/slides/hwsieve-slides.ppt`