Implementação e análise comparativa de variações do criptossistema RSA

Cesar Alison Monteiro Paixão

DISSERTAÇÃO APRESENTADA

AO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DA

UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE

EM

CIÊNCIA DA COMPUTAÇÃO

Área de Concentração : Ciência da Computação
Orientador : Prof. Dr. Routo Terada

- São Paulo, abril de 2003 -

Implementação e análise comparativa de variações do criptossistema RSA

Este exemplar corresponde à redação
final da dissertação devidamente corrigida e
apresentada por Cesar Alison Monteiro Paixão e
aprovada pela Comissão Julgadora.

São Paulo, 11 de abril de 2003

Banca Examinadora :

- Prof. Dr. Routo Terada (orientador) -IME-USP
- Prof. Dr. Fábio Kon -IME-USP
- Prof. Dr. João José Neto - EP-USP

Aos meu pais Cesar Paixão e Maria Izilda

Agradecimentos

Ao meu orientador Prof. Dr. Routo Terada que me auxíliou na escolha do tema, na elaboração do trabalho e me introduziu, através de suas aulas, à fascinante ciência denominada criptologia.

Aos meus amigos do laboratório de processamento de imagens do IME-USP. Dentre eles agradeço especialmente a Celina M. Takemura e David S. Pires pela amizade que fizemos e pelas importantes observações no decorrer do desenvolvimento desta dissertação. Agradeço também aos meus amigos Eduardo T. Ueda, Jorge A. Tonami e Ney B. Luz pelas fundamentais discussões que tivemos principalmente no ínicio do mestrado.

À minha namorada Karina S. C. Miras pela compreensão, carinho, paciência e pelo tempo dispensado na revisão deste trabalho.

À toda minha família, pela ajuda financeira, intelectual e emocional; principalmente às minhas irmãs Patricia S. M. Paixão e Tatiana M. M. Paixão, por me ajudarem na pontuação do trabalho.

Não poderia deixar de agradecer a Deus, que possibilitou todos estes acontecimentos e que me dá forças sempre que preciso.

Abstract

In 1976 the article “New Directions on Cryptography” was published by W. Diffie and M. E. Hellman [13], inspiring the creation of the so called public key algorithms. Since then, several cryptosystems from this category have been created, being RSA the most used and divulged one. That is due to the characteristics like ease of implementation, possibility of digital signatures generation and verification, allied with high reliability related to security. Nevertheless this cryptosystem demands high computional time and, in virtue of dissemination and popularization of the Internet and by the extraordinary increase corcerning the number of users and secret information, it has become necessary to research more efficient alternatives. Hence, in the last years, experts of diverse areas related to criptology have created some variants of the most famous public key cryptosystem, aiming at reducing its execution time.

The purpose of this document is to analyse the four most popular ones from these variants (Batch RSA, Mprime RSA, Mpower RSA and Rebalanced RSA), related to speed, security, and interoperability with current systems, besides verifing, with the same parameters, the results obtained by the combination of two proposals studied (Rebalanced RSA and Mprime RSA).

Resumo

Em 1976 foi publicado o artigo “New Directions in Cryptography” por W. Diffie e M. E. Hellman [13], inspirando a criação dos chamados algoritmos de chave pública. Desde então, vários criptossistemas desta categoria foram criados, sendo o RSA o mais utilizado e difundido. Isto se deve a características como facilidade de implementação, possibilidade de geração e verificação de assinaturas digitais, aliadas com a alta confiabilidade relativa à segurança. Todavia, esse criptossistema exige um alto tempo computacional e, em virtude da disseminação e popularização da Internet e do espantoso crescimento referente ao número de usuários e informações sigilosas, se fez necessário a busca de alternativas mais eficazes. Como resultado, nos últimos anos estudiosos de diversas áreas ligadas à criptologia criaram algumas variações do mais famoso criptossistema de chave pública, com intuito de diminuir o tempo de execução do mesmo.

O objetivo desta dissertação é analisar as quatro mais populares destas variações (RSA em Lote, RSA com Múltiplos Primos, RSA com Múltiplas Potências e RSA Rebalanceado), com relação à velocidade, segurança e interoperabilidade com sistemas atuais, além de verificar, com os mesmos parâmetros, os resultados obtidos pela combinação de duas das propostas estudadas (RSA Rebalanceado e RSA com Múltiplos Primos).

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO

1

Introdução

Criado no *Massachusetts Institute of Tecnology - (M.I.T)* por Ron L. Rivest, Adi Shamir e Len Adleman [1], o RSA é atualmente o criptossistema de chave pública mais difundido e implementado. Sua atuação abrange tanto a codificação e decodificação de mensagens, quanto a geração e verificação de assinaturas digitais. Além disso, este criptossistema é de fácil implementação e, apesar de estar sob ataques há mais de vinte e cinco anos, desde sua apresentação para o grande público em agosto de 1977 pela revista *Scientific American*, nenhum desses ataques exigiu sequer uma mudança na estrutura do mesmo.

As características citadas acima, explicam a ampla utilização e propagação do RSA nas mais diversas aplicações, como por exemplo, na criação de certificados digitais, na segurança no tráfego da *Web* e na autenticação de usuários através do uso de *smart cards*. Mais que isso, somadas às recentes padronizações e especificações, essas características nos fazem prever a permanência do RSA como ferramenta de segurança por ainda muitos anos.

Contudo, o RSA possui pontos fracos e o principal deles é o alto tempo computacional necessário para a criptografia e decriptografia. Objetivando diminuir esse tempo, algumas implementações do RSA utilizam expoentes pequenos, mas como veremos na seção 3.5 é

necessário ter cuidado na escolha desses expoentes (pois estão intimamente ligados à segurança do criptossistema). Neste contexto, nos últimos anos alguns algoritmos que buscam melhorar o desempenho de criptografia e principalmente de decriptografia do RSA têm sido propostos, merecendo alguns destes atenção especial.

1.1 Objetivo

Esta dissertação tem como objetivo apresentar um resumo dos tópicos relacionados com a criptografia de chave pública, concentrando a atenção no criptossistema RSA e em quatro variações recentes deste. As variações estudadas: RSA em Lote, RSA com Múltiplos Primos, RSA com Múltiplas Potências e RSA Rebalanceado, estão focadas na diminuição do tempo de decriptografia do criptossistema original tornando mais aceitável a utilização deste em sistemas com recursos computacionais limitados (por exemplo computadores de mão).

Buscando encontrar a melhor solução, apresentamos uma análise comparativa entre essas variações e incluímos nessa análise um novo esquema, baseado na combinação das técnicas utilizadas pelo RSA Rebalanceado e pelo RSA com Múltiplos Primos.

1.2 Contribuições

As quatro principais contribuições desta dissertação são descritas abaixo:

- Uma revisão bibliográfica de conceitos e métodos importantes inerentes à Teoria dos Números e Criptografia, criando um arcabouço necessário para o entendimento de algumas variações recentes do RSA.
- A proposição de um novo método, denominado RSA Rebalanceado com Múltiplos Primos (principal contribuição original deste trabalho), que constitui uma combinação dos métodos denominados RSA Rebalanceado (*Rebalanced RSA*) e RSA com Múltiplos Primos (*Mprime RSA*).
- A implementação das variações estudadas, incluindo a implementação do novo método proposto, em bibliotecas que facilitam a utilização e permitem um maior aproveitamento de código.

- A análise comparativa de todos os esquemas estudados, tratando características como o desempenho relativo à velocidade, segurança, padronização e uma breve discussão sobre as facilidades de implementação e implantação destes algoritmos.

Como outras contribuições não tão significativas, podemos citar um resumo dos principais ataques relacionados ao RSA e os resultados práticos obtidos pela utilização do Algoritmo Clássico [30] e pelo algoritmo de Garner [31] (ambos utilizados na aplicação do Teorema Chinês do Resto [30]).

1.3 Organização da Dissertação

Neste capítulo foram apresentadas as motivações que levaram ao estudo do criptossistema RSA e algumas de suas variações.

No segundo capítulo são introduzidos os conceitos necessários para a compreensão dos algoritmos de chave pública, abordando desde a definição de números inteiros até a introdução de teoremas fundamentais na Criptografia.

No terceiro capítulo descrevemos o criptossistema RSA por meio dos algoritmos de geração de chaves, criptografia e decriptografia. Em seguida, apresentamos o método de *Quisquater-Couvreur* que tem por fim aumentar o desempenho da decriptografia RSA. Provamos a correção e analisamos a segurança do criptossistema, descrevendo os ataques mais conhecidos sobre este.

O quarto capítulo apresenta os algoritmos de geração de chaves, criptografia e decriptografia de cada uma das variações seguidos de um exemplo numerico [1] de cada uma delas.

Com base no conhecimento fornecido pelo quarto capítulo, o capítulo seguinte visa criar condições para a implementação e análise de todas as variações estudadas. Para isso, introduz a metodologia e as ferramentas utilizadas durante este processo, bem como a especificação que viabiliza a comunicação com sistemas que já fazem uso do RSA.

O sexto capítulo contém os detalhes e resultados inerentes à implementação dos algoritmos apresentados no quarto capítulo. Nele também encontram-se uma análise do ganho de cada

[1] Os exemplos numéricos apresentados neste documento, foram produzidos utilizando o programa Maple V.

variação sobre o criptossistema RSA tradicional e detalhes de segurança de cada uma destas.

Traçando algumas conclusões, o sétimo capítulo sintetiza os resultados apresentados no capítulo precedente facilitando a escolha de uma das propostas mediante a uma dada aplicação.

O apêndice A é dividido em duas seções. Na primeira encontramos os resultados referentes a implementação do algoritmo de Garner e do algoritmo clássico (seções 2.4 e 2.5). Na segunda seção, encontramos exemplos de arquivos de saída gerados pelas implementações dos algoritmos apresentados no quarto capítulo.

O apêndice B disponibiliza a listagem das implementações analisadas no sexto capítulo.

CAPÍTULO

2

Conceitos Fundamentais

Para iniciarmos nosso estudo em criptografia de chave pública, mais precisamente no estudo do criptossistema RSA, necessitamos relembrar algumas definições e conceitos da teoria dos números e, assim, formar o alicerce indispensável para a compreensão dos algoritmos apresentados posteriormente. Portanto, este capítulo serve como base para todos os algoritmos e idéias apresentadas ao longo deste documento e deve ser considerado primeira referência para as dúvidas que surgirem.

2.1 Inteiros, Primos e Outras Definições

Na Criptografia o ínicio de tudo está nos números inteiros. Na criptografia de chave pública, mais especificamente, temos como ponto fundamental os números primos. A seguir, enunciaremos estes conceitos e alguns outros considerados importantes para o entendimento do texto.

O conjunto dos inteiros {... -2, -1, 0, 1, 2, ...} é denotado pelo símbolo Z.

Definição 2.1.1 *Sejam a e b inteiros. Então a divide b se existe um inteiro c tal que b = ac. a divide b é denotado por* $a|b$ *e pode ser lido como a é um divisor de b, ou a é um fator de b.*

Definição 2.1.2 *Um inteiro c é um divisor comum de a e b se* $c|a$ *e* $c|b$*.*

Definição 2.1.3 *Um inteiro positivo d denotado por mdc(a,b) é o máximo divisor comum dos inteiros a e b se d é um divisor comum de a e b e sempre que* $c|a$ *e* $c|b$ *então* $c|d$*. Da mesma forma podemos dizer que d é o maior inteiro positivo que divide tanto a como b, com exceção de mdc(0,0) = 0.*

Definição 2.1.4 *Dois inteiros a e b são ditos relativamente primos ou coprimos se mdc(a,b) = 1.*

Definição 2.1.5 *Um inteiro p* ≥ 2 *é dito primo, se seus únicos divisores positivos são 1 e p. Caso contrário p é chamado composto.*

2.2 Resto de divisão, Módulo, Z_N e Z_N^*

Se a e N são inteiros e $N > 0$, o resto de a dividido por N é o menor inteiro $r \geq 0$ que difere de a por um múltiplo de N e será representado por $a \ mod \ N$ (a módulo N). Escrevemos $a = qN + r$ ou $a \equiv r$ (mod N), onde o inteiro q é chamado quociente e o símbolo $\equiv$ é conhecido como *congruência* [1]. Exemplo: 7, 17, -3 possuem resto 7 quando divididos por 10.

O conceito de congruência foi introduzido por Carl F. Gauss em 1801 [23] em que foi definido que a e b são congruentes ou "equivalentes módulo N" se $N|(a - b)$ para $N > 0$, ou seja, $a - b = kN$ para algum inteiro k. Repare que isso é o mesmo que dizer que a e b são congruentes módulo N se possuem o mesmo resto sobre N e, por isso 7, 17, -3 são congruentes módulo 10. Observe ainda, que existem apenas N inteiros que não são equivalentes módulo N, pois o resto da divisão por N é tal que $0 \leq r \leq N - 1$. Definimos então:

Definição 2.2.1 *O conjunto dos inteiros módulo N é denotado por* Z_N*.*

Observe que Z_N forma o conjunto dos inteiros $0, 1, 2, ..., N - 1$, onde a adição, a subtração e a multiplicação são executadas módulo N, esse conjunto é também conhecido como anel Z_N [30], e é sobre ele que o criptossistema RSA atua.

[1] Neste documento trataremos $a \equiv b$ (mod N) por $a = b \ mod \ N$, por questão de conveniência.

Definição 2.2.2 *O conjunto de todos os elementos em Z_N que são relativamente primos a N é denotado por Z_N^*.*

Definição 2.2.3 *O número de elementos em Z_N^* é simbolizado por $\phi(N)$, este número é também conhecido como* ordem *de Z_N^* ou função de $Euler$. Em particular para N = $\prod_{i=1}^{k} p_i$ onde N é o produto de primos distintos $\phi(N)$ = $\prod_{i=1}^{k}(p_i - 1)$.*

Com os conceitos e definições vistos acima podemos enunciar o seguinte teorema que garante o funcionamento do algoritmo RSA (veja seção 3.4).

Teorema 2.2.1 (Euler) *Sejam $p_1, p_2 ... p_k$, k primos distintos e N = $\prod_{i=1}^{k} p_i$. Então para todo a $\in Z_N^*$ temos $a^{\phi(N)} \equiv 1$ mod N.* [2]

Através do Teorema de Euler chegamos ao seguinte corolário.

Corolário 2.2.1 *Sejam $p_1, p_2 ... p_k$, k primos distintos e N = $\prod_{i=1}^{k} p_i$. Então para todo a $\in Z_N^*$ e um inteiro positivo r temos $a^{r\phi(N)+1} \equiv a$ (mod N). Pois, $a^{r\phi(N)+1} \equiv a^{r\phi(N)}a \equiv 1^r a \equiv a$ mod N.*

2.3 Inversa Multiplicativa módulo N

Seja $a \in Z_N$. A inversa multiplicativa de *a* módulo N é um inteiro $x \in Z_N$ tal que $ax \equiv 1\ (modN)$. Se tal x existir ele é único, neste caso dizemos que *a* é inversível e sua inversa é denotada por a^{-1}. Sabemos que *a* é inversível se e somente se $mdc(a, N) = 1$, e podemos calcular a inversa de *a* pelo algoritmo de Euclides estendido [31].

2.4 Teorema Chinês do Resto - TCR

O teorema chinês do resto pode ser visto, informalmente, como um algoritmo que permite resolver sistemas de congruências lineares. Possui este nome porque um dos primeiros lugares em que apareceu foi no livro *Manual de Aritmética do Mestre Sun*, escrito em torno de 100 D.C., pelo matemático Sun-Tsu. O matemático resolveu o problema de calcular inteiros x que possuem restos 2, 3, 2 quando divididos respectivamente por 3, 5, 7 [24, 30]. A solução é dada

[2] Este teorema é uma extensão do Teorema de Fermat [30, 31].

por $x = 23$ ou, mais especificamente, x = 23 + 105s para qualquer inteiro s (Repare que 105 é o produto de 3, 5 e 7). O problema pode ser enunciado da seguinte forma:

Se os inteiros $n_1, n_2, ..., n_k$ são primos entre si, então o sistema de congruências

$$\begin{aligned} x &\equiv a_1 \ (mod\ n_1) \\ x &\equiv a_2 \ (mod\ n_2) \\ &\vdots \\ x &\equiv a_k \ (mod\ n_k) \end{aligned}$$

tem uma única solução módulo N = $n_1 n_2 ... n_k$, onde $a_1, a_2, ..., a_k$ são inteiros.

O algoritmo clássico para resolução do sistema acima é descrito a seguir:

Sejam N_1 = ($\prod_{j=1}^{k} n_j$)/n_1, ..., N_k = ($\prod_{j=1}^{k} n_j$)/n_k. Pelo Algoritmo de Euclides Estendido (veja [31] e seção 2.3) calculamos os únicos inteiros,

$$N_1' = N_1^{-1}\ mod\ n_1, ..., N_k' = N_k^{-1}\ mod\ n_k$$

desde que cada n_j é relativamente primo ao correspondente N_j. Para terminar, como mostrado em [27] fazemos:

$$x = \sum_{j=1}^{k} a_j N_j N_j'\ mod\ N \tag{2.1}$$

Enunciaremos este resultado mais formalmente a seguir:

Teorema 2.4.1 (TCR) *Suponha $n_1, ..., n_k$ inteiros positivos primos entre si e sejam $a_1, ..., a_k$ inteiros. O sistema de k congruências x $\equiv a_j$ (mod n_j) (1 $\leq$ j $\leq$ k) possui uma única solução módulo N = $n_1 n_2 ... n_k$ que é dada por*

$$x = \sum_{j=1}^{k} a_j N_j N_j' \ \ mod\ \ N$$

onde $N_j = N/n_j$ e $N_j' = N_j^{-1}\ mod\ n_j$, para $1 \leq j \leq k$.

Ou seja, para o caso particular de $k = 2$ temos $N_1 = (n_1 n_2)/n_1 = n_2$ e $N_2 = (n_1 n_2)/n_2 = n_1$ e portanto

$$x = a_1 N_1 N_1' + a_2 N_2 N_2' \ \ mod\ (n_1 n_2) \tag{2.2}$$

onde $N_1' = n_2^{-1}\ mod\ n_1$ e $N_2' = n_1^{-1}\ mod\ n_2$. Note que se a_1 = a_2 então podemos reescrever a solução para o caso $k = 2$ da seguinte maneira

$$\begin{aligned} x &= a_1 N_1 N_1' + a_2 N_2 N_2' \ \ mod\ (n_1 n_2) \\ x &= a_1 (N_1 N_1' + N_2 N_2') \ \ mod\ (n_1 n_2) \\ x &= a_1 \ \ mod\ (n_1 n_2) \end{aligned}$$

pois $N_1 N_1' + N_2 N_2' = s n_1 n_2 + 1$, para algum inteiro $s \geq 1$ [30].

2.5 Algoritmo de Garner

Na seção anterior vimos a utilização do algoritmo clássico na aplicação do Teorema Chinês do Resto (teorema 2.4.1). Apesar de simples, o algoritmo apresentado anteriormente não é a melhor solução, principalmente quando utilizamos módulos grandes. Descrevemos aqui, uma solução apresentada por Garner [31] e que possui algumas vantagens sobre a anterior.

Entrada: Um inteiro positivo $N = \prod_{i=1}^{k} p_i > 1$, com $mdc(p_i, p_j) = 1$ para todo $i \neq j$, e uma representação modular $(a_1, a_2, ...a_k)$ de x para p_i.

Saída: O inteiro x, que representa a solução do sistema de congruências.

Para $i \leftarrow 2$ até k faça
    $C_i \leftarrow 1$
    Para $j \leftarrow 1$ até $(i-1)$ faça
        $h \leftarrow {p_j}^{-1}\ mod\ p_i$
        $C_i \leftarrow u C_i\ mod\ p_i$
$h \leftarrow a_1$
$x \leftarrow h$
$P \leftarrow 1$
Para $i \leftarrow 2$ até k faça

$P \leftarrow p_{i-1}$
$h \leftarrow (a_i - x)C_i \; mod \; p_i$
$x \leftarrow x + h \prod_{j=1}^{i-1} p_j$
retorna x

Podemos observar que no algoritmo clássico necessitamos de uma redução módulo N, enquanto esta não é necessária na versão de Garner. Os atuais algoritmos que calculam reduções modulares levam tempo de execução assintótico [3] igual a $O(n)$ para um módulo de n bits [31]. Suponha assim que N é um inteiro de rs bits e cada p_i é um inteiro de r bits. Desse modo, uma redução modular de N leva $O((rs)^2)$ e uma redução modular de p_i leva $O(r^2)$ operações em bits. Como o algoritmo de Garner só faz uso de reduções módulo p_i, temos que este leva tempo $O(sr^2)$ e, portanto, é mais eficiente.

Neste trabalho, tivemos a oportunidade de comparar o desempenho obtido tanto pelo algoritmo clássico quanto pelo algoritmo de Garner. Esse resultado pode ser visto na primeira seção do apêndice A.

[3] Para saber mais sobre notação assintótica veja [24].

CAPÍTULO

3

Criptografia e Criptografia de Chave Pública

Em grego, *kryptós* significa secreto ou oculto e *grápho* se refere a escrita, portanto, poderíamos definir a Criptografia como a ciência de escrever em código secreto. Em termos mais claros, a Criptografia estuda os métodos para codificar uma mensagem de modo que só seu destinatário legítimo consiga interpretá-la [5, 20]. O método mais simples consiste em substituir uma letra pela seguinte no alfabeto, isto é, transladar o alfabeto uma casa para diante de forma circular. Uma codificação semelhante à esta foi utilizada pelo imperador romano Julio César a fim de estabelecer uma comunicação segura com as legiões em combate pela Europa, ficando assim conhecida como cifra de César, um dos primeiros métodos de codificação de que se tem notícia.

A cifra de César pertence a uma classe de algoritmos conhecidos como algoritmos de chave secreta. A chave secreta, no caso da cifra de César, seria o número de posições deslocadas em relação ao início do alfabeto (no caso acima somente 1 posição) . Observando este método notamos que a chave deve ser previamente combinada entre o emissor e o receptor da mensagem

através de um meio sigiloso[1], além disso, deve ser mantida em segredo para evitar que uma pessoa não autorizada consiga ler a mensagem.

3.1 Criptografia de Chave Pública

Os criptossistemas estudados daqui em diante são conhecidos como algoritmos de chave pública ou assimétricos. Eles se baseiam na idéia de que cada usuário possui um par de chaves (d, e) sendo *d* a chave particular e portanto secreta do usuário (conhecida também como chave de decriptografia) e *e* a chave pública que pode ser fornecida a qualquer pessoa (conhecida também como chave de criptografia).

Para facilitar o entendimento, suponha que possuímos uma caixa inquebrável com uma fechadura que permite a entrada de duas chaves e e d. Estas chaves possuem a seguinte propriedade: a chave e só permite o movimento de giro para a esquerda enquanto que a chave d só permite movimentos giratórios para a direita. Suponha que a caixa esteja aberta quando a tranca da fechadura está na posição central, ou seja, qualquer outra posição fecha terminantemente a caixa. Dessa maneira, um emissor pode colocar uma mensagem na caixa, trancá-la com sua chave, enviá-la através de um meio qualquer e ter certeza de que ela só será aberta pelo destinatário. Repare que qualquer uma das chaves pode trancar a caixa (codificar a mensagem), usando a outra para abri-la (decodificar a mensagem) [2].

Tratando os sistemas de chave pública de forma mais genérica, dado um par de chaves (d, e) e um inteiro M, devemos ter as seguintes relações [30]:

- Denotando por $D()$ a aplicação da chave particular d, que transforma M em $D(M) = C$. Então, $E(C) = M$ onde $E()$ denota a aplicação da chave e. Ou seja, $E(D(M)) = M$ (e é a chave inversa da chave d).
- O cálculo do par de chaves (d, e) é computacionalmente fácil, ou seja, pode ser dado por um algoritmo de tempo polinomial.
- É computacionalmente difícil calcular *d* a partir do conhecimento de *e*.

[1] Este problema é conhecido como *problema da distribuição de chaves.*

[2] Isso não acontece em todos os criptossistemas de chave pública (veja em [12]), mas é uma propriedade referente ao RSA estudado neste documento.

- Os cálculos de $D()$ e $E()$ são computacionalmente fáceis para quem conhece as chaves.
- É computacionalmente difícil calcular $D()$ sem conhecer a chave d.

Os criptossistemas de chaves públicas, além de resolverem o problema da distribuição de chaves que ocorrem nos criptossistemas de chave secreta ou simétricos, permitem solucionar problemas conhecidos como:

- **Autenticação de Destino** - esconder informações sigilosas das pessoas que controlam as linhas de comunicação e os computadores intermediários (provedores), garantindo que só o verdadeiro destinatário consiga ler a informação enviada.
- **Autenticação da Origem** - evitar que uma pessoa mal-intencionada personalize ou falsique a identidade do emissor enviando informação para o destinatário, ou seja, o destinatário quer ter certeza de que foi o verdadeiro emissor que enviou a informação.
- **Detecção de Integridade de Informação** - Evitar que uma pessoa mal-intenciona- da altere parte da informação que transita na linha de comunicação, antes de chegar ao destinatário ou após o recebimento e armazenamento no computador deste. Assim, o emissor gostaria de detectar se alguma alteração foi feita na linha ou no local de armazenamento dos dados.

Para entendermos como a criptografia de chave pública auxilia na resolução dos problemas acima mencionados, introduziremos a seguir o criptossistema RSA. Para tanto, denotaremos daqui para frente Beto como sendo emissor ou remetente de uma mensagem $M \in Z_N$, Alice como sendo a receptora ou destinatária desta mensagem e Carlos um mal-intencionado que deseja ler informação sigilosa alheia. Denotaremos C como um texto ilegível, resultado da codificação da mensagem M (que supomos legível).

3.2 Criptossistema RSA

Em linhas gerais, podemos descrever três algoritmos que constituem o criptossistema RSA. O primeiro é responsável pela geração do par de chaves pública e particular, o segundo e o terceiro são responsáveis pela criptografia e decriptografia[3] de uma dada mensagem $M \in Z_N$. Estes algoritmos são enunciados a seguir:

[3]O significado dos termos criptografia, decriptografia, codificação e decodificação são encontrados no glossário.

Geração de Chaves - O algoritmo de geração de chaves utiliza um parâmetro de segurança n como entrada, que indica o tamanho em bits do módulo RSA (explicado adiante). Para obter um par de chaves, um usuário ou uma entidade U calcula através de um algoritmo probabilístico [30] dois primos p e q de $\lfloor n/2 \rfloor$ bits [4] e um inteiro $N \leftarrow pq$ conhecido como módulo RSA (assim N possui n bits) . A seguir, U escolhe um $e \in Z$ relativamente primo a $\phi(N) = (p-1)(q-1)$. O inteiro e é chamado expoente de criptografia, e é normalmente fixado com o valor 65537 [25]. A chave pública de U é dada por $< N, e >$ e a chave particular de U é dada por $< N, d >$, satisfazendo *ed* = 1 mod $\phi(N)$. Normalmente a chave pública $< N, e >$ é enviada para uma autoridade certificadora, que gera um certificado para esta.

Criptografia - Para enviar um texto T criptografado para Alice, Beto primeiro o formata usando o padrão PKCS#1 (veja seção 5.2) obtendo assim uma mensagem M pertencente ao anel Z_N. A seguir, Beto obtém $< N_A, e_A >$ (chave pública da Alice) através da própria Alice ou de um repositório público e codifica M, fazendo $C \leftarrow M^{e_A} \ mod \ N_A$ (figura 3.1). A codificação usando a chave particular de Beto é efetuada da mesma forma e o texto gerado por esta é conhecido como mensagem assinada por Beto, sendo a verificação da assinatura a operação de decodificação, usando a chave pública correspondente (figura 3.2).

Decriptografia - Para decodificar C, Alice usa sua chave particular $< N_A, d_A >$ para calcular a e-ésima raiz de C, fazendo $M \leftarrow C^{d_A} \ mod \ N_A$ (figura 3.1). Como d_A e N_A são números grandes (cada um com aproximadamente n bits), a decriptografia exige um alto tempo computacional. A seguir, o processo de formatação do algoritmo de criptografia é revertido obtendo T através de M.

Figura 3.1: Beto codifica a mensagem formatada M com a chave pública $< N_A, e_A >$ da Alice. Carlos intercepta o texto ilegível C, enquanto Alice recupera M através de sua chave particular $< N_A, d_A >$.

[4] O símbolo $\lfloor x \rfloor$ denota o maior inteiro menor ou igual a x. Lê-se: chão de x.

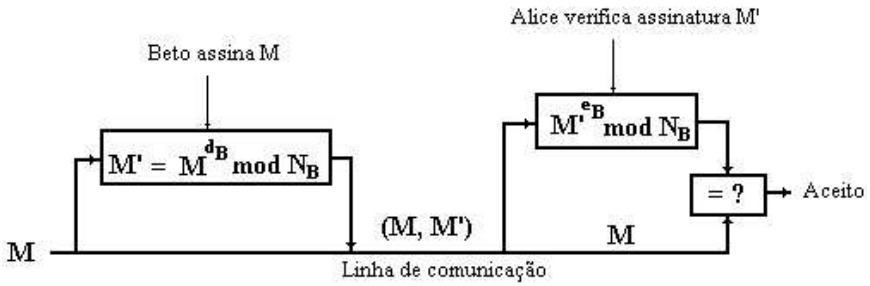

Figura 3.2: Beto aplica sua chave particular $< N_B, d_B >$ gerando a mensagem assinada M'. A seguir, envia o par (M, M') para Alice que calcula um $T = M'^{e_B} \ mod \ N_B$ e o compara com M, validando a assinatura se forem iguais.

RSA - Exemplo Numérico

Geração das Chaves:

$p = 11, \ q = 13$: Inicialmente escolhemos 2 primos p e q

$N = pq = 143, \ \phi(N) = (p-1)(q-1) = 120$: A seguir calculamos N e $\phi(N)$ utilizando p e q

$e = 7$: Escolhemos um e relativamente primo a $\phi(N)$

$d = e^{-1} \ mod \ \phi(N) = 103$: Calculamos a inversa de e módulo $\phi(N)$

Chave pública $< N, e >=< 143, 7 >$

Chave particular $< N, d >=< 143, 103 >$

Criptografia:

Para uma mensagem $M = 15$

$C = M^e \ mod \ N$: Aplicamos a chave pública em M

$C = 15^7 \ mod \ 143 = 115$

Decriptografia:

$M = C^d \ mod \ N$: Aplicamos a chave particular, inversa da chave pública em C

$M = 115^{103} \ mod \ 143 = 15$

3.3 Método de Quisquater-Couvreur

O método de *Quisquater-Couvreur* proposto em [6] é uma técnica utilizada na decriptografia RSA que faz uso do Teorema Chinês do Resto. O método consiste em calcular $M_p = C^d \ mod \ p$ e $M_q = C^d \ mod \ q$ (com p, q e d de aproximadamente $(logN)/2$ bits), obtendo M através da combinação de M_p e M_q pelo TCR (Teorema 2.4.1). Para um resultado ainda melhor, quando calculamos M_p e M_q, podemos reduzir o expoente particular d módulo $(p-1)$ e módulo $(q-1)$. Em outras palavras, podemos calcular $M_p = C^{d_p} \ mod \, p$ e $M_q = C^{d_q} \ mod \, q$, onde $d_p = d \ mod \ (p-1)$ e $d_q = d \ mod \ (q-1)$. O tempo de execução do método de *Quisquater-Couvreur* é cerca de 4 vezes mais rápido que o tempo gasto pela aplicação de $C^d \ mod \ N$ diretamente (a demonstração desse resultado pode ser vista na seção 6.2). Essa técnica ficou tão popular que pode ser considerada o padrão de implementação atual e neste documento será referenciada através do termo RSA QC, deixando a expressão RSA tradicional para o criptossistema da forma como foi criado por Rivest, Shamir e Adleman.

RSA QC - Exemplo Numérico

Geração das Chaves:

$p = 11, \ q = 13$: Inicialmente escolhemos 2 primos p e q

$N = pq = 143, \ \phi(N) = (p-1)(q-1) = 120$: A seguir calculamos N e $\phi(N)$ utilizando p e q

$e = 7$: Escolhemos um e relativamente primo a $\phi(N)$

$d = e^{-1} \ mod \ \phi(N) = 103$: Calculamos a inversa de e módulo $\phi(N)$

Chave pública $< N, e >=< 143, 7 >$

Chave particular $< N, d >=< 143, 103 >$

Criptografia:

Para uma mensagem $M = 15$

$C = M^e \ mod \ N$: Aplicamos a chave pública em M

$C = 15^7 \ mod \ 143 = 115$

Decriptografia:

Reduzimos d módulo $(p-1)$ e $(q-1)$

e fazemos:

$M_p = 115^3 \ mod \ 11$: Aplicamos d módulo $(p-1)$ a C

$M_q = 115^7 \ mod \ 13$: Aplicamos d módulo $(q-1)$ a C

aplicando a equação 2.2 obtemos,

$M = (M_q(p)(p^{-1}\ mod\ q)$
$+M_p(q)(q^{-1}\ mod\ p)\ \ mod\ pq$: Aplicando o TCR temos M de M_p e M_q
$M = (115^7(11)(6) + 115^3(13)(6))\ mod\ 143$
$M = 15$

3.4 Correção e Segurança do RSA

A seguir, provaremos que o criptossistema RSA realmente funciona, mostrando que $(M^e)^d = M\ mod\ N$, onde *ed* = 1 mod $\phi(N)$. A prova para M $\in Z_N^*$ é muito simples e é descrita a seguir:

$$(M^e)^d = M^{k\phi(N)+1} \text{ mod N}$$

Vale lembrar que, pela definição de congruência, se *ed* = 1 mod $\phi(N)$, então *ed* - 1 = k$\phi(N)$, logo *ed* = k$\phi(N)$ + 1. Continuando:

$$\begin{aligned}(M^e)^d &= (M^{\phi(N)})^k M\ mod\ N \\ &= M\ mod\ N \quad \rightarrow\ Pelo\ Corol\acute{a}rio\ 2.2.1\end{aligned}$$

Mostraremos, entretanto, que o RSA funciona quando M $\in Z_N$ e não somente quando M $\in Z_N^*$. Para isso, podemos provar que $(M^e)^d = M\ mod\ p$ e $(M^e)^d = M\ mod\ q$ (onde $N = pq$ e p, q são dois primos distintos), e a seguir resolvemos o sistema com o TCR. Como o cálculo é análogo a ambos os primos p e q, faremos apenas um deles.

Sabemos que *ed* = 1 + k$\phi(N)$ e, portanto, *ed* = 1 + k(p - 1)(q - 1) (Veja a definição 2.2.3), assim temos que:

$$\begin{aligned}(M^e)^d &= M^{1+k(p-1)(q-1)}\ mod\ p \\ &= M(M^{(p-1)})^{k(q-1)}\ mod\ p\end{aligned}$$

Considerando que o $mdc(M, p) = 1$, [5] temos pelo Teorema 2.2.1 (Euler) que $M^{\phi(p)} \equiv 1\ mod\ p$ (e como p é primo temos que $\phi(p) = p - 1$) e, assim, como desejávamos:

[5] Note que caso contrário p divide M, ou seja $M = 0\ mod\ p$ e assim $(M^e)^d = M\ mod\ p$ trivialmente.

$$(M^e)^d = M(1)^{k(q-1)}\ mod\ p$$
$$= M\ mod\ p$$

Da mesma forma podemos mostrar que $(M^e)^d = M\ mod\ q$. Ou seja, sabemos que $(M^e)^d = M\ mod\ p$ e $(M^e)^d = M\ mod\ q$ e pela equação 2.2 concluímos que $(M^e)^d = M\ mod\ N$, como queríamos.

Ao analisar a segurança do criptossistema RSA, percebemos que este só pode ser considerado seguro caso seja inviável tanto calcular o expoente particular *d* quanto extrair a e-ésima raiz módulo N, quando apenas *e* e N são conhecidos. Ora, a única maneira conhecida para o cálculo de *d* é usar o algoritmo de Euclides estendido a *e* módulo $\phi(N)$, mas para calcular $\phi(N)$ precisamos fatorar N e obter p e q. Por outro lado, não conhecemos um algoritmo eficiente capaz de calcular a e-ésima raiz módulo N, o que nos leva a verificar a segurança do criptossistema RSA através da dificuldade gerada pela fatoração de N. Mesmo assim, alguém poderia propor os seguintes ataques:

1 - Calcular $\phi(N)$ sem fatorar N.

2 - Determinar *d* sem fatorar N e sem calcular $\phi(N)$.

3 - Calcular um *d'* equivalente a *d*.

Porém, como é mostrado em [14, 30], se obtivéssemos tais resultados, então por conseqüência, teríamos um algoritmo viável computacionalmente para fatorar N, o que implica que estes ataques são tão ou mais dispendiosos computacionalmente que o melhor algoritmo de fatoração conhecido.

Devido à demonstrada importância da fatoração para a garantia de sigilo do criptossistema, introduziremos a seguir os dois melhores algoritmos com tal finalidade conhecidos atualmente: o *Number Field Sieve* e o *Elliptic Curve Method*. O *Number Field Sieve*, ou também conhecido como NFS, é atualmente o algoritmo de fatoração mais rápido e seu tempo de execução pode ser dado assintoticamente por:

$$NFS[N] = exp((1.923 + o(1))(logN)^{1/3}(loglogN)^{2/3})$$

O tempo de execução assintótico para o *Elliptic Curve Method* (também conhecido como ECM) é dado por:

$$ECM[N, p_1] = (logN)^2 exp((\sqrt{2} + o(1))(logp_1)^{1/2}(loglogp_1)^{1/2})$$

Podemos observar que o tempo de execução do NFS depende só do tamanho do N que estamos tentando fatorar, ao passo que o tempo de execução do ECM não depende só de N, mas também do tamanho do menor fator primo deste, descrito na equação como p_1. Como implicação deste fato, o ECM pode ser muito mais rápido que o NFS na fatoração de N, para o caso particular em que N seja constituído de um primo muito pequeno. Ou seja, para maximizar o tempo de execução do ECM, precisamos maximizar também o tamanho do menor fator primo de N, por isso, é recomendado utilizar *primos balanceados*. Para a garantia de boas chaves devemos saber também qual é o limite de atuação, tanto do NFS quanto do ECM, para um dado módulo N (ou seja, o tamanho de N para o qual seja computacionalmente inviável a utilização destes algoritmos) supondo o uso de primos balanceados. Atualmente se usarmos primos maiores que 256 bits estamos imunes aos dois algoritmos de fatoração [25].

Para se ter uma idéia da inviabilidade computacional proporcionada pela fatoração de números inteiros, para fatorar um módulo de 428 bits (129 algarismos decimais), estima-se que levaríamos cerca de 5 mil *MIPS* anos [30]. Apostando na segurança oferecida por seu criptossistema, os laboratórios RSA [18] divulgam em seu sítio um desafio para a fatoração de módulos de diversos tamanhos. Em 1999, um grupo de pesquisadores estabeleceu um recorde de fatoração para uma chave RSA de 512 bits. O grupo utilizou o NFS e levou um pouco mais de 7 meses com 292 computadores, onde a grande maioria era formada por estações Sun e computadores pessoais Pentium II, num esforço computacional estimado em 8 mil *MIPS* anos.

Discutimos aqui, a segurança do criptossistema RSA baseado na dificuldade de fatorar N. Lembramos, porém, que nem todos os ataques têm como objetivo a fatoração ou o conhecimento da chave particular. Alguns, por exemplo, buscam ler somente uma mensagem específica. Tendo em vista a segurança de um modo mais geral, na próxima seção descreveremos alguns dos principais ataques relatados sobre o RSA.

3.5 Ataques mais conhecidos ao RSA

Objetivando minimizar o tempo gasto pelo criptossistema RSA, alguém poderia propor, por exemplo, a redução no valor dos expoentes público e particular, reduzindo assim o tempo de criptografia/verificação de assinatura e de decriptografia/geração de assinatura; ou ainda, utilizar um único módulo N para inúmeras chaves a fim de diminuir o tempo de geração destas. Apesar de alcançar o resultado desejado, estas medidas comprometem a segurança das mensagens.
A seguir descrevemos os principais ataques que atuam tanto sobre a má escolha de parâmetros, quanto na inexperiência do usuário ou conhecimento da chave pública.

3.5.1 Expoente Público Pequeno (*Low Public Exponent*)

Enunciamos aqui o ataque mais conhecido sobre expoentes públicos pequenos, também denominado como *Hastad Broadcast Attack* [2, 11]. Suponha que Beto deseja enviar uma mensagem M criptografada para um número k de participantes $P_1, P_2, ..., P_k$ e que cada participante possua sua própria chave RSA $< N_i, e_i >$. Assumindo que M é menor que todos os N_{is}, Beto codifica M usando o expoente público e_i e envia o i-ésimo C para P_i $(1 \leq i \leq k)$. Com má-intenção, Carlos pode coletar os k textos codificados ($C_1, C_2, ..., C_k$) através da linha de comunicação ou local de armazenamento.

Supondo que todos os expoentes e_i são iguais a 3, podemos mostrar de uma maneira bem simples que Carlos pode recuperar M se $k \geq 3$. Com posse de C_1, C_2, C_3, onde $C_1 = M^3 \ mod \ N_1$, $C_2 = M^3 \ mod \ N_2$, $C_3 = M^3 \ mod \ N_3$ tal que o $mdc(N_i, N_j) = 1$ para todo $i \neq j$ (caso contrário, pode-se fatorar algum dos N_i's), Carlos pode aplicar o Teorema Chinês do Resto (Teorema 2.4.1) obtendo um $C' \in Z_{N_1 N_2 N_3}$ tal que $C' = M^3$ sobre os inteiros. E assim, pode recuperar M calculando a raiz cúbica de C'. Mais genericamente, se todos os expoentes públicos são iguais a *e*, Carlos pode recuperar M desde que $k \geq e$ e o expoente público *e* utilizado seja pequeno.

Para impedir a eficácia deste e de outros ataques sobre expoentes públicos pequenos é recomendado utilizar o valor $e = 2^{16} + 1 = 65537$. Além de seguro [2], este valor diminui drasticamente o número de multiplicações necessárias ao algoritmo de exponenciação, se comparado a um valor de e aleatório menor ou igual a $\phi(N)$ (17 contra 1000 multiplicações [2]). Como outra medida de prevenção, poderíamos concatenar algo na mensagem antes da

criptografia. Por exemplo, se M é de m bits, Beto poderia enviar $M_i = i2^m + M$ para o participante P_i, fazendo com que Carlos obtivesse diferentes mensagens, o que tornaria o ataque ineficaz. Infelizmente Hastad mostrou em [11], que uma concatenação linear não é segura e para solucionar este problema deve-se utilizar uma concatenação aleatória (essa técnica é conhecida como *salting*).

3.5.2 Expoente Particular Pequeno (*Low Private Exponent*)

Wiener mostrou em [32], que se o expoente particular d for menor que $N^{1/4}$, onde N é o módulo RSA, então pode-se obter total conhecimento de d analisando somente a chave pública correspondente. O ataque é baseado em aproximações da relação entre e, d, N, ϕ e ocorre quando $q < p < 2q$ e $d < \frac{1}{3}N^{1/4}$.

Como já é conhecido, $ed = 1\ mod\ \phi(N)$, ou seja, existe um k tal que $ed - k\phi(N) = 1$. Logo,

$$\left|\frac{e}{\phi(N)} - \frac{k}{d}\right| = \frac{1}{d\phi(N)}.$$

Pela equação anterior, vemos que $\frac{k}{d}$ é uma aproximação de $\frac{e}{\phi(N)}$ e, embora não conhecemos $\phi(N)$, podemos usar N para aproximá-lo. Como $\phi(N) = N-p-q+1$ e $p+q-1 < 3\sqrt{N}$, temos que $|N - \phi(N)| < 3\sqrt{N}$. E, finalmente, utilizando estas aproximações, podemos recuperar d.

O estudo deste ataque levou ao surgimento de muitos outros que exploram o tamanho pequeno dos expoentes no RSA. Mais recentemente, D. Boneh e G. Durfee em [7] melhoraram o resultado acima mostrando que é possível obter o expoente particular d, conhecendo somente a chave pública, se este for menor que $N^{0.292}$. A descrição completa do ataque de Wiener não será apresentada neste documento. Para uma visão mais detalhada veja [2, 7, 32].

3.5.3 Ataque com Parte da Chave Exposta (*Partial Key Exposure Attack*)

Quando alguns bits do expoente particular d são conhecidos (mais precisamente pelo menos 1/4 dos bits mais ou menos significativos) e o expoente público $e < \sqrt{N}$, é possível recuperar

totalmente o expoente particular d. Este ataque é uma conseqüência do Teorema de Coopersmith, que afirma que dados $\lceil n/4 \rceil$ bits mais ou menos significativos de p, podemos reconstruir d em tempo $O(e\ log\ e)$.

Como no ataque anterior, sabemos que $ed - k\phi(N) = 1$, ou seja, $ed - k(N - p - q + 1) = 1$ e que $0 < k \leq e$ pois $d < \phi(N)$. Reduzindo este resultado módulo $2^{n/4}$ e fazendo $q = N/p$ obtemos:

$$(ed)p - kp(N - p + 1) + kN = p \pmod{2^{n/4}}$$

Logo, se conhecemos os n/4 bits menos significativos de d, conhecemos também quanto vale $ed\ mod\ 2^{n/4}$, obtendo assim uma equação em função de k e p. Como $0 < k \leq e$, pode-se calcular todos os possíveis valores de k e, com esses valores, obter o conjunto formado por $p\ mod\ 2^{n/4}$. Finalmente, o teorema de Coppersmith pode ser usado para fatorar N através da reconstrução de d.

À primeira vista pode parecer inviável a aplicação deste método, já que, parece difícil conseguir parte do expoente particular. Na verdade, existem algumas fórmulas para calcular uma boa aproximação d' para d quando e é pequeno [2] e para a maioria dos $d's$ calculados, metade dos bits mais significativos de d são iguais a d', o que torna o ataque eficaz. Mais uma vez, constatamos a importância da escolha cuidadosa do expoente público e.

3.5.4 Módulo Comum (*Commom Modulus*)

A fim de diminuir o tempo computacional gasto ao gerar módulos diferentes para cada usuário, alguém poderia fixar o mesmo $N = pq$ para todos. Uma autoridade certificadora confiável forneceria para um usuário i um único par e_i, d_i. Logo, este usuário possuiria uma chave pública dada por $< N, e_i >$ e uma chave particular dada por $< N, d_i >$. Esta idéia parece funcionar - um texto codificado $C = M^{e_A}\ mod\ N$ que deve ser enviado para Alice não poderia ser decodificado por Beto, pois este não possui d_A. Todavia, como todos os usuários possuem o mesmo módulo, Beto pode usar seus próprios expoentes e_B, d_B para fatorar N e após isso, estará apto a conseguir a chave particular d_A da Alice e de qualquer outro usuário i, através da chave pública correspondente. Esta observação nos mostra que um módulo RSA não deve ser usado para mais que uma entidade.

3.5.5 Ocultamento (*Blinding*)

Seja $< N, d >$ a chave particular de Beto e $< N, e >$ sua chave pública, suponhamos que Carlos deseja obter uma assinatura de Beto na mensagem $M \in Z_N^*$. É claro que Beto, após ler a mensagem, pode se recusar a assiná-la. Porém, se Carlos insistir escolhendo um número aleatório $r \in Z_N$ e calcular $M' = r^e M \ mod \ N$, ele pode convencer Beto a assinar a aparentemente inocente mensagem M', obtendo uma assinatura S'. O que Beto não sabe é que $S' = (M')^d \ mod \ N$ e que agora Carlos pode calcular $S = S'/r \ mod \ N$, conseguindo assim uma assinatura de Beto na mensagem M escolhida por ele. Isso ocorre pois,

$$S = (S')/r = (r^e M)^d/r = r(M^d)/r = M^d \ (modN)$$

Esta técnica, conhecida como *blinding*, capacita Carlos a obter uma assinatura válida na mensagem escolhida, pedindo a Beto que assine uma mensagem aleatória. Beto não tem informação sobre o que está realmente assinando. Felizmente, a maioria dos esquemas de assinaturas aplicam uma função de *hash* na mensagem M antes de assinar e, portanto, o ataque não é uma séria preocupação.

3.6 Considerações sobre o criptossistema RSA

Na seção 3.2 vimos que o criptossistema RSA é baseado na exponenciação módulo números grandes (centenas ou mesmo milhares de bits). Este tipo de operação é excessivamente lenta se comparada as operações de deslocamento de bits, "ou exclusivo" ou até mesmo de soma e subtração, utilizadas pelos criptossistemas simétricos. Em ambientes com recursos limitados essa demora pode se tornar um problema, afetando o desempenho do sistema como um todo. Um exemplo disso seria aplicações de decriptografia RSA em um pequeno computador de mão tal como o PalmPilot III que pode levar cerca de 30 segundos [25]. Da mesma maneira, em um servidor SSL, a baixa velocidade da decriptografia reduz significativamente o número de requisições por segundo que o servidor pode manipular. O que se faz geralmente é usar um hardware com a finalidade exclusiva de melhorar estas operações [6].

[6] Os coprocessadores RSA atuais podem executar cerca de 10.000 decriptografias RSA por segundo (usando um módulo de 1024 bits) [25].

Mencionamos como exemplo, a decriptografia e geração de assinatura, já que o expoente público e pode ser utilizado com valores pequenos (por exemplo, $e = 65537$), fazendo com que o tempo da criptografia e a verificação de assinatura seja menor. O mesmo não pode ser feito com relação ao expoente d (normalmente da ordem de N), pois caso contrário, o sistema RSA se torna inseguro (veja na seção 3.5). Nestas condições temos um alto tempo computacional para o cálculo da decriptografia e geração de assinatura, se comparado ao tempo de criptografia e verificação de assinatura.

Entretanto, nem em todas as aplicações é desejável ter o comportamento referido acima. Na verdade, muitas vezes é necessário obter o comportamento oposto. Por exemplo, se um telefone celular precisa gerar uma assinatura RSA que será posteriormente verificada em um servidor rápido, é desejável que a geração de assinatura seja mais rápida que a verificação. Da mesma maneira, navegadores *Web* que utilizam a criptografia RSA através do protocolo SSL normalmente possuem ciclos ociosos para queimar, ao passo que servidores SSL responsáveis pelas decriptografias estão sobrecarregados [3, 25]. Neste último caso, para decriptografar um texto C um servidor *Web* usa sua chave particular $< N, d >$ para obter $C^d \ mod \ N$ e, como foi mencionado acima, alto tempo computacional é exigido para esta operação.

Nos últimos anos alguns algoritmos foram propostos com objetivo de diminuir o tempo computacional da decriptografia/geração de assinatura. A maioria deles sugerem para isso uma maneira de diminuir o tempo da exponenciação sobre o módulo RSA. As propostas que consideramos mais significativas são descritas no próximo capítulo.

CAPÍTULO

4

Variações do RSA

A seguir, estudaremos algumas propostas de otimização do criptossistema RSA que visam melhorar o desempenho da geração de assinatura e decriptografia tanto do criptossistema original quanto do aqui denotado RSA QC.

A primeira variação a ser estudada é conhecida como RSA em Lote ou *Batch RSA*, caracterizada por fazer diversas decriptografias com o custo de aproximadamente uma. A segunda e a terceira variação são baseadas no uso do módulo nas formas $N = \prod_{i=1}^{k} p_i$ (para $k > 1$) e $N = p^k q$ (para $k \geq 1$), respectivamente, e por esse motivo são conhecidas como RSA com múltiplos fatores. A quarta variação é conhecida como RSA Rebalanceado ou *Rebalanced RSA* e aumenta o desempenho da decriptografia, deslocando a maior parte do trabalho para o algoritmo de criptografia. Ao final do capítulo apresentamos um esquema baseado na mistura das técnicas utilizadas pelo RSA com múltiplos fatores (módulo na forma $N = pqr$) e do RSA Rebalanceado.

4.1 RSA em Lote (*Batch RSA*)

O método denominado RSA em Lote surgiu devido à observação de Fiat [9] de que, quando usamos expoentes públicos pequenos e_1 e e_2, é possível decriptografar dois textos com o custo de aproximadamente um. Suponha que C_1 é o texto codificado obtido da criptografia de M_1 usando a chave pública $< N, 3 >$ e que C_2 é o texto codificado obtido da criptografia de M_2 usando a chave pública $< N, 5 >$. Para decodificar C_1 e C_2, precisamos calcular $C_1^{1/3} = M_1 \ mod \ N$ e $C_2^{1/5} = M_2 \ mod \ N$ (lembrando que a *e*-ésima raiz de C em Z_n pode ser obtida por $C^d \ mod \ N$). Fiat percebeu que fazendo $M = (C_1^5.C_2^3)^{1/15}$ obtém-se

$$M_2 = C_2^{1/5} = \frac{M^6}{C_1^2.C_2} \ mod \ N \qquad \text{e} \qquad M_1 = C_1^{1/3} = \frac{M}{M_2} \ mod \ N$$

e assim, é possível decodificar C_1 e C_2 com o custo de aproximadamente uma raiz 15° de M. É claro que alguns outros cálculos são necessários, mas para expoentes e_1 e e_2 pequenos (por exemplo 3 e 5) esses cálculos são desprezíveis [1]. Outro fato que deve ser apontado é o de que a decriptografia em lote só é possível se utilizarmos expoentes públicos distintos entre si, ou seja, a técnica não é válida para $C_1^{1/3}$ e $C_2^{1/3}$. Este fato foi mostrado usando teoria de Galois por D. Boneh e H.Shacham no Apêndice A de [3]. Para generalizar a obsevação acima para a decriptografia de b textos codificados, Fiat utilizou a estrutura de uma árvore binária completa, com a propriedade de que todo nó interno possui dois filhos. Esta generalização pode ser enunciada como segue:

Para b expoentes públicos $e_1, ..., e_b$ distintos e primos entre si, compartilhando o mesmo módulo $N = pq$ e b mensagens codificadas $C_1, ..., C_b$ (onde cada C_i foi codificada usando $< N, e_i >$), desejamos decodificar simultaneamente $M_i = C_i^{1/ei}$ para 1 ≤ i ≤ b.

Definido o problema, descrevemos agora os algoritmos de geração de chaves, criptografia e decriptografia:

- **Geração de Chaves** - Pode ser usado o mesmo algoritmo de geração de chaves do RSA tradicional. É claro que devem ser evitados expoentes públicos grandes a fim de aumentar

[1] Lembramos que a 15^a raiz de M é calculada fazendo $M^{d_1 d_2} \ mod \ N$, e que os expoentes particulares devem ser grandes para garantir a segurança do sistema (veja seção 3.5.2), exigindo um esforço computacional muito maior que o apresentado pela utilização de expoentes públicos pequenos (veja seção 3.5.1)

o desempenho do algoritmo de decriptografia. A chave pública é dada por $< N, e >$ e a chave particular por $< N, d >$.

- **Criptografia** - Dada uma chave pública $< N, e >$ e uma mensagem $M \in Z_N$, Beto criptografa M exatamente como no RSA tradicional, ou seja, $M^e \ mod \ N$.

- **Decriptografia**- Para facilitar o entendimento deste algoritmo denotaremos abaixo nós subscritos por L ou R (de *left* e *right*) referindo-se ao valor correspondente do filho esquerdo ou direito destes, respectivamente. Por exemplo, para uma variável M, M_R é o valor de M no nó do filho direito. O processo de decriptografia proposto por Fiat pode ser dividido em três fases [3, 9], como apresentado abaixo:

  Cálculo do Produto - Nesta fase, percorremos a árvore calculando o produto das mensagens codificadas C_i e formando na raiz da árvore de *batch* o valor $C = \prod_{i=1}^{b} C_i^{e/e_i}$, onde $e = \prod_{i=1}^{b} e_i$. Inicialmente, associamos a cada nó folha um expoente público $E \leftarrow e_i$. O E de cada nó interno é obtido através do cálculo do produto de seus filhos , $E \leftarrow E_L E_R$. (Note que o nó raiz E será igual a e, ou seja, o produto de todos os expoentes públicos). Cada C_i é armazenada em uma variável C no nó folha que contém seu e_i correspondente. Os $C's$ restantes (nós internos), são calculados em um processo *bottom-up* na árvore usando o seguinte passo recursivo (veja figura 4.1).

  $$C \leftarrow C_L^{E_R} C_R^{E_L}$$

  Exponenciação - Ao término da fase anterior, o nó raiz contém $C = \prod_{i=1}^{b} C_i^{e/e_i}$. Nesta fase, a e-ésima raiz deste C é extraída [2]. A exponenciação resulta em $C^{1/e} = \prod_{i=1}^{b} C_i^{1/e_i}$, e este valor é então armazenado em M no nó raiz. Ou seja, M contém o produto das decriptografias dos $C's$ (veja figura 4.2).

  Fatoração do Produto - Agora, buscamos fatorar o produto M em subprodutos M_L e M_R, e eventualmente, em mensagens decodificadas M_i nas folhas. O método proposto por Fiat para efetuar esta fatoração é baseado na solução de congruências

[2] Note que este é o ponto no qual é necessário o conhecimento das chaves particulares.

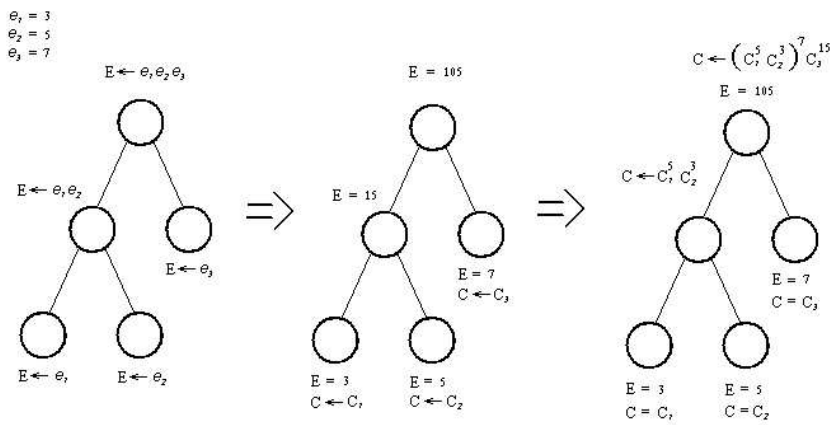

Figura 4.1: Primeira Fase - Cálculo do Produto (Exemplo para os expoentes $e_1 = 3$, $e_2 = 5$, $e_3 = 7$).

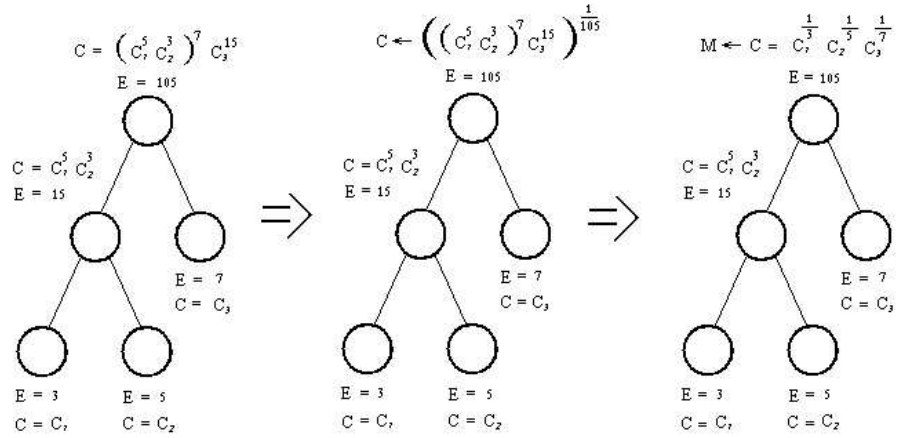

Figura 4.2: Segunda Fase - Exponenciação.

utilizando o Teorema Chinês do Resto (Teorema 2.4.1), mais propriamente utilizando a equação 2.2. Em cada nó interno obtemos um X satisfazendo o par de congruências:

$$X \equiv \begin{cases} 0 & mod\ E_L \\ 1 & mod\ E_R \end{cases}$$

Agora, calculamos dois números auxiliares, X_L e X_R, definidos em cada nó como segue:

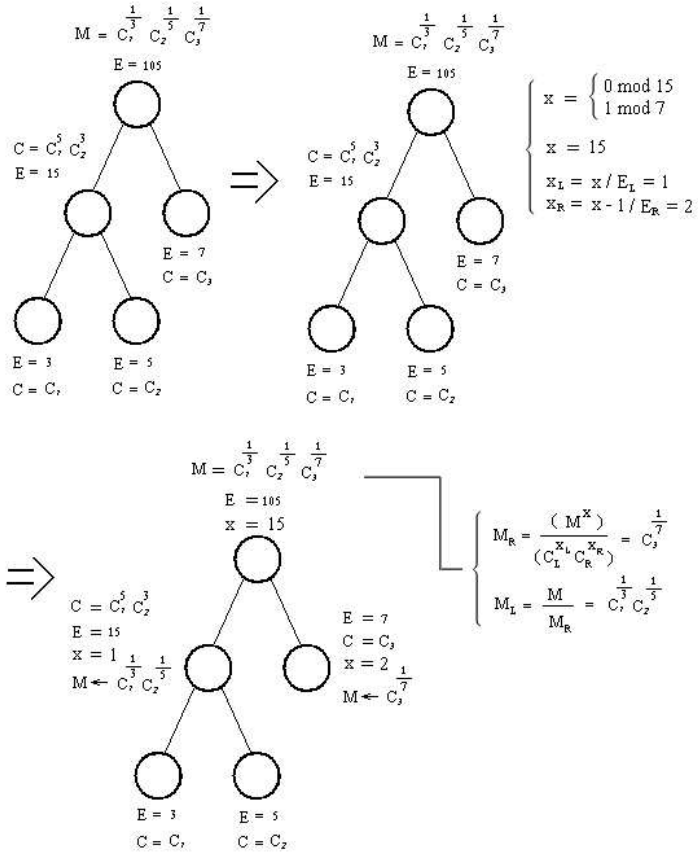

Figura 4.3: Terceira Fase - Fatoração do Produto.

$$X_L = X/E_L \qquad X_R = (X-1)/E_R$$

Ambas as divisões são feitas sobre inteiros e X, X_L e X_R são tais que, cada nó interno, M^X é igual a $C_L^{X_L} C_R^{X_R} M_R$ (demonstração em [9]). Então, podemos aplicar o seguinte passo recursivo:

$$M_R \leftarrow M^X/(C_L^{X_L} C_R^{X_R}) \qquad M_L \leftarrow M/M_R$$

Ao final desta fase, cada folha M_i contém a decriptografia do C_i colocado nela inicialmente (veja figuras 4.3 e 4.4).

Só uma exponenciação de grande valor foi necessária, ao invés de b delas. Outras quatro

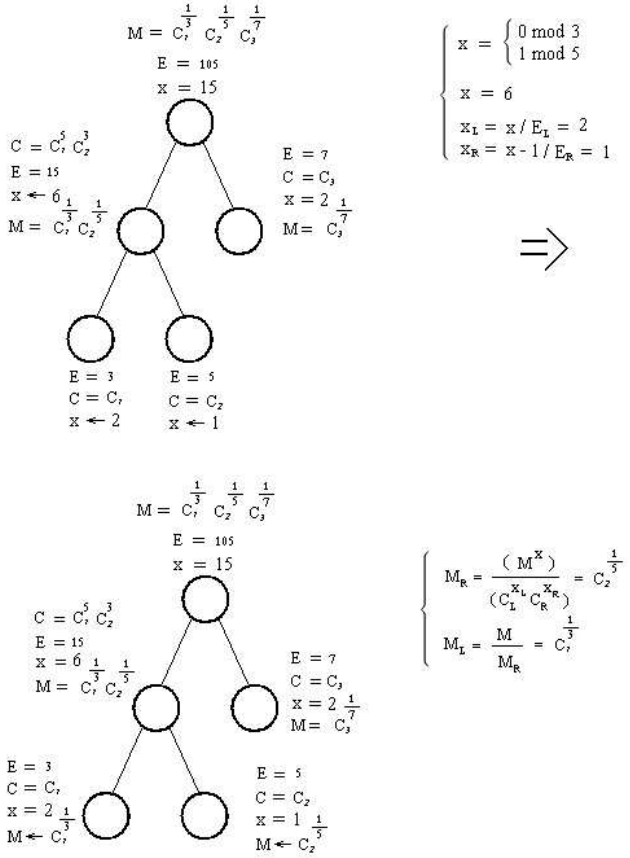

Figura 4.4: Terceira Fase - Fatoração do Produto (Continuação).

pequenas exponenciações, duas inversões e quatro multiplicações em cada um dos $b-1$ nós internos, são necessárias.

RSA em Lote - Exemplo Numérico

Geração das Chaves:

$p = 17,\ q = 23$	: Inicialmente escolhemos 2 primos p e q
$N = pq = 391,\ \phi(N) = (p-1)(q-1) = 352$	: A seguir calculamos N e $\phi(N)$ utilizando p e q
$e_1 = 3$	: Escolhemos um e_1 relativamente primo a $\phi(N)$
$e_2 = 5$	: Escolhemos um e_2 relativamente primo a $\phi(N)$
$d_1 = {e_1}^{-1}\ mod\ \phi(N) = 235$	: Calculamos a inversa de e_1 módulo $\phi(N)$

$d_2 = {e_2}^{-1}\ mod\ \phi(N) = 141$: Calculamos a inversa de e_1 módulo $\phi(N)$

1^a Chave pública $< N, e_1 >$ = $< 391, 3 >$	1^a Chave particular $< N, d_1 >$=$< 391, 235 >$
2^a Chave pública $< N, e_2 >$ = $< 391, 141 >$	2^a Chave particular $< N, d_2 >$=$< 391, 141 >$

Criptografia:

Dada as mensagens $M_1 = 382$ e $M_2 = 168$

$C_1 = {M_1}^{e_1}\ mod\ N = 53$: Aplicamos a 1^a chave pública a M_1

$C_2 = {M_2}^{e_2}\ mod\ N = 155$: Aplicamos a 2^a chave pública a M_2

Decriptografia:

Efetuamos as 3 fases:

$C = ({C_1}^5 {C_2}^3)\ mod\ 391$: Fase 1 - Cálculo do Produto

$M = C^{(d_1 d_2)}\ mod\ 391 = 52$: Fase 2 - Exponenciação

$M_1 = \frac{M^{10}}{C_1^3 C_2^2}\ mod\ N = 382$: Fase 3 - Fatoração do Produto em M_1

$M_2 = \frac{M^6}{C_1^2 C_2}\ mod\ N = 168$ Fatoração do Produto em M_2

4.2 RSA com múltiplos fatores (*Multi-factor RSA*)

Os próximos dois métodos são baseados na modificação da estrutura do módulo RSA. O primeiro utiliza o módulo na forma $N = \prod_{i=1}^{k} p_i$ (para $k > 1$) e, assim, quando N é de 1024 bits e $k = 3$, cada primo é representado com aproximadamente 341 bits. Para este adotaremos o nome RSA com Múltiplos Primos ou *multi-prime RSA* [25]. O segundo, denominado RSA com Múltiplas Potências ou *Multi-power RSA*, utiliza o módulo na forma $N = p^k q$, para $k \geq 1$. Ambos os métodos são totalmente compatíveis com o RSA, considerando que não é conhecido um algoritmo eficiente que diferencie as chaves públicas destes das utilizadas pelo RSA tradicional (onde $N = pq$).

4.2.1 RSA com Múltiplos Primos (*Multi-prime RSA*)

O RSA com Múltiplos Primos foi proposto por um grupo de cientistas [8], que modificaram o módulo RSA constituindo-o de k primos ($N = p_1 p_2 ... p_k$) ao invés dos tradicionais dois

primos p e q. A descrição dos algoritmos de geração de chaves, criptografia e decritografia segue abaixo:

Geração de Chaves - O algoritmo de geração de chaves recebe como entrada o parâmetro de segurança n e um parâmetro adicional k, que indica o número de primos a ser utilizado. O par de chaves pública e particular é gerado através dos passos que seguem:

1 - Inicialmente um usuário ou uma entidade U gera aleatoriamente k primos distintos $p_1, ..., p_k$ cada um com $\lfloor n/k \rfloor$ bits de tamanho. A seguir calcula $N \leftarrow \prod_{i=1}^{k} p_i$. Para um módulo de 1024 bits deve-se usar no máximo $k = 3$ (isto é, $N = pqr$), por razões de segurança consideradas no sexto capítulo.

2 - U escolhe um expoente público e aleatório ou o mesmo utilizado como padrão nas chaves públicas RSA ($e = 65537$) e calcula $d = e^{-1} \; mod \; \phi(N)$. Naturalmente, U deve garantir que e seja relativamente primo a $\phi(N) = \prod_{i=1}^{k}(p_i - 1)$ para que exista a inversa multiplicativa. A chave pública é dada por $< N, e >$ e a chave particular é dada por $< N, d >$ [3].

Criptografia - Dada uma chave pública $< N, e >$ e uma mensagem $M \in Z_N$, Beto criptografa M exatamente como no RSA tradicional, ou seja, $M^e \; mod \; N$ (figura 4.5).

Decriptografia - A decriptografia é feita usando o TCR (Teorema 2.4.1). Seja $d_i = d \, mod \, (p_i - 1)$. Para decriptograr um texto C, Alice primeiramente deve calcular $M_i = C^{d_i} \; mod \; p_i$ para cada i, $1 \leq i \leq k$. A seguir, deve aplicar o TCR nos M_i's para obter $M = C^d \; mod \; N$ (figura 4.5). O TCR faz uso de apenas algumas inversões e multiplicações (k inversões e $2k$ multiplicações no algoritmo clássico), tomando assim, tempo desprezível se comparado com as k exponenciações de $\lfloor n/k \rfloor$ bits.

Observe que este método propõe diminuir o tempo gasto pela exponenciação modular, reduzindo o tamanho do módulo de n para $\lfloor n/k \rfloor$ bits, além de reduzir o tamanho do expoente particular d para aproximadamente $\lfloor n/k \rfloor$ bits. Dessa maneira, ao invés de termos na decriptografia uma única exponenciação utilizando um módulo de n bits e com um expoente particular grande, teremos k exponenciações sobre módulos de $\lfloor n/k \rfloor$ bits e sobre um expoente particular

[3] Para adaptação com o PKCS #1 veja seção 6.4.

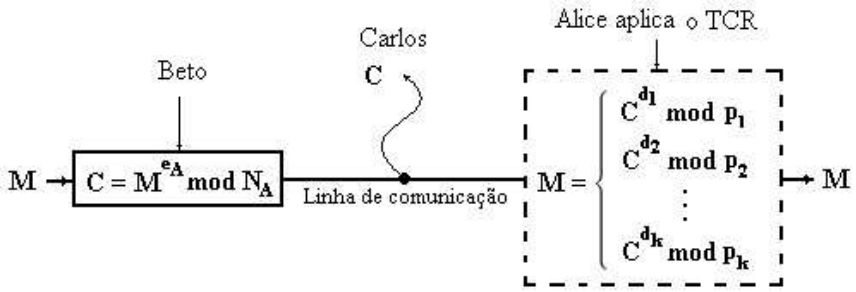

Figura 4.5: Beto codifica M usando a chave pública $< N_A, e_A >$ da Alice e envia o texto codificado C para ela, que a decodifica usando o TCR no sistema de congruências $M_i = C^{d_i} \; mod \; p_i$, onde $d_i = d \; mod \; (p_i - 1)$.

reduzido, o que é mais eficiente (a prova desta afirmação pode ser encontrada na seção 6.4).

RSA com Múltiplos Primos - Exemplo Numérico

Geração das Chaves ($k = 3$):

$p_1 = 3, \; p_2 = 5, \; p_3 = 7$: Inicialmente escolhemos 3 primos p_1, p_2 e p_3

$N = p_1 p_2 p_3 = 105$: A seguir calculamos N e $\phi(N)$

$\phi(N) = (p_1 - 1)(p_2 - 1)(p_3 - 1) = 48$ utilizando p_1, p_2 e p_3

$e = 5$: Escolhemos um e relativamente primo a $\phi(N)$

$d = e^{-1} \; mod \; \phi(N) = 29$: Calculamos a inversa de e módulo $\phi(N)$

Chave pública $< N, e >=< 105, 5 >$

Chave particular $< N, d >=< 105, 29 >$

Criptografia:

Para uma mensagem $M = 75$

$C = M^e \; mod \; N$: Aplicamos a chave pública a M

$C = 75^5 \; mod \; 105 = 45$

Decriptografia:

Redução modular de d

$d_1 = d \; mod \; (p_1 - 1) = 1$: Reduzimos d módulo $(p_1 - 1)$

$d_2 = d \; mod \; (p_2 - 1) = 1$: Reduzimos d módulo $(p_2 - 1)$

$d_3 = d \; mod \; (p_3 - 1) = 5$: Reduzimos d módulo $(p_3 - 1)$

Calculamos os Mis

$M_1 = C^{d_1} \; mod \; p_1 = 0$: Aplicamos d módulo $(p_1 - 1)$ a C

$M_2 = C^{d_2} \; mod \; p_2 = 0$: Aplicamos d módulo $(p_2 - 1)$ a C

$M_3 = C^{d_3} \; mod \; p_3 = 5$: Aplicamos d módulo $(p_3 - 1)$ a C

aplicando a equação 2.2 obtemos,

$M' = (M_1(p_2)(p_2{}^{-1} \; mod \; p_1)$: Aplicando o TCR obtemos M dos Mis

$+M_2(p_1)(p_1{}^{-1} \; mod \; p_2) \;\; mod \; p_1p_2$

$M' = 0$

$M = (M'(p_3)(p_3{}^{-1} \; mod \; p_1p_2)$

$+M_3(p_1p_2)(p_1p_2{}^{-1} \; mod \; p_3) \;\; mod \; p_1p_2p_3$

$M = 75$

4.2.2 RSA com Múltiplas Potências (*Multi-power RSA*)

No RSA com Múltiplas Potências, proposto por Takagi [29], modificamos a estrutura do módulo RSA de $N = pq$ para $N = p^k q$. Os algoritmos de geração de chaves, criptografia e decriptografia são descritos abaixo:

Geração das chaves - O algoritmo de geração de chaves recebe como entrada o parâmetro de segurança n e um parâmetro adicional *k*. As chaves são geradas pelos procedimentos subseqüentes:

1 - Inicialmente, um usuário ou uma entidade U gera dois primos aleatórios distintos de $\lfloor \frac{n}{k+1} \rfloor$ bits, p e q, e calcula $N \leftarrow p^k q$.

2 - Usando o expoente público e aleatório ou padrão RSA ($e = 65537$), U calcula $d \leftarrow e^{-1} \; mod \; (p-1)(q-1)$. U deve garantir também que $mdc(e, p) = 1$.

3 - Finalmente, U calcula $d_p \leftarrow d \; mod \; (p-1)$ e $d_q \leftarrow d \; mod \; (q-1)$. A chave pública é $< N, e >$ e a chave particular é $< p, q, d_p, d_q >$.[4]

[4]Para adaptação com o PKCS #1 veja seção 6.5.

Criptografia - Dada uma chave pública $< N, e >$ e uma mensagem $M \in Z_N$, Beto criptografa M exatamente como no RSA tradicional, ou seja, $M^e \ mod \ N$.

Decriptografia - Para decriptografar C, Alice procede da seguinte maneira (figura 4.6) :

1 - Obtém $M_q = M \ mod \ q$ fazendo $M_q = C^{d_q} \ mod \ q$ e $M_p = M \ mod \ p$ fazendo $M_p = C^{d_p} \ mod \ p$. Observe que $M_p^e = C \ mod \ p$ e $M_q^e = C \ mod \ q$.

2 - Usando o algoritmo proposto por T. Takagi em [29], que foi baseado no algoritmo *Hensel lift* [4, 28], Alice constrói um M_p' tal que $(M_p')^e = C \ mod \ p^k$. A vantagem deste método está no fato de que transformar M_p em M_p' pelo algoritmo proposto é muito mais rápido que calcular a exponenciação de $\lfloor \frac{n}{k+1} \rfloor$ bits módulo p^k (veja a próxima seção).

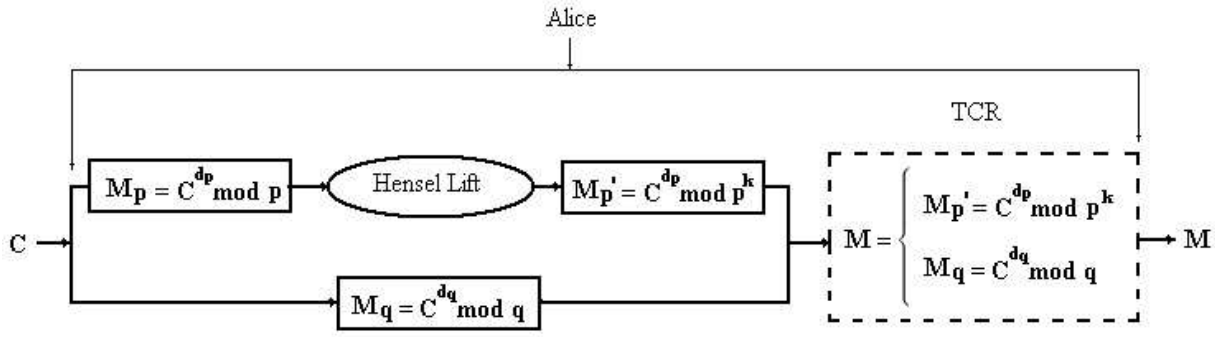

Figura 4.6: Alice recebe o texto codificado C e calcula M_p e M_q. Após isso, através do algoritmo *Hensel Lift* expande M_p para $M_p' \ mod \ p^k$ e usa o TCR para resolver o sistema $M = M_p'$, $M = M_q$.

3 - Usando o TCR (Teorema 2.4.1), Alice calcula o único $M \in Z_N$ tal que $M = M_p' \ mod \ p^k$ e $M = M_q \ mod \ q$. Este M corresponde à decriptografia de C, ou seja, $M = C^d \ mod \ N$.

Apresentamos na próxima seção, a expansão especificada no segundo passo, através do algoritmo proposto por Takagi [29].

4.2.3 Algoritmo para o cálculo de M_p'

Analisando a ordem do grupo $Z_{p^k}{}^*$, vemos que esta pode ser dada por $p^{k-1}(p-1)$, ou seja $p^k - p^{k-1}$, desde que só os múltiplos de p em Z_{p^k} não são relativamente primos a p^k e

existem p^{k-1} destes múltiplos. Esse resultado nos mostra que, se adaptássemos o algoritmo de decriptografia do RSA tradicional para calcular M'_p, não obteríamos ganho na velocidade sobre a decriptografia do RSA QC com $N = pq$ (veja seção 3.3), pelo contrário, desde que para isso teríamos que calcular $M'_p = C^d \ (mod\, p^k)$ com $d = e^{-1} \ (mod\, p^{k-1}(p-1)(q-1))$.

No entanto, pelo método proposto por Takagi em [29] calculamos inicialmente $M_p = C^{d_p} \ (mod\, p)$ e, através do algoritmo *Hensel Lifting* descrito em [4], podemos expandir M_p para $M'_p = M_p \ mod\, p^k$ com um tempo equivalente a calcular M_p com $d = e^{-1} \ (mod\, (p-1)(q-1))$. Para demonstrar esse método, denotaremos o texto criptografado reduzido módulo p^k como C_p (isto é, $C_p = M'^e_p \ (mod\, p^k)$).

Note que M'_p possui a seguinte expansão

$$M'_p = K_0 + pK_1 + p^2K_2 + ... + p^{k-1}K_{k-1} \ \ (mod\, p^k) \tag{4.1}$$

então, se definirmos a função $F_i(X_0, X_1, ..., X_i)$ como

$$F_i(X_0, X_1, ..., X_i) \ = \ (X_0 + pX_1 + ... + p^iX_i)^e,$$

onde i = 0, 1, ..., k - 1 , vemos que $F_{k-1}(X_0, X_1, ..., X_{k-1}) = (X_0 + pX_1 + ... + p^{k-1}X_{k-1})^e$ é equivalente a função que criptografa o texto M'_p na equação 4.1. Reduzindo agora esta equação módulo p^{i+1}, temos a relação

$$F_i(X_0, X_1, ..., X_i) \ = \ F_{i-1} + p^iG_{i-1}X_i \ \ (\text{mod}\ p^{i+1}),$$

onde $F_{i-1} \ = \ (X_0 + pX_1 + ... + p^{i-1}X_{i-1})^e$ e $G_{i-1} \ = \ e(X_0 + pX_1 + ... + p^{i-1}X_{i-1})^{e-1}$ para $i = 0, 1, ..., k-1$. Fazendo $K_0 = C_p^d \ mod\ p$, podemos calcular recursivamente através da relação acima e do texto criptografado C os valores $K_1, ..., K_{k-1}$. Ou seja, para i = 1, K_1 é a solução da seguinte equação linear de X_1:

$$C \ = \ F_0(K_0) \ + \ pG_0(K_0)X_1 \ (mod\, p^2)$$

ou,

$$X_1 = K_1 = (C - F_0(K_0))(pG_0(K_0))^{-1} \ mod\, p^2$$

Para generalizar o resultado acima podemos assumir que $K_1, K_2, ..., K_{i-1}$ já estão calculados, e assim obtemos $F_{i-1}(K_0, K_1, ..., K_{i-1})$ e $G_{i-1}(K_0, K_1, ..., K_{i-1})$ em Z, denotados anteriormente por F_{i-1} e G_{i-1}, respectivamente. Portanto, de uma maneira mais geral, K_i é a solução da seguinte equação linear de X_i:

$$C = F_{i-1} + p^i G_{i-1} X_i \ (mod\ p^{i+1})$$

Após calcular $K_0, K_1, ..., K_{k-1}$ podemos obter $M_p \ (mod\ p^k)$ da equação 4.1 e o texto original $M \ (mod\ p^k q)$, pela aplicação do TCR (Teorema 2.4.1) em $M_p \ (mod\ p^k)$ e $M_q \ (mod\ q)$.

Para finalizar, descrevemos abaixo este algoritmo em pseudo-código [28] (incluindo o cálculo de M_q e o resultado final M).

Função Decript
Entrada: d,p,q,e,k,C
Saída: M

$d_p \leftarrow d\ mod\ (p-1)$: Reduz d módulo $(p-1)$
$d_q \leftarrow d\ mod\ (q-1)$: Reduz d módulo $(q-1)$
$K_0 \leftarrow C^{d_p}\ mod\ p$: K_0 faz o papel de M_p
$M_q \leftarrow C^{d_q}\ mod\ q$: Calcula M_q
$A_0 \leftarrow K_0$
Para i ← 1 até (k - 1) faça : Expande K_0 para M_p (M'_p)
  $F_i \leftarrow A_{i-1}^e$ mod (p^{i+1})
  $E_i \leftarrow$ C - F_i mod (p^{i+1})
  $B_i \leftarrow E_i/p^i \in Z$
  $K_i \leftarrow (eF_i)^{-1} A_{i-1} B_i\ mod\ p$
  $A_i \leftarrow A_{i-1}$ + $p^i K_i \in Z$
$M_p \leftarrow A_{k-1}$: Aplicamos o TCR utilizando
$p_1 \leftarrow (p^k)^{-1}\ mod\ q,\ q_1 \leftarrow q^{-1}\ mod\ p^k$ a equação 2.2
$M \leftarrow (q_1 q M_p + p_1 p^k M_q)\ mod\ p^k q$

Veja que este método permite a expansão de M_p para M'_p utilizando somente $k-1$ exponenciações (com o expoente público e, que supomos relativamente pequeno), $k-1$ inversões e algumas multiplicações.

RSA com Múltiplas Potências - Exemplo Numérico

Geração das Chaves ($k = 2$):

$p = 5,\ q = 13$: Inicialmente escolhemos 2 primos p e q

$N = p^k q = 325$: A seguir calculamos N utilizando p e q

$e = 7$: Escolhemos um e relativamente primo a p

$d = e^{-1}\ mod\ (p-1)(q-1) = 7$: Calculamos a inversa de e módulo $(p-1)(q-1)$

Redução modular de d

$d_p = d\ mod\ (p-1) = 3$: Reduzimos d módulo $(p-1)$

$d_q = d\ mod\ (q-1) = 7$: Reduzimos d módulo $(q-1)$

Chave pública $< N, e >=< 325, 7 >$

Chave particular $< p, q, d_p, d_q >=< 5, 13, 3, 7 >$

Criptografia:

Para uma mensagem $M = 257$: Aplicamos a chave pública a M

$C = M^e\ mod\ N$

$C = 257^7\ mod\ 325 = 218$

Decriptografia:

$M_p = C^{d_p}\ mod\ p$: Aplicamos d módulo $(p-1)$ a C

$M_q = C^{d_q}\ mod\ q = 10$: Aplicamos d módulo $(q-1)$ a C

$M'_p = M_p\ mod\ p^k = 7$: Expandimos M_p para M'_p (*Hensel Lift*)

aplicando a equação 2.2 obtemos,

$M = (M_q(p^k)((p^k)^{-1}\ mod\ q)$: Aplicando o TCR temos M de M_p e M_q

$+M'_p(q)(q^{-1}\ mod\ p^k)\ \ mod\ p^k q$

$M = 257$

4.3 RSA Rebalanceado (*Rebalanced RSA*)

O RSA Rebalanceado é baseado na observação de Wiener [25, 32] sobre o estudo do expoente particular d. Esta variação sugere a melhora no desempenho do algoritmo de decriptografia deslocando o trabalho deste para o algoritmo de criptografia. É de nosso conhecimento que não podemos reduzir o tamanho do expoente particular d, uma vez que, se este for menor que $N^{0.292}$, o RSA torna-se inseguro. O que se faz então é escolher um d grande (na ordem de

N), tal que $d \; mod \; p-1$ e $d \; mod \; q-1$ sejam números pequenos, reduzindo assim o tempo de decriptografia. Descrevemos mais detalhadamente este processo através dos algoritmos de geração de chaves, criptografia e decriptografia.

Geração das Chaves - O algoritmo de geração de chaves leva dois parâmetros de segurança n e s onde $s \leq n/2$ (normalmente utilizamos $s = 160$ bits). Um usuário ou uma entidade U calcula as chaves da seguinte maneira:

1 - Gera dois primos aleatórios distintos de $\lfloor n/2 \rfloor$ bits p e q com $mdc(p-1, q-1) = 2$. E calcula $N \leftarrow pq$.

2 - Mais dois números aleatórios de s-bits d_p e d_q são gerados, tais que $mdc(d_p, p-1) = 1$, $mdc(d_q, q-1) = 1$ e $d_p = d_q \; mod \; 2$.

3 - A seguir, U encontra um d tal que $d = d_p \; mod \; p-1$ e $d = d_q \; mod \; q-1$ (Explicado adiante).

4 - E, finalmente, calcula $e \leftarrow d^{-1} \; mod \; \phi(N)$. A chave pública é $< N, e >$ e a chave particular é $< p, q, d_p, d_q >$ [5].

Repare que para encontrar d no passo 3, não podemos utilizar o Teorema Chinês do Resto (Teorema 2.4.1), uma vez que $p-1$ e $q-1$ não são relativamente primos (são ambos pares). Entretanto, $(p-1)/2$ é relativamente primo a $(q-1)/2$, além disso $d_p = d_q \; mod \; 2$. Portanto, se fizermos $a = d_p \; mod \; 2$, podemos encontrar através do Teorema Chinês do Resto um elemento d' tal que

$$d' = \tfrac{d_p - a}{2} \pmod{\tfrac{p-1}{2}} \qquad \text{e} \qquad d' = \tfrac{d_q - a}{2} \pmod{\tfrac{q-1}{2}}.$$

Para encontrar o d necessário no passo 3, basta fazer $d = 2d' + a$. Atente que, $d = d_p \; mod \; p-1$ e $d = d_q \; mod \; q-1$.

Diferentemente das demais variações apresentadas esta força um valor para o expoente particular d e somente após isso, calcula o expoente público e. Devemos mostrar então que o expoente d calculado no passo 3 é inversível módulo $\phi(N)$ no passo subseqüente. Para isso,

[5] Para adaptação com o PKCS #1 veja seção 6.6

lembramos que o $mdc(d_p, p-1) = 1$ e $mdc(d_q, q-1) = 1$, logo, $mdc(d, p-1) = 1$ e $mdc(d, q-1) = 1$. Resultando que o $mdc(d, (p-1)(q-1)) = 1$ como desejávamos.

Outra característica importante desta variação é que o expoente público e calculado terá um valor muito grande (da ordem de N) e foi isso que tentamos dizer no início desta seção, quando afirmamos que o RSA Rebalanceado diminui o tempo de decriptografia, deslocando o trabalho para o algoritmo de criptografia. Esta característica não é comum às implementações do RSA, que utilizam normalmente um e igual a 65537, entretanto, segundo D. Boneh e H. Shacham em [25], as atuais Autoridades Certificadoras são capazes de gerar certificados para tais chaves públicas.

Vejamos agora os algoritmos de criptografia e decriptografia:

Criptografia - A criptografia usando a chave pública $< N, e >$ é idêntica à criptografia do RSA tradicional (figura 4.7). Lembrando, entretanto, que o expoente público e é muito maior que o utilizado normalmente no RSA e, assim, a entidade (ou usuário) que criptografar a mensagem M precisa estar disposta ou capacitada a utilizar tais chaves públicas.

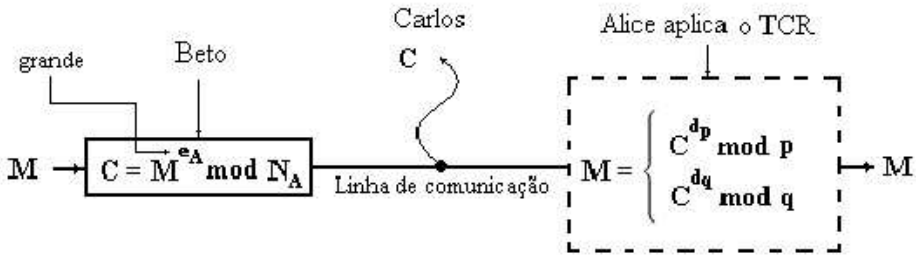

Figura 4.7: Beto codifica M usando a chave pública da Alice $< N_A, e_A >$ (com tamanho muito maior do que de costume) e envia o texto codificado C para ela. Alice recebe o texto C e utiliza o TCR para resolver o sistema $M = C^{d_p} \ mod \ p$, $M = C^{d_q} \ mod \ q$, onde d_p e d_q são calculados no algoritmo de geração de chaves.

Decriptografia - Para decodificar C usando a chave particular $< p, q, d_p, d_q >$ Alice deve proceder de acordo com os passos abaixo (figura 4.7):

1 - Alice calcula $M_p \leftarrow C^{d_p} \ mod \ p$ e $M_q \leftarrow C^{d_q} \ mod \ q$.

2 - Usando a equação 2.2 do TCR Alice obtém o único $M \in Z_N$ tal que $M = M_p \; mod \; p$ e $M = M_q \; mod \; q$, ou seja, a decriptografia de C ($M = C^d \; mod \; N$).

RSA Rebalanceado - Exemplo Numérico

Geração das Chaves:

$p = 7, \; q = 5$

$N = pq = 35, \; \phi(N) = (p-1)(q-1) = 24$: Inicialmente escolhemos 2 primos p e q

$d_p = 5$: A seguir calculamos N e $\phi(N)$ utilizando p e q

$d_q = 7$: Repare que $mdc(d_p, p-1) = 1$

$a = d_p \; mod \; 2 = 1$: Repare que $mdc(d_q, q-1) = 1$ e $d_p = d_q \; mod \; 2$

$d1' = \frac{(d_p - a)}{2} \; mod \; \frac{(p-1)}{2} = 2$: Calculamos o auxiliar a

$d2' = \frac{(d_q - a)}{2} \; mod \; \frac{(q-1)}{2} = 1$

$d' = 5$

Calculando d : Aplicando o TCR obtemos d' de $d1'$ e $d2'$

$d = 2d' + a = 11$: Calculamos d (inversível)

$e = d^{-1} \; mod \; \phi(N) = 11$: Calculamos a inversa de d módulo $\phi(N)$

Chave pública $< N, e > = < 35, 11 >$	Chave particular $< p, q, d_p, d_q > = < 7, 5, 5, 7 >$

Criptografia:

Para uma mensagem $M = 17$

$C = M^e \; mod \; N$: Aplicamos a chave pública a M

$C = 17^{11} \; mod \; 35 = 33$

Decriptografia:

$M_p = C^{d_p} \; mod \; p = 3$: Aplicamos d módulo $(p-1)$ a C

$M_q = C^{d_q} \; mod \; q = 2$: Aplicamos d módulo $(q-1)$ a C

aplicando a equação 2.2 obtemos,

$M = (M_q(p)(p^{-1} \; mod \; q)$: Aplicando o TCR temos M de M_p e M_q

$+M_p(q)(q^{-1} \; mod \; p) \;\; mod \; pq$

$M = 17$

Outras variações

Ao analisarmos as variações enunciadas até o momento, podemos observar que algumas combinações podem ser efetuadas. Como primeira combinação, podemos utilizar em conjunto a técnica utilizada pelo RSA com múltiplos ou a utilizada pelo RSA QC, com a utilizada pelo RSA em Lote. Ou seja, podemos reduzir cada C_i (utilizado pelo RSA em Lote) módulo p_i $(1 \leq i \leq k)$, combinando posteriormente estes resultados através do Teorema Chinês do Resto[6]. Isto na verdade, reflete na utilização de k árvores: a primeira executando módulo p_1, a segunda executando módulo p_2 e assim por diante. Ao final do processo combinamos os resultados obtidos pelas árvores, por meio do algoritmo clássico ou pelo algoritmo de Garner.

Considerando, entretanto, que o ganho obtido pelo RSA em Lote está na fase de exponenciação (veja seção 3.3 e figura 4.2), e que sua decriptografia implica na execução de pequenas exponenciações para cada árvore utilizada[7], optamos por implementar este método com parâmtero $k = 2$, ou seja, em nossa implementação do RSA em Lote fizemos uso da técnica utilizada pelo RSA QC.

Uma outra combinação que pode ser efetuada é a do RSA Rebalanceado com o RSA com Múltiplos Primos. Esta mistura é totalmente possível e em nosso trabalho resolvemos avaliar o ganho ou prejuízo trazido por ela, fazendo um estudo comparativo com as demais variações apresentadas. O novo esquema foi denominado como RSA Rebalanceado com Múltiplos Primos e será detalhado na próxima seção.

Alguém poderia propor também que se combinassem as técnicas utilizadas pelo RSA Rebalanceado e pelo RSA com Múltiplas Potências. Mas infelizmente, esta combinação gera uma variação lenta tanto na criptografia, quanto na decriptografia. Isso acontece porque o algoritmo utilizado para o cálculo de Mp' (seção 4.2.3) faz uso do expoente público e e como o e utilizado pelo RSA Rebalanceado é muito maior que o padrão, teremos grandes exponenciações modulares na criptografia e na decriptografia e por esta razão resolvemos não implementar esta variação.

[6] No caso da combinação com o RSA QC, temos $k = 2$

[7] Observe que quanto maior for o parâmetro k, mais árvores serão utilizadas e conseqüentemente mais exponenciações pequenas serão necessárias, comprometendo o ganho obtido na fase de exponenciação.

A seguir, descrevemos de forma mais detalhada a nova variação proposta nesta dissertação (RSA Rebalanceado com Múltiplos Primos).

4.4 RSA Rebalanceado com Múltiplos Primos

Ao misturarmos as técnicas utilizadas pelo RSA Rebalanceado e pelo RSA com Múltiplos Primos obtemos uma nova variação, que denominamos neste trabalho como RSA Rebalanceado com Múltiplos Primos. Os algoritmos de geração de chaves, de criptografia e decriptografia são apresentados abaixo:

Geração das Chaves - O algoritmo de geração de chaves leva dois parâmetros de segurança n e s onde $s \leq n/k$ (o significado de s e k é mostrado adiante). Um usuário ou uma entidade U calcula as chaves RSA através dos passos:

1 - Gera k primos aleatórios distintos de $\lfloor n/k \rfloor$ bits $p_1, p_2, ..., p_k$, com $mdc(p_1 - 1, p_2 - 1, ..., p_k - 1) = 2$. E calcula $N \leftarrow p_1 p_2 ... p_k$.

2 - Mais k números aleatórios de s-bits $d_{p_1}, d_{p_2}, ..., d_{p_k}$ são gerados, tais que $mdc(d_{p_1}, p_1 - 1) = 1$, $mdc(d_{p_2}, p_2 - 1) = 1$, ..., $mdc(d_{p_k}, p_k - 1) = 1$ e $d_{p_1} = d_{p_2} = ... = d_{p_k} \ mod \ 2$.

3 - A seguir, U encontra um d tal que $d = d_{p_1} \ mod \ p_1 - 1$, $d = d_{p_2} \ mod \ p_2 - 1$ ... $d = d_{p_k} \ mod \ p_k - 1$ (Explicado à frente).

4 - Finalmente, U calcula $e \leftarrow d^{-1} \ mod \ \phi(N)$. A chave pública é dada por $< N, e >$ e a chave particular é dada por $< p_1, p_2, ..., p_k, d_{p_1}, d_{p_2}, ..., d_{p_k} >$ [8].

Como no RSA Rebalanceado, para encontrar d no passo 3, não podemos utilizar o Teorema Chinês do Resto (Teorema 2.4.1), já que $p_1 - 1$, $p_2 - 1$, ..., $p_3 - 1$ não são relativamente primos. Entretanto, sabemos que, $(p_1 - 1)/2$ é relativamente primo a $(p_2 - 1)/2$ ... $(p_k - 1)/2$, além disso $d_{p_1} = d_{p_2} = d_{p_k} \ mod \ 2$. Portanto, como anteriormente, se fizermos $a = d_p \ mod \ 2$, podemos encontrar através do Teorema Chinês do Resto um elemento d' tal que

[8] Para adaptação com o PKCS #1 veja seção 6.7

$$d' \equiv \begin{cases} \frac{d_{p_1}-a}{2} \ (mod \ \frac{p_1-1}{2}) \\ \frac{d_{p_2}-a}{2} \ (mod \ \frac{p_2-1}{2}) \\ \vdots \\ \frac{d_{p_k}-a}{2} \ (mod \ \frac{p_k-1}{2}) \end{cases}$$

e obteremos d fazendo $d = 2d' + a$.

A justificativa do porque d é inversível módulo $\phi(N)$ no passo 4, nada mais é que uma generalização da justificativa mostrada na seção anterior.

Criptografia - Novamente a criptografia usando a chave pública $< N, e >$ é idêntica a criptografia utilizada pelo RSA tradicional (figura 4.8). Lembrando, entretanto, que como no RSA Rebalanceado, o expoente público e é muito maior que o utilizado normalmente e, assim, a entidade responsável por criptografar a mensagem M precisa estar disposta ou capacitada a utilizar tal expoente.

Decriptografia - Seja $d_i = d \ mod \ (p_i - 1)$. Para decodificar C, Alice primeiramente deve calcular $M_i = C^{d_i} \ mod \ p_i$ para cada i, $1 \leq i \leq k$. Logo após, Alice deve combinar os M_i's através do TCR para obter $M = C^d \ mod \ N$ (figura 4.8). Novamente, a utilização do TCR leva tempo desprezível se comparada com as k exponenciações (veja seção 4.2.1).

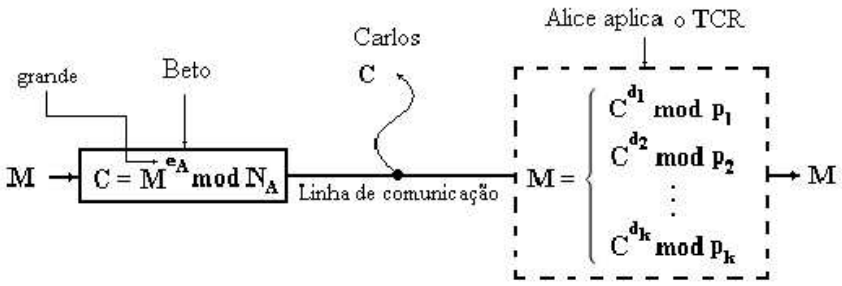

Figura 4.8: Beto codifica M usando a chave pública da Alice $< N_A, e_A >$ (com tamanho muito maior do que de costume) e envia o texto codificado C para Alice que o decodifica aplicando o TCR no sistema de congruências $M_i = C^{d_i} \ mod \ p_i$, onde $d_i = d \ mod \ (p_i - 1)$.

RSA Rebalanceado com Múltiplos Primos - Exemplo Numérico

Geração das Chaves $(k = 3)$**:**

$p_1 = 3,\ p_2 = 5,\ p_3 = 7$: Inicialmente escolhemos 2 primos p e q

$N = p_1 p_2 p_3 = 105,$: A seguir calculamos N e $\phi(N)$

$\phi(N) = (p_1 - 1)(p_2 - 1)(p_3 - 1) = 48$ utilizando p_1, p_2 e p_3

$d_{p_1} = 95$: Repare que $mdc(d_{p_1}, p_1 - 1) = 1$

$d_{p_2} = 89$: Repare que $mdc(d_{p_2}, p_2 - 1) = 1$

$d_{p_3} = 59$: Repare que $mdc(d_{p_3}, p_3 - 1) = 1$

$a = d_p \ mod\ 2 = 1$: Calculamos o auxiliar a

$d1' = \frac{(d_{p_1} - a)}{2}\ mod\ \frac{(p_1 - 1)}{2} = 47$

$d2' = \frac{(d_{p_2} - a)}{2}\ mod\ \frac{(p_2 - 1)}{2} = 0$

$d3' = \frac{(d_{p_3} - a)}{2}\ mod\ \frac{(p_3 - 1)}{2} = 2$

$d' = 2$: Aplicando o TCR obtemos d'

Calculando d

$d = 2d' + a = 5$: Calculamos d (inversível)

$e = d^{-1}\ mod\ \phi(N) = 29$: Calculamos a inversa de d módulo $\phi(N)$

Chave pública $< N, e >=< 105, 29 >$

Chave particular $< p_1, p_2, p_3, d_{p_1}, d_{p_2}, d_{p_3} >$

Criptografia:

Para uma mensagem $M = 73$

$C = M^e\ mod\ N$: Aplicamos a chave pública a M

$C = 73^{29}\ mod\ 105 = 103$

Decriptografia:

$M_{p_1} = C^{d_{p_1}}\ mod\ p_1 = 1$: Aplicamos d módulo $(p_1 - 1)$ a C

$M_{p_2} = C^{d_{p_2}}\ mod\ p_2 = 3$: Aplicamos d módulo $(p_2 - 1)$ a C

$M_{p_3} = C^{d_{p_3}}\ mod\ p_3 = 3$: Aplicamos d módulo $(p_3 - 1)$ a C

aplicando a equação 2.2 obtemos,

$M' = (M_q(p)(p^{-1}\ mod\ q)$

$+M_p(q)(q^{-1}\ mod\ p)\ \ mod\ pq$

$M = (M'(p_3)({p_3}^{-1}\ mod\ p_1 p_2)$: Aplicando o TCR temos M de M_p e M_q

$+M_{p_3}(p_1 p_2)((p_1 p_2)^{-1}\ mod\ p_3)$

$mod\ p_1 p_2 p_3$

$M = 73$

RSA Rebalanceado com Múltiplos Primos - Aplicações

Sabendo-se que o RSA com Múltiplos Primos e o RSA com Múltiplas Potências diminuem o tempo da decriptografia RSA sem provocar prejuízo à criptografia, podemos citar como aplicações desses métodos as mesmas empregadas pelo RSA Tradicional ou RSA QC (por exemplo, aplicações de comércio eletrônico). O mesmo acontece com o RSA em lote, lembrando contudo, que este necessita de b mensagens (utilizando o mesmo módulo e expoentes públicos diferentes entre si) para iniciar o processo de decriptografia, sendo utilizado em aplicações nas quais desejamos trabalhar com grupos de mensagens. Boneh e Shacham [3] obtiveram bons resultados com a aplicação do RSA em Lote em um servidor SSL.

Os métodos utilizados pelo RSA Rebalanceado e pelo RSA Rebalanceado com Múltiplos Primos, que diminuem o tempo da decriptografia RSA em detrimento da criptografia, parecem não apresentar vantagens em termos práticos. Todavia, existem aplicações onde o rebalanceamento provocado por esses algoritmos é desejável. Como primeiro exemplo, podemos citar a pré-gravação de informações sigilosas em um DVD. Poderíamos criptografar essas informações na pré-gravação utilizando um desses métodos, e assim garantiríamos a autenticação de destino (veja seção 3.1). Repare que o prejuízo gerado pelo processo de decriptografia é amenizado pela pré-gravação, além disso, teríamos um ótimo desempenho da decriptografia (momento da leitura dos dados). Uma outra situação em que essas variações funcionam bem, é aquela onde a geração de assinatura é executada em maior número que sua verificação. Um banco, por exemplo, pode emitir diversas assinaturas digitais em um único dia (em documentos, recibos), enquanto o usuário que recebe esta assinatura, pode ter maior disponibilidade. Nesta situação é razoável transferir o esforço exigido pela geração de assinaturas para a verificação destas.

Como último exemplo podemos citar o caso dos computadores de mão (PDAs), que possuem geralmente recursos limitados. Em uma comunicação com servidores (ou até mesmo com *notebooks* e computadores de mesa), poderíamos deixar a cargo dos computadores de mão o papel da decriptografia (no caso, rápida) e o papel da criptografia (lenta) para os computadores que possuem mais recursos computacionais. Poderíamos, inclusive, utilizar uma implementação do RSA com Múltiplos Primos com chaves tanto do RSA com Múltiplos Primos, quanto do RSA Rebalanceado com Múltiplos primos, dependendo do tipo de comunicação (computador de mesa/computador de mesa, ou, computador de mesa/computador de mão).

CAPÍTULO

5

Método e Ferramentas utilizadas

Antes de apresentarmos a análise das implementações dos algoritmos já estudados, introduzimos neste capítulo as ferramentas que viabilizaram este processo e que facilitam a utilização das implementações com sistemas já em funcionamento.

Inicialmente, descrevemos a plataforma utilizada, especificando também o tipo de linguagem de programação e versão. Descrevemos também as características dos dados de entrada e de saída e a forma como foram produzidos. Finalizamos com a apresentação das ferramentas de padronização e suporte para números grandes, conhecidas como PKCS e GMP, respectivamente.

5.1 Hardware, Características de Execução e Comparação

Os resultados disponíveis nos capítulos posteriores foram obtidos utilizando um computador com um processador AMD Atlhon XP 1600+, 256 MB de memória RAM, executando o sistema operacional Linux Mandrake versão 9.0. A linguagem escolhida para as implementações foi

a linguagem C, com compilador gnu gcc versão 3.2. Durante a execução dos programas, procuramos evitar o uso de processos desnecessários, visando à minimização de interferência nas medidas.

Tomamos como medida inicial de comparação o tempo, em microssegundos, gasto pela criptografia e/ou decriptografia de uma dada mensagem M. É importante ressaltar que, nesta medição, não consideramos o tempo de iniciação de variáveis e leitura dos dados. Consideramos apenas o tempo gasto pelo procedimento em execução, seja criptografia ou decriptografia.

Ao escolher o tamanho dos módulos a serem utilizados, consideramos inicialmente o valor recomendado atualmente pelos Laboratórios RSA [19], ou seja, 1024 bits. Porém, para estabelecermos uma ponte com sistemas que ainda utilizam módulos inferiores (um número ainda muito relevante) resolvemos analisar módulos de 768 bits; e para apresentarmos uma projeção para futuros sistemas, pensando também nos usuários mais cautelosos e nas informações que requerem um nível maior de segurança, consideramos módulos com 2048 bits.

O processo cumpriu então o seguinte roteiro: para cada variação do RSA, utilizamos 60 chaves geradas aleatoriamente (20 para cada tipo de módulo utilizado) e, para cada chave utilizada, executamos cinqüenta mensagens também aleatórias. No total, cada variação foi executada 3000 vezes, 1000 vezes utilizando módulos de 768 bits, 1000 vezes utilizando módulos de 1024 bits e 1000 vezes utilizando módulos de 2048 bits.

As mensagens utilizadas foram geradas através de um pequeno programa em C e de um *shell script* que, combinados, construíam mensagens aleatórias menores que um determinado módulo[1]. Mais precisamente, as mensagens possuíam tamanho próximo ao módulo a ser testado, ou seja, próximos a 768, 1024 e 2048 bits.

Com exceção do RSA em Lote, em que por convêniencia, foram agrupadas b mensagens em cada arquivo de mensagem, em todas as outras variações cada mensagem gerada era armazenada em um arquivo diferente, resultando, em cada um destes arquivos, uma única mensagem. Assim, ao utilizarmos chaves com $n = 768$ bits, utilizávamos também mensagens de tamanho próximo a 768 bits; ao utilizarmos chaves com $n = 1024$ bits utilizávamos mensagens de tamanho próximo a 1024 bits e, finalmente, quando utilizávamos chaves com $n = 2048$ bits,

[1] Um exemplo de uma mensagem de 1024 bits pode ser vista na seção A.2 do Apêndice A.

utilizávamos mensagens de tamanho próximo a 2048 bits. Um diagrama de execução para $n = 768$ bits é ilustrado na figura 5.1.

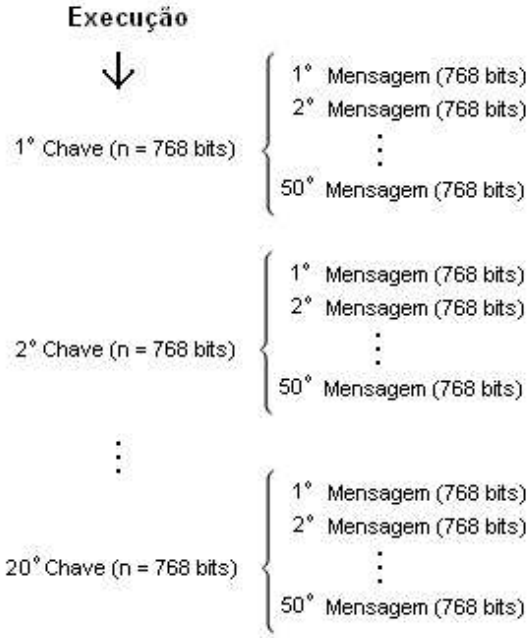

Figura 5.1: Diagrama de Execução para $n = 768$ bits.

A cada execução era registrado, em um arquivo de saída, o tempo, em microssegundos, necessário para codificar ou decodificar a mensagem corrente. Ao terminar a execução de uma chave, registramos neste arquivo a média do tempo gasto em microssegundos para as 50 mensagens utilizadas por ela. Após a execução completa das 20 chaves da variação em teste, registramos no arquivo a média aritmética dos tempos relativos às 1000 criptografias ou 1000 decriptografias. Logo, para cada variação, geramos seis arquivos de saída, três para a criptografia (módulos 768, 1024, e 2048 bits) e três para decriptografia (módulos 768, 1024, 2048 bits). Um exemplo de tal arquivo pode ser visto na seção A.2 do Apêndice A.

Tendo em vista a dispersão das medidas em torno das médias calculadas, acrescentamos a estas, o desvio padrão amostral correspondente. Denotando a média aritmética dos tempos

calculados por $\overline{x}$ e, x_i, como o tempo necessário para codificar ou decodificar o i-ésimo texto, o desvio padrão amostral pode ser obtido pela fórmula:

$$\sigma = \sqrt{\sum_{i=1}^{1000} \frac{(x_i - \overline{x})^2}{n-1}} \tag{5.1}$$

lembrando que $\overline{x} = (\sum_{i=1}^{1000})/n$.

Considerando também, que o tempo em microssegundos gasto por um procedimento é uma medida um tanto volátil (visto que bastaria a alteração do computador ou sistema operacional para que os resultados fossem muitas vezes bruscamente alterados), adotamos uma medida que acreditamos ser menos instável denominada *Speedup*. *Speedup* pode ser entendido como ganho relativo, ou seja, descreve quantas vezes um algoritmo é mais rápido que outro.

Denotamos os termos *SpeedupRSA* e *SpeedupQC* sobre um módulo de n bits, que serão simbolizados como $S_{RSA}(n)$ e $S_{QC}(n)$, respectivamente, e definidos da seguinte maneira:

$$S_{RSA}(n) = \frac{T_{RSA}(n)}{T_{AP}(n)} \tag{5.2}$$

$$S_{QC}(n) = \frac{T_{QC}(n)}{T_{AP}(n)} \tag{5.3}$$

onde $T_{RSA}(n)$ é a média aritmética do tempo das decriptografias usando o RSA tradicional em microssegundos sobre módulos de n bits, $T_{QC}(n)$ é a média aritmética do tempo das decriptografias usando RSA QC em microssegundos sobre módulos n bits e $T_{AP}(n)$ é a média aritmética do tempo das decriptografias do algoritmo proposto em microssegundos sobre módulos de n bits.

5.2 *Public-Key Cryptography Standards* - PKCS

O PKCS [17] é uma série de especificações produzidas pelos Laboratorios RSA em cooperação com desenvolvedores de sistemas de segurança de várias partes do mundo, que visa acelerar, através de padronização, a utilização e o desenvolvimento de algoritmos de chave pública. Surgiu em 1991, como resultado de encontros de um pequeno grupo de precursores no uso da tecnologia de chave pública e desde então tem se tornado referência até mesmo para padrões já estabelecidos, como ANSI X9, PKIX, SET, S/MIME e SSL. Atualmente seu desenvolvimento

ocorre basicamente através de lista de discussões e ocasionais *workshops*.

Como pode ser visto na tabela 5.1, a especificação PKCS responsável pela padronização da criptografia/verificação de assinatura e pela decriptografia/geração de assinatura utilizando o criptossistema RSA é dada pelo PKCS#1. As especificações referentes ao PKCS#2 e PKCS#4 foram incorporadas a esta. Além de especificar as primitivas criptográficas , o PKCS#1 define também padrões de conversões de dados, como o *I2OSP* e *OS2IP* (permitindo que uma dada mensagem $X > N - 1$ possa ser codificada ou decodificada corretamente), funções *hash* (para serem utilizadas, por exemplo, para assinaturas) e tipos de chaves a serem utilizadas com as primitivas criptográficas.

Número	**Tema**
1	RSA Cryptography Standard
3	Diffie-Hellman Key Agreement Standard
5	Password-Based Cryptography Standard
6	Extended-Certificate Syntax Standard
7	Cryptographic Message Syntax Standard
8	Private-Key Information Syntax Standard
9	Selected Attribute Types
10	Certification Request Syntax Standard
11	Cryptographic Token Interface Standard
12	Personal Information Exchange Syntax Standard
13	Elliptic Curve Cryptography Standard
15	Cryptographic Token Information Format Standard

Tabela 5.1: Temas tratados pelas especificações PKCS.

Em nossas implementações tentamos gerar chaves que se aproximavam ao máximo das especificadas nas primitivas criptográficas da versão 2.1 do PKCS#1 (versão mais recente até a presente data, lançada em 14 de junho de 2002). A representação da chave pública no PKCS#1 é a mesma utilizada no decorrer deste documento, ou seja, $< N, e >$. Já a representação da chave particular pode ser dada tanto pelo par $< N, d >$ (veja seção 3.2), como pela quíntupla $< p, q, d_p, d_q, qInv >$ e uma opcional seqüência de triplas $< p_i, d_i, t_i >$, $i = 3, ..., u$, uma para cada primo não pertencente à quíntupla, onde os componentes destas seqüências são descritos abaixo:

- p : o primeiro fator primo, um inteiro positivo.

- q : o segundo fator primo, um inteiro positivo.
- d_p : primeiro coeficiente do TCR, $d \; mod \; p-1$, um inteiro positivo.
- d_q : segundo coeficiente do TCR, $d \; mod \; q-1$, um inteiro positivo.
- $qInv$: a inversa de q módulo p, um inteiro positivo.
- p_i : i-ésimo fator primo (para i= 3, 4, ...), um inteiro positivo.
- d_i : i-ésimo coeficiente do TCR, $d \; mod \; p_i - 1$ (para i= 3, 4, ...), um inteiro positivo.
- t_i : $p_1 p_2 ... p_{i-1} t_i \equiv 1 \; (mod \; p_i)$, um inteiro positivo (onde p_1 = p e $p_2 = q$).

Este tipo de chave é necessária pois esta especificação utiliza o algoritmo de Garner na aplicação do Teorema Chinês do Resto (seção 2.5). Assim, ao implementarmos os algoritmos descritos no capítulo anterior, se quisermos que estes estejam de acordo com a especificação acima, devemos antes adaptá-los de tal forma que produzam e sejam capazes de utilizar as chaves na forma apresentada aqui. Este modelo de chaves foi criado com o intuito de suportar tanto o RSA tradicional quanto o RSA QC e o RSA com Múltiplos Primos. Todavia, sabemos que o RSA Rebalanceado utiliza o mesmo tipo de chaves que o RSA QC (com a única diferença que o expoente público e é maior que o convencional) e portanto, também está de acordo com a especificação. O mesmo ocorre com o RSA Rebalanceado com Múltiplos Primos, que utiliza o mesmo tipo de chaves que o RSA com Múltiplos Primos. Concluindo, as únicas variações do quarto capítulo que não se enquadram no PKCS #1 são as do RSA em Lote e do RSA com Múltiplas Potências, que utilizam o expoente público na decriptografia.

O algoritmo de decriptografia utilizado pelo PKCS#1 em sua última versão depende então do tipo da chave particular utilizada. Se a chave particular for da forma $< N, d >$ calculamos M fazendo $M = C^d \; mod \; N$ como habitual. Se a chave particular for da segunda forma $< p, q, d_p, d_q, qInv >$ e $< p_i, d_i, t_i >$, procedemos como segue:

Entrada: C, $(p, q, d_p, d_q, qInv)$ e triplas opcionais (p_i, d_i, t_i)
Saida: M

$M_1 \leftarrow C^{d_p} \ mod \ p$: Calcula M_1 e M_2 como no RSA QC
$M_2 \leftarrow C^{d_q} \ mod \ q.$
Se $(u > 2)$: Se utilizarmos mais de 2 primos,
  Para $i \leftarrow 3$ até u faça calcula os Ms restantes
    $M_i \leftarrow C^{d_i} \ mod \ p_i$
$h \leftarrow (m_1 - m_2)qInv \ mod \ p.$: Utiliza o algoritmo de Garner para 2 primos
$M \leftarrow M_2 + qh.$
Se $(u > 2)$: Utiliza o algoritmo de Garner para
  $R = p_1$ os primos restantes
  Para $i \leftarrow 3$ até u faça
    $R = Rp_{i-1}.$
    $h = (m_i - m)t_i \ (mod \ p_i).$
    $M = M + Rh.$

Observe a conformidade do código acima com os algoritmos de decriptografia do RSA QC e do RSA com Múltiplos Primos, e note a possibilidade de utilização das chaves do RSA Rebalanceado e do RSA Rebalanceado com Múltiplos Primos.

A adequação das variações com o PKCS #1 permite a comunicação entre sistemas padronizados, porém, não garante que as chaves utilizadas sejam seguras. Para isso, devemos seguir as recomendações descritas no terceiro capítulo. A primeira destas recomendações é a utilização de primos grandes (com centenas de bits). Mas para efetuarmos tal processo necessitamos de uma ferramenta denominada *GMP*, que descrevemos a seguir.

5.3 *GMP - GNU Multiple Precision Arithmetic Library*

Para implementar o criptossistema RSA, assim como os algoritmos propostos com primos e conseqüentemente módulos satisfatórios (grandes tais como 768, 1024 e 2048 bits), precisamos de uma ferramenta que consiga trabalhar com tais números. Para a linguagem C e C++ encontramos disponível a biblioteca GMP, que utilizamos no presente trabalho devido às vantagens especificadas abaixo:

- É distribuída gratuitamente sob a *GNU Lesser General Public License* (veja [10, 21].

- Não impõe limites de precisão sob o tamanho dos números a serem utilizados. Ou seja, a precisão depende somente da quantidade de memória disponível na máquina onde o GMP está sendo executado.
- Possui um rico conjunto de funções, incluindo funções de teoria de números.
- Seus projetistas tiveram como principal preocupação a velocidade. Como conseqüência, suas funções apresentam um bom desempenho tanto para operandos pequenos, quanto para operandos extremamente grandes.

Os principais alvos de aplicação do GMP são: criptografia, segurança na Internet, sistemas algébricos e álgebra computacional. Desde a sua criação em 1991, a biblioteca vem sendo aprimorada e atualizada com novas versões, e sua popularidade tem crescido entre os desenvolvedores de sistemas.

Ao adotarmos esta biblioteca, consideramos principalmente o uso de algoritmos eficientes pelas funções relacionadas à criptografia e teoria de números (para tornar os resultados próximos aos desejados em uma situação real).

Para melhor compreensão da listagem dos programas encontrados no Apêndice B, descrevemos de forma sucinta as funções da biblioteca GMP [21] que utilizamos durante nosso trabalho:

- void mpz_init(mpz_t *var*) - inicia a variável *var*, atribuindo a ela o valor 0.
- void mpz_init_set_ui(mpz_t *var*, unsigned long *op*) - inicia a variável *var*, atribuindo da variável *op* de tipo *unsigned long*.
- void mpz_set(mpz_t *var*, mpz_t *op*) - associa o valor de *op* a variável já iniciada *var*.
- void mpz_set_ui(mpz_t *var*, unsigned long *op*) - associa o valor de uma variável do tipo *unsigned long* à já iniciada variável *var*.
- void mpz_set_str(mpz_t *var*, char **str*, int *base*) - converte o valor da string *str* em um inteiro *mpz_t* na base *base* e armazena o resultado em *var*.
- void mpz_get_str(char **str*, int *base*, mpz_t *var*) - converte o valor da variável *var* para uma string na base *base* e armazena o resultado em *str*.

- void mpz_add(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - armazena na variável *var* o valor de *op1* + *op2*.
- void mpz_add_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*) - armazena na variável *var* o valor de *op1* + *op2*.
- void mpz_sub(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - armazena na variável *var* o valor de *op1* - *op2*.
- void mpz_sub_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*) - armazena na variável *var* o valor de *op1* - *op2*.
- void mpz_mul(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - armazena na variável *var* o valor de *op1* * *op2*.
- void mpz_mul_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*) - armazena na variável *var* o valor de *op1* * *op2*.
- void mpz_fdivq(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - armazena na variável *var* o valor de $\lfloor op1/op2 \rfloor$.
- void mpz_fdivq_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*) - armazena na variável *var* o valor de $\lfloor op1/op2 \rfloor$.
- void mpz_mod(mpz_t *var*, mpz_t *op1*, mpz_t *op2*)) - Armazena na variável *var* o valor de $op1 \ mod \ op2$.
- void mpz_mod_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*)) - Armazena na variável *var* o valor de $op1 \ mod \ op2$.
- void mpz_pow_ui(mpz_t *var*, mpz_t *op1*, unsigned long *op2*) - Armazena na variável *var* o valor de $op1^{op2}$, onde $op2$ é do tipo *unsigned long*.
- void mpz_powm -(mpz_t *var*, mpz_t *op1*, mpz_t *op2*, mpz_t *op3*) - Armazena na variável *var* o valor de $op1^{op2} \ mod \ op3$.
- int mpz_invert(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - Armazena em *var* a inversa de *op1* módulo *op2*. Utiliza para isso o algoritmo de Euclides Estendido.
- void mpz_gcd(mpz_t *var*, mpz_t *op1*, mpz_t *op2*) - Armazena na variável *var* o máximo divisor comum de *op1* e *op2*.

- int mpz_probab_prime_p(mpz_t *var*, int *op1*) - Retorna 2 se *var* é primo, 1 se *var* é um provável primo, ou 0 se é composto. Utiliza o teste probabilístico de Miller-Rabin. O número de testes a serem feitos são controlados por *op1*.

- int mpz_cmp (mpz_t *op1*, mpz_t *op2*) - Compara *op1* com *op2*. Retorna um número positivo se $op1 > op2$, negativo se $op1 < op2$ e zero caso $op1 = op2$.

- int mpz_cmp_ui (mpz_t *op1*, unsigned long *op2*) - Compara *op1* com *op2*. Retorna um número positivo se $op1 > op2$, negativo se $op1 < op2$ e zero caso $op1 = op2$.

- void mpz_clear(mpz_t *var*) - Desaloca o espaço ocupado pela variável *var*.

Todas as funções apresentadas atuam somente com operandos inteiros e representam apenas um pequeno subconjunto das funções pertencentes a biblioteca *GMP*.

Adiante, apresentamos os resultados práticos provenientes da implementação dos algoritmos já introduzidos. Em tal implementação, utilizamos a versão 4.1 [2] da biblioteca GMP.

[2]Resultados obtidos com versões posteriores, ou até mesmo com outras bibliotecas, podem ser diferentes dos apresentados no próximo capítulo, devido ao desempenho proporcionado por suas funções. Lembramos contudo, que o desempenho foi o fator decisivo na escolha da biblioteca GMP.

CAPÍTULO

6

Implementação e Análise Comparativa

Este capítulo é destinado à análise das implementações dos algoritmos estudados no capítulo quatro. Por questão de conveniência, nas tabelas trataremos as variações por seus respectivos nomes em inglês, ou seja, Batch, Mprime, Mpower, Rebalanced. No caso do RSA Rebalanceado com Múltiplos Primos, tratá-lo-emos por RPrime.

Utilizamos rigorosamente as medidas de comparação tratadas no quinto capítulo, bem como as ferramentas lá descritas.

6.1 Implementação e Desempenho do RSA tradicional

Para medida de comparação com os algoritmos que seguem, implementamos o RSA tradicional utilizando o seguinte par de chaves:

- Chave pública : $< N, e >$
- Chave particular: $< N, d >$

Como não poderia deixar de ser, estas chaves estão completamente de acordo com o PKCS#1 desde sua primeira versão. Utilizamos, como expoente público e, o valor 65537.

A tabela 6.1 descreve a média aritmética e o desvio padrão para cada módulo analisado, dos tempos em microssegundos correspondentes à execução de 1000 criptografias ou 1000 decriptografias, segundo a seção 5.1. Podemos notar que por utilizarmos um expoente público e padrão, que é muito menor que o expoente particular d (na ordem de N), os resultados obtidos pelo algoritmo de criptografia são muito inferiores aos obtidos pela decriptografia.

n (em bits)	**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ
768	185	29	7903	179
1024	308	15	17619	206
2048	1080	41	124160	4780

Tabela 6.1: Média aritmética e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA tradicional para os módulos de 768, 1024 e 2048 bits.

Separamos os resultados da tabela 6.1 em dois gráficos (figura 6.1). O primeiro gráfico corresponde às criptografias e o segundo corresponde às decriptografias do criptossistema RSA. Essa divisão facilita a visualização dos desvios em torno da média, por ela podemos observar que nas decriptografias para módulos de 768 e 1024 bits os valores concernentes ao desvio padrão são insignificantes com relação a média do tempo em microsegundos.

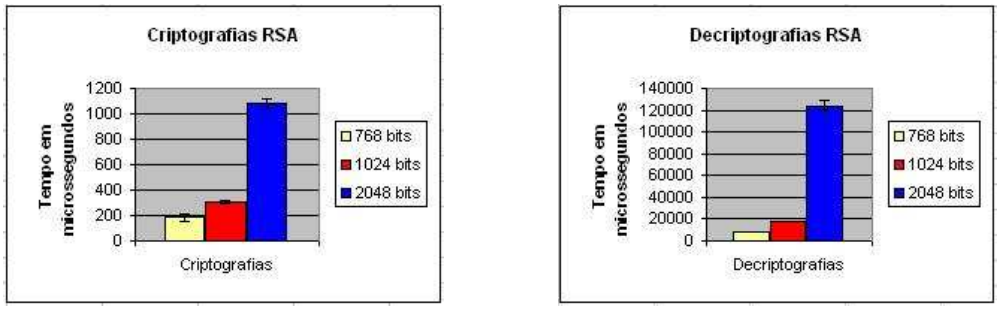

Figura 6.1: Criptografias e decriptografias do RSA tradicional (referente à tabela 6.1).

Atentamos ao fato de que, em todas as implementações analisadas, mais de 82% das medidas se encontravam no intervalo $\{\overline{x} - \sigma, \overline{x} + \sigma\}$.

6.2 Implementação e Desempenho do RSA QC

Reforçando o que dissemos na seção 3.3, o esquema utilizado pelo RSA QC pode ser considerado o padrão de implementação atual; principalmente após sua inclusão na especificação PKCS#1, que desde a versão 1.5 (datada de novembro de 1993), fornece algum suporte ao RSA QC. A implementação utilizada neste trabalho, de acordo com a versão atual desta especificação, utiliza o seguinte par de chaves [1]:

- Chave pública : $< N, e >$
- Chave particular: $< p, q, dp, dq, qInv > \ < r, dr, rInv >$

onde $qInv = q^{-1} \ mod \ p$ e $rInv = (pq)^{-1} \ mod \ r$ (veja seção 5.2). A exemplo da implementação do RSA tradicional, utilizamos como expoente público o padrão $e = 65537$.

Alguém poderia se perguntar se usar chaves desse tipo traz algum prejuízo à segurança do sistema. Lembramos que a única mudança feita com relação às chaves do RSA tradicional ocorre na chave particular que supomos, portanto, totalmente secreta. Devemos considerar também que conhecer os primos p e q não é privilégio somente de quem possui a chave particular dessa forma, pois existem algoritmos rápidos capazes de fatorar N conhecendo somente $< N, d >$ (um exemplo pode ser visto em [30]).

Na tabela 6.2, encontramos a média aritmética e o desvio padrão do tempo em microssegundos utilizado pelas criptografias e decriptografias, segundo os critérios descritos na seção 5.1. Por utilizar o mesmo algoritmo de criptografia e fazer uso do mesmo expoente público, os resultados referentes às criptografias do RSA QC se assemelham aos obtidos pelas criptografias do RSA tradicional. Porém, na decriptografia, como esperado, o RSA QC obteve um ganho considerável em relação ao RSA tradicional.

[1] Maiores detalhes sobre as implementações dos algoritmos de geração de chaves leia seção 6.9.

n (em bits)	Criptografia		Decriptografia	
	$\overline{x}$	σ	$\overline{x}$	σ
768	185	12	2438	93
1024	307	14	5294	129
2048	1080	43	35745	2913

Tabela 6.2: Média aritmética e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA QC para os módulos de 768, 1024 e 2048 bits.

Para uma melhor visualização dos dados descritos na tabela 6.2 construímos os gráficos dispostos na figura 6.2. É fácil perceber a semelhança dos resultados concernentes às criptografias do RSA e do RSA QC e sua discrepância em relação às decriptografias (resultados ilustrados nas figuras 6.1 e 6.2).

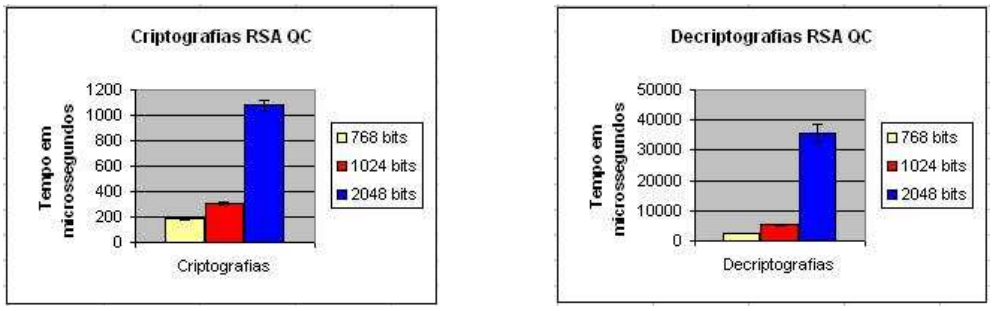

Figura 6.2: Criptografias e decriptografias do RSA QC (referente à tabela 6.2).

Podemos fazer uma estimativa teórica do *Speedup* do RSA QC em relação ao RSA tradicional em termos do número de exponenciações executadas. Sabemos que o RSA tradicional executa uma única exponenciação sobre o módulo de n bits. Já o RSA QC necessita de duas exponenciações usando um módulo de $n/2$ bits. É conhecido, que os algoritmos que calculam exponenciações da forma $C^d \ mod \ p$ levam tempo $O(log \ d \ log^2 \ p)$. Assim quando d é da ordem de p, estes algoritmos levam tempo $O(log^3 p)$. Calculamos então um *Speedup* teórico do RSA QC em relação ao RSA tradicional, fazendo:

$$\frac{n^3}{2(n/2)^3} = 4.$$

O cálculo acima nos mostra que temos um ganho de cerca de 4 vezes do RSA QC em relação ao RSA tradicional nas exponenciações inerentes à decriptografia. Na realidade, para uma análise completa deste ganho, deveríamos adicionar o tempo gasto pela aplicação do TCR

no RSA QC, o que traria uma redução no resultado obtido acima. Portanto, pelo *Speedup* prático dado pela equação 5.3, obtivemos um valor inferior, e que varia de acordo com o tamanho do módulo, chegando a 3,47 para um módulo de 2048 bits (tabela 6.3).

n	$S_{RSA}(n)$
768 bits	3,241591
1024 bits	3,328107
2048 bits	3,473493

Tabela 6.3: *SpeedupRSA* relativo ao RSA QC, para os módulos de 768, 1024 e 2048 bits.

6.3 Implementação, Desempenho e Segurança do RSA em Lote

O método empregado pelo RSA em Lote para decriptografar mensagens não se adapta às versões existentes do PKCS #1. Acreditamos que por utilizar b chaves juntamente com b expoentes públicos na decriptografia, esta variação será dificilmente adaptada a futuras versões desta especificação.

As chaves utilizadas em nossa implementação do RSA em Lote são da seguinte forma:

- Chave pública : $< N, e_1, e_2, ..., e_b >$.
- Chave particular: $< p, q, N, d_1, d_2, ..., d_b >$

onde $e_1, e_2, ..., e_b$ são os expoentes públicos responsáveis por codificar as b mensagens originais e $d_1, d_2, ..., d_b$ são os expoentes particulares responsáveis por decodificar as b mensagens codificadas. Os expoentes foram agrupados pelo algoritmo de geração de chaves para facilitar tanto a criptografia, quanto a decriptografia. É claro que, para o uso real do RSA em Lote, devemos implementar uma espécie de receptador de mensagens (a serem decodificadas ou assinadas) e chaves particulares que possuam o mesmo módulo RSA, para depois utilizar seu algoritmo de decriptografia (para mais detalhes veja seção 6.9). Dan Boneh e Hovav Shacham propuseram um algoritmo para tal finalidade que pode ser visto em [3].

O RSA em Lote foi implementado com parâmetro $b = 2, 4, 6, 8$ utilizando expoentes públicos menores possíveis tais como 3, 5, 7, 11 (pois pela própria natureza do algoritmo de decriptografia, se utilizarmos expoentes públicos grandes temos mais desvantagens que vantagens), calculando o tempo gasto pelas decriptografias em microssegundos. Um outro fato importante e que já foi detalhado no capítulo anterior é o de que nossa implementação do RSA em Lote utiliza a técnica empregada na decriptografia do RSA QC.

Os resultados obtidos estão dispostos nas tabelas 6.4 e 6.3. Podemos analisar que o tempo de criptografia é baixo se comparado com o RSA ou com o RSA QC (a criptografia de quatro mensagens sobre um módulo de 1024 bits leva 378 microssegundos, enquanto a criptografia de uma mensagem no RSA ou RSA QC sobre o módulo de 1024 bits leva cerca de 308 microssegundos). Isso acontece, pois utilizamos expoentes públicos extremamente pequenos no RSA em Lote, enquanto utilizamos $e = 65537$ no RSA e no RSA QC.

n (em bits)	**Batch b = 2**				**Batch b = 4**			
	Criptografia		**Decriptografia**		**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
768	98	26	2891	98	231	8	3943	121
1024	159	10	5955	106	378	13	7598	175
2048	538	14	37565	1638	1284	50	41750	346

Tabela 6.4: Desempenho do RSA em Lote para b = 2 e b = 4.

n (em bits)	**Batch b = 6**				**Batch b = 8**			
	Criptografia		**Decriptografia**		**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
768	369	26	5221	132	535	12	6590	210
1024	609	31	9572	195	888	47	11696	228
2048	2097	90	48001	755	3066	68	54192	452

Tabela 6.5: Desempenho do RSA em Lote para b = 6 e b = 8.

Para compararmos o desempenho obtido pelo RSA em Lote com as demais variações, resolvemos considerar o tempo gasto para criptografar e decriptografar uma única mensagem quando usamos o parâmetro $b = 4$. Dividimos então o tempo das criptografias e decriptografias por quatro. O resultado pode ser visto na figura 6.3.

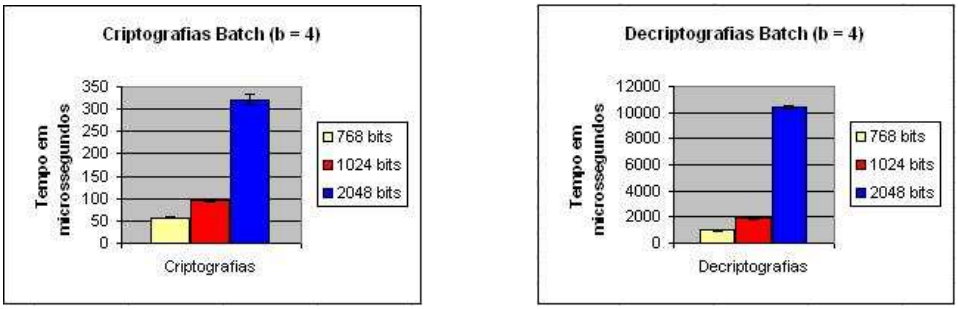

Figura 6.3: Criptografias e decriptografias do RSA em Lote ($b = 4$).

Se considerarmos os valores descritos na figura 6.3 podemos estimar, através da equação 5.3, o ganho do RSA em Lote em relação ao RSA QC nas decriptografias (*SpeedupQC*). O resultado para $b = 2, 4, 6, 8$ pode ser visto na tabela 6.6 e na figura 6.4.

n	$S_{QC}(n)$ b=2	$S_{QC}(n)$ b=4	$S_{QC}(n)$ b=6	$S_{QC}(n)$ b=8
768 bits	1,686614	2,473244	2,801762	2,959636
1024 bits	1,778002	2,787049	3,318429	3,621067
2048 bits	1,903101	3,424671	4,468032	5,276794

Tabela 6.6: *SpeedupQC* relativo ao RSA em Lote usando b mensagens.

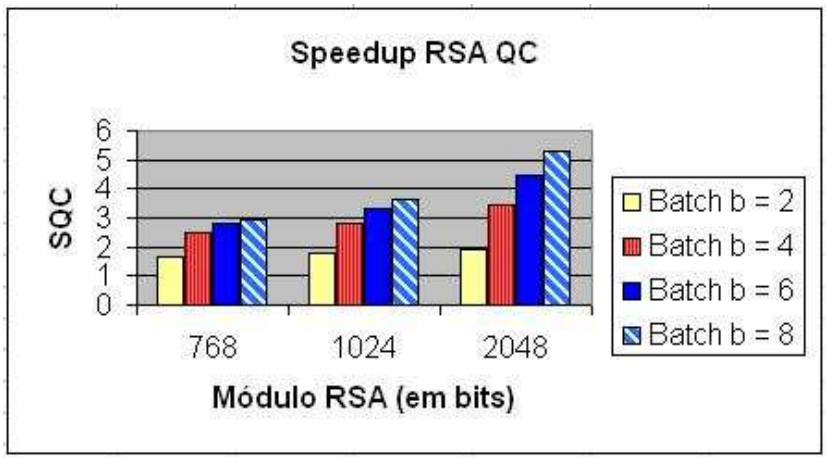

Figura 6.4: *SpeedupQC* relativo ao RSA em Lote, para b = 2, 4, 6, 8.

Constatamos pela figura 6.4 que o ganho cresce conforme aumentamos o parâmetro b. Este fato pode nos levar a concluir erroneamente que, para obtermos resultados melhores, basta

incrementarmos tal parâmetro. Isso não é verdade uma vez que o parâmetro b está intimamente ligado ao agrupamento de chaves e mensagens, que é um processo lento. Para ficar mais claro, suponha que escolhemos um valor de b muito grande, por exemplo 1000, e suponha que cada requisição demorou cerca de 1 segundo para chegar. Nestas condições, para decriptografar as primeiras 1000 mensagens simples utilizando qualquer um dos módulos tratados, levaremos muito mais tempo para agrupá-las do que para decriptografá-las. Então, ao implementarmos o RSA em Lote, por exemplo em um servidor, devemos fazer antes uma análise de qual é o melhor valor de b, de acordo com a quantidade de requisições de decriptografias.

Segurança do RSA em Lote

Baseando-se em um limite assintótico necessário para obter $m^{1/e} \; mod \; N$, Fiat [9] mostrou que o RSA em Lote é tão seguro quanto o RSA tradicional. Esta análise só diz respeito à dificuldade de se extrair a e-ésima raiz módulo N. Podemos analisar a segurança do RSA em Lote de outra forma se considerarmos os parâmetros por ele utilizados.

Devido ao fato de o desempenho do RSA em Lote depender do uso de expoentes públicos pequenos, os ataques sobre este tipo de expoentes (seção 3.5.1) atuam com maior chance de sucesso quando utilizamos este método. Ademais, por ser preciso usar chaves que utilizam o mesmo módulo, este esquema também apresenta uma maior fragilidade ao ataque do Módulo Comum (seção 3.5.4). Uma alternativa que ameniza o dano produzido pelo uso de expoentes públicos pequenos seria a concatenação de alguns bits aleatórios na mensagem (veja seção 3.5.1). Já para resolver o problema do módulo comum, podemos fazer uso de certificados digitais, garantindo assim a autenticidade dos usuários. Como no RSA tradicional ou no RSA QC, devemos tomar as devidas precauções contra os ataques de Ocultamento e de expoente particular pequeno. No último caso devemos garantir que o algoritmo de geração de chaves não forneça expoentes particulares com estas características. Em nossa implementação este ataque não é um problema, já que nosso algoritmo de geração de chaves fornece expoentes particulares da ordem de N.

6.4 Implementação, Desempenho e Segurança do RSA com Múltiplos Primos

O RSA com Múltiplos Primos está descrito na especificação PKCS#1 desde a versão 2.0 (julho de 2000), onde foi adicionado como uma emenda ao documento original. Seguindo a especificação, ao implementar o algoritmo, utilizamos as chaves como segue :

- Chave pública : $< N, e >$
- Chave particular: $< p, q, dp, dq, qInv > \ < r, dr, rInv >$

onde $qInv = q^{-1} \ mod \ p$ e $rInv = (pq)^{-1} \ mod \ r$ (veja seção 5.2). Novamente, o fornecimento das chaves se dá através do algoritmo de geração de chaves e usar a chave particular da forma apresentada acima, não traz nenhum prejuízo para a segurança do sistema, pelo contrário, possibilita o uso do algoritmo de Garner (2.5) para a aplicação do TCR.

Apesar de implementarmos o RSA com Múltiplos Primos de forma genérica (ou seja, para $k > 2$), pelas razões de segurança tratadas na seção 6.4, analisamos seu desempenho somente para o caso $k = 3$. Já a respeito do valor estipulado para o expoente público e, adotamos como habitual, o valor 65537.

Na tabela 6.7 vemos a média aritmética e o desvio padrão (equação 5.1) para os três módulos analisados, dos tempos em microssegundos correspondentes à execução de 1000 criptografias ou 1000 decriptografias (veja seção 5.1).

n (em bits)	**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ
768	185	27	1244	54
1024	324	19	2798	127
2048	1096	44	18113	199

Tabela 6.7: Média aritmética e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA com Múltiplos Primos para os módulos de 768, 1024 e 2048 bits.

Os resultados relativos às criptografias foram semelhantes aos revelados pelo RSA tradicional e pelo RSA QC, por utilizarmos o mesmo algoritmo de criptografia e o mesmo expoente

público empregado por estes esquemas (veja também a figura 6.5). Já nas decriptografias, o RSA com Múltiplos Primos obteve um ganho considerável em relação a essas implementações, perdendo apenas para o RSA em Lote com parâmetro $b \geq 4$.

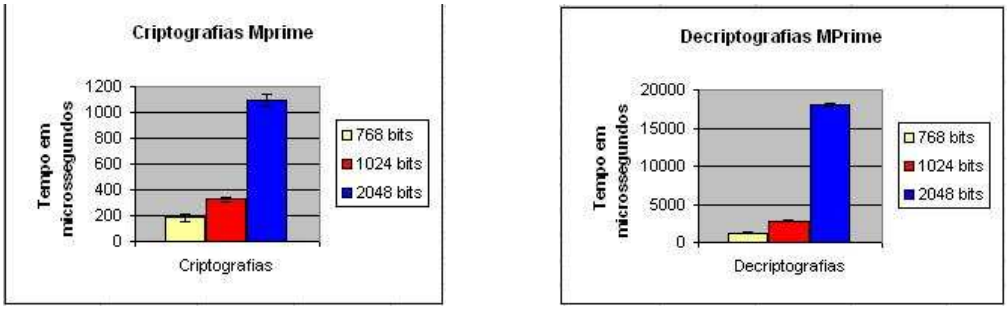

Figura 6.5: Criptografia e decriptografia do RSA com Múltiplos Primos (referente à tabela 6.7).

Se compararmos o RSA com Múltiplos Primos com o RSA QC em termos do número de exponenciações executadas, vemos que o RSA QC necessita de duas exponenciações usando um módulo de $n/2$ bits, já o RSA com Múltiplos Primos necessita de k exponenciações com módulo de n/k bits. Já dissemos que os algoritmos que calculam exponenciações da forma $C^d \ mod \ p$ levam tempo $O(log^3 p)$, quando d é da ordem de p. Então, podemos calcular um *Speedup* teórico do RSA com Múltiplos Primos em relação ao RSA QC fazendo:

$$\frac{2(n/2)^3}{k(n/k)^3} = k^2/4.$$

Assim, usando $k = 3$ teríamos um ganho de 2,25 em relação a decriptografia do RSA QC. Por este resultado, o *Speedup* do RSA com Múltiplos Primos em relação ao RSA QC não depende do tamanho do módulo, o que não é verdade. Isso ocorre porque estamos medindo somente em relação à exponenciação modular, que é a operação mais cara. Utilizando a equação 5.3, que retrata o *Speedup* prático com relação ao RSA QC, obtivemos, portanto, um resultado inferior e que varia de acordo com o tamanho do módulo chegando a 1,97 para um módulo de 2048 bits (tabela 6.8).

n	$S_{RSA}(n)$	$S_{QC}(N)$
768 bits	6,352894	1,959807
1024 bits	6,296998	1,892066
2048 bits	6,854745	1,973444

Tabela 6.8: *Speedup* do RSA com Múltiplos Primos em relação ao RSA tradicional (S_{RSA}) e em relação RSA QC (S_{QC}), para os módulos de 768, 1024 e 2048 bits.

Segurança do RSA com Múltiplos Primos

A segurança do RSA com Múltiplos Primos depende da dificuldade de fatorar inteiros da forma $N = p_1, p_2, ..., p_k$ com $k > 2$. Como visto na seção 3.4, o algoritmo de fatoração *Number Field Sieve* não obtém nenhuma vantagem ao usar o módulo N na forma anterior, já que seu desempenho depende somente do tamanho do inteiro que estamos tentando fatorar, ou seja, n. Já a fatoração pelo *Elliptic Curve Method* (ECM) depende, além do tamanho de N, do tamanho do menor fator primo, e assim, devemos ter cuidado com o tamanho dos primos a serem utilizados. Atualmente o ECM atua com primos até 256 bits de tamanho, ou seja, um módulo de 512 bits com 2 primos usado no RSA tradicional viabiliza o sucesso do ataque, mesmo usando primos balanceados. Diante deste fato, ao implementar o RSA com Múltiplos Primos não devemos usar $k > 2$ para módulos de 768 bits e $k > 3$ para módulos de 1024 bits.

Em uma análise do criptossistema de chave simétrica AES (*American Encryption Standard*), Lenstra [16] estimou para os próximos anos o maior valor possível para k, onde usar o ECM não é mais rápido que usar o NFS, baseando-se na equivalência computacional[2] destes algoritmos. Os resultados obtidos por Lenstra para os módulos de 1024, 2048, 4096 e 8192 estão dispostos na tabela 6.9.

Por este modelo de comparação, se usarmos quatro primos com módulos de 1024 bits no ano 2020, teremos uma equivalência computacional entre o ECM e o NFS, possibilitando a fatoração do módulo não só pelo ECM, como foi descrito anteriormente. Observe também que, mesmo havendo equivalência computacional ao utilizarmos quatro primos com módulos de 2048 bits em 2010, cada fator primo deste módulo deverá ter aproximadamente 512 bits, o que inviabiliza a fatoração segundo o atual limite de atuação do ECM (primos de 256 bits) e do

[2]Dois algoritmos são computacionalmente equivalentes se ao executá-los exigem, na média, o mesmo esforço computacional.

ANO	Módulo (em bits)			
	1024	2048	4096	8192
2001	3	3	4	4
2010	3	4	4	5
2020	4	4	4	5
2030	5	5	5	5

Tabela 6.9: Equivalência computacional ECM vs NFS para RSA com Múltiplos Primos

NFS (módulos de 512 bits).

Lenstra também comparou o tamanho das chaves que igualam o nível de segurança do RSA com Múltiplos Primos com criptossistemas de chaves simétricas. Para igualar a segurança do AES com chave de 256 bits, por exemplo, precisamos usar cinco primos, cada um com 3078 bits.

Com relação aos ataques enunciados na seção 3.5, M. Jason Hinek [14] fez recentemente uma análise dos ataques de Expoente Particular Pequeno e do Ataque com parte da chave exposta sobre o RSA com múltiplos primos. Foi concluído que o ataque de chave parcial com três e quatro primos é ineficaz; obteve-se também uma evidência experimental que sugere que o tempo de execução do ataque é exponencial no tamanho da chave do módulo RSA. Já em relação ao ataque com Expoente Particular Pequeno, Hinek concluiu que o ataque se estende naturalmente ao RSA com Múltiplos Primos e a exemplo do RSA tradicional, deve-se ter cuidado na geração dos expoentes particulares.

Todos os outros ataques vistos na seção 3.5 se aplicam da mesma maneira que o RSA tradicional ao RSA com Múltiplos Primos e, por isso, devem ser seguidas as recomendações descritas nesta seção. Nenhum ataque específico ao RSA com Múltiplos Primos foi encontrado em nossas pesquisas.

6.5 Implementação, Desempenho e Segurança do RSA com Múltiplas Potências

O RSA com Múltiplas Potências, diferentemente do RSA com Múltiplos Primos, não está especificado em nenhuma das versões existentes do PKCS #1. Ademais, no processo de decriptografia esta variação faz uso do expoente público (o que não é comum) podendo dificultar a padronização. Para utilizarmos o algoritmo de Garner, tornando a comparação com o RSA com Múltiplos Primos mais realista, fizemos uso das chaves pública e particular na forma:

- Chave pública : $< N, e >$
- Chave particular: $< p, q, dp, dq, qInv, e >$

onde $qInv = q^{-1} \, mod \, p$ segundo a seção 5.2. Mais uma vez, utilizamos o expoente público e padrão, ou seja, $e = 65537$. A escolha deste expoente é muito importante nesta variação, pois valores grandes ocasionariam a queda no desempenho da decriptografia. Como medida de segurança (veja seção 6.5), adotamos $k = 2$ para os módulos analisados.

Na tabela 6.10 verificamos os resultados obtidos pela execução de 1000 criptografias ou 1000 decriptografias, segundo o critério descrito na seção 5.1. Como era de se esperar, na criptografia em que utilizamos o expoente público padrão com o mesmo algoritmo utilizado pelo RSA tradicional ou RSA QC, obtivemos resultados semelhantes à implementação desses esquemas. Na decriptografia, porém, o RSA com Múltiplas Potências obteve um ganho considerável em relação a estas implementações, sendo melhor até mesmo que o RSA com Múltiplos Primos.

n (em bits)	**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ
768	184	25	977	43
1024	321	15	2083	60
2048	1101	33	12773	191

Tabela 6.10: Média aritmética e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA com Múltiplas Potências para os módulos de 768, 1024 e 2048 bits.

Analisando a média aritmética e o desvio padrão do RSA em Lote com parâmetro $b = 4$ e do RSA com Múltiplas Potências, percebemos que existe uma certa igualdade (com uma leve vantagem para o RSA com Múltiplas Potências) para o módulo de 768 bits, entretanto essa proximidade não se mantém para os demais módulos, para os quais o RSA em Lote se mostra mais eficiente (veja figuras 6.3 e 6.6).

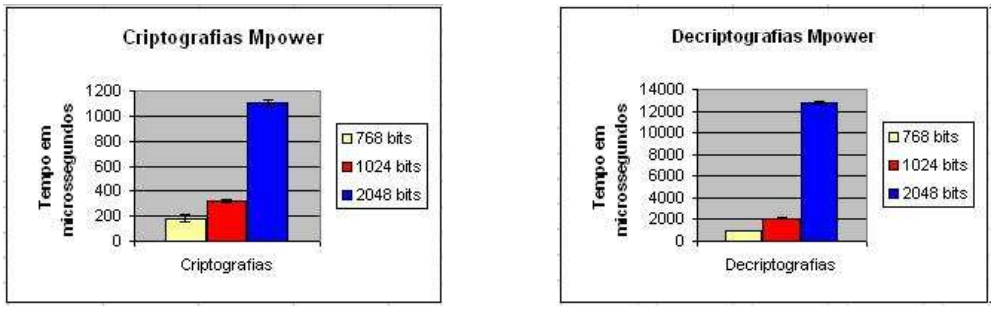

Figura 6.6: Criptografias e decriptografias do RSA com Múltiplas Potências (referente à tabela 6.10).

Para melhor visualizarmos os ganhos obtidos pelo RSA com Múltiplas Potências podemos compará-lo com o RSA QC, em termos do número de exponenciações executadas. Sabemos que o RSA QC necessita de duas exponenciações usando um módulo de $n/2$ bits, enquanto o RSA com Múltiplas Potências necessita de duas exponenciações com módulo de $n/(k+1)$ bits. Considerando que os algoritmos para calcular exponenciações levam tempo $O(log^3 p)$ quando d é da ordem de p, calculamos um *Speedup* teórico fazendo:

$$\frac{2(n/2)^3}{2(n/(k+1))^3} = (k+1)^3/8$$

Por esta fórmula quando usamos $k = 2$ obtemos um ganho de 3,37. Veja que a fórmula acima não depende do tamanho do módulo. Isso ocorre porque estamos comparando os métodos somente em relação a exponenciação modular, que é a operação mais cara. Na prática, entretanto, devemos considerar a atuação do algoritmo utilizado para o cálculo do Mp' (seção 4.2.3) assim como algoritmos de inversão, multiplicação modular e outros.

Ao utilizar a equação 5.3 que descreve o *Speedup* prático em relação ao RSA QC, obtivemos como esperado, um resultado inferior e que varia de acordo com o tamanho do módulo, chegando a 2,79 para um módulo de 2048 bits (tabela 6.11).

n	$S_{RSA}(n)$	$S_{QC}(n)$
768 bits	8,089048	2,495394
1024 bits	8,458473	2,541527
2048 bits	9,720504	2,798481

Tabela 6.11: *Speedup* do RSA com Múltiplas Potências em relação ao RSA tradicional e ao RSA QC, para os módulos de 768, 1024 e 2048 bits.

Segurança do RSA com Múltiplas Potências

A segurança do RSA com Múltiplas Potências depende da segurança proporcionada pelo módulo $p^k q$. Como já foi dito na seção 3.4, o tempo de execução do *Number Field Sieve* (NFS) é estimado no total do número de bits do inteiro a ser fatorado, sendo impraticável sua aplicação a inteiros com mais de 768 bits. Como no RSA com múltiplas potências temos o módulo na forma $N = p^k q$, devemos utilizar um N com mais de 768 bits (ou seja, primos maiores que 256 bits) para evitar a fatoração mencionada. A técnica utilizada pelo *Elliptic Curve Method* (ECM) é diferente, e busca encontrar primos que são divisores do inteiro a ser fatorado. O seu tempo de execução é estimado em termos do tamanho de bits do menor fator primo. Sabemos, contudo, que escolhendo primos com mais que 256 bits tornamos este método impraticável, ou seja, ao escolhermos valores para p e q maiores que 256 bits, nenhum dos dois algoritmos mencionados acima é viável e o criptossistema pode ser considerado seguro contra a fatoração (por meio destes métodos). Salientamos que, como conseqüência do que foi dito acima, não devemos usar $k > 1$ para módulos de 768 bits e $k > 2$ para módulos de 1024 bits.

Analisado o RSA com Múltiplas Potências com relação aos dois métodos de fatoração mais conhecidos, devemos nos perguntar se existem outros algoritmos de fatoração que exploram especificamente a forma do módulo $N = p^k q$. Boneh, Durfee e Howgrave-Graham [15] propuseram um algoritmo para fatorar inteiros com esta forma, porém, o algoritmo proposto por eles trabalha bem somente com valores grandes de k (próximos a $\sqrt{logp}$), caso em que chega a ser na verdade até assintoticamente mais rápido que o ECM. Para $N = p^2 q$, portanto, este algoritmo não traz melhoras na fatoração. Um algoritmo de fatoração baseado no ECM, que explora números na forma $p^2 q$, foi apresentado por Peralta e Okamoto [22]. O tempo de execuçao deste algoritmo é só um pouco mais rápido que o ECM, o que não o torna uma ameaça

para primos maiores que 256 bits.

Atualmente, não existem algoritmos de fatoração que consigam fatorar números da forma p^2q em tempo polinomial. Takagi [29] nos lembra que o RSA com Múltiplas Potências não é o único criptossistema que se baseia na dificuldade de fatorar inteiros na forma p^2q. Existem dois outros criptossistemas de chave pública e um sistema de assinatura digital apoiados no mesmo conceito.

Os ataques citados na seção 3.5 atuam no RSA com Múltiplas Potências da mesma maneira que atuam sobre o RSA tradicional e portanto, devemos seguir as recomendações referidas nessa seção. Nenhum ataque específico ao RSA com Múltiplas Potências foi encontrado em nossa pesquisa.

6.6 Implementação, Desempenho e Segurança do RSA Rebalanceado

Por utilizar o mesmo algoritmo de criptografia e decriptografia do RSA QC, o RSA Rebalanceado é totalmente compatível com a especificação PKCS #1. Seguindo esta especificação, utilizamos as chaves pública e particular na forma:

- Chave pública : $< N, e >$
- Chave particular: $< p, q, dp, dq, qInv, e >$

onde $qInv = q^{-1}\ mod\ p$ segundo a seção 5.2; deixando, mais uma vez, a cargo do algoritmo de geração de chaves a responsabilidade de fornecer chaves com expoentes adequados.

A média aritmética e o desvio padrão dos tempos em microssegundos referentes às criptografias e decriptografias, se encontram na tabela 6.12 e na figura 6.7. Podemos observar que, como neste método, o algoritmo de criptografia utiliza um expoente público e muito maior que o padrão, obtivemos um péssimo desempenho na criptografia. Em contrapartida, na decriptografia, o RSA Rebalanceado faz uso de um expoente particular d que é pequeno módulo $p - 1$ e $q - 1$ e, portanto, obteve um ótimo desempenho. É fácil notar que, usando módulos de 2048 bits, esta variação obteve os melhores resultados dentre as apresentadas até o momento.

n (em bits)	**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ
768	7959	232	967	60
1024	17613	190	1748	67
2048	124065	5412	5977	180

Tabela 6.12: Média aritmética e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA Rebalanceado para os módulos de 768, 1024 e 2048 bits.

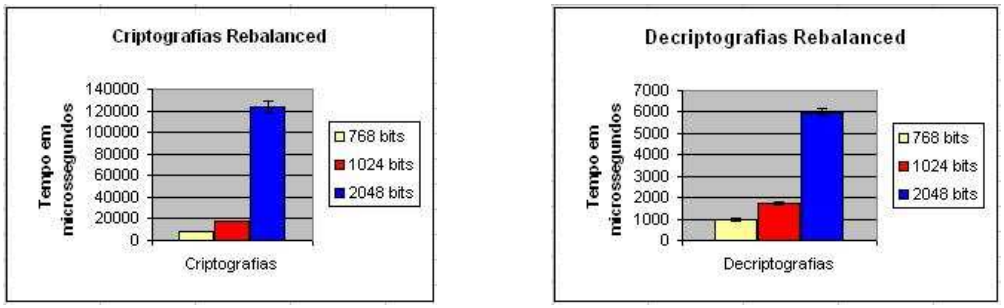

Figura 6.7: Criptografias e decriptografias do RSA Rebalanceado (referente à tabela 6.12).

Para módulos de 768 bits, se considerarmos a média aritmética ($\overline{x}$) e o respectivo desvio padrão (σ) das 1000 mensagens, tanto do RSA Rebalanceado quanto do RSA com Múltiplas Potências e do RSA em lote, notaremos que a diferença entre os valores é mínima, isto é, neste caso particular, as três variações obtêm resultados semelhantes (veja a figura 6.8). Analisando, por fim, os tempos das decriptografias usando módulos de 1024 bits, o RSA Rebalanceado só obteve desempenho inferior à implementação do RSA em Lote (com parâmetros $b = 6$ e $b = 8$), e à implementação do RSA Rebalanceado com Múltiplos Primos, vista na próxima seção (seção 6.7).

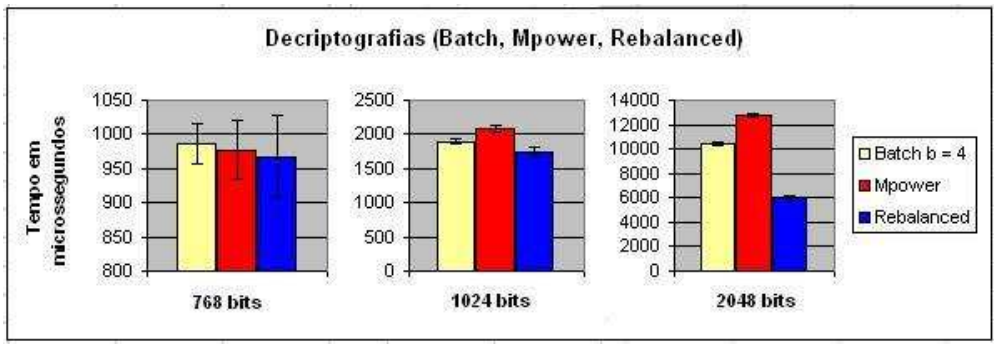

Figura 6.8: Situação conflituosa (RSA em Lote, RSA com Múltiplas Potências e RSA Rebalanceado).

Esses resultados ficam mais claros se compararmos o RSA Rebalanceado com o RSA QC em termos do número de exponenciações executadas. Nesta análise, verificamos que ambos necessitam de duas exponenciações usando um módulo de $n/2$ bits, porém o RSA Rebalanceado executa estas exponenciações com expoentes particulares de s bits de tamanho. Considerando que os algoritmos para calcular exponenciações levam tempo $O(logdlog^2p)$ e que $d\ mod\ (p-1)$ (ou $q-1$) é da ordem de s, podemos calcular um *Speedup* teórico fazendo:

$$\frac{2(n/2)^3}{2(s(n/2)^2)} = n/2s$$

Em outras palavras, para um módulo de 1024 bits e $s = 160$ obtemos um ganho de 3,20. Mais uma vez, este resultado é uma aproximação teórica do *Speedup* em relação às exponenciações modulares dos dois algoritmos e portanto não considera, por exemplo, a aplicação do TCR. Ao calcular o *Speedup* prático em relação ao RSA QC dado pela equação 5.3, obtivemos um ganho de 3,02 para um módulo de 1024 bits e de 5,98 para um módulo de 2048 bits (tabela 6.13).

n	$S_{RSA}(n)$	$S_{QC}(n)$
768 bits	8,172699	2,5212
1024 bits	10,07952	3,028604
2048 bits	20,77296	5,980425

Tabela 6.13: *Speedup* do RSA Rebalanceado em relação ao RSA tradicional (S_{RSA}) e ao RSA QC (S_{QC}), para os módulos de 768, 1024 e 2048 bits.

Observe que, como esperado, os resultados obtidos pelo *Speedup* prático variam bruscamente com o módulo. Isso ocorre devido ao tamanho do expoente particular, que, quando reduzido módulo $p-1$ (ou $q-1$), assume s bits de tamanho.

Segurança do RSA Rebalanceado

A segurança do RSA Rebalanceado depende da segurança oferecida pelo expoente particular d quando este é pequeno módulo $p-1$ e $q-1$. Trata-se de um problema ainda em aberto. O que sabemos é que, como o d utilizado por este esquema é da ordem de N (ou seja, grande) os ataques a expoentes particulares pequenos se tornam ineficazes. Também fica claro que os

ataques a expoentes públicos pequenos não são um problema, desde que o expoente público e seja bem maior que o utilizado como padrão. Neste aspecto, o RSA Rebalanceado é até mais seguro que as outras variações, se estas utilizarem o expoente público padrão.

Com relação à fatoração, por ter o módulo na forma $N = pq$ como no RSA tradicional, o RSA Rebalanceado pode fazer uso de módulos de 768 bits sem se preocupar com os atuais algoritmos de fatoração, diferentemente das variações com múltiplos fatores.

Na verdade, o melhor ataque conhecido sobre o RSA Rebalanceado [2] é capaz de fatorar N em tempo $O(min(\sqrt{d_p}, \sqrt{dq}))$, ou seja, para obtermos uma segurança de 2^{80} (2^{80} tentativas)[3] precisamos garantir que d_p e d_q sejam de pelos menos 160 bits de tamanho (pois $\sqrt{2^{160}} = 2^{80}$), valor que utilizamos em nossa implementação.

6.7 Implementação, Desempenho e Segurança do RSA Rebalanceado com Múltiplos Primos

É fácil perceber que o RSA Rebalanceado com Múltiplos Primos é totalmente compatível com o PKCS #1. Isso ocorre porque dos três algoritmos utilizados pelo RSA Rebalanceado com Múltiplos Primos, o algoritmo de geração de chaves é o único que difere dos três algoritmos utilizados pelo RSA com Múltiplos Primos. Fazendo uso desta compatibilidade, utilizamos as chaves como segue :

- Chave pública : $< N, e >$
- Chave particular: $< p, q, dp, dq, qInv, r, dr, rInv >$

onde $qInv = q^{-1}\ mod\ p$ e $rInv = (pq)^{-1}\ mod\ r$ segundo a seção 5.2.

Pelas razões de segurança descritas na seção 6.7, analisamos somente módulos formados por três primos. A tabela 6.14 e a figura 6.9 ilustram os resultados concernentes às criptografias e decriptografias. Como no RSA Rebalanceado, o algoritmo de criptografia utiliza um expoente público e muito maior que o padrão, o que proporciona um péssimo desempenho. Contudo, na

[3]Para medida de comparação, lembramos que o algoritmo DES original possui 2^{56} chaves ou possibilidades para a quebra.

decriptografia o RSA Rebalanceado com Múltiplos Primos (usando três primos), faz uso de um expoente particular d que é pequeno módulo $p-1$, $q-1$ e $r-1$, além de fazer exponenciações sobre módulos menores, ocasionando o melhor desempenho entre todas as variações estudadas.

n (em bits)	**Criptografia**		**Decriptografia**	
	$\overline{x}$	σ	$\overline{x}$	σ
768	7882	127	811	55
1024	18667	245	1363	60
2048	126918	6889	4564	136

Tabela 6.14: Média e Desvio Padrão em microssegundos da criptografia e decriptografia do RSA Rebalanceado com Múltiplos Primos para módulos de 768, 1024 e 2048 bits.

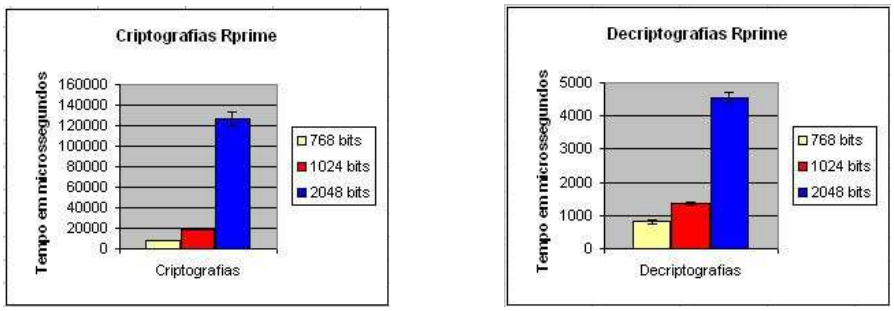

Figura 6.9: Criptografias e decriptografias do RSA Rebalanceado com Múltiplos Primos (referente à tabela 6.14).

Para melhor compreender o ganho proporcionado pela decriptografia do RSA Rebalanceado com Múltiplos Primos, novamente tomamos como base as exponenciações relativas ao RSA QC. O RSA QC utiliza duas exponenciações com módulos de $n/2$ bits; o RSA Rebalanceado com Múltiplos Primos, por sua vez, faz uso de k exponenciações sobre módulos de n/k bits (lembrando que cada uma destas exponenciações é realizada com expoentes particulares de s bits de tamanho). Já foi dito que os algoritmos para calcular exponenciações levam tempo $O(logdlog^2p)$ e sabendo que $d\ mod\ p_i - 1$ é da ordem de s, podemos calcular um *Speedup* teórico fazendo:

$$\frac{2(n/2)^3}{k(s(n/k)^2)} = nk/4s.$$

Este resultado nos mostra que, para um módulo de 1024 bits, usando três primos e $s = 160$, obtemos um ganho de 4,8. Por ser uma aproximação teórica do *Speedup* em relação às

exponenciações modulares dos dois algoritmos, este resultado não considera, por exemplo, a aplicação do TCR. Na prática, através da equação 5.3, calculamos o *SpeedupQC*, obtendo um ganho de 3,88 para um módulo de 1024 bits e 7,83 para um módulo de 2048 bits (tabela 6.15).

n	$S_{RSA}(n)$	$S_{QC}(n)$
768 bits	9,74476	3,006165
1024 bits	12,92663	3,884079
2048 bits	27,20421	7,831946

Tabela 6.15: *Speedup* do RSA Rebalanceado com Múltiplos Primos em relação ao RSA tradicional e ao RSA QC, para os módulos de 768, 1024 e 2048 bits.

Comparando o RSA Rebalanceado com Múltiplos Primos com o RSA Rebalanceado, obtivemos um ganho de cerca de 30% para um módulo de 2048 bits. Veja a tabela 6.16, onde $T_{Reb}(n)$ é o tempo da decriptografia do RSA Rebalanceado em microssegundos usando um módulo de n bits e $T_{Rprime}(n)$ é o tempo da decriptografia do RSA Rebalanceado com Múltiplos Primos usando um módulo de n bits.

n	$T_{Reb}(n)/T_{Rprime}(n)$
768 bits	1,192355
1024 bits	1,282465
2048 bits	1,309597

Tabela 6.16: *Speedup* do RSA Rebalanceado com Múltiplos Primos com relação ao RSA Rebalanceado.

Repare que, pelo mesmo motivo apresentado ao RSA Rebalanceado, esta variação aumenta bruscamente seu desempenho na decriptografia à medida que incrementamos o tamanho do módulo RSA.

Segurança do RSA Rebalanceado com Múltiplos Primos

Claramente, a segurança do RSA Rebalanceado com Múltiplos Primos, assim como no RSA Rebalanceado, depende da segurança oferecida pelo expoente particular d (com as características descritas acima) e do tamanho dos primos utilizados (como no RSA com Múltiplos

Primos). Sabemos que o d utilizado é suficientemente grande para tornar ineficazes os ataques a expoentes particulares pequenos. E, novamente, os ataques a expoentes públicos pequenos não são um problema, devido ao uso do expoente público e maior que o utilizado como padrão.

Utilizando expoentes d_p, d_q e d_r de 160 bits cada (para $k = 3$), conseguimos evitar a fatoração de N pelo ataque mencionado na seção 6.6. Já, para evitar a fatoração pelos métodos NFS e ECM, devemos utilizar primos maiores que 256 bits, ou seja, para um módulo de 1024 bits devemos utilizar no máximo três primos, e não podemos usar mais que dois primos para um módulo de 768 bits, seguindo os mesmos critérios utilizados pelo RSA com Múltiplos Primos.

6.8 Facilidade de Implementação e Implantação das Variações

Dentre as variações estudadas, por utilizar uma estrutura em árvore e por necessitar de um agrupamento de mensagens e de suas respectivas chaves para realizar a decriptografia, a do RSA em Lote pode ser considerada a que oferece maior dificuldade de implementação. Dan Boneh e Hovav Shacham apresentaram um algoritmo iterativo e que não faz uso de uma arvóre para valores de $b \leq 8$. Os mesmos propuseram um algoritmo para fazer o agrupamento necessário pelo RSA em Lote [3, 25], conseguindo dobrar o número de requisições atendidas em um servidor SSL [3] (usando $b = 8$). Em nossa implementação, como já dissemos, consideramos apenas o tempo das decriptografias sem considerar o tempo do agrupamento de mensagens e chaves. Entretanto, ao implementá-la para fins práticos, devemos ter em mente que seu desempenho está intimamente ligado a esse agrupamento. Outra característica que devemos considerar é que cada par de chaves necessita de um certificado digital para garantir a autenticidade, o que aumenta o custo da implantação.

Os algoritmos de criptografia e de decriptografia utilizados pelo RSA Rebalanceado são de fácil implementação, por serem os mesmos utilizados pelo RSA QC. Assim, quem já faz uso do RSA QC em suas aplicações, pode usufruir das vantagens apresentadas pelo RSA Rebalanceado, bastando implementar o algoritmo de geração de chaves deste ou utilizar uma entidade externa confiável capaz de gerar tais chaves. O custo de conversão de um sistema para outro é, portanto, relativamente baixo. O mesmo acontece com o RSA com Múltiplos Primos e com o RSA Rebalanceado com Múltiplos Primos, pois quem já utiliza o primeiro não precisa

nada mais que chaves apropriadas para utilizar o segundo, podendo criar seu próprio algoritmo de geração de chaves, como foi feito em nosso trabalho, ou então receber tais chaves de uma entidade confiável.

Apesar de ser de fácil implementação, o RSA com Múltiplas Potências só mantém do RSA QC o algoritmo de criptografia. Portanto, o desenvolvedor que possui um sistema utilizando o RSA tradicional, o RSA QC ou o RSA com Múltiplos Primos e deseja convertê-lo para utilizar o RSA com Múltiplas Potências, precisará alterar tanto o algoritmo de geração de chaves, quanto o algoritmo de decriptografia.

Um outro detalhe que deve ser analisado é que implementações como as do RSA com Múltiplos Primos, RSA Rebalanceado com Múltiplos Primos e até a do próprio RSA em Lote, pela natureza de seus algoritmos, podem levar vantagem sobre as outras se utilizadas com máquinas multiprocessadas ou se implementadas em hardware. Por exemplo, uma máquina com quatro processadores executando o RSA com Múltiplos Primos (com quatro primos) pode paralelizar as exponenciações e obter resultados ainda melhores.

6.9 Considerações sobre os Algoritmos de Geração de Chaves

Mesmo não sendo o foco do nosso trabalho a comparação entre os algoritmos de geração de chaves utilizados, descrevemos aqui os principais aspectos notados durante a implementação destes. Inicialmente podemos considerar que estes algoritmos são, entre todos estudados aqui, os que apresentam maior complexidade com relação à implementação. Isso se deve, principalmente, às condições impostas por estes, que podem tornar o processo de geração de chaves, muitas vezes, excessivamente lento se comparado com os demais.

Em referência aos aspectos de parametrização, utilizamos, em todas as variações, primos balanceados (veja seção 3.4), a fim de dificultar a atuação do algoritmo de fatoração ECM. Ademais, utilizamos sempre que viável o valor 65537 para o expoente público e, visando uma análise mais consistente. O tamanho do expoente particular d também respeitou os limites de segurança impostos pela observação de Wiener (veja seção 3.5.2) ficando em torno de n bits.

Essa medida também contribui para uma melhor análise comparativa das variações.

Tratando mais especificamente do reaproveitamento de código, os algoritmos de geração de chaves utilizados pelo RSA tradicional e pelo RSA QC são praticamente idênticos, diferenciando-se apenas por alguns valores referentes à saída da chave particular. No RSA tradicional, a saída da chave particular corresponde a $< N, d >$, enquanto que no RSA QC a saída é dada por $< p, q, dp, dq, qInv >$ [4]. Assim, o algoritmo de geração de chaves do RSA QC teria que calcular a inversa de q módulo p dada por $qInv$, e os coeficientes dp e dq dados por $d \ mod \ (p-1)$ e $d \ mod \ (q-1)$, respectivamente. A inclusão dessas operações ao algoritmo de geração de chaves do RSA tradicional não traz prejuízos significativos ao seu desempenho, em termos de tempo para geração de chaves.

No RSA em Lote utilizamos um algoritmo de geração de chaves semelhante ao utilizado pelo RSA QC. Modificamos, entretanto, este algoritmo para gerar não duas, mas $2b$ chaves (b públicas e b particulares) com expoentes públicos menores possíveis e que fossem primos entre si. Todas as chaves públicas foram armazenadas em um único arquivo público, enquanto todas as chaves particulares foram armazenadas em um único arquivo particular [5]. Estas condições fizeram com que o algoritmo de geração de chaves do RSA em Lote obtivesse o pior desempenho em termos do tempo para a geração de chaves do que os demais. Contudo, lembramos que, na prática, os algoritmos de criptografia e decriptografia do RSA em Lote podem usar chaves geradas pelo algoritmo de geração de chaves do RSA QC e fazer a leitura de b chaves (públicas ou particulares), em lugar de uma.

Os algoritmos de geração de chaves utilizados pelo RSA com múltiplos primos e pelo RSA com Múltiplas Potências, além de serem parecidos entre si, não diferem muito do utilizado pelo RSA QC. O algoritmo de geração de chaves utilizado pelo RSA com Múltiplos Primos utiliza k primos, cada um com n/k bits de tamanho, em lugar dos dois primos com $n/2$ bits de tamanho utilizado pelo RSA QC. Portanto, para o RSA com Múltiplos Primos utilizando três primos, teríamos que acrescer ao tempo gasto para gerar os coeficientes do TCR, o tempo de um primo a mais a ser calculado (lembrando, é claro, que os primos utilizados pelo RSA com Múltiplos Primos possuem tamanho menor do que os gerados pelo RSA QC para um mesmo módulo) o

[4] Lembramos que essas saídas são apropriadas para o algoritmo de Garner. Para aplicar o algoritmo clássico do TCR, devemos alterá-las.

[5] Este procedimento foi realizado apenas para facilitar o trabalho do algoritmo de criptografia e decriptografia, e não é representativo do uso real de uma aplicação.

que não traz prejuízos significativos. É fácil perceber que o mesmo ocorre com o RSA com Múltiplas Potências usando $k = 2$.

O RSA Rebalanceado e o RSA Rebalanceado com Múltiplos Primos possuem algoritmos mais complexos de geração de chaves [6], devido ao grande número de condições necessárias para a geração de suas chaves, contudo, apresentam desempenho semelhante aos demais (ao fazer esta afirmação estamos considerando que o RSA em Lote, em condições reais, pode usar o algoritmo de geração de chaves do RSA QC). Atentamos também, que não encontramos nenhum estudo sobre a quantidade de chaves que pode ser gerada seguindo as restrições impostas a estes algoritmos.

Diante dos fatos levantados nesta seção, alguém poderia se perguntar se seria conveniente escolher o método a ser utilizado, baseando-se no desempenho obtido pelos algoritmos de geração de chaves. Todavia, o problema ocasionado por estes algoritmos pode ser contornado, repassando-o para entidades externas (desde que confiáveis) ou até mesmo, pré-calculando as chaves para distribuí-las posteriormente (de preferência codificando-as, é claro). Outro fato, que devemos considerar, é que a geração de chaves é um processo raramente executado se comparado à criptografia e à decriptografia, e, portanto, deve ser analisada apenas em segundo plano. Por este último motivo, neste trabalho nos ocupamos apenas com a geração correta das chaves e não com o tempo gasto para gerá-las.

[6] Um exemplo de uma chave gerada pelo algoritmo de geração de chaves do RSA Rebalanceado com Múltiplos Primos pode ser vista na seção A.2 do Apêndice A.

CAPÍTULO

7

Conclusão

No capítulo anterior analisamos separadamente o desempenho de cada variação, comparado ao desempenho do RSA tradicional ou do RSA QC. Neste capítulo, apresentamos um resumo desta análise, objetivando definir, de maneira mais clara, por qual variação devemos optar em uma dada situação.

7.1 Criptografias

No que diz respeito à criptografia, para todas as variações foi utilizado o mesmo algoritmo e portanto poderíamos esperar resultados semelhantes. Contudo, sabemos que tanto o RSA Rebalanceado, quanto o RSA Rebalanceado com Múltiplos Primos utilizam o expoente público e muito maior que o padrão. Além disso, sabemos que o RSA em Lote, utiliza expoentes públicos extremamente pequenos. Esses fatos fizeram com que os métodos mencionados obtivessem resultados divergentes dos demais (veja tabela 7.1). As pequenas oscilações medidas entre as outras variações se devem principalmente, à aleatoriedade na geração de chaves e de mensagens. O mesmo fato ocorre para a diferença no tempo de criptografia do RSA Rebalanceado com relação ao RSA Rebalanceado com Múltiplos Primos. Na figura 7.1 podemos visualizar estas

observações. Note que aplicamos a função logarítmica sobre o tempo em microsegundos visando possibilitar a comparação entre os valores obtidos no RSA, RSA em Lote, RSA QC, Mprime e Mpower ínfimos em relação à escala original.

n	RSA	Batch 4	RSA QC	Mprime	Mpower	Rebalanced	Rprime
768 bits	185	231	185	185	184	7959	7882
1024 bits	308	378	307	324	321	17613	18667
2048 bits	1080	1284	1080	1096	1101	124065	126918

Tabela 7.1: Tempo em microsegundos relativo à criptografia dos algoritmos estudados.

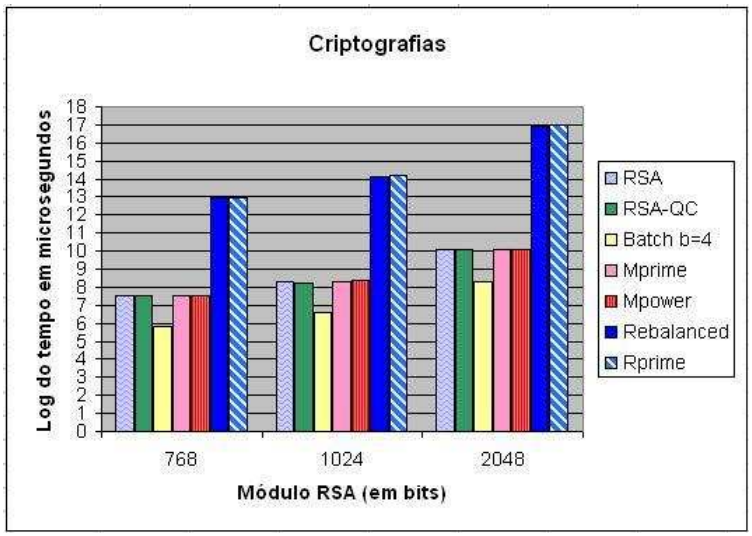

Figura 7.1: Tempo em microsegundos relativo à criptografia dos algoritmos estudados.

7.2 Decriptografias

Os tempos em microsegundos obtidos pela decriptografia de todos os algoritmos analisados estão dispostos na tabela 7.2. O melhor desempenho foi alcançado pelo RSA Rebalanceado com Múltiplos Primos. Lembramos porém, que esta variação junto com o RSA Rebalanceado, obteve o pior desempenho na criptografia por utilizar um expoente público grande. Outra característica que deve ser mencionada é que a exemplo da criptografia, os resultados ilustrados tanto pela tabela quanto pela figura 7.2 com relação ao RSA em Lote, descrevem o tempo em

microsegundos para decodificar apenas uma mensagem em lugar de quatro[1].

n	RSA	Batch 4	RSA QC	Mprime	Mpower	Rebalanced	Rprime
768 bits	7903	3943	2438	1244	977	967	811
1024 bits	17619	7598	5294	2798	2083	1748	1363
2048 bits	124160	41750	35745	18113	12773	5977	4564

Tabela 7.2: Tempo em microsegundos relativo à decriptografia dos algoritmos estudados.

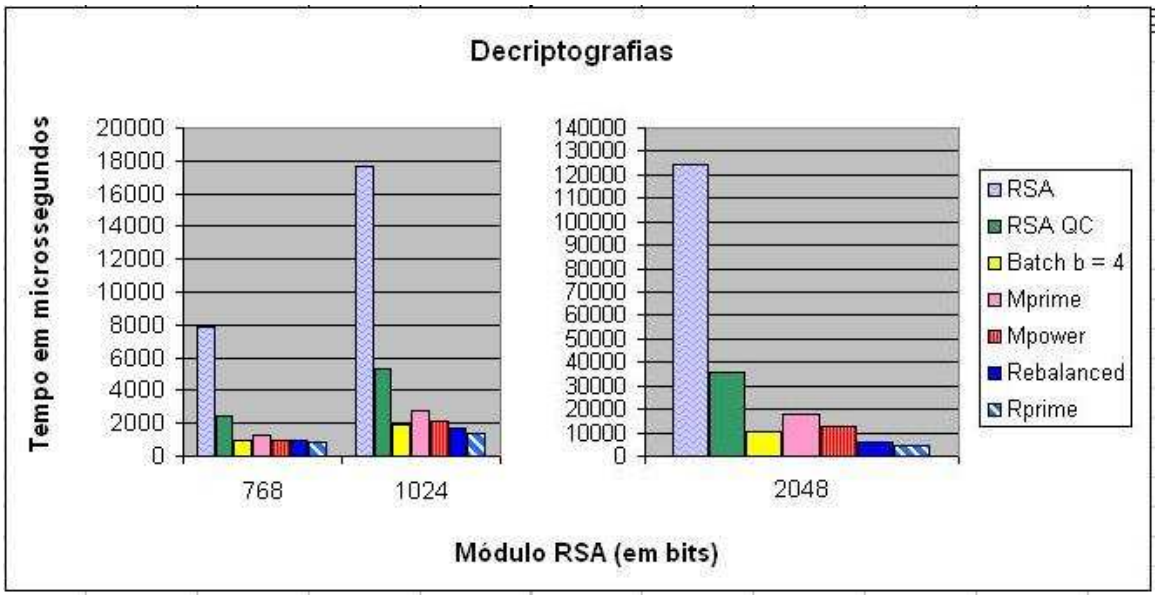

Figura 7.2: Tempo em microsegundos relativo à decriptografia dos algoritmos estudados.

Objetivando facilitar o entendimento dos resultados, a tabela 7.3 e a figura 7.3 resumem o *Speedup* obtido pelas variações com relação a decriptografia do RSA tradicional. Para tais cálculos utilizamos a equação 5.2, com exceção aos valores referentes ao RSA em Lote para o qual usamos $T_{RSA}(n)/(T_{AP}(n)/b)$ (lembrando que, neste caso $b = 4$), segundo a notação descrita no quinto capítulo. No entanto, esta comparação é um pouco falha, pois como descrito na seção 6.9 o RSA em Lote pode gastar mais tempo na aglomeração de mensagens do que na própria decriptografia. Deixando de lado este fato, podemos observar que, para $n = 1024$ e $n = 2048$, o RSA em Lote só perde para o RSA Rebalanceado e para o RSA Rebalanceado

[1]Para estimar tal cálculo dividimos o tempo gasto pela implementação do RSA em Lote pelo número de mensagens decriptografadas (parâmetro b, no caso quatro).

com Múltiplos Primos. Podemos notar também que, para $n = 768$, o RSA em Lote, o RSA com Múltiplas Potências e o RSA Rebalanceado obtiveram desempenho semelhante, entretanto pela natureza de cada uma destas variações, esses resultados não se mantiveram próximos nos demais módulos.

n	RSA QC	Batch 4	Mprime	Mpower	Rebalanced	Rprime
768 bits	3,241591	8,017246	6,352894	8,089048	8,172699	9,74476
1024 bits	3,328107	9,275599	6,296968	8,458473	10,07952	12,92663
2048 bits	3,473493	11,89557	6,854745	9,720504	20,77296	27,20421

Tabela 7.3: *SpeedupRSA* das variações estudadas.

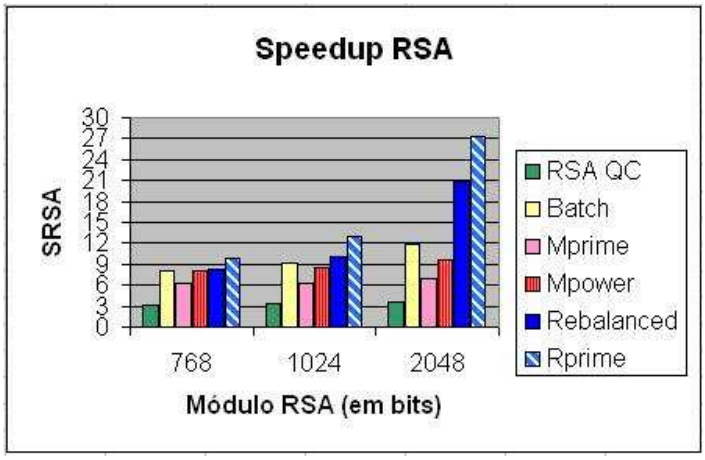

Figura 7.3: *SpeedupRSA* das variações estudadas.

Para quem já utiliza o RSA QC como criptossistema é interessante avaliar o ganho obtido pelas variações apresentadas. Através da tabela 7.4 e da figura 7.4 constatamos que todas as variações revelam um ganho acima de 89% para todos os módulos analisados, chegando ao máximo de 783% no RSA Rebalanceado com Múltiplos Primos usando um módulo de 2048 bits[2].

[2]Com exceção do RSA em Lote, todos os valores ilustrados na tabela 7.4 foram calculados utilizando a equação 5.3.

n	Batch 4	Mprime	Mpower	Rebalanced	Rprime
768 bits	2,473244	1,959807	2,495394	2,5212	3,006165
1024 bits	2,787049	1,892066	2,541527	3,028604	3,884079
2048 bits	3,424671	1,973444	2,798481	5,980425	7,831946

Tabela 7.4: *SpeedupQC* das variações estudadas.

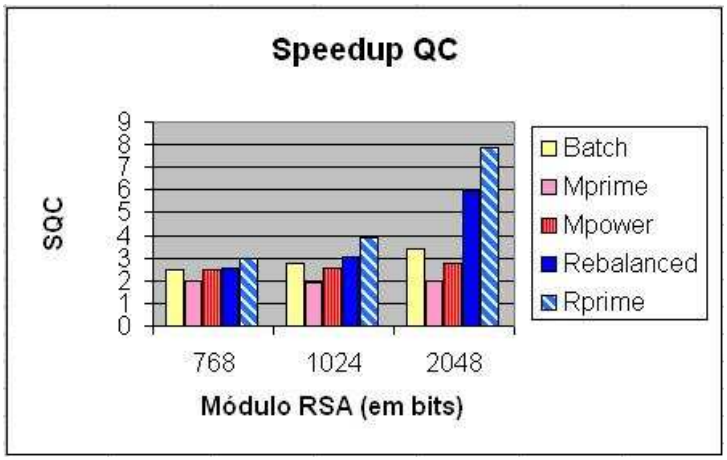

Figura 7.4: *SpeedupQC* das variações estudadas.

7.3 Discussão

Visto que todas as variações podem ser consideradas *seguras*[3], desde que tomadas as providências especificadas no sexto capítulo, o RSA com Múltiplos Primos pode ser considerado a melhor opção para quem deseja obter um bom desempenho na criptografia e decriptografia e além disso deseja interoperar com sistemas que já utilizam o PKCS#1. Apesar de obterem melhor desempenho que o RSA com Múltiplos Primos, o RSA com Múltiplas Potências e o RSA em Lote não são especificados nas versões existentes do PKCS#1, servindo assim como as melhores opções para quem deseja obter uma boa velocidade tanto na criptografia quanto na decriptografia, mas por alguma razão não precisa fazer uso do PKCS #1. Lembramos, mais uma vez, que além de fazer necessário o uso de certificados digitais para garantir a autenticidade das chaves, o RSA em Lote necessita de um aglomerador de mensagens, o que certamente afetará

[3]Lembramos que normalmente, a confiança com relação à segurança de um criptossistema é conseguida empiricamente. No caso, o adjetivo *"seguras"* indica a não existência de ataques eficazes sobre as variações em nossa pesquisa.

o desempenho obtido aqui.

Para as aplicações que priorizam o desempenho na decriptografia e na geração de assinaturas e que desejam interoperar com sistemas que utilizam o PKCS #1, a melhor escolha é o RSA Rebalanceado com Múltiplos Primos, que para módulos de 2048 bits obteve um ganho de 30% com relação ao RSA Rebalanceado (veja seção 4.4) sendo cerca de 27,2 vezes mais rápido que o RSA tradicional (tabela 7.3) e cerca de 7,8 vezes mais rápido que o RSA QC (tabela 7.4). Um outro fato importante a favor desta nova variação é que sistemas que atualmente utilizam o RSA com Múltiplos Primos podem facilmente adaptar-se a ela, bastando alterar o algoritmo de geração de chaves. Podemos até mesmo criar um sistema híbrido de geração de chaves.

7.4 Considerações Finais

Em suma, devido às diferenças existentes entre as variações, é difícil chegar a um acordo sobre qual delas é melhor. Na verdade, como referido acima, para cada aplicação temos uma solução que melhor se enquadra. Devemos então, ao implementar uma das propostas estudadas neste documento, analisar o que buscamos, como por exemplo, velocidade na criptografia, velocidade na decriptografia, padronização; a fim de encontrar a solução que melhor satisfaça nossa necessidade. Para facilitar esse trabalho, sintetizamos na tabela 7.5, os principais resultados observados sobre cada uma das variações.

Variação	Vantagens	Desvantagens
Batch	- Excelente desempenho na criptografia (expoentes pequenos). - Ótimo desempenho na decriptografia, obtendo um ganho de 3,42 sobre o RSA QC, para $b = 4$ e $n = 2048$.	- Necessidade de um aglomerador de mensagens. - Apresenta maior fragilidade sob ataques com expoentes públicos pequenos e módulos comuns, exigindo a emissão de múltiplos certificados digitais. - Não é tratado pela especificação PKCS#1.

Mprime	- Fácil Implementação. - Bom desempenho tanto na criptografia quanto na decriptografia (ganho de 1,97 sobre RSA QC para $n = 2048$) - Especificada pelo PKCS#1.	- Não é recomendado seu uso para módulos menores ou iguais a 768 bits.
Mpower	- Fácil Implementação. - Bom desempenho na criptografia e ótimo desempenho na decriptografia (ganho de 2,79 sobre RSA QC para $n = 2048$).	- Desempenho depende do tamanho do expoente público, que não pode ser muito grande. - Não é recomendado seu uso para módulos menores ou iguais a 768 bits. - Não é tratada pela especificação PKCS#1.
Rebalanced	- Utiliza o mesmo algoritmo de decriptografia que o RSA QC. - Excelente desempenho na decriptografia (ganho de 5,98 sobre RSA QC para $n = 2048$). - Desempenho aumenta consideravelmente com módulos maiores. - Se enquadra na especificação PKCS#1.	- Péssimo desempenho na criptografia.
Rprime	- Utiliza o mesmo algoritmo de decriptografia que o Mprime RSA. - Obteve o melhor desempenho na decriptografia (ganho de 7,83 sobre o . RSA QC para $n = 2048$). - Desempenho aumenta consideravelmente com módulos maiores. - Se enquadra na especificação PKCS#1.	- Péssimo desempenho na criptografia. - Não é recomendado seu uso para módulos menores ou iguais a 768 bits.

Tabela 7.5: Resumo comparativo das variações estudadas.

APÊNDICE

A

Dados Experimentais

A.1 Algoritmos utilizados na aplicação do TCR

Como descrito na seção 2.5, podemos aplicar o Teorema Chinês do Resto usando dois algoritmos. O primeiro é o algoritmo clássico enunciado em [24, 30], o segundo criado por Garner [31] é teoricamente mais rápido por utilizar reduções sobre módulos menores.

Tivemos a oportunidade de medir o tempo gasto em microsegundos pela decriptografia de cada uma das variações analisadas utilizando tanto o algoritmo clássico, quanto o algoritmo de Garner. Os resultados obtidos estão dispostos nas tabelas A.1 e A.2.

n	RSA	Batch 4	RSA QC	Mprime	Mpower	Rebalanced	Rprime
768 bits	7925	4017	2534	1359	1170	1165	925
1024 bits	17622	7676	5442	2932	2191	1901	1514
2048 bits	124155	42280	36307	18673	13196	6292	4915

Tabela A.1: Tempo em microsegundos das decriptografias usando o algoritmo clássico do TCR.

n	RSA	Batch 4	RSA QC	Mprime	Mpower	Rebalanced	Rprime
768 bits	7903	3943	2438	1244	977	967	811
1024 bits	17619	7598	5294	2798	2083	1748	1363
2048 bits	124160	41750	35745	18113	12773	5977	4564

Tabela A.2: Tempo em microsegundos das decriptografias usando o algoritmo de Garner.

Condizendo com as observações vistas na seção 2.5, para todas as variações, ao utilizar o algoritmo de Garner obtivemos um ganho em relação ao tempo de execução, se comparado ao tempo utilizado pelo algoritmo clássico. Note que, por não fazer uso do teorema chinês do resto, o RSA tradicional não se beneficiou com a alteração de algoritmos. Para melhor visualizar estes resultados, calculamos a relação entre o tempo de execução medido utilizando o algoritmo clássico sobre o tempo de execução medido usando o algoritmo de Garner (veja tabela A.3).

n	Batch 4	RSA QC	Mprime	Mpower	Rebalanced	Rprime
768 bits	1,0187	1,0393	1.0924	1.1975	1.2047	1.1405
1024 bits	1,0102	1,0279	1.0478	1.0518	1.0875	1.1107
2048 bits	1,0126	1,0157	1.0309	1.0331	1.0527	1.0769

Tabela A.3: Relação de desempenho do algoritmo clássico sobre o algoritmo de Garner.

Percebemos, pela tabela acima, que os maiores beneficiados pela utilização do algoritmo de Garner foram o RSA Rebalanceado e o RSA com Múltiplas Potências com módulos de 768 bits. Estes obtiveram um ganho de cerca de 20% se comparado com a utilização do algoritmo clássico. As demais variações obtiveram ganhos que variaram de 1 até 14%. Apesar de não parecerem muito significativos, lembramos que quando trabalhamos com um número excessivo de mensagens esses valores podem se tornar essenciais. Além disso, não podemos esquecer que, se quisermos estar de acordo com a especificação PKCS #1, devemos utilizar o algoritmo de Garner para a aplicação do Teorema Chinês do Resto (veja seção 5.2).

A.2 Dados de entrada e saída

Nesta seção apresentamos um exemplo das saídas geradas pela implementação de uma das variações estudadas. Optamos por mostrar os arquivos de saída gerados pelos algoritmos de geração de chaves e decriptografia do RSA Rebalanceado com Múltiplos Primos (com $k = 3$),

já que as chaves geradas por este novo esquema mostram características comuns tanto do RSA Rebalanceado quanto do RSA com Múltiplos Primos. A geração das mensagens e execução dos dados seguiram o processo descrito na seção 5.1.

Chave Pública (Exemplo)

Abaixo segue um exemplo de uma chave pública gerada pelo algoritmo de geração de chaves do RSA Rebalanceado com Múltiplos Primos. No caso utilizamos $n = 1024$ bits.

$$N = 10918999605f9bfdf387d02d34c072a76abfbec676131838693704f58eedd27aabc3438 \\ 203eeea8784c14fb66a5f161746c738c714fd691edcf8d9698da0619e1c056b963baad28 \\ d8ebccc1d557d8058915d8e26f61848468acf84d58bc192a176351211d155367a7ce3f30 \\ 10e2728fa2c9c87ea2b48d300173877f12b199ae2b1$$

$$e = 64b9eb88b21f64667168296a93d224bb4bd702fe1c46ecca084d1cfd7b1b15619601ff7c \\ e055ada7255009d2effa102978f13c61f5defb09bf4cf8b47b4ac6e542fd2e781559d9 \\ 26c97ae767894a9c82c78064166d017d4d01cc926e0272f7cd97de4c0a3d54b399ac7412 \\ c14ad09780bafbdfcc40d6c1b38856dd44300fb3ad1$$

Note que, como descrito nas seções 4.3 e 4.4, o expoente público e gerado é um número realmente grande (em torno de N), diferentemente do expoente público padrão (65537) que possui apenas 17 bits.

Chave Particular (Exemplo)

O exemplo que segue descreve a chave particular correspondente à chave pública vista acima. Lembramos que utilizamos o parâmetro s igual a 160 e por isso d_p, d_q e d_r possuem 160 bits de tamanho. Já os primos p, q e r, seguindo a explicação da seção 4.4, possuem aproximadamente n/k bits de tamanho (lembrando que no caso $n = 1024$ e $k = 3$).

$$p = 4150ecab66948736a8e74390d1aba00a177c8e27fb35f4ed067cc46e1d4b10dbe72098 \\ c1da812384cd1809$$

$$q = 9f57f52a450b4d9ea696bb061f16d64a698554ea9b171e3f171e4b38a3c3d6196f0232\\ 8602aa816c45aa23$$

$$d_p = 8ec2add777c1b23ad146416ca11bad79f5be7d41$$

$$d_q = 5b4239d4f50e339ea191acce782c943bfb010bff$$

$$qInv = 31b2d02aafc9ccb570fc948ec6e835a8b78a6d1f767f0a139c44cb4f67d3c18f8d3\\ 17c7c8b757fb63b9646$$

$$r = 68546dbcb82b6b4717cfb6a4cf5282fed6f1beadaa53847b075352797d0c7e2520909a\\ ec291a66a735bb03$$

$$d_r = 500e575b2e781a6dad96438fd05ef4759ca83075$$

$$rInv = dabbbfc0360c66d400440cdd1419de6a935e2f6e38cfdb697fc18f47ab6574624fb\\ 66e62cb455a2b1ecec$$

Mensagens, Criptografia e Decriptografia (Exemplo)

A título de curiosidade, incluímos abaixo um exemplo de uma mensagem M antes de ser codificada e após termos aplicado os algoritmos de criptografia e decriptografia do RSA Rebalanceado com Múltiplos Primos. A fim de melhor indentificar este processo, denominamos como C a criptografia da mensagem M e como M' a decriptografia do texto C.

$$M = 801179a6a08d979ba8f73fa6d16411a1015fc56533af85f2d117e1ae1d752e2eeed4cf\\ 7b6c6a2463a9ca70c0b8ad021862774c01fd3ed30dbcfdef844c2733e1f6ae4d60d2b099\\ c20c9a8aa9ac931c13df1e011df11ecaeefe4f2fcb25dd1d42a61a6da6b43fa34eba4ceb\\ 4d58ff1c7702a681ee5470d3466cde6c$$

$$C = 2ab49d223c1419520b62391a84ddc9c4f71b14c74e25196fe2ac9bd357af5cc3559f4c\\ dad3d4fd0cf0921624ee47f38e4157a9273ea151a2726961750798bfc8770a8907efc\\ f771af00a7319c8fcb6c1d9837e57b3289e63a7135c20bb09fe8c6c0437c2c70b8959c8\\ bf2b616326ae1894ed751c0c5768200ba5e9b22b9a42404$$

$$M' = 801179a6a08d979ba8f73fa6d16411a1015fc56533af85f2d117e1ae1d752e2eeed4cf$$
$$7b6c6a2463a9ca70c0b8ad021862774c01fd3ed30dbcfdef844c2733e1f6ae4d60d2b099$$
$$c20c9a8aa9ac931c13df1e011df11ecaeefe4f2fcb25dd1d42a61a6da6b43fa34eba4ceb$$
$$4d58ff1c7702a681ee5470d3466cde6c$$

Como era de se esperar, $M = M'$. Em nossos testes utilizamos as funções *cmp* e *diff* do Unix para verificar que os algoritmos funcionavam corretamente.

Tempo em microsegundos (Exemplo)

Para finalizar, apresentamos aqui um exemplo de parte da saída gerada pelo RSA Rebalanceado com Múltiplos Primos, referente ao tempo gasto pela decriptografia das primeiras 100 mensagens codificadas, utilizando duas diferentes chaves. Ao final da execução de cada chave, temos a média do tempo gasto pelas decriptografias precedentes.

Número da da chave	n	Número da Mensagem	Tempo em Microsegundos	Número da da chave	n	Número da Mensagem	Tempo em Microsegundos
1	1024	1	1450	2	1024	1	1330
1	1024	2	1349	2	1024	2	1330
1	1024	3	1362	2	1024	3	1330
1	1024	4	1348	2	1024	4	1330
1	1024	5	1346	2	1024	5	1342
1	1024	6	1346	2	1024	6	1329
1	1024	7	1347	2	1024	7	1330
1	1024	8	1346	2	1024	8	1330
1	1024	9	1347	2	1024	9	1330
1	1024	10	1403	2	1024	10	1347
1	1024	11	1347	2	1024	11	1341
1	1024	12	1368	2	1024	12	1330
1	1024	13	1348	2	1024	13	1330
1	1024	14	1349	2	1024	14	1330
1	1024	15	1349	2	1024	15	1330
1	1024	16	1364	2	1024	16	1330
1	1024	17	2134	2	1024	17	1330
1	1024	18	1348	2	1024	18	1342
1	1024	19	1346	2	1024	19	1330
1	1024	20	1346	2	1024	20	1330
1	1024	21	1350	2	1024	21	1329
1	1024	22	1369	2	1024	22	1329
1	1024	23	1349	2	1024	23	1330
1	1024	24	1349	2	1024	24	1330
1	1024	25	1349	2	1024	25	1330

Número da da chave	**n**	**Número da Mensagem**	**Tempo em Microsegundos**	**Número da da chave**	**n**	**Número da Mensagem**	**Tempo em Microsegundos**
1	1024	26	1349	2	1024	26	1345
1	1024	27	1349	2	1024	27	1329
1	1024	28	1370	2	1024	28	1330
1	1024	29	1574	2	1024	29	1330
1	1024	30	1349	2	1024	30	1329
1	1024	31	1348	2	1024	31	1328
1	1024	32	1348	2	1024	32	1328
1	1024	33	1348	2	1024	33	1330
1	1024	34	1350	2	1024	34	1330
1	1024	35	1359	2	1024	35	1331
1	1024	36	1348	2	1024	36	1535
1	1024	37	1348	2	1024	37	1331
1	1024	38	1349	2	1024	38	1330
1	1024	39	1349	2	1024	39	1330
1	1024	40	1349	2	1024	40	1330
1	1024	41	1349	2	1024	41	1330
1	1024	42	1424	2	1024	42	1331
1	1024	43	1350	2	1024	43	1391
1	1024	44	1368	2	1024	44	1330
1	1024	45	1348	2	1024	45	1331
1	1024	46	1348	2	1024	46	1329
1	1024	47	1349	2	1024	47	1330
1	1024	48	1374	2	1024	48	1330
1	1024	49	1347	2	1024	49	2042
1	1024	50	1349	2	1024	50	1333
Média do tempo em microsegundos = 1376				Média do tempo em microsegundos = 1363			

A listagem da implementação do RSA com Múltiplos Primos e das demais variações enunciadas no quarto capítulo são apresentadas a seguir.

APÊNDICE

B

Listagem dos programas

Para facilitar a utilização para outros usuários, implementamos cinco bibliotecas contendo as principais funções utilizadas. As três primeiras podem ser divididas da seguinte maneira:

1. **rsa_package** - Implementa os algoritmos de criptografia e decriptografia estudados no quarto capítulo, com exceção dos descritos pelo RSA em Lote.

2. **print_package** - Implementa funções para saidas dos dados e gerenciamento de arquivos.

3. **random_package** - Implementa funções que possibilitam a geração de números aleatórios (Para geração de chaves).

A quarta biblioteca utilizada implementa a função **diftime**, responsável por calcular a diferença de tempo em microssegundos entre os momentos da passagem por dois pontos de um programa. Logo, esta biblioteca foi utilizada por todas as variações. As implementações das funções de criptografia e decriptografia do RSA em Lote foram agrupadas na biblioteca denominada como **batch_package**.

A seguir, apresentamos a listagem das bibliotecas mencionadas acima, bem como dos programas de geração de chaves utilizados durante o trabalho. Apresentamos também um exemplo de uso das funções implementadas nos três primeiros pacotes enunciados.

rsapackage.h

```
#include <gmp.h>        //Para as funções gmp
#include <stdio.h>      //Para a função printf
#include <stdlib.h>     //Para a função exit

//------------------------------------------------------------------------
// Definições do RSA com Múltiplos Primos

#define KPRIME 3   //Define o número de primos que será utilizado no programa

//------------------------------------------------------------------------
// Definições do RSA com Múltiplas Potências

#define KPOWER 2  //Define o tamanho da potência do modulo p^kq

struct takagi {  //Estrutura utiliza pelo algoritmo de Takagi
  mpz_t A;
  mpz_t B;
  mpz_t E;
  mpz_t F;
  mpz_t K;
};

typedef struct takagi alg;  //Define a estrutura de Takagi como alg

//------------------------------------------------------------------------
//Prototipo das funções implementadas em rsa_package.c

void rsacrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void rsadecrypt(mpz_t C, mpz_t d, mpz_t N, mpz_t M);
void rsaqccrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void rsaqcdecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t C, mpz_t M);
void mprimecrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void mprimedecrypt(mpz_t p[], mpz_t d[], mpz_t t[], mpz_t C, mpz_t M);
void mpowercrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void mpowerdecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t e, mpz_t C, mpz_t M);
void rebalancedcrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void rebalanceddecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t C, mpz_t M);
void rprimecrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void rprimedecrypt(mpz_t p[], mpz_t d[], mpz_t t[], mpz_t C, mpz_t M);
//------------------------------------------------------------------------
```

rsapackage.c

```
#include "rsa_package.h"

//------------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro Mˆe mod N
void  rsacrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
 mpz_powm(C, M, e, N);
}

//------------------------------------------------------------------------------
//Recebe inteiros gmp C, d e N e guarda em M o inteiro Cˆd mod N
void rsadecrypt(mpz_t C, mpz_t d, mpz_t N, mpz_t M){
  mpz_powm(M, C, d, N);
}

//------------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro Mˆe mod N
void  rsaqccrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
 mpz_powm(C, M, e, N);
}

//------------------------------------------------------------------------------
//Armazena em M  através do algoritmo de Garner
//o valor da combinação de Cˆdp mod p e Cˆdq mod q  mdc(p, q) = 1
void rsaqcdecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t C, mpz_t M){
  mpz_t M1, M2, h;

   //Inicializa as variaveis gmp utilizadas
   mpz_init(M1);
   mpz_init(M2);
   mpz_init(h);
   //Calcula Cˆdp mod p e Cˆdq mod q
   mpz_powm(M1, C, dp, p);
   mpz_powm(M2, C, dq, q);
   //Armazena em M através do algoritmo de Garner a combinação de M1 e M2
   mpz_sub(h, M1, M2);
   mpz_mul(h, h, qInv);
   mpz_mod(h, h, p);
   mpz_mul(M1, q, h);
   mpz_add(M, M2, M1);
   //Desaloca memoria das variaveis utilizadas
   mpz_clear(M1);
   mpz_clear(M2);
   mpz_clear(h);
}
```

```
//------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro M^e mod N
void mprimecrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
  mpz_powm(C, M, e, N);
}

//------------------------------------------------------------------------
// Recebe um vetor p com KPRIME primos, um vetor d com KPRIME expoentes
// e um vetor t com KPRIME-1 coeficientes do TCR
// Armazena em M a combinacao dos C^{d[i]} mod p[i] para i = 1,2 ... KPRIME
void mprimedecrypt(mpz_t p[], mpz_t d[], mpz_t t[], mpz_t C, mpz_t M){
  mpz_t h, P, m[KPRIME];
  int i;

   //inicializa variaveis utilizadas
   mpz_init(h);
   mpz_init(P);

   //Calcula os Mis intermediarios atraves de C
   for (i = 0; i < KPRIME; i++){
     mpz_init(m[i]);
     mpz_powm(m[i], C, d[i], p[i]);
   }

   //Recupera M utilizando o TCR pelo algoritmo de Garner
   //segundo o padrao PKCS#1 v2.1
   mpz_sub(h, m[0], m[1]);
   mpz_mul(h, h, t[1]);
   mpz_mod(h, h, p[0]);
   mpz_mul(h, h, p[1]);
   mpz_add(M, m[1], h);
   mpz_set(P, p[0]);
   for (i = 2; i < KPRIME; i++){
     mpz_mul(P, P, p[i-1]);
     mpz_sub(h, m[i], M);
     mpz_mul(h, h, t[i]);
     mpz_mod(h, h, p[i]);
     mpz_mul(h, h, P);
     mpz_add(M, M, h);
   }
   //Desaloca memoria das variaveis utilizadas
   mpz_clear(h);
   mpz_clear(P);
   for (i = 0; i < KPRIME; i++)
     mpz_clear(m[i]);
}
```

```
//-------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro M^e mod N
void mpowercrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
  mpz_powm(C, M, e, N); }

//-------------------------------------------------------------------------
// Recebe os primos p e q; os expoentes dp e dq ;o coeficiente do TCR qInv
// o expoente publico e e o inteiro gmp criptografado C
// Guarda em M através do algoritmo de Takagi o resultado
// da decriptografia de C modulo p^{KPOWER}q
void mpowerdecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t e, mpz_t C, mpz_t M){

  mpz_t  Mp, Mq, pi, pii, pk;
  alg a[KPOWER];   //Cria a estrutura utilizada pelo algoritmo
  int i;

    //Inicializa variaveis utilizadas
    mpz_init(Mp);
    mpz_init(Mq);
    mpz_init(pi);
    mpz_init(pii);
    mpz_init(pk);

    for (i = 0; i < KPOWER; i++){
      mpz_init(a[i].A);
      mpz_init(a[i].B);
      mpz_init(a[i].E);
      mpz_init(a[i].F);
      mpz_init(a[i].K);
    }

    //Recupera M de C - Algoritmo proposto por Takagi
    //Inicialmente calculamos a[0].K = C^{dp} mod p e Mq = C^{dq} mod q
    mpz_powm(a[0].K, C, dp, p);
    mpz_powm(Mq, C, dq, q);
    mpz_set(a[0].A, a[0].K);
    //A seguir expandimos a[0].K para Mp onde (Mp)^e = C mod p^k
    for (i = 1; i < KPOWER; i++){
      mpz_pow_ui(pii, p, i+1);
      mpz_powm(a[i].F, a[i-1].A, e, pii);
      mpz_sub(a[i].E , C, a[i].F);
      mpz_mod(a[i].E, a[i].E, pii);
      mpz_pow_ui(pi, p, i);
      mpz_fdiv_q(a[i].B, a[i].E, pi);
      mpz_mul(a[i].K, e, a[i].F);
      if((mpz_invert(a[i].K, a[i].K, p)) == 0){
        printf("Inversa nao existe - 1!\n\n");
        exit(1);
      }
```

```
      mpz_mul(a[i].K, a[i].K, a[i-1].A);
      mpz_mul(a[i].K, a[i].K, a[i].B);
      mpz_mul(a[i].A, a[i].K, pi);
      mpz_mul(a[i].A, a[i].K, p);
      mpz_add(a[i].A, a[i-1].A, a[i].A);
    }

    mpz_set(Mp, a[KPOWER-1].A);
    //Calcula o TCR usando o algoritmo de Garner
    mpz_sub(pi, Mp, Mq);
    mpz_mul(pi, pi, qInv);
    mpz_pow_ui(pk, p, KPOWER);
    mpz_mod(pi, pi, pk);
    mpz_mul(Mp, q, pi);
    mpz_add(M, Mq, Mp);

    //Desaloca memoria utilizada pelas variaveis gmp
    for (i = 0; i < KPOWER; i++){
      mpz_clear(a[i].A);
      mpz_clear(a[i].B);
      mpz_clear(a[i].E);
      mpz_clear(a[i].F);
      mpz_clear(a[i].K);
    }

    mpz_clear(pi);
    mpz_clear(pii);
    mpz_clear(pk);
    mpz_clear(Mp);
    mpz_clear(Mq);
}

//------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro M^e mod N
void rebalancedcrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
  mpz_powm(C, M, e, N);
}

//------------------------------------------------------------------------
//Utiliza a funcao do rsaqc para decriptografar C
void rebalanceddecrypt(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t C, mpz_t M){
  rsaqcdecrypt(p, q, dp, dq, qInv, C, M);
}

//------------------------------------------------------------------------
//Recebe inteiros gmp M, e e N e guarda em C o inteiro M^e mod N
void rprimecrypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
  mpz_powm(C, M, e, N);
}
```

```
//---------------------------------------------------------------------------
//Utiliza a funcao do rsa com multiplos primos para decriptografar C
void rprimedecrypt(mpz_t p[], mpz_t d[], mpz_t t[], mpz_t C, mpz_t M){
  mprimedecrypt(p, d, t, C, M);
}
```

****

print_package.h

```
#define MSGFILE "../Mensagens/m2048/msgrsa"  //prefixo do arquivo de mensagem
#define PUBFILE "./c2048/pubkey"             //prefixo do arq. da chave publica
#define SECFILE "./c2048/seckey"          //prefixo do arq. da chave particular
#define CRIFILE "./Cript2048/rsacri"      //prefixo do arq. criptografado
#define DECFILE "./Decri2048/rsadec"      //prefixo do arq. decriptografado
#define RESULTC "./SaidaC/saida2048.rsa"  //arquivo de saida das criptografias
#define RESULTD "./SaidaD/saida2048.rsa"  //arquivo de saida das decriptografias
#define SIZEKEY 2048     //Tamanho do modulo RSA
#define NUMMSGS 50       //Numero de mensagens utilizadas
#define NUMKEYS 20       //Numero de chaves utilizadas

//Bibliotecas utilizadas em print_package.c
#include <stdlib.h>  //Para exit
#include <stdio.h>   //Para fopen
#include <string.h>  //Para strcat

//---------------------------------------------------------------------------
//Prototipos das funcoes implementadas em print_package.c

void open_file(FILE **, char *, char);
void print_heading(FILE **);
void print_time(FILE **, long spend_time, int i);
void print_average(FILE **, long total_time, int i);
void file_manager(char, char , int, FILE **, FILE **, FILE **, FILE **);
void print_result(int, long *total_time, long spend_time, FILE **);
```

****

print_package.c

```
#include "print_package.h"

//---------------------------------------------------------------------------
//Recebe um arquivo de nome file apontado por fp e o abre
//para leitura ou escrita de acordo com a variavel op
void open_file(FILE ** fp, char * file, char op){
```

```
  if (op == 'w') {
    if ((*fp = fopen(file, "w")) == NULL) {
      printf("Nao abriu %s", file);
      exit(-1);
    }
  }

  else if (op == 'r') {
    if ((*fp = fopen(file, "r")) == NULL) {
      printf("Nao encontrou %s", file);
      exit(-1);
    }
  } else {
      printf("Terceiro parametro de open_file invalido!\n");
      exit(-1);
  }
}

//---------------------------------------------------------------------------
//Imprime no arquivo apontado por fp o cabecalho do arquivo de saida
//referente ao tempo utilizado pela criptografia ou decriptografia
void print_heading(FILE **fp){

  fprintf(*fp,
    "-----------------------------------------------------------------\n");
  fprintf(*fp,
    "  Número   |  Tamanho da chave  |  Número da  |   Tempo em     |\n");
  fprintf(*fp,
    " da chave  |      em bits       |  mensagem   | microsegundos  |\n");
  fprintf(*fp,
    "-----------------------------------------------------------------\n");
}

//---------------------------------------------------------------------------
//Imprime no arquivo apontado por fp o tempo gasto pela criptografia
// ou decriptografia
void print_time(FILE **fp, long spend_time, int i){
 fprintf(*fp, "    %2.d     |      %6.d        |     %2.d      |%12.ld    |\n",
 i/NUMMSGS + 1, SIZEKEY, i%NUMMSGS + 1,  spend_time);
}

//---------------------------------------------------------------------------
//Imprime em fp a media dos tempos necessarias para as i criptografias ou
//decriptografias
void print_average(FILE **fp, long total_time, int i){
```

```
  fprintf(*fp,
  "\nMédia do tempo em microsegundos = %ld\n\n", (total_time)/(i+1));
  fprintf(*fp,
  "----------------------------------------------------------------\n");
}
//-------------------------------------------------------------------------
//Funcao que gerencia a abertura e fechamento dos arquivos apontados por
//fp1, fp2, fp3, fp4. A variavel c indica se queremos abrir ('o') ou fechar ('c') os arquivos
//e a variavel status indica se o processo é de criptografia ('c') ou decriptografia ('d')
void file_manager(char c, char status, int i, FILE **fp1, FILE **fp2, FILE **fp3, FILE **fp4){

  char dir1[30], dir2[30];

  switch (c) {

    case 'o' :  //Abre os arquivos que serao utilizados

                if (i == 0){
                  if (status == 'c'){
                    open_file(&(*fp4), RESULTC, 'w');
                  }
                  else if (status == 'd'){
                         open_file(&(*fp4), RESULTD, 'w');
                  }
                  else {
                    printf(
                  "Segundo parametro incorreto no gerenciador de arquivos \n");
                    exit(1);
                  }
                  print_heading(&(*fp4));
                }

                //Coloca em dir1 o nome do arquivo a ser aberto
                dir1[0] = '\0';
                dir2[0] = '\0';
                if (status == 'c'){
                  strcat(dir1, MSGFILE);
                  sprintf(dir2, "%d", i%NUMMSGS+1);
                }
                else {
                  strcat(dir1, CRIFILE);
                  sprintf(dir2, "%d", i+1);
                }

                strcat(dir2, ".rsa");
                strcat(dir1, dir2);
                open_file(&(*fp1), dir1, 'r');
```

```
                //Coloca em dir1 o nome do arquivo a ser aberto
                dir1[0] = '\0';
                dir2[0] = '\0';
                if (status == 'c') strcat(dir1, PUBFILE);
                else strcat(dir1, SECFILE);
                sprintf(dir2, "%d", i/NUMMSGS + 1);
                strcat(dir2, ".rsa");
                strcat(dir1,dir2);
                open_file(&(*fp2), dir1, 'r');

                //Coloca em dir1 o nome do arquivo a ser aberto
                dir1[0] = '\0';
                dir2[0] = '\0';
                if (status == 'c') strcat(dir1, CRIFILE);
                else strcat(dir1, DECFILE);
                sprintf(dir2, "%d", i+1);
                strcat(dir2, ".rsa");
                strcat(dir1, dir2);
                open_file(&(*fp3), dir1, 'w');
                break;

  case 'c':     //Fecha os arquivos
                fclose(*fp1);
                fclose(*fp2);
                fclose(*fp3);
                if ((i+1) == NUMMSGS*NUMKEYS) fclose(*fp4);
                break;

  }
}

//---------------------------------------------------------------------------
//Imprime em fp4 o tempo gasto pela criptografia ou decriptografia e
//eventualmente imprime a media das i criptografias ou decriptografias
void print_result(int i, long *total_time, long spend_time, FILE **fp4){

   *total_time += spend_time;

   print_time(&(*fp4), spend_time, i);

   //Se acabou a execucao de uma chave entao imprime a media
   if ((((i+1)%NUMMSGS == 0) && (i != 0)) || ((i+1) == NUMMSGS*NUMKEYS))
     print_average(&(*fp4), *total_time, i);

}
//---------------------------------------------------------------------------
```

random_package.h

```
//Bibliotecas utilizadas por random_package.c
#include <sys/time.h>      //Para a função setitimer
#include <signal.h>        //Para a função signal
#include <stdio.h>         //Para função printf
#include <gmp.h>           //Para as variáveis gmp de random_package.c

volatile unsigned int i, counter, value;     //Usadas por get_random e handler

//-------------------------------------------------------------------------
//Prototipo das funções implementadas em random_package.c
unsigned int get_random(void);
void handler(void);
void create_random(mpz_t, int, int);

*************************************************************************
```

random_package.c

```
#include "random_package.h"

//---------------------------------------------------------------------
//Gera um inteiro aleatório entre 0-255 - Função criada por Aggelos Keromitis <kermit@gr.forthnet>
unsigned int get_random(void){

  struct itimerval x, y;

  i = 0;
  counter = 0;
  value = 0;

  x.it_interval.tv_sec = 0;
  x.it_interval.tv_usec = 1;
  x.it_value.tv_sec = 0;
  x.it_value.tv_usec = 1;

  //Põe o timer (de tempo real) com valor de x e obtém o antigo valor através de y
  //Quando o timer expira, é enviado o sinal SIGALRM.
  if (setitimer(ITIMER_REAL, &x, &y) == -1){
    printf("Erro na função get_random!\n");
    return 0;
  }

  //Executa handler caso receba o sinal SIGALRM
  signal(SIGALRM, handler);
   while (counter < 8)
    i++;
```

```
  //Ignora o sinal recebido (SIGALRM)
  signal(SIGALRM, SIG_IGN);


  //Põe o timer (de tempo real) com valor de y; quando o timer expira é enviado o sinal SIGALRM
  if (setitimer(ITIMER_REAL, &y, (struct itimerval *) NULL) == -1)
    printf("Erro na funcao get_random!\n");

  return value;
}

//---------------------------------------------------------------------
//Manipulador do  SIGALRM - função criada por Aggelos Keromitis <kermit@gr.forthnet>
void handler(void){
  value = (value << 1) | (i & 0x1);  //value = 2*value+(i & 1)
  counter++;
  i = 0;
  //Executa handler caso receba o sinal SIGALRM
  signal(SIGALRM, handler);
}

//---------------------------------------------------------------------
//Retorna em x um inteiro gmp de 4i/c bits
void create_random(mpz_t x, int i, int c){

  while ((i -= c) > 0){
    mpz_mul_ui(x, x, 16);
    mpz_add_ui(x, x, get_random());
  }
}
```

****

diftime.h

```
#include <sys/time.h>    //Para struct timeval


long diftime(struct timeval, struct timeval);   //Protótipo da função implementada em diftime.c
```

****

diftime.c

```
#include "diftime.h"
```

```
//---------------------------------------------------------------------------
//Recebe quatro inteiros longos contidos nas estruturas tf e t0 e retorna a diferença entre eles (tf - t0)
long diftime(struct timeval tf, struct timeval t0){

  int sec, usec;

   sec  = tf.tv_sec - t0.tv_sec;   //Calcula a diferença em segundos
   usec = tf.tv_usec - t0.tv_usec; //Calcula a diferença em microsegundos
   if (usec < 0){          //Ajusta o tempo quando usec é negativo
     usec += 1000000L;
     sec -= 1;
   }

   return (sec * (1000000) + usec);
}

*****************************************************************************
```

batch_package.h

```
#include <gmp.h>        // Para as funcoes gmp
#include <stdlib.h>     // Para exit

#define b 4             //Numero de mensagens em lote
#define SIZE_BATCH 2*b  //Numero de nos na arvore

struct node {  //Estrutura que forma um no da arvore de batch
  mpz_t E;
  mpz_t Cp;
  mpz_t Cq;
  mpz_t Mp;
  mpz_t Mq;
  mpz_t Xp;
  mpz_t Xq;
};

typedef struct node batch_tree;  //Arvore de batch

//---------------------------------------------------------------------
//Prototipos das funcoes implementadas em batch_package.c

void batch_crypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C);
void Product(int i, mpz_t p, mpz_t q, mpz_t N, batch_tree a[]);
void Exponentiation(mpz_t p, mpz_t q, mpz_t dn, batch_tree a[]);
void Break_Product(int i, mpz_t p, mpz_t q, mpz_t N, batch_tree a[]);
void batch_decrypt(mpz_t p, mpz_t q, mpz_t dn, mpz_t N, batch_tree a[]);
void init_batch_tree(batch_tree a[], int aux[]);
void clear_batch_tree(batch_tree a[]);
```

****

batch_package.c

```
#include "batch_package.h"

//-----------------------------------------------------------------------
//Recebe inteiros M, e e N e guarda em C o inteiro M^e mod N
void batch_crypt(mpz_t M, mpz_t e, mpz_t N, mpz_t C){
  mpz_powm(C, M, e, N);
}

//-----------------------------------------------------------------------
//Calcula recursivamente na raiz da arvore de batch (batch_tree)
//o produtorio dos Cpi^{E/ei} e o produtorio dos Cqi^{E/ei}, onde E = produtorio// dos eis (para i= 1, 2...b)
void Product(int i, mpz_t p, mpz_t q, mpz_t N, batch_tree a[]){

  mpz_t aux1, aux2;

  if (i <= (SIZE_BATCH-1)/2){
    //Chama recursivamente Product até encontrar as folhas da árvore de batch
    Product(2*i, p, q, N, a);
    Product(2*i+1, p, q, N, a);
    //Calcula o produtorio dos eis na raiz da arvore
    mpz_mul(a[i].E, a[2*i].E, a[2*i+1].E);
    //Calcula Cp^{E/ei} (para i= 1, 2...b)
    mpz_init(aux1);
    mpz_init(aux2);
    mpz_powm(aux1, a[2*i].Cp, a[2*i+1].E, p);
    mpz_powm(aux2, a[2*i+1].Cp, a[2*i].E, p);
    mpz_mul(a[i].Cp, aux1, aux2);
    mpz_mod(a[i].Cp, a[i].Cp, p);
    //Calcula Cq^{E/ei} (para i= 1, 2...b)
    mpz_powm(aux1, a[2*i].Cq, a[2*i+1].E, q);
    mpz_powm(aux2, a[2*i+1].Cq, a[2*i].E, q);
    mpz_mul(a[i].Cq, aux1, aux2);
    mpz_mod(a[i].Cq, a[i].Cq, q);
    mpz_clear(aux1);
    mpz_clear(aux2);
  }
}

//-----------------------------------------------------------------
//Extrai a e-esima raiz (dada por dn) de Cp e Cq modulo p-1 e q-1
//Armazena os resultados em Mp e Mq, ou seja, armazena o produto
//dos textos decriptografados modulo p-1 e modulo q-1 em Mp e Mq
void Exponentiation(mpz_t p, mpz_t q, mpz_t dn, batch_tree a[]){

  mpz_t dp, dq;
```

```
  mpz_init(dp);
  mpz_init(dq);
  mpz_sub_ui(dp, p, 1);
  mpz_sub_ui(dq, q, 1);
  mpz_mod(dp, dn, dp);
  mpz_mod(dq, dn, dq);
  //Extrai a E-esima raiz (Produto dos d's) de C modulo p
  mpz_powm(a[1].Mp, a[1].Cp, dp, p);
  //Extrai a E-esima raiz (Produto dos d's) de C modulo q
  mpz_powm(a[1].Mq, a[1].Cq, dq, q);
  mpz_clear(dp);
  mpz_clear(dq);
}

//-------------------------------------------------------------
//Quebra o produto dos textos decriptografados encontrados na raiz
//em subprodutos, até obter os textos decriptografados modulo p-1
//e q-1 nas folhas
void Break_Product(int i, mpz_t p, mpz_t q, mpz_t N, batch_tree a[]){

  mpz_t aux1, aux2;

  if (i <= (SIZE_BATCH-1)/2){
    mpz_init(aux1);
    mpz_init(aux2);
    //Calculo de Xp pelo TCR
    if ((mpz_invert(aux1, a[2*i].E, a[2*i+1].E)) == 0){
      printf("inversa nao existe 1");
      exit(-1);
    }
    mpz_mul(a[i].Xp, a[2*i].E, aux1);
    mpz_mod(a[i].Xp, a[i].Xp, a[i].E);

    //Calculo de Xq pelo TCR
    mpz_set(a[i].Xq, a[i].Xp);

    //Calculo de XL modulo p
    if ((mpz_invert(aux1, a[2*i].E, p)) == 0){
      printf("inversa nao existe 2");
      exit(-1);
    }
    mpz_mul(a[2*i].Xp, a[i].Xp, aux1);
    mpz_mod(a[2*i].Xp, a[2*i].Xp, p);

    //Calculo de XL modulo q
    if ((mpz_invert(aux1, a[2*i].E, q)) == 0){
      printf("inversa nao existe 2");
      exit(-1);
    }
    mpz_mul(a[2*i].Xq, a[i].Xq, aux1);
    mpz_mod(a[2*i].Xq, a[2*i].Xq, q);
```

```
//Calculo de XR modulo p
if ((mpz_invert(aux2, a[2*i+1].E, p)) ==0){
  printf("inversa nao existe 3");
  exit(-1);
}
mpz_sub_ui(aux1, a[i].Xp, 1);
mpz_mul(a[2*i+1].Xp, aux1, aux2);
mpz_mod(a[2*i+1].Xp, a[2*i+1].Xp, p);

//Calculo de XR modulo q
if ((mpz_invert(aux2, a[2*i+1].E, q)) ==0){
  printf("inversa nao existe 3");
  exit(-1);
}
mpz_sub_ui(aux1, a[i].Xq, 1);
mpz_mul(a[2*i+1].Xq, aux1, aux2);
mpz_mod(a[2*i+1].Xq, a[2*i+1].Xq, q);

//Calculo de MR modulo p
mpz_powm(aux1, a[2*i].Cp, a[2*i].Xp, p);
mpz_powm(aux2, a[2*i+1].Cp, a[2*i+1].Xp, p);
mpz_mul(aux1, aux1, aux2);
mpz_mod(aux1, aux1, p);
if ((mpz_invert(aux1, aux1, p)) == 0){
  printf("inversa nao existe 4");
  exit(-1);
}
mpz_powm(a[2*i+1].Mp, a[i].Mp, a[i].Xp, p);
mpz_mul(a[2*i+1].Mp, a[2*i+1].Mp, aux1);
mpz_mod(a[2*i+1].Mp, a[2*i+1].Mp, p);

//Calculo de MR modulo q
mpz_powm(aux1, a[2*i].Cq, a[2*i].Xq, q);
mpz_powm(aux2, a[2*i+1].Cq, a[2*i+1].Xq, q);
mpz_mul(aux1, aux1, aux2);
mpz_mod(aux1, aux1, q);
if ((mpz_invert(aux1, aux1, q)) == 0){
  printf("inversa nao existe 4");
  exit(-1);
}
mpz_powm(a[2*i+1].Mq, a[i].Mq, a[i].Xq, q);
mpz_mul(a[2*i+1].Mq, a[2*i+1].Mq, aux1);
mpz_mod(a[2*i+1].Mq, a[2*i+1].Mq, q);

//Calculo de ML modulo p
if ((mpz_invert(aux2, a[2*i+1].Mp, p))==0) {
  printf("inversa nao existe 5");
  exit(-1);
}
mpz_mul(a[2*i].Mp, a[i].Mp, aux2);
mpz_mod(a[2*i].Mp, a[2*i].Mp, p);
```

```
    //Calculo de ML modulo q
    if ((mpz_invert(aux2, a[2*i+1].Mq, q))==0) {
      printf("inversa nao existe 5");
      exit(-1);
    }
    mpz_mul(a[2*i].Mq, a[i].Mq, aux2);
    mpz_mod(a[2*i].Mq, a[2*i].Mq, q);
    Break_Product(2*i, p, q, N, a);    //Quebra o produto contido nos nós do lado ‘‘esquerdo’’ da árvore
    Break_Product(2*i+1, p, q, N, a);  //Quebra o produto contido nos nós do lado ‘‘direito’’ da árvore
    mpz_clear(aux1);
    mpz_clear(aux2);
  }
}

//-------------------------------------------------------------------------
//Calcula a decriptografia de b mensagens armazenadas em batch_tree segundo o metodo de Fiat
//Utiliza os primos p e q na aplicação do metodo de Quisquater-Couvreur.
//dn contem o produto dos expoentes particulares (ds)
void batch_decrypt(mpz_t p, mpz_t q, mpz_t dn, mpz_t N, batch_tree a[]){

  Product(1, p, q, N, a);
  Exponentiation(p, q, dn, a);
  Break_Product(1, p, q, N, a);

}
//-------------------------------------------------------------------------
//Inicializa a arvore de batch
void init_batch_tree(batch_tree a[], int aux[]){

  int i;

  for (i = 0; i < SIZE_BATCH; i++){
    mpz_init_set_ui(a[i].E, aux[i]);
    mpz_init(a[i].Cp);
    mpz_init(a[i].Cq);
    mpz_init(a[i].Mp);
    mpz_init(a[i].Mq);
    mpz_init(a[i].Xp);
    mpz_init(a[i].Xq);
  }
}

//-------------------------------------------------------------------------
//Desaloca a memoria utilizada pela arvore de batch
void clear_batch_tree(batch_tree a[]){

  int i;
```

```
  for (i = 0; i < SIZE_BATCH; i++){
    mpz_clear(a[i].E);
    mpz_clear(a[i].Cp);
    mpz_clear(a[i].Cq);
    mpz_clear(a[i].Mp);
    mpz_clear(a[i].Mq);
    mpz_clear(a[i].Xp);
    mpz_clear(a[i].Xq);
  }
}
//---------------------------------------------------------------------------
```

Apresentamos a seguir os programas responsáveis pela geração das chaves utilizadas pelas variações estudadas em nosso trabalho[1].

rsaqckg.c : RSA QC - Geração de Chaves

```
#include <stdio.h>              //Para printf e fprintf
#include <stdlib.h>             //Para a função exit
#include <gmp.h>                //Para as funções gmp
#include "random_package.h"
#include "print_package.h"

//------------------------------------------------------------------------
Inicializa as variáveis gmp utilizadas
void init_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d){

  mpz_init_set_ui(p, 0);
  mpz_init_set_ui(q, 0);
  mpz_init(dp);
  mpz_init(dq);
  mpz_init(qInv);
  mpz_init(n);
  mpz_init(e);
  mpz_init(d);
 }

//------------------------------------------------------------------------
Desaloca memória utilizada pelas variáveis gmp utilizadas
void clear_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d){

  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(dp);
  mpz_clear(dq);
  mpz_clear(qInv);
  mpz_clear(n);
  mpz_clear(e);
  mpz_clear(d);
}
```

[1]Estes programas foram baseados na implementação de Aggelos Keromitis na geração de chaves do RSA tradicional.

```
//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY segundo PKCS#1 v2.1
void save_seckey(FILE **fp, mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv){

  fprintf(*fp, "p = %s\n",  mpz_get_str((char *) NULL, 16, p));
  fprintf(*fp, "q = %s\n", mpz_get_str((char *) NULL, 16, q));
  fprintf(*fp, "dp = %s\n", mpz_get_str((char *) NULL, 16, dp));
  fprintf(*fp, "dq = %s\n", mpz_get_str((char *) NULL, 16, dq));
  fprintf(*fp, "qInv = %s\n", mpz_get_str((char *) NULL, 16, qInv));
}

//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY segundo PKCS#1 v2.1
void save_pubkey(FILE **fp, mpz_t n, mpz_t e){

   fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
   fprintf(*fp, "e = %s\n", mpz_get_str((char *) NULL, 16, e));
}

//---------------------------------------------------------------------
void create_primes(mpz_t x, int i){

  create_random(x, i, 2)

  while (!mpz_probab_prime_p(x, 25))    /* Encontra um primo */
    mpz_add_ui(x, x, 1);
}

//---------------------------------------------------------------------
int create_exponents(mpz_t p, mpz_t q, int p_exponent, int i, mpz_t e, mpz_t d){

  mpz_t phi;

   mpz_sub_ui(p, p, 1);
   mpz_sub_ui(q, q, 1);
   mpz_init(phi);
   mpz_mul(phi, p, q);       /* Calcula phiN = (p - 1)(q - 1) */

   if (p_exponent) {      /* expoente publico escolhido pelo usuario */
     mpz_set_ui(e, p_exponent);
   }
   else {
     while ((i--) > 0){
       mpz_mul_ui(e, e, 16);
       mpz_add_ui(e, e, get_random());
     }
```

```
    while (mpz_cmp(e, phi) >= 0)        // Divide ele caso seja maior que n
      mpz_div_ui(e, e, 2);

    do {
      mpz_add_ui(e, e, 1);
      mpz_gcd(d, e, phi);              ///d = mdc(e, phi)
    } while (mpz_cmp_ui(d, 1) && mpz_cmp(d, phi)<=0);   // até mdc(e, phi) = 1 para garantir inversa multiplicativa
  }

  if (mpz_invert(d, e, phi) == 0){
    printf("Inversa de e nao existe, saindo...\n");
    exit(1);
  }

  if ( !mpz_cmp_ui(q, 1) || mpz_cmp(q, phi) >= 0){
    printf("Falhou ao encontrar um d, reiniciando ...\n" );
    return 0;
  }

  if (mpz_cmp_ui(d, 0) < 0)           // Se d é negativo, adiciona-se o módulo
    mpz_add(d, d, phi);

  mpz_clear(phi);

  return 1;
}

//---------------------------------------------------------------------
void create_coefficients(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t d){

  //Calculando dp, dq
  mpz_mod(dp, d, p);
  mpz_mod(dq, d, q);

  //Recuperando p, q e calculando qInv
  mpz_add_ui(p, p, 1);
  mpz_add_ui(q, q, 1);
  mpz_invert(qInv, q, p);
}

//---------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    :  N, e                    em hexa
// Chave Particular :  p, q, dp, dq, qInv      em hexa

int main(int argc, char **argv) {

  mpz_t p, q, dp, dq, n, d, e, qInv;
  int  bits, p_exponent = 0, create;
  FILE *fp1, *fp2;
```

```
   if (argc < 2){
     printf("Usar: %s <modulo> [<expoent publico>]\n", argv[0]);
     printf("\ttamanho do modulo em bits\n");
     printf("\texpoente publico opcional\n");
     exit(-1);
   } if ( argc > 2 ) p_exponent = atoi(argv[2]);

   bits = atoi(argv[1]);
   if (bits < 32){
     printf("Tamanho da chave Invalido!.\n");
     exit(-1);
   }

   init_mpz_vars(p, q, dp, dq, qInv, n, e, d);
   create = 0;
   while (!create) {
     create_primes(p, bits/4);
     printf("Pegou o primeiro primo.\n");
     create_primes(q, bits/4);
     printf("Pegou o segundo primo.\n");
     mpz_mul(n, p, q);                  // Calcula o módulo RSA
     create = create_exponents(p, q, p_exponent, bits/4, e, d);
   }

   //Cria os coeficientes necessarios para a aplicacao do TCR
   create_coefficients(p, q, dp, dq, qInv, d);

   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, p, q, dp, dq, qInv);
   save_pubkey(&fp2, n, e);

   printf("Chave publica escrita em \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);

   //Desaloca memoria das variaveis utilizadas
   clear_mpz_vars(p, q, dp, dq, qInv, n, e, d);
   fclose(fp1);
   fclose(fp2);

   return 0; //Avisa ao SO que tudo OK
}
```

****

batchkg.c : RSA em Lote - Geração de Chaves

/∗ **Entrada:** 1 - Tamanho do módulo RSA em bits.
2 - Sequência de b expoentes públicos separados por um espaço.

Saida: 1 - Chave pública na forma:
$< N, e_1, N, e_2, N, e_3, N, e_b >$, onde b é o tamanho do lote, e_i é o i-ésimo expoente público e N é o módulo RSA.
2 - Chave particular na forma:
$< p, q, N, d_1, N, d_2, ..., N, d_b >$, onde d_i corresponde ao i-ésimo expoente secreto e p, q são números primos tal que $N = pq$. ∗/

```
#include <stdio.h>              //Para printf e fprint
#include <stdlib.h>             //Para a função exit
#include <gmp.h>                //Para as funções gmp
#include "random_package.h"
#include "print_package.h"

#define MAXEXP 32

//-------------------------------------------------------------------------
//Inicializa as variaveis gmp utilizadas durante o programa
void init_mpz_vars(mpz_t p, mpz_t q, int argc, mpz_t e, mpz_t d[], mpz_t n, mpz_t phi){

 int i;

   mpz_init(p);
   mpz_init(q);
   mpz_init (e);
   for (i = 0; i < argc; i++)
     mpz_init(d[i]);
   mpz_init(n);
   mpz_init(phi);

}
//-------------------------------------------------------------------------

//Desaloca variaveis gmp utilizadas durante o programa
void clear_mpz_vars(mpz_t p, mpz_t q, int argc, mpz_t e, mpz_t d[], mpz_t n, mpz_t phi){

  int i;
```

```
   mpz_clear(p);
   mpz_clear(q);
   mpz_clear(e);
   for (i = 0; i < argc; i++)
     mpz_clear(d[i]);
   mpz_clear(n);
   mpz_clear(phi);

}

//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY
void save_seckey(FILE **fp, mpz_t p, mpz_t q, int argc, mpz_t n, mpz_t d[]){

  int i;

  fprintf(*fp, "p = %s\n", mpz_get_str((char *) NULL, 16, p));
  fprintf(*fp, "q = %s\n", mpz_get_str((char *) NULL, 16, q));

  for (i = 0; i < argc - 2; i++){
     fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
     fprintf(*fp, "d = %s\n", mpz_get_str((char *) NULL, 16, d[i]));
  }
}

//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY
void save_pubkey(FILE **fp, int argc, mpz_t n, int p_exponent[]){

  int i;

  for (i = 0; i < argc - 2; i++){
    fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
    fprintf(*fp, "e = %x\n", p_exponent[i]);
  }
}

//---------------------------------------------------------------------
//Recebe um numero inteiro apontado por x e faz com que este seja
//primo e que x-1 seja relativamante primo aos expoentes publicos
//em p_exponent
//argc: indica a quantidade de expoentes publicos
// e p_exponent é um vetor contendo os expoentes publicos utilizados
void create_p(mpz_t x, int argc, int p_exponent[]){

  mpz_t e, phi;
  int j, Fail = 0;
```

```
   mpz_init(e);
   mpz_init(phi);
   mpz_sub_ui(phi, x, 1);
   //Verifica que os es sao relativamente primos a x-1
   for (j = 2; (j < argc) && !Fail; j++){
     mpz_set_ui(e, p_exponent[j-2]);
     if (!mpz_invert(e, e, phi)) Fail = 1;
   }

   //Garante que os es sao relativamente primos a x-1
   while (!mpz_probab_prime_p(x, 25) || Fail){
     Fail = 0;
     mpz_add_ui(x, x, 1);
     mpz_sub_ui(phi, x, 1);
     for (j = 2; (j < argc) && !Fail; j++){
       mpz_set_ui(e, p_exponent[j-2]);
       if (!mpz_invert(e, e, phi)) Fail = 1;
     }
   }

   mpz_clear(e);
   mpz_clear(phi);
}

//---------------------------------------------------------------------
//Gera dois primos apontados por p e q com de tamanho
//argc : indica a quantidade de expoentes publicos utilizados
//p_exponent é o vetor que contem os expoentes publicos utilizados
void create_primes(mpz_t p, mpz_t q, int nibbles, int argc, int p_exponent[]){

  create_random(p, nibbles, 2);
  create_p(p, argc, p_exponent);
  printf("Pegou o primeiro primo.\n");

  create_random(q, nibbles, 2);
  create_p(q, argc, p_exponent);
  printf("Pegou segundo primo.\n");

}

//---------------------------------------------------------------------
//Cria os expoentes particulares apontados pelo vetor d[]
//phi: contem o produto dos (pi -1)
//argc : indica a quantidade de expoentes publicos utilizados
//p_exponent é o vetor dos expoentes publicos utilizados
int create_exponents(mpz_t e, mpz_t d[], mpz_t phi, int argc, int p_exponent[]){

  int j;
```

```
  for (j = 2; j < argc; j++){
    mpz_set_ui(e, p_exponent[j-2]);

    //Calcula a inverda de e mod phi
    mpz_invert(d[j-2], e, phi);
    if (mpz_cmp(d[j-2], phi) >= 0){
      printf("Falhou ao econtrar d%d, reiniciando ...\n", j-1);
      return 0;
    }

    //se d for negativo adicionamos phi
    if (mpz_cmp_ui(d[j-2], 0) < 0)
      mpz_add(d[j-2], d[j-2], phi);
  }
  return 1;
}


//---------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    :  N, e                     em hexa
// Chave Particular :  p, q, N, d1, N, d2 ...   em hexa

int main(int argc, char **argv){

  mpz_t p, q, phi, n, e, d[MAXEXP];
  int i, bits, nibbles, j, create, status = 0,
      p_exponent[8] = {0,0,0,0,0,0,0,0};
  FILE *fp1, *fp2;

   if (argc < 2){
     printf("Uso: %s <modulo> <expoente publico 1> <expoente publico 2> ... <expoente publico k> \n", argv[0]);
     printf("\t modulo = tamanho do modulo em bits\n");
     printf("\texponentes publicos separados por um espaco\n");
     exit(-1);
   } else if (argc > 2){
            for (j = 2; j < argc; j++)
               p_exponent[j-2] = atoi(argv[j]);
   } else if (argc > MAXEXP) {
            printf("Numero excessivo de expoentes maximo = %d", MAXEXP);
            exit(-1);
          }

   bits = atoi(argv[1]);
   if (bits < 32) {
     printf("Tamanho da chave Invalido!\n");
     exit(-1);
   }
```

```
   //Cria os dois primos e os expoentes necessarios pela chaves
   init_mpz_vars(p, q, argc, e, d, n, phi);
   create = 0;
   while (!create) {
     create_primes(p, q, bits/4, argc, p_exponent);
     mpz_mul(n, p, q);           // Calcula o modulo RSA
     mpz_sub_ui(p, p, 1);
     mpz_sub_ui(q, q, 1);
     mpz_mul(phi, p, q);         // Calcula phi = (p - 1)*(q - 1)
     mpz_add_ui(p, p, 1);
     mpz_add_ui(q, q, 1);
     create = create_exponents(e, d, phi, argc, p_exponent);
   }

   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, p, q, argc, n, d);
   save_pubkey(&fp2, argc, n, p_exponent);

   printf("Chave publica escrita em \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);

   //Desaloca memoria das variaveis utilizadas
   clear_mpz_vars(p, q, argc, e, d, n, phi);
   fclose(fp1);
   fclose(fp2);

   return 0;    //Indica ao SO que tudo OK
}
```

****

mprimekg.c : RSA com Múltiplos Primos - Geração de Chaves

/* **Entrada:** 1 - Tamanho do módulo RSA em bits.
2 - Expoente público em hexadecimal (Opcional).

Saida: 1 - Chave pública na forma:
$< N, e >$, onde e correspoente ao expoente público e N corresponde ao módulo RSA.
2 - Chave particular na forma:
$< p, q, d_p, d_q, qInv >$ e triplas $< r_i, d_i, t_i >$ (para i = 3, ... k) , onde k indica o
número de primos utilizados, d_i é o i-ésimo expoente particular e t_i o i-ésimo coeficiente do TCR. */

```
#include <stdio.h>              //Para printf e fprintf
#include <stdlib.h>             //Para a função exit
#include <gmp.h>                //Para as funções gmp
#include "random_package.h"
#include "print_package.h"
```

```
//---------------------------------------------------------------------
#define k 3

//Inicializa variáveis gmp utilizadas
void init_mpz_vars(mpz_t p[], mpz_t di[], mpz_t t[], mpz_t n, mpz_t e, mpz_t d){

  int i;

  for (i = 0; i < k; i++){
     mpz_init_set_ui(p[i], 0);
     mpz_init(di[i]);
  }

  for (i = 0; i < k-1; i++)
    mpz_init(t[i]);

  mpz_init_set_ui(n, 1);
  mpz_init(e);
  mpz_init(d);
}

//---------------------------------------------------------------------
//Desaloca memória das variáveis gmp utilizadas
void clear_mpz_vars(mpz_t p[], mpz_t di[], mpz_t t[], mpz_t n, mpz_t e, mpz_t d){

  int i;

   for (i = 0; i < k; i++){
     mpz_clear(p[i]);
     mpz_clear(di[i]);
   }
   for (i = 0; i < k-1; i++)
     mpz_clear(t[i]);

   mpz_clear(n);
   mpz_clear(d);
   mpz_clear(e);
}

//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY segundo PKCS#1 v2.1

void save_seckey(FILE **fp, mpz_t p[], mpz_t di[], mpz_t t[]){

  int i;
```

```
  for (i = 0; i < 2; i++)
    fprintf(*fp, "%c = %s\n", 'p'+ i , mpz_get_str((char *) NULL, 16, p[i]));
  for (i = 0; i < 2; i++)
    fprintf(*fp, "d%c = %s\n", 'p'+ i, mpz_get_str((char *) NULL, 16, di[i]));
  fprintf(*fp, "qInv = %s\n", mpz_get_str((char *) NULL, 16, t[0]));
  for (i = 2; i < k; i++) {
    fprintf(*fp, "%c = %s\n", 'p' + i , mpz_get_str((char *) NULL, 16, p[i]));
    fprintf(*fp, "d%c = %s\n", 'p' + i, mpz_get_str((char *) NULL, 16, di[i]));
    fprintf(*fp,"%cInv = %s\n", 'p' + i, mpz_get_str((char *) NULL, 16,t[i-1])); }
}

//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY segundo PKCS#1 v2.1
void save_pubkey(FILE **fp, mpz_t n, mpz_t e){

   fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
   fprintf(*fp, "e = %s\n", mpz_get_str((char *) NULL, 16, e));
}

//---------------------------------------------------------------------
//Armazena em x um primo de 4i/k bits
void create_primes(mpz_t x, int i){

  create_random(x, i, k);

  while (!mpz_probab_prime_p(x, 25))    /* Encontra um primo */
    mpz_add_ui(x, x, 1);
}

//---------------------------------------------------------------------
//Recebe um vetor de primos em p[] e retorna em e e d os expoentes das chaves publica e particular
int create_exponents(mpz_t p[], int p_exponent, int nibbles, mpz_t e, mpz_t d){
  int i;
  mpz_t phi;

   //Calcula phi
   mpz_init_set_ui(phi, 1);
   for (i = 0; i < k; i++){
     mpz_sub_ui(p[i], p[i], 1);
      mpz_mul(phi, phi, p[i]);       /* Calcula phi = (p - 1)*(q - 1)*(r - 1) */
   }

   i = nibbles;
```

```
   //Calcula e e d
   if (p_exponent){                    /* expoente publico */
     mpz_set_ui(e, p_exponent);  /* escolhido pelo usuario */
   }
   else {
     mpz_set_ui(e, 0);          /* Pega um e aleatorio */
     while ((i--) > 0){
       mpz_mul_ui(e, e, 16);
       mpz_add_ui(e, e, get_random());
     }

     while (mpz_cmp(e, phi) >= 0)        /*Divide e por 2, caso seja maior que phi*/
       mpz_div_ui(e, e, 2);

     do {
       mpz_add_ui(e, e, 1);
       mpz_gcd(d, e, phi);               //d = mdc(e, phi) */
     } while (mpz_cmp_ui(d, 1) && mpz_cmp(d,phi)<=0);   //Até mdc(e,phi) = 1 para garantir inversa multiplicativa
   }

   if (mpz_invert(d, e, phi) == 0){
     printf("Inversa de e não existe, saindo ... \n");
     exit(-1);
   }

   if ( !mpz_cmp_ui(d, 1) || mpz_cmp(d, phi) >= 0){
     printf("Falhou ao encontrar d, reiniciando ...\n" );
     return 0;
   }

   if (mpz_cmp_ui(d, 0) < 0)       //Se d é negativo adiciona-se o módulo*/
     mpz_add(d, d, phi);

   mpz_clear(phi);

   return 1;
}

//-------------------------------------------------------------------------
//Cria os coeficientes necessários pelo algoritmo de Garner (TCR)
void create_coefficients(mpz_t p[], mpz_t di[], mpz_t t[], mpz_t d){

  int i, j;

   //Calcula dp, dq, dr ...
   for (i = 0; i < k; i++)
      mpz_mod(di[i], d, p[i]);
```

```
   //Recupera p, q e r
   for (i = 0; i < k; i++)
     mpz_add_ui(p[i], p[i], 1);

   //Calcula qInv, rInv ... necessarios para o alg. de Garner (TCR)
   mpz_invert(t[0], p[1], p[0]);
   for (i = 1; i < k-1; i++){
     mpz_set_ui(t[i], 1);
     for (j = 0; j < i + 1; j++)
        mpz_mul(t[i], t[i], p[j]);
     mpz_invert(t[i], t[i], p[i+1]);
   }
}

//---------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    :  N,e           em hexa
// Chave Particular :  p, q, dp. dq, qInv, r, dr, rInv
int main (int argc, char **argv){

   mpz_t p[k], di[k], t[k-1], d, e, n;
   int i, bits, p_exponent = 0, create;
   FILE *fp1, *fp2;


   if (argc < 2){
     printf("Uso: %s <modulo> [<expoente_publico>]\n", argv[0]);
     printf("\to tamanho do modulo em bits\n");
     printf("\ta escolha do expoente publico eh opcional\n");
     exit(-1);
   } else if ( argc > 2 ) p_exponent = atoi(argv[2]);

   bits = atoi(argv[1]);
   if (bits < 32){
     printf("Tamanho da chave invalido!.\n");
     exit(-1);
   }

   init_mpz_vars(p, di, t, n, e, d);
   create = 0;
   while (!create){
     for (i = 0; i < k; i++){
       create_primes(p[i], bits/ 4);
       printf("Pegou %c.\n", 'p' + i);
     }
     for (i = 0; i < k; i++)
       mpz_mul(n, n, p[i]);
     create = create_exponents(p, p_exponent, bits/4, e, d);
   }
```

```
   //Cria os coeficientes necessarios para a aplicacao do TCR
   create_coefficients(p, di, t, d);

   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, p, di, t);
   save_pubkey(&fp2, n, e);

   printf("Chave publica escrita em \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);

   //Desaloca memoria das variaveis utilizadas
   clear_mpz_vars(p, di, t, n, e, d);
   fclose(fp1);
   fclose(fp2);

   return 0;  // Avisa ao SO que tudo Ok
}
```

****

mpowerkg.c : RSA com Múltiplas Potências - Geração de Chaves

/* **Entrada:** 1 - Tamanho do módulo RSA em bits.
2 - Expoente público em hexadecimal (Opcional).

Saida: 1 - Chave pública na forma:
$< N, e >$, onde e correspoente ao expoente público e N corresponde ao módulo RSA na forma $p^k q$ (k indica o tamanho da potência utilizada)
2 - Chave particular na forma:
$< p, q, d_p, d_q, qInv, e >$. onde p, q são primos de aproximadamente $(logN)/k + 1$ bits, d_p e d_q são os expoentes particulares; $qInv$ é o coeficiente do TCR; e é o expoente público. */

```
#include <stdio.h>                //Para printf e fprintf
#include <stdlib.h>               //Para exit
#include <gmp.h>                  //Para as funções gmp
#include "random_package.h"
#include "print_package.h"

#define k 2
```

```
//---------------------------------------------------------------------
//Inicializa variáveis gmp utilizadas
void init_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d, mpz_t pk){

  mpz_init(p);
  mpz_init(q);
  mpz_init(dp);
  mpz_init(dq);
  mpz_init(qInv);
  mpz_init(n);
  mpz_init(e);
  mpz_init(d);
  mpz_init_set_ui(pk, 1);
}

//---------------------------------------------------------------------
//Desaloca memória das variáveis gmp utilizadas
void clear_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d, mpz_t pk){

  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(dp);
  mpz_clear(dq);
  mpz_clear(qInv);
  mpz_clear(n);
  mpz_clear(e);
  mpz_clear(d);
  mpz_clear(pk);
}

//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY segundo PKCS#1 v2.1
void save_seckey(FILE **fp, mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t e){

   fprintf(*fp,"p = %s\n", mpz_get_str((char *) NULL, 16, p));
   fprintf(*fp,"q = %s\n", mpz_get_str((char *) NULL, 16, q));
   fprintf(*fp,"dp = %s\n", mpz_get_str((char *) NULL, 16, dp));
   fprintf(*fp,"dq = %s\n", mpz_get_str((char *) NULL, 16, dq));
   fprintf(*fp,"qInv = %s\n", mpz_get_str((char *) NULL, 16, qInv));
   fprintf(*fp,"e = %s\n", mpz_get_str((char *) NULL, 16, e));
}

//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY segundo PKCS#1 v2.1
void save_pubkey(FILE **fp, mpz_t n, mpz_t e){

   fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
   fprintf(*fp, "e = %s\n", mpz_get_str((char *) NULL, 16, e));
}
```

```
//---------------------------------------------------------------------
//Armazena em p e q primos de 4*nibbles/k+1 bits
void create_primes(mpz_t p, mpz_t q, int nibbles, int p_exponent, mpz_t e){

  mpz_t gcd;

   create_random(p, nibbles, k+1);

   if (p_exponent)  mpz_set_ui(e, p_exponent); //Se usuário entrou com expoente público
   else create_random(e, nibbles, k);       //caso contrário é criado
    mpz_init(gcd);
   mpz_gcd(gcd, e, p);
   while ((!mpz_probab_prime_p(p, 25)) || (!mpz_cmp_ui(gcd, 1) == 0)){
     mpz_add_ui(p, p, 1);
     mpz_gcd(gcd, e, p);
   }
   printf("Pegou o primeiro primo.\n");

   create_random(q, nibbles, k+1);
   while (!mpz_probab_prime_p(q, 25))   /* Encontra o primo q */
     mpz_add_ui(q, q, 1);

   printf("Pegou o segundo primo.\n");

}

//---------------------------------------------------------------------
//Calcula o valor do expoente particular
int create_exponent_d(mpz_t p, mpz_t q, int i, int p_exponent, mpz_t e, mpz_t d){

  mpz_t phi;

   mpz_sub_ui(p, p, 1);
   mpz_sub_ui(q, q, 1);
   mpz_init(phi);
   mpz_mul(phi, p, q);

   if (mpz_invert(d, e, phi) == 0){
     printf("Inversa de e nao existe, saindo ...\n");
     exit(-1);
   }

   if ( !mpz_cmp_ui(d, 1) || mpz_cmp(d, phi) >= 0){
     printf("Falhou ao encontrar d, reiniciando ...\n" );
     return 0;
   }
```

```
   if (mpz_cmp_ui(d, 0) < 0)             //Se d é negativo adiciona-se o módulo
     mpz_add(d, d, phi);

   mpz_clear(phi);
   return 1;
}

//------------------------------------------------------------------------
//Cria os coeficientes necessários pelo algoritmo de Garner (TCR)
void create_coefficients(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t d, mpz_t pk){

   //Calcula dp e dq
   mpz_mod(dp, d, p);
   mpz_mod(dq, d, q);


   //Recupera p e q
   mpz_add_ui(p, p, 1);
   mpz_add_ui(q, q, 1);

   //Calcula qInv
   mpz_init(qInv);
   mpz_invert(qInv, q, pk);
}

//------------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    :  N, e                        em hexa
// Chave Particular :  p, q, dp, dq, qInv, e       em hexa

int main(int argc, char **argv){

  mpz_t p, q, e, d, dp, dq, n, qInv, pk;
  int i, bits, p_exponent = 0, create;
  FILE *fp1, *fp2;

   if (argc < 2){
     printf("Uso: %s <modulo> [<expoente_publico>]\n", argv[0]);
     printf("\to modulo = tamanho do modulo em bits\n");
     printf("\ta escolha do expoente publico eh opcional\n");
     exit(-1);
   } else  if ( argc > 2 ) p_exponent = atoi(argv[2]);

   bits = atoi(argv[1]);
   if (bits < 32){
     printf("Tamanho da chave Invalido!.\n");
     exit(-1);
   }
```

```
   init_mpz_vars(p, q, dp, dq, qInv, n, e, d, pk);
   create = 0;
   while (!create){
     create_primes(p, q, bits/ 4, p_exponent, e);
     for (i = 0; i < k; i++)
       mpz_mul(pk, pk, p);
     mpz_mul(n, pk, q);
     create = create_exponent_d(p, q, bits/ 4, p_exponent, e, d);
   }

   //Cria os coeficientes necessarios para a aplicacao do TCR
   create_coefficients(p, q, dp, dq, qInv, d, pk);

   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, p, q, dp, dq, qInv, e);
   save_pubkey(&fp2, n, e);

   printf("Chave publica escrita em \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);

   //Desaloca variaveis utilizadas
   clear_mpz_vars(p, q, dp, dq, qInv, n, e, d, pk);
   fclose(fp1);
   fclose(fp2);

   return 0; //Indica ao SO que tudo Ok
}
```

****

rebalancedkg.c : RSA Rebalanceado - Geração de Chaves

/* **Entrada:** 1 - Tamanho do módulo RSA em bits.

Saida: 1 - Chave pública na forma:
$< N, e >$, onde e correspoente ao expoente público (da ordem de N de tamanho) e N
corresponde ao módulo RSA.
2 - Chave particular na forma:
$< p, q, d_p, d_q, qInv >$, onde p, q são primos tais que $N = pq$
d_p e d_q são expoentes particulares (pequenos módulo p - 1 e q - 1, respectivamente) e $qInv$
é o coeficiente do TCR. */

```
#include <stdio.h>                 //Para printf e fprintf
#include <stdlib.h>                //Para a função exit
#include <gmp.h>                   //Para as funções gmp
#include "random_package.h"
#include "print_package.h"
```

```
//---------------------------------------------------------------------
//Inicializa variáveis gmp utilizadas
void init_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d){

  mpz_init(p);
  mpz_init(q);
  mpz_init(dp);

  mpz_init(dq);
  mpz_init(qInv);
  mpz_init(n);
  mpz_init(e);
  mpz_init(d);
}

//---------------------------------------------------------------------
//Desaloca memória das variáveis gmp utilizadas
void clear_mpz_vars(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t n, mpz_t e, mpz_t d){

  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(dp);
  mpz_clear(dq);
  mpz_clear(qInv);
  mpz_clear(n);
  mpz_clear(e);
  mpz_clear(d);
}

//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY segundo PKCS#1 v2.1
void save_seckey(FILE **fp, mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv){

   fprintf(*fp, "p = %s\n"   , mpz_get_str((char *) NULL, 16, p));
   fprintf(*fp, "q = %s\n"   , mpz_get_str((char *) NULL, 16, q));
   fprintf(*fp, "dp = %s\n"  , mpz_get_str((char *) NULL, 16, dp));
   fprintf(*fp, "dq = %s\n"  , mpz_get_str((char *) NULL, 16, dq));
   fprintf(*fp, "qInv = %s\n", mpz_get_str((char *) NULL, 16, qInv));
}

//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY segundo PKCS#1 v2.1
void save_pubkey(FILE **fp, mpz_t n, mpz_t e){

   fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
   fprintf(*fp, "e = %s\n", mpz_get_str((char *) NULL, 16, e));
}
```

```
//------------------------------------------------------------------------
//Cria dois primos p e q de 4*nibbles/2 bits tal que mdc(p-1,q-1) = 2
void create_primes(int nibbles, mpz_t p, mpz_t q){

  mpz_t p1, q1, gcd;

   create_random(p, nibbles, 2);
   while (!mpz_probab_prime_p(p, 25))   /* Encontra o primo p */
     mpz_add_ui(p, p, 1);
   printf("Pegou o primeiro primo.\n");

   create_random(q, nibbles, 2);
   mpz_init(p1);
   mpz_init(q1);
   mpz_init(gcd);
   mpz_sub_ui(p1, p, 1);
   mpz_sub_ui(q1, q, 1);
   mpz_gcd(gcd, p1, q1);
   while ((!mpz_probab_prime_p(q, 25)) || (mpz_cmp_ui(gcd, 2))){
     /* Encontra o primo q */
     mpz_set(q1, q);
     mpz_add_ui(q, q, 1);
     mpz_gcd(gcd, p1, q1);
   }
   printf("Pegou o segundo primo.\n");

   mpz_clear(p1);
   mpz_clear(q1);
   mpz_clear(gcd);
}

//------------------------------------------------------------------------------
//Cria os expoentes das chaves (e e d)
int create_exponents(int nibbles, mpz_t p, mpz_t q, mpz_t d, mpz_t e){

  mpz_t p1, q1, dp, dq, dp1, dq1, gcd, phi;

   mpz_init(p1);
   mpz_init(q1);

   mpz_init(dp1);
   mpz_init(dq1);
   mpz_init(gcd);
   mpz_init(phi);
   mpz_sub_ui(p1, p, 1);
   mpz_sub_ui(q1, q, 1);
```

```
//----------Gera um dp --------------------------------
mpz_init_set_ui(dp, 0);
create_random(dp, nibbles, 2);
mpz_gcd(gcd, dp, p1);
while (mpz_cmp_ui(gcd, 1)){  /* Garante que mdc(dp,p-1)=1 */
  mpz_add_ui(dp, dp, 1);
  mpz_gcd(gcd, dp, p1);
}

printf("Pegou dp. \n");

//----------Gera um dq --------------------------------
mpz_init_set_ui(dq, 0);
create_random(dq, nibbles, 2);

mpz_gcd(gcd, dq, q1);
mpz_mod_ui(dp1, dp, 2);
mpz_mod_ui(dq1, dq, 2);
while ((mpz_cmp_ui(gcd, 1)) || (mpz_cmp(dp1, dq1))){
  /* Garante que mdc(dq,q-1)=1 */
  mpz_add_ui(dq, dq, 1);
  mpz_gcd(gcd, dq, q1);
  mpz_mod_ui(dq1, dq, 2);
}

printf("Pegou dq.\n");

mpz_mul(phi, p1, q1);

//Encontra um d tal que d = dp mod p - 1 e d = dq mod q - 1
mpz_sub(dp, dp, dp1);  // Calcula dp - (dp mod 2)
mpz_sub(dq, dq, dp1);  // Calcula dq - (dq mod 2)

mpz_fdiv_q_ui(dp, dp, 2);
mpz_fdiv_q_ui(dq, dq, 2);

mpz_fdiv_q_ui(p1, p1, 2);  //Calcula (p - 1)/2
mpz_fdiv_q_ui(q1, q1, 2);  //Calcula (q - 1)/2

mpz_mod(dp, dp, p1);
mpz_mod(dq, dq, q1);
```

```
   mpz_mul(dp, dp, q1);
   if (mpz_invert(d, q1, p1)==0){ //Calcula ((q-1)/2)^{-1} mod (p-1)/2
     printf("inversa 1 nao existe!");
     exit(0);
   }
   mpz_mul(dp, dp, d);

   mpz_mul(dq, dq, p1);
   if (mpz_invert(d, p1, q1) ==0){ //Calcula ((p-1)/2)^{-1} mod (q-1)/2
     printf("inversa 2 nao existe!");
     exit(0);
   }
   mpz_mul(dq, dq, d);


   mpz_add(d, dp, dq);
   mpz_mul(p1, p1, q1);
   mpz_mod(d, d, p1);

   //Calcula d fazendo d = 2d + dp mod 2;
   mpz_mul_ui(d, d, 2);
   mpz_add(d, d, dq1);

   if (mpz_invert(e, d, phi)==0){
     printf("Inversa de d nao existe!");
     return 0;
   }

   if ( !mpz_cmp_ui(e, 1) || mpz_cmp(e, phi) >= 0) {
     printf("Falhou ao encontrar e, reiniciando...\n" );
     return 0;
   }
   else  if (mpz_cmp_ui(e, 0) < 0)           //Se e é negativo, adiciona-se o módulo
           mpz_add(e, e, phi);

   mpz_clear(phi);
   mpz_clear(gcd);
   mpz_clear(p1);
   mpz_clear(q1);
   mpz_clear(dp);
   mpz_clear(dq);
   mpz_clear(dp1);
   mpz_clear(dq1);

   return 1;
}
```

```
//---------------------------------------------------------------------
//Cria os coeficientes utilizados pelo algoritmo de Garner (TCR)
void create_coefficient(mpz_t p, mpz_t q, mpz_t dp, mpz_t dq, mpz_t qInv, mpz_t d){

   //Calcula dp e dq
   mpz_sub_ui(p, p, 1);
   mpz_sub_ui(q, q, 1);
   mpz_mod(dp, d, p);
   mpz_mod(dq, d, q);
   mpz_add_ui(p, p, 1);
   mpz_add_ui(q, q, 1);
   //Calcula qInv
   mpz_invert(qInv, q, p);

}

//---------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    : N,e                          em hexa
// Chave Particular : p, q, dp, dq, qInv           em hexa
int main(int argc, char **argv){

   mpz_t p, q, n, dp, dq, d, e, qInv;
   int bits, create = 0;
   FILE *fp1, *fp2;

   if (argc < 2) {
     printf("Uso: %s <modulo>\n", argv[0]);
     printf("\tmodulo = tamanho do modulo RSA em bits\n");
     exit(-1);
   }

   bits = atoi(argv[1]);
   if (bits < 32){
     printf("Tamanho da chave Invalido!.\n");
     exit(-1);
   }

   init_mpz_vars(p, q, dp, dq, qInv, n, e, d);
   create = 0;
   while (!create) {
     create_primes(bits/ 4 , p, q);
     mpz_mul(n, p, q);            // Calcula o módulo RSA
     create = create_exponents(320/4, p, q, d, e);
   }
```

```
   create_coefficient(p, q, dp, dq, qInv, d);

   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, p, q, dp, dq, qInv);
   save_pubkey(&fp2, n, e);

   printf("Chave publica escrita em \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);


   //Desaloca variaveis utilizadas
   clear_mpz_vars(p, q, dp, dq, qInv, n, e, d);
   fclose(fp1);
   fclose(fp2);

   return 0; //Avisa ao SO que tudo Ok
}
```

****

rprimekg : RSA Rebalanceado com Múltiplos Primos - Geração de Chaves

/* **Entrada:** 1 - Tamanho do módulo RSA em bits.

Saida: 1 - Chave pública na forma:
$< N, e >$, onde e correspoente ao expoente público (da ordem de N em tamanho) e N corresponde ao módulo RSA.
2 - Chave particular na forma:
$< p, q, d_p, d_q, qInv >$ e triplas $< r_i, d_i, t_i >$ (para i = 3, ... k) , onde k indica o número de primos utilizados, d_i é o i-ésimo expoente particular e t_i o i-ésimo coeficiente do TCR. */

```
#include <stdio.h>              //Para printf e fprintf
#include <stdlib.h>             //Para a função exit
#include <gmp.h>                //Para as funções gmp
#include "random_package.h"
#include "print_package.h"

#define k 3

//---------------------------------------------------------------------
//Função responsável pela inicialização das variáveis gmp utilizadas
void init_mpz_vars(mpz_t pi[], mpz_t di[], mpz_t t[], mpz_t n, mpz_t e, mpz_t d){
  int i;
```

```
  for (i = 0; i < k; i++){
    mpz_init(pi[i]);
    mpz_init(di[i]);
    mpz_init(t[i]);
  }
  mpz_init_set_ui(n, 1);
  mpz_init(e);
  mpz_init(d);

}

//---------------------------------------------------------------------
//Função responsável pelo desalocamento da memória utilizada pelas variáveis gmp
void clear_mpz_vars(mpz_t pi[], mpz_t di[], mpz_t t[], mpz_t n, mpz_t e, mpz_t d){

  int i;

  for (i = 0; i < k; i++){
     mpz_clear(pi[i]);
     mpz_clear(di[i]);
     mpz_clear(t[i]);
   }
   mpz_clear(n);
   mpz_clear(e);
   mpz_clear(d);

}

//---------------------------------------------------------------------
//Escreve a chave particular no arquivo SECKEY segundo PKCS#1 v2.1
void save_seckey(FILE **fp, mpz_t pi[], mpz_t di[], mpz_t t[]){

  int i;

  for (i = 0; i < 2; i++)
    fprintf(*fp, "%c = %s\n", 'p'+ i , mpz_get_str((char *) NULL, 16, pi[i]));
  for (i = 0; i < 2; i++)
    fprintf(*fp, "d%c = %s\n", 'p'+ i, mpz_get_str((char *) NULL, 16, di[i]));

  fprintf(*fp, "qInv = %s\n", mpz_get_str((char *) NULL, 16, t[0]));
  for (i = 2; i < k; i++) {
    fprintf(*fp, "%c = %s\n", 'p' + i , mpz_get_str((char *) NULL, 16, pi[i]));
    fprintf(*fp, "d%c = %s\n", 'p' + i, mpz_get_str((char *) NULL, 16, di[i]));
    fprintf(*fp,"%cInv = %s\n", 'p' + i, mpz_get_str((char *) NULL, 16,t[i-1])); }
}
```

```
//---------------------------------------------------------------------
//Escreve a chave publica no arquivo PUBKEY segundo PKCS#1 v2.1
void save_pubkey(FILE **fp, mpz_t n, mpz_t e){

   fprintf(*fp, "n = %s\n", mpz_get_str((char *) NULL, 16, n));
   fprintf(*fp, "e = %s\n", mpz_get_str((char *) NULL, 16, e));
}

//---------------------------------------------------------------------
//Cria 3 primos de 4*nibles/3 bits onde mdc(p1-1, p2-1, p3-1) = 2
void create_primes(int nibbles, mpz_t pi[]){

  int i;
  mpz_t pa[3], gcd, gcd2;


   for (i = 0; i < k; i++)
     mpz_init(pa[i]);

   mpz_init(gcd);
   mpz_init(gcd2);

   // -------- Pega o primeiro primo --------------------------------
   create_random(pi[0], nibbles, k);
   while (!mpz_probab_prime_p(pi[0], 25))
     mpz_add_ui(pi[0], pi[0], 1);

   printf("Pegou o primeiro primo.\n");

   // -------- Pega o segundo primo ---------------------------------
   create_random(pi[1], nibbles, k);
   mpz_sub_ui(pa[0], pi[0], 1);
   mpz_sub_ui(pa[1], pi[1], 1);
   mpz_gcd(gcd, pa[0], pa[1]);
   while ((!mpz_probab_prime_p(pi[1], 25)) || mpz_cmp_ui(gcd, 2)){
     /* Encontra um primo */
     mpz_set(pa[1], pi[1]);
     mpz_add_ui(pi[1], pi[1], 1);
     mpz_gcd(gcd, pa[0], pa[1]);
   }

   printf("Pegou o segundo primo.\n");

  // -------- Pega o terceiro primo --------------------------------
   create_random(pi[2], nibbles, k);
   mpz_sub_ui(pa[2], pi[2], 1);
   mpz_gcd(gcd, pa[1], pa[2]);
   mpz_gcd(gcd2, pa[0], pa[2]);
```

```
   while ((!mpz_probab_prime_p(pi[2], 25)) || mpz_cmp_ui(gcd, 2) || mpz_cmp_ui(gcd2, 2)){
     /* Encontra um primo */
     mpz_set(pa[2], pi[2]);
     mpz_add_ui(pi[2], pi[2], 1);
     mpz_gcd(gcd, pa[1], pa[2]);
     mpz_gcd(gcd2, pa[0], pa[2]);
   }
   printf("Pegou o terceiro primo.\n");

   mpz_clear(pa[0]);
   mpz_clear(pa[1]);
   mpz_clear(pa[2]);
   mpz_clear(gcd);
   mpz_clear(gcd2);
}

//------------------------------------------------------------------------
//Função que implementa o algoritmo de Garner
void garner(mpz_t di[], mpz_t da[], mpz_t pa[], mpz_t d){

  mpz_t n_aux, u_aux, resul, m;

   //Teorema Chinês do Resto
   mpz_init(n_aux);
   mpz_init(u_aux);
   mpz_init(resul);
   mpz_init(m);

   mpz_mul(m, pa[0], pa[1]);
   mpz_mul(m, m, pa[2]);
   for (i = 0; i < k; i++){
     mpz_fdiv_q(n_aux, m, pa[i]);
     if ((mpz_invert(u_aux, n_aux, pa[i])) == 0)
       printf("Inversa 2 nao existe");
     mpz_mul(resul, di[i], n_aux);
     mpz_mul(resul, resul, u_aux);
     mpz_add(di[k], di[k], resul);
     mpz_mod(d, di[k], m);
   }

  mpz_clear(n_aux);
  mpz_clear(u_aux);
  mpz_clear(resul);
  mpz_clear(m);
}
```

```
//------------------------------------------------------------------------------
//Recebe um vetor de primos apontado por pi e calcula os expoentes e e d
int create_exponents(int nibbles, mpz_t pi[], mpz_t d, mpz_t e){

  int i;
  mpz_t gcd, phi, di[k+1], da[k], pa[k];

   mpz_init(gcd);
   mpz_init(phi);

   for (i = 0; i < k; i++){
     mpz_init_set_ui(di[i],0);
     mpz_init_set_ui(da[i],0);
     mpz_init(pa[i]);
     mpz_sub_ui(pa[i], pi[i], 1);
   }

   // Calcula dp
   create_random(di[0], nibbles, k-1);
   mpz_gcd(gcd, di[0], pa[0]);
   while (mpz_cmp_ui(gcd, 1)){  /* Garante que gcd(dp,p-1)=1 */
     mpz_add_ui(di[0], di[0], 1);
     mpz_gcd(gcd, di[0], pa[0]);
   }

   printf("Pegou dp. \n");

   //  Calcula os dis restantes
   for (i = 1; i < k; i++){
     create_random(di[i], nibbles, k-1);
     mpz_gcd(gcd, di[i], pa[i]);
     mpz_mod_ui(da[i-1], di[i-1], 2);
     mpz_mod_ui(da[i], di[i], 2);
     while ((mpz_cmp_ui(gcd,1)) || (mpz_cmp(da[i-1], da[i]))){
       /* Garante que gcd(dq,q-1)=1 */
       mpz_add_ui(di[i], di[i], 1);
       mpz_gcd(gcd, di[i], pa[i]);
       mpz_mod_ui(da[i], di[i], 2);
     }
     printf("Pegou d%c.\n", 'p'+i);
   }

   mpz_set_ui(phi, 1);
   for (i = 0; i < k; i++)
      mpz_mul(phi, phi, pa[i]);  //Calcula (p -1)(q - 1)...(pk - 1)
```

```
   // Calcula d e e
   //Encontra um d tal que d = dp mod p - 1 e d = dq mod q - 1
   for (i = 0; i < k; i++){
     mpz_sub(di[i], di[i], da[i]);  // Calcula dpi - (dpi mod 2)
     mpz_fdiv_q_ui(di[i], di[i], 2);
     mpz_fdiv_q_ui(pa[i], pa[i], 2);  //Calcula (pi - 1)/2
     mpz_mod(di[i], di[i], pa[i]);

   }

   mpz_init_set_ui(di[k],0);

   garner(di, da, pa, d);

   //Calcula d fazendo d = 2d + dp mod 2;
   mpz_mul_ui(d, d, 2);
   mpz_add(d, d, da[0]);

   if (mpz_invert(e, d, phi)==0){
     printf("Inversa de d nao existe!");
     return 0;
   }

   if ( !mpz_cmp_ui(e,1) || mpz_cmp(e,phi) >= 0){
     printf("Falhou ao encontrar e, reiniciando...\n" );
     return 0;
   }
   else if (mpz_cmp_ui(e, 0) < 0)   //Se e é negativo, adiciona-se o modulo
          mpz_add(e, e, phi);

   //Desaloca memoria das variaveis utilizadas
   for (i = 0; i < k; i++){
     mpz_clear(di[i]);
     mpz_clear(da[i]);
     mpz_clear(pa[i]);
   }
   mpz_clear(gcd);
   mpz_clear(phi);

   return 1;
}

//-------------------------------------------------------------------------
//Calcula os coeficientes necessários pelo algoritmo de Garner (TCR)
void create_coefficients(mpz_t pi[], mpz_t di[], mpz_t t[], mpz_t d){

  int i, j;
```

```
  //Calcula dp, dq e dr
  for (i = 0; i < k; i++){
    mpz_sub_ui(t[i], pi[i], 1);
    mpz_mod(di[i], d, t[i]);
  }

  //Calcula qInv, rInv ...
  mpz_invert(t[0], pi[1], pi[0]);
  for (i = 1; i < k-1; i++){
    mpz_set_ui(t[i], 1);
     for (j = 0; j < i + 1; j++)
       mpz_mul(t[i], t[i], pi[j]);
     mpz_invert(t[i], t[i], pi[i+1]);
  }
}

//---------------------------------------------------------------------
// Grava em arquivo o par de chaves publica e particular na forma:
// Chave Publica    :  N,e                                    em hexa
// Chave Particular :  p, q, dp. dq, qInv < r, dr, rInv> ...    em hexa
int main(int argc, char **argv) {

   mpz_t  pi[k], t[k], di[k], n, d, e;
   int bits, create, i;
   FILE *fp1, *fp2;

   if (argc < 2) {
     printf("Uso: %s <modulo>\n", argv[0]);
     printf("\t modulo = tamanho do modulo em bits\n");
     exit(-1);
   }

   bits = atoi(argv[1]);

   if (bits < 32) {
     printf("Tamanho da chave invalido!.\n");
     exit(-1);
   }

   init_mpz_vars(pi, di, t, n, e, d);
   create = 0;
   while (!create){
     create_primes(bits/ 4, pi);
     for (i = 0; i < k; i++) {            //Calcula o modulo RSA
       mpz_mul(n, n, pi[i]);
     }
     create = create_exponents(320/4, pi, d, e);
   }

   create_coefficients(pi, di, t, d);
```

```
   //Grava em arquivo as chaves publica e particular
   open_file(&fp1, SECFILE, 'w');
   open_file(&fp2, PUBFILE, 'w');
   save_seckey(&fp1, pi, di, t);
   save_pubkey(&fp2, n, e);

   printf("Chave publica escrita em  \"%s\"\n", PUBFILE);
   printf("Chave particular escrita em \"%s\"\n", SECFILE);


   //Desaloca variaveis utilizadas
   clear_mpz_vars(pi, di, t, n, e, d);
   fclose(fp1);
   fclose(fp2);

   return 0;    //Avisa ao SO que tudo OK
}
//--------------------------------------------------------------------
```

Devido à similaridade entre os programas responsáveis pela chamadas das rotinas de criptografia e decriptografia, colocamos abaixo apenas o caso particular da decriptografia do RSA QC, atentando que os demais programas podem ser obtidos com a simples alteração do exemplo abaixo nas partes referentes às leituras dos dados e chamadas das primitivas criptográficas.

rsaqcdecrypt.c : RSA QC - Decriptografia

/∗ **Entrada:** 1 - arquivo apontado por SECFILE, que contem uma chave particular na seguinte forma:
$< p, q, dp, dq, qInv >$ onde p, q são primos,
dp = expoente particular d reduzido módulo $p - 1$
dq = expoente particular d reduzido módulo $q - 1$
qInv = coeficiente do TCR, ou seja, $q^{-1} \ mod \, p$

2 - arquivo apontado por CRYFILE, que contem uma mensagem cifrada C na seguinte forma:
C = <numero_menor_que_N>, onde N é parte da chave particular

Saida: Mensagem M gravada no arquivo apontado por DECFILE.
M contem a decriptografia da mensagem C utilizando a chave secreta apontada por SECFILE.

OBS: CRYFILE, DECFILE, SECFILE, NUMMSGS e NUMKEYS são especificadas em print_package.h ∗/

```
#include <gmp.h>               //Para funcoes gmp
#include "diftime.h"           //Para diftime
#include "print_package.h"     //Para file_manager e print_result
#include "rsa_package.h"       //Para rsaqcdecrypt

//---------------------------------------------------------------------------
#define MAXBUF 1000     //Tamanho maximo de caracteres de N (4000 bits)

int main(int argc, char **argv){

  mpz_t  p, q, dp, dq, qInv, M, C;
  char  buf[MAXBUF], buf2[MAXBUF];
  struct timeval t0, tf;
  long spend_time, total_time = 0;
  int  i;
  FILE *fp1, *fp2, *fp3, *fp4;  //Arquivos utilizados por file_manager

    //Inicializa as variaveis GMP
    mpz_init(p);
    mpz_init(q);
    mpz_init(dp);
    mpz_init(dq);
    mpz_init(qInv);
    mpz_init(M);
    mpz_init(C);
```

```
    //Executa NUMMSGS*NUMKEYS decriptografias
    for (i = 0; i < NUMMSGS*NUMKEYS; i++){

      //Abre os arquivos utilizados
      file_manager('o', 'd', i, &fp1, &fp2, &fp3, &fp4);

      //Le a mensagem cifrada C
      fscanf(fp1, "C = %s", buf);
      mpz_set_str(C, buf, 16);

      //Le a chave particular (p,q,dp,dq,qInv) segundo PKCS#1 v2.1
      fscanf(fp2, "%*s %*c %s %*s %*c %s", buf, buf2);
      mpz_set_str(p, buf, 16);
      mpz_set_str(q, buf2, 16);
      fscanf(fp2, "%*s %*c %s %*s %*c %s", buf, buf2);
      mpz_set_str(dp, buf, 16);
      mpz_set_str(dq, buf2, 16);
      fscanf(fp2, "%*s %*c %s ", buf);
      mpz_set_str(qInv, buf, 16);

      //Recupera M de C pelo TCR usando o algoritmo de Garner
      gettimeofday(&t0, NULL);
      rsaqcdecrypt(p, q, dp, dq, qInv, C, M);
      gettimeofday(&tf, NULL);

      //Imprime a mensagem decriptografada no arquivo de saida
      fprintf(fp3, "M = %s\n", mpz_get_str((char *) NULL, 16, M) );
      //Calcula o tempo gasto pela decriptografia
      spend_time = diftime(tf, t0);
      //Imprime o tempo gasto pela decriptografia
      print_result(i, &total_time, spend_time, &fp4);
      //Fecha os arquivos utilizados
      file_manager('c', 'd', i, &fp1, &fp2, &fp3, &fp4);
    }

    //Desaloca memoria das variaveis utilizadas
    mpz_clear(p);
    mpz_clear(q);
    mpz_clear(dp);
    mpz_clear(dq);
    mpz_clear(qInv);
    mpz_clear(M);
    mpz_clear(C);

    return 0;   //Indica ao SO que tudo OK
}
//--------------------------------------------------------------------------
```

Glossário

assinatura digital Codificação produzida pela aplicação da chave particular em uma dada mensagem.

ataque Busca ou tentativa de exploração de falhas através de um método não necessariamente matemático.

autoridade certificadora(CA) Uma autoridade certificadora (CA) é uma entidade responsável por garantir a autenticidade das chaves de uma determinada pessoa ou companhia.

cifra Nome dado a criptografias elementares.

codificar Por em código, transfigurar uma dada mensagem.

criptografia Definimos criptografia ou verificação de assinatura como a aplicação da chave pública em uma dada mensagem; *veja também* decriptografia.

criptossistema Um algoritmo de criptografia-decriptografia.

decodificar Processo inverso da codificação, tornar legível uma dada mensagem codificada.

decriptografia Decriptografia ou geração de assinatura a aplicação da chave particular. Portanto, ambos os processos podem codificar ou decodificar uma mensagem; *veja também* criptografia.

DES *Data Encryption Standard*- criptossistema de chave simétrica desenvolvido pela IBM juntamente com o governo americano na década de 70 como padrão oficial.

funções de hash Uma função que recebe um tamanho de entrada variável produzindo um tamanho fixo na saída. São normalmente utilizadas para diminuir o tamanho das assinaturas digitais além de trazer um nível a mais de segurança [26, 30].

MIPS *Million of Instructions per second*- termo designa a execução de um milhão de execuções por segundo, utilizado como medida de velocidade computacional; *veja também* MIPS anos.

MIPS anos indica a execução de um milhão de execuções por segundo durante o período ininterrupto de um ano; *veja* MIPS.

mpz *multiple precision integer*- utilizado como prefixo das funções GMP que operam com números inteiros.

PDA Personal Digital Assistant- termo designado aos equipamentos com recursos de armazenamento de informações e processamento limitado, por exemplo, computadores de mão e celulares.

PKCS *Public-key cryptography Standards*- Um conjunto de especificações relativos à criptografia publicados pelo laboratórios RSA.

primitiva criptográfica Algoritmo utilizado para codificar ou decodificar uma mensagem já formatada.

primos balanceados Termo utilizado para designar dois ou mais primos que possuam aproximadamente o mesmo tamanho em bits.

RSA Nome dado ao criptossitema de chave assimétrico criado por Rivest, Shamir e Adleman. A sigla RSA representa a inicial do nome de cada um dos pesquisadores.

I2OSP *Integer-to-Octet-String primitive*: padrão de conversão de uma mensagem X em uma mensagem formatada $0 \leq M \leq N - 1$.

OS2IP *Octet-String-to-Integer primitive*: padrão de conversão de uma mensagem formatada $0 \leq M \leq N - 1$ para uma mensagem X; *veja também* I2OSP.

Índice Remissivo

Referências Bibliográficas

[1] R. Rivest , A. Shamir , L. Adleman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*. Commum. Of the ACM 21(2): 120 - 126, 1978.

[2] D. Boneh. Twenty Years of Attacks on the RSA. *Notices of the American Mathematical Society*, vol 46(2):203–213, 1999.

[3] H. Shacham, D. Boneh. Improving SSL Handshake Performance via Batching. *In D. Naccache, ed., Proceedings of RSA 2001*, 2020 of LNCS:28–43, 2001.

[4] H. Cohen. *A Course in Computational Algebraic Number Theory, Springervol 138 do Graduate Texts in Mathematics*. Springer, p.137, 1996.

[5] S. C. Coutinho. *Números Inteiros e Criptografia RSA*. IMPA/SBM, Rio de Janeiro, 1997.

[6] J. Quisquater, C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Eletronic Letters*, vol 18:905–907, 1982.

[7] D. Boneh, G. Durfee. Cryptanalysis of RSA with Private Key d Less than $n^{0.292}$. *IEEE Transactions on Information Theory*, 46(4)::I339–I349, 2000.

[8] T. Collins , D. Hopkins , S. Langford e M. Sabin. Public Key Cryptographic Apparatus and Method. *US Patent #5,848,159*, Jan. 1997.

[9] A. Fiat. Batch RSA. *Advances in Cryptology: Proceedings of Crypto '89*, 435:175–185, 1989.

[10] GNU. Gnu Lesser General Public License. *Disponível em http://www.gnu.org/copyleft/lesser.htm*.

[11] J. Hastad. Solving simultaneous modular equations of low degree. *SIAM J. of Computing*, pages 17:336–341, 1988.

[12] R. Merkle, M. Hellman. Hiding Information and Signatures in Trapdoor Knapsacks. *IEEE Transactions on Information Theory*, IT 24-5:525–530, 1978.

[13] W. Diffie, M. E. Hellman. New Directions in Cryptografy. *IEEE Transactions on Information Theory*, vol 22:pp. 644–654, 1976.

[14] M. Jason Hinek. Low Public Exponent Partial Key and Low private Exponent Attacks on Multi-prime RSA. *Waterloo, Ontario, Canada*, 2002.

[15] D. Boneh, Glenn Durfee, Nick Howgrave-Graham. Factoring $n = p^r q$ for Large r. *Proceedings of Crypto '99, LNCS*, vol 1666, Springer-Verlag:326–337, 1999.

[16] A. k. Lenstra. (Unbelievable security: Matching AES security using public key systems. *In Advances of Cryptology - ASIACRYPT 2001*, vol 2248 of LNCS:67–86, 2001.

[17] RSA Labs. Public Key Criptography Standards (PKCS). *Version 2.1 - 2002, disponível em http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html*.

[18] RSA Labs. Factorization of RSA-155. *http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html*, 1999.

[19] RSA Labs. Has the RSA algorithm been compromised as a result of Bernstein's Paper? *http://www.rsasecurity.com/rsalabs/technotes/bernstein.html*, 2002.

[20] C. L. Lucchesi. *Introdução à Criptografia*. Quarta Escola de Computação - Instituto de Matemática e Estatística - Universidade de São Paulo, São Paulo, 1984.

[21] GNU MP. GMP. *Version 4.1 -2002, disponível em http://www.swox.com/gmp/*.

[22] R. Peralta, E. Okamoto. Faster Factoring of Integers of a Special Form. *IEICE Trans. Fundamentals*, vol E79-A, No.4:489–493, 1996.

[23] M. Y. Rhee. *Cryptography and Secure Communications*. McGraw-Hill - Series on Computer Communications, Hanyang University - Seoul, Korea, 1994.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, The MIT Press, 1990.

[25] D. Boneh, H. Shacham. Fast Variants of RSA. *RSA Laboratories*, 2002.

[26] W. Stallings. *Protect your Privacy - A Guide for PGP Users*. Prentice Hall, Inc, New Jersey, 1995.

[27] D. R. Stinson. *Cryptography - Theory and Practice*. CRC Press, Boca Raton, 1995.

[28] T. Takagi. Fast RSA-type cryptosystems using n-adic expansion. *Advances in Cryptology - CRYPTO '97*, LNCS 1294:372–384, 1997.

[29] T. Takagi. Fast RSA-type Cryptosystem Modulo $p^k q$. *In H. Krawczyk, ed., Proceedings of Crypto '98*, 1462 of LNCS. Springer-Verlag:318–326, 1998.

[30] R. Terada. *Segurança de Dados - Criptografia em Redes de Computador*. Editora Edgard Blücher Ltda, 2000.

[31] A. Menezes, P. Van Oorschot, S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[32] M. Wiener. Cryptanalysis of Short RSA Secret Exponents. *IEEE Transactions on Information Theory*, pages 36(3):553–558, 1990.