Implementação de um protótipo do *RSA*

Marcos José Cândido Euzébio

UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E CIÊNCIA DA COMPUTAÇÃO

CAMPINAS - SÃO PAULO
BRASIL

Implementação de um Protótipo do RSA

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Marcos José Cândido Euzébio e aprovada pela Comissão Julgadora.

Campinas, 2 de Setembro de 1987.

Prof. Dr. Cláudio L. Lucchesi

(Orientador)

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Conteúdo

Lista de Figuras

Lista de Tabelas

Implementação de um protótipo do *RSA*

Marcos José Cândido Euzébio
Orientador: Cláudio Leonardo Lucchesi

Agosto 1987

Abstract

The main goal of this work is to present a description of the implementation of a prototype of the RSA criptosystem. The RSA is a public key cipher and has been considered one of the best inventions in the last years in the criptography area. Its implementation brings a lot of problems, ranging from Number Theory to Analysis of Complexity of Algorithms.

The implemented prototype was used to investigate some computing bottlenecks that make the practical use of the RSA criptosystem not interesting today, and to study others problems that arise in the implementation.

This work is divided in three chapters and five appendixes. In Chapter 1, in order to show the context where the work was made, are presented some basic concepts of criptography. The RSA is described in this chapter too.

The chapter 2 is formed with the main algorithms that the program use. There are some algorithms more easily found in criptography literature and some commonly found in seminumerical-algorithm literature.

Chapter 3 describes other aspects of the program. There are informations about the way multiple precision numbers are represented internally by the program, one overview of the organization of the program modules and some tests made with the prototype in three machines: PC, VAX - 785 and D - 8000.

The appendixes contain some results obtained by the prototype and some demonstrations of its use.

Prefácio

O objetivo deste trabalho é apresentar uma descrição da implementação de um protótipo para o RSA: o PRSA. O RSA é um sistema criptográfico de chave pública e tem sido reconhecido como uma das idéias mais brilhantes que apareceram nos últimos anos na área da criptografia. Sua implementação traz uma série de desafios, dentro de um espectro que vai da Teoria dos Números à Análise de Complexidade de Algoritmos.

A implementação de um protótipo para o RSA serve para estudar os gargalos que tornam a utilização prática do sistema inviável hoje em dia, mas além disso, permitiu o estudo de vários problemas correlatos à implementação do RSA, que se apresenta como um fértil campo para pesquisas.

Este trabalho está organizado em três capítulos, onde são expostos vários aspectos da implementação do protótipo e cinco apêndices.

No capítulo 1 são feitas algumas considerações a respeito de conceitos básicos de criptografia, com intenção de situar o contexto principal em que o trabalho foi feito. São feitas algumas definições gerais de sistemas criptográficos e é apresentado o RSA.

O capítulo 2 se constitui dos principais algoritmos utilizados na implementação do programa. Além dos algoritmos mais de alto nível em geral encontrados em artigos e livros da área, tentou-se apresentar alguns pontos a respeito da geração de números aleatórios e da aritmética de números de precisão múltipla.

Já o capítulo 3 contém informações mais específicas do protótipo. É apresentada a organização modular do programa e a forma de representação interna dos números de precisão múltipla. Além disso são mostrados dois testes de desempenho do protótipo: 1) Velocidade de ciframento e deciframento e 2) Eficiência da geração de números primos. Estas duas operações são as mais importantes na implementação do RSA. É falado

também de algumas experiências feitas para permitir uma maior visualização prática do desempenho do algoritmo de Rabin para teste de primalidade.

Finalmente nos apêndices podem ser encontrados demonstrações de uso do PRSA, tabelas de números primos e uma descrição mais pormenorizada dos módulos componentes do programa.

AGRADECIMENTOS Este trabalho não teria sido possível sem o apoio, incentivo e mesmo participação de uma série de pessoas amigas e companheiras de trabalho. Assim eu não poderia deixar de agradecer ao Cláudio Lucchesi pela orientação não-paternalista mas paciente. Ao Tomasz Kowaltowski por ter nos incentivado a usar o sistema LaTeX; dentre muitas outras coisas. Ao João Meidanis pelo apoio técnico e moral e que apostou nas minhas possibilidades. Aos meus colegas de trabalho do DCC, aos colegas do curso de pós-gradução, Unicamp e Campinas. Aos meus pais, irmãos, tios e avós.

Campinas, 2 de setembro de 1987.

Capítulo 1

Criptografia e RSA: conceitos

1.1 Introdução

Criptografia é a ciência que estuda meios de garantir sigilo de informações, sejam em trânsito ou armazenadas, e a **autenticidade** dessas informações e de quem as transmite ou recebe [DAHA84]. Um **cifrador** ou **criptossistema** é um metódo que permite a escrita, em sigilo, das informações. O processo consiste em transformar um conjunto de **mensagens** em **criptogramas**.

A transformação de mensagens em criptogramas recebe o nome de **ciframento** enquanto que a transformação inversa, que torna legíveis os criptogramas , tem como nome **deciframento**. Estas duas transformações são controladas através das chamadas **chaves** criptográficas.

Estudos visando a utilização de computadores em atividades criptográficas aumentaram bastante nos últimos quinze anos em parte devido a importantes descobertas na área, tais como: criptossistemas de chave pública, assinatura digital, o **DES**(Data Encryption Standard), esquemas de proteção de chaves e protocolos para distribuição de chaves. Este desenvolvimento da criptografia teve e tem tido impacto de/em áreas diversas como a Teoria dos Números, Complexidade de Algoritmos, Algoritmos Probabilísticos, Algoritmos Semi-Numéricos etc.

Neste trabalho estaremos descrevendo a implementação de um protótipo de uma grande descoberta da criptografia moderna: o criptossistema de chave pública conhecido como **RSA**[RIVE78]. Não estaremos preocupados em apresentar um abordagem profunda dos aspec-

tos teóricos envolvidos nas várias fases da implementação do RSA e nem no papel desempenhado pelo mesmo quando utilizado em protocolos criptográficos; ao leitor interessado em tais aspectos sugerimos uma consulta aos trabalhos de Lucchesi[LUCC86], Dahab[DAHA84], Denning[DENN82], Konheim[KONH81] além de muitos outros citados na bibliografia. As duas primeiras citações acima correspondem a trabalhos escritos em português. A criptografia é uma arte/ciência que tem evoluído bastante nos últimos anos e as referências acima apresentam definições precisas dos conceitos da área dando ao leitor interessado condições de adquirir os conhecimentos necessários ao acompanhamento dos progressos que surgem frequentemente.

Nosso trabalho foi feito em software, entretanto já existem implementações do RSA por hardware. Rivest[RIVE80] descreve uma tal implementação da qual podem ser obtidos alguns detalhes em TreLeaven e Mansi[TREL85].

Até algumas décadas atrás o uso da criptografia era confinado aos meios militares e diplomáticos e só há pouco tempo esta realidade mudou com a criptografia ganhando o domínio público. Grandes corporações e companhias privadas começaram a utilizar a criptografia como defesa contra a escalada da espionagem industrial para garantir o sigilo de dados importantes. A necessidade de utilizar as facilidades oferecidas pelos avanços recentes da computação e novas tecnologias de comunicação também tiveram um papel importante no crescimento do uso da criptografia. Isto porque a proliferação de redes de computadores, transferência eletrônica de fundos, caixa automático, correio eletrônico, terminais de ponto de vendas, conferência eletrônica, etc., serviu para aumentar a vulnerabilidade de dados importantes em trânsito através de *grampeamentos*.

Assim a tecnologia que tornou as maravilhas da sociedade computadorizada viável tem pés de barro necessitando de soluções para o problema da segurança, além de que deve ser levado em consideração que o problema se torna mais difícil pela possibilidade da utilização de computadores no comprometimento desta mesma segurança. Daí o interesse tanto de cidadãos comuns como de empresas comerciais na aquisição de criptossistemas indexcriptossistema mais seguros, obrigando a comunidade acadêmica à pesquisa e criação de tais sistemas.

Uma das soluções mais elegantes já surgidas para resolver este pro-

blema baseia-se no conceito de sistema criptográfico[1] de chave pública descoberto por Diffie e Hellman[DIFF77], que provê mecanismos não apenas para resolver os problemas de sigilo e autenticação de mensagens mas também para proteção contra falsificação ou dissimulação de remetentes através da **assinatura digital**. Diffie e Hellman não apresentaram um metódo prático de implementação baseado em tal conceito, mas motivaram Rivest, Shamir e Adleman a criar um que, em homenagem aos seus autores, é conhecido como RSA.

Nos cifradores clássicos ou convencionais, tanto o processo de ciframento quanto o de deciframento utilizam uma mesma chave. Isto cria uma dificuldade na distribuição das mesmas em segredo, o que só pode ser feito através de encontros pessoais, mensageiros fiéis etc. , procedimentos estranhos e ineficientes em um ambiente de alta tecnologia.

Os sistemas de chave pública, por outro lado, utilizam chaves diferentes nos processos de ciframento e deciframento. A chave de ciframento é de domínio público enquanto que a chave de deciframento deve ser mantida em segredo. Pelo fato de utilizarem duas chaves diferentes estes criptossistemas são também classificados como **assimétricos**, em contraposição aos sistemas convencionais de chave única que são classificados como **simétricos**.

De uma forma geral qualquer sistema criptográfico, seja ele convencional ou de chave pública, consiste basicamente de: (1) algoritmos que permitam o ciframento e deciframento de mensagens e (2) chaves necessárias à garantia da segurança e sigilo destas transformações. Uma das condições impostas a tais sistemas é que a sua segurança deve se basear estritamente no segredo das chaves secretas e nunca na ignorância de um **criptanalista** com relação aos algoritmos de ciframento utilizados.

Atualmente o criptossistema simétrico mais famoso ainda é o DES, que foi adotado como padrão pelo NBS(National Bureau of Standarts) dos Estados Unidos. Jamais um cifrador foi tão atacado e defendido como o DES e muitos trabalhos podem ser encontrados na literatura sobre ele. Muitas críticas ao DES estão relacionadas com a utilização de uma chave de apenas 56 bits e também na não divulgação, por completo, dos critérios utilizados no seu projeto. Passados mais de dez anos desde o seu lançamento não se sabe se algum criptanalista conseguiu quebrar o DES, apesar das

[1] Sistema criptográfico, criptossistema e cifrador são sinônimos e serão usados de forma alternante apenas por uma questão de estilo.

muitas tentativas já feitas.

1.2 Algumas definições básicas

Nesta seção vamos descrever algumas notações e características de criptossistemas e outras noções e conceitos básicos fundamentais em Criptografia.

Um criptossistema pode ser definido em termos de cinco componentes [DENN82]:

1. O espaço de mensagens, M.
2. O espaço de criptogramas, C.
3. O espaço das chaves, K.
4. Uma família de tranformações de ciframento, $E_{ke}: M \rightarrow C$, onde $ke \in K$.
5. Uma família de transformações de deciframento, $D_{kd}: C \rightarrow M$, onde $kd \in K$.

Cada transformação de ciframento E_{ke} é definida por um algoritmo de ciframento E, igual para qualquer transformação na família, e por uma chave ke, que distingue uma transformação de outra. Da mesma forma, cada transformação de deciframento D_{kd} é definida por um algoritmo de deciframento D e uma chave kd. Para um dado par (ke, kd) D_{kd} é a inversa de E_{ke}, ou seja, $D_{kd}(E_{ke}(M)) = M$ para toda mensagem $M \in M$.

Qualquer criptossistema deve satisfazer aos seguintes requisitos:

1. As transformações de ciframento e de deciframento devem ser eficientes para todas as chaves;
2. O sistema deve ser fácil de usar;
3. A segurança do sistema deve depender apenas das chaves e não do desconhecimento dos algoritmos de E e D.

O primeiro requisito é essencial em aplicações que usam computadores, já que muitas vezes o processamento de ciframentos e decifamentos é feito simultaneamente à transmissão e recepção dos dados e esta exigência evita que as transformações de ciframento e decifamento se tornem um gargalo no sistema. O segundo exige que seja fácil a obtenção de chaves de ciframento para a qual exista a transformação inversa de decifamento. Já o terceiro indica que os algoritmos de ciframento e decifamento devem ser intrinsicamente fortes, de modo que não seja possível **quebrar** o sistema pelo simples conhecimento de como ele funciona.

Outros requisitos específicos são necessários para a garantia de sigilo e autenticidade. Caso sigilo seja garantido então nenhum criptoanalista será capaz de obter as mensagens originais a partir dos criptogramas correspondentes. Assim, de uma forma mais precisa, para a garantia de sigilo devem ser cumpridos dois requisitos:

1. Determinar sistematicamente a transformação de decifamento D_{kd} a partir de criptogramas C é um problema computacionalmente intratável, mesmo se muitas mensagens M respectivas a cada criptograma são conhecidas;

2. Determinar sistematicamente as mensagens M diretamente a partir de criptogramas respectivos C também é um problema computacionalmente intratável.

O primeiro requisito assegura que ninguém é capaz de obter sistematicamente a transformação de decifamento, arriscando quanto quiser e assim ninguém é capaz de decifrar qualquer criptograma C cifrado com a transformação E_{ke}. O segundo requisito indica a força da transformação de decifamento sem a qual ninguém é capaz de decifrar as mensagens cifradas. Estes requisitos devem valer ainda que o número de criptogramas interceptados seja grande.

Para garantir sigilo é necessário então que a transformação D_{kd} seja mantida em segredo enquanto que nenhuma restrição é feita quanto à proteção da transformação E_{ke}, que pode ser revelada, a não ser que ela comprometa a segurança de D_{kd}.

Garantir **autenticidade** de uma mensagem[2] implica na garantia de que nenhum criptoanalista consiga substituir um criptograma C por outro

[2]ou integridade de uma mensagem

C' sem que isto seja detectado; de um modo mais preciso:

1. Determinar a transformação de ciframento E_{ke} a partir de um criptograma C, é um problema computacionalmente intratável, mesmo se a mensagem M correspondente é conhecida.

2. Obter um criptograma C' de forma que $D_{kd}(C')$ seja uma mensagem válida $M' \in \mathcal{M}$ é também um problema computacionalmente intratável.

O primeiro requisito afirma que ninguém é capaz de determinar sistematicamente a transformação de ciframento E_{ke}. Assim nenhum criptanalista é capaz de cifrar uma mensagem M' e substituir o criptograma C por $C' = E_{ke}(M')$. O segundo requisito garante que ninguém pode encontrar um criptograma C' que quando decifrado fornece uma mensagem M' que faça sentido. Estes requisitos devem valer independentemente da quantidade de criptogramas conhecidos por um criptoanalista.

Para a garantia da autenticidade das mensagens é necessário que apenas a transformação de ciframento E_{ke} seja guardada em segredo, ao contrário do que é exigido na garantia de sigilo. A transformação D_{kd} pode aqui ser revelada desde que ela não comprometa a segurança de E_{ke}.

Nos criptossistemas simétricos as chaves de ciframento e de decifra-mento são iguais[3]. Isto significa que as transformações de ciframento e de decifra-mento são derivadas facilmente uma a partir da outra, o que por um lado obriga que ambas as transformações seja mantidas em segredo, mas que por outro, feito isto, garante tanto sigilo quanto autenticidade das mensagens. Sigilo e autenticidade não podem ser garantidos separadamente porque neste caso E_{ke} e/ou D_{kd} ficariam expostas.

Nos criptossistemas assimétricos as chaves de ciframento e decifra-mento são diferentes, sendo computacionalmente inviável obter uma a partir da outra e assim uma das transformações E_{ke} ou D_{kd} pode ser revelada sem comprometer a outra: a garantia de sigilo e autenticidade são problemas independentes.

[3]ou ainda que não sejam, são facilmente obteníveis uma a partir da outra

1.3 Criptossistemas de chave pública

Diffie e Hellman elaboraram a idéia de criptossistema de chave pública preocupados principalmente em resolver um sério problema, que é o da distribuição das chaves criptográficas. Consideremos uma rede com n usuários. Quando se usa um sistema criptográfico convencional, o número de chaves criptográficas necessárias para garantir a segurança das informações trocadas cresce com o quadrado de n. Como em situações práticas o número de usuários pode ser grande e existe a necessidade de que a troca de chaves se processe através de canais seguros de comunicação, o problema fica realmente difícil. Quando se usa um criptossistema de chave pública não existe mais o problema de distribuição de chaves e além disto o número de chaves envolvidas cresce linearmente com o número n de usuários.

Podemos fazer uma analogia entre um sistema de chave pública e um sistema telefônico, onde cada usuário tem o número de seu aparelho em um lista telefônica. Potencialmente então qualquer usuário pode conversar com outro, bastando para isto que consulte a lista e obtenha o número do telefone do outro. Em um sistema de chave pública cada usuário gera suas próprias chaves de ciframento e deciframento, publicando a chave de ciframento em um diretório público de chaves criptográficas, juntamente com outras informações de caráter individual que o identificam, como por exemplo, nome, endereço, etc. Se um usuário B quer enviar uma mensagem para outro A, ele procura no diretório a chave pública de ciframento de A, cifra a mensagem que é enviada para A. O usuário A decifra o criptograma recebido de B usando a chave secreta de deciframento.

Expressando isto de uma forma mais precisa[DENN82], diríamos que em um sistema de chave pública cada usuário A tem um transformação pública de ciframento E_A e uma transformação secreta de deciframento D_A. A transformação D_A utiliza uma chave secreta enquanto que a transformação E_A usa uma chave pública e é computacionalmente inviável o cálculo de uma delas a partir da outra.

Se o usuário A deseja enviar uma mensagem M sigilosa para o usuário B, A calcula o criptograma $C = E_B(M)$ e o envia para B. Para decifrar o criptograma recebido e obter a mensagem original M, B aplica a transformação secreta D_B em C, obtendo

$$D_B(C) = D_B(E_B(M)) = M.$$

Este procedimento não garante a autenticidade de M tendo em vista que qualquer usuário com acesso a transformação E_B pode enviar uma mensagem M' para B, se passando por A.

Para garantir autenticidade das mensagens M enviadas a B, A deve aplicar a transformação secreta D_A a M, obtendo o criptograma $C = D_A(M)$, que é enviado a B. B então aplica a transformação pública E_A em C, e obtém

$$E_A(C) = E_A(D_A(M)) = M,$$

que é a mensagem original enviada por A.

Aqui a autenticidade da mensagem M está garantida uma vez que somente A conhece a transformação D_A, mas qualquer usuário com acesso à transformação pública E_A pode decifrar C e obter M, ou seja, não está garantido o sigilo da mensagem.

Entretanto é possível garantir tanto sigilo quanto autenticidade de M. Para isto A faz uso das transformações D_A e E_B. A aplica à mensagem M inicialmente a transformação secreta D_A e ao resultado obtido é aplicada a transformação pública E_B, obtendo assim o criptograma C, que é enviado a B. Para decifrar C e obter a mensagem original, B deve lançar mão das transformações D_B e E_A. Inicialmente B aplica ao criptograma C a transformação secreta D_B e ao resultado obtido é aplicado a transformação secreta E_A, obtendo assim a mensagem original M:

$$\begin{aligned} E_A(D_B(C)) &= E_A(D_B(E_B(D_A(M)))) \\ &= E_A(D_A(M)) \\ &= M. \end{aligned}$$

1.4 O esquema RSA

O RSA pertence a uma família de cifradores conhecidos como **cifradores de exponenciação**. Ele foi o primeiro criptossistema que incorporou as idéias de Diffie e Hellman de sistema de chave pública. Aparentemente sua segurança baseia-se na dificuldade de fatoração de inteiros, um problema conhecido desde a antiguidade e que até hoje não apresenta soluções satisfatórias do ponto de vista computacional.

O aparecimento do RSA, conhecido assim devido às iniciais dos sobrenomes dos autores: Rivest, Shamir e Adleman, veio provocar um

grande rebolico nos meios acadêmicos e mesmo na imprensa não especializada. Um artigo da revista *Scientific American* saudava a descoberta do RSA com o título "A New Kind of Cipher That Would Take Millions of Years to Break"[GARD77]. Infelizmente a revista conta com um mau precedente: ela teria publicado em 1917 um artigo sobre os cifradores de Vigenère em que eles eram dados como "impossible of translation"[DENN82,KANH67].

Depois do RSA alguns outros criptossistemas de chave pública foram criados. O criptossistema baseado no problema da mochila de Merkle e Hellman[MERK78] foi quebrado por Shamir em 1981[SHAM82]. O criptossistema baseado em códigos corretores de erro de McEliece[MCEL78] não parece ter sido muito estudado mas de qualquer forma não permite a garantia simultânea de sigilo e autenticidade de mensagens.

Os cifradores de exponenciação de um modo geral implementam as transformações de ciframento e deciframento através da aplicação da operação de exponenciação modular às mensagens ou criptogramas. É conveniente considerar aqui mensagens e criptogramas como sendo inteiros $M \in [0, n-1]$ onde n é um número inteiro positivo livre de quadrados. A transformação de ciframento é definida então como sendo:

$$E_{ke}(M) := M^e \bmod n \tag{1.1}$$

e a transformação de deciframento é definida como:

$$D_{kd}(C) := C^d \bmod n. \tag{1.2}$$

Aqui, ke é o par (e, n) e kd é o par (d, n).

Estas transformações de ciframento e deciframento são utilizadas com base na generalização devida a Euler do teorema de Fermat, que afirma que para todo M que é primo relativo com n vale a igualdade

$$M^{\phi(n)} \bmod n = 1$$

Esta propriedade [4] implica que se e e d satisfazem a relação

$$(ed) \bmod \phi(n) = 1, \tag{1.3}$$

[4] $\phi(n)$ denota a função *totient* de Euler, isto é, o número de inteiros positivos menores que n e primos relativos com n. Dada a fatoração canônica $p_1^{r_1} \cdots p_s^{r_s}$ de n em fatores primos $p_1, \cdots, p_s$ distintos, com $r_1, \cdots, r_s \geq 1$ e $s \geq 1$, sabe-se que $\phi(n) = (p_1 - 1)p_1^{r_1 - 1} \cdots (p_s - 1)p_s^{r_s - 1}$ [NIVE72,VINO77,HARD59]

e se n é livre de quadrados, então

$$M^{ed} \bmod n = M$$

para todo M em $[0, n-1]$. Assim, a operação 1.2 é inversa da operação 1.1, garantindo assim que a operação de deciframento restaura a mensagem M a partir do criptograma C [LUCC86,DENN82].

A implementação destes cifradores não é difícil, pois dado $\phi(n)$ é fácil gerar um par (e, d) que satisfaça a equação 1.3. Para isto escolhe-se d tal que $\text{mdc}(d, \phi(n)) = 1$ e e é igual ao inverso multiplicativo de d módulo $\phi(n)$.

Dado e é fácil calcular d se $\phi(n)$ é conhecido. A chave de ciframento é o par (e, n) e a de deciframento o par (d, n), desta forma para que o conhecimento de (e, n) não comprometa (d, n) é necessário que o valor de $\phi(n)$ também não seja comprometido. No caso de n ser um número primo, então $\phi(n) = n - 1$, facilmente obtenível a partir de n. Portanto, é necessário que n seja produto de pelo menos dois números primos para que um cifrador de exponenciação possa ser utilizado em um sistema de chave pública.

No RSA o módulo n é um número composto com dois fatores primos p e q. Assim, $\phi(n) = (p-1)(q-1)$.

Para gerar as chaves de ciframento e de deciframento é necessário a geração dos dois primos p e q. Podemos imaginar que p e q têm por volta de 100 casas decimais. Pode-se então calcular $n := pq$, que será de conhecimento público; entretanto istô não vai revelar os valores de p e q devido à dificulade de fatorar um número inteiro grande. Na verdade p e q só serão utilizados para computar n e $\phi(n) := (q-1)(p-1)$ e depois disto podem ser eliminados.

Para calcular a chave pública de ciframento (e, n) falta obter um número e que seja primo relativo a $\phi(n)$. Observe que qualquer número primo maior do que p e q serve, porém e não precisa ser primo. A chave secreta de deciframento (d, n) é calculada através da resolução da congruência

$$ed \equiv 1 \bmod \phi(n),$$

a qual tem solução pois e é primo relativo a $\phi(n)$[VINO77,NIVE72].

Em resumo os passos a seguir são[LUCC86]:

1. O usuário gera dois primos grandes distintos p e q, da ordem de 10^{100} cada um.

2. O usuário computa $n := pq$ e $\phi(n) := (p-1)(q-1)$.

3. O usuário escolhe, ao acaso, um inteiro $e \in [0, \phi(n)]$ tal que $\text{mdc}(e, \phi(n)) = 1$.

4. O usuário determina d, a solução, no intervalo $[0, \phi(n)]$, da equação $(ed) \bmod \phi(n) = 1$

5. O usuário publica (e, n), mantém em segredo d e esquece p, q e $\phi(n)$.

1.4.1 Sigilo e autenticação com o RSA

O RSA tem como característica importante o fato de que as transformações de ciframento e deciframentocomutam, de forma que ele pode ser utilizado independentemente para garantir sigilo e autenticação.

Considere um cenário onde estão presentes dois usuários: A e B. O usuário A gera sua chave pública de ciframento (e_A, n_A), a qual é registrada em um diretório de fácil acesso ao público. O usuário A gera também a sua chave secreta de deciframento (d_A, n_A), o mesmo fazendo B. Assim podemos definir as transformações de ciframento e deciframento respectivamente, de A e B, respectivamente, como sendo:

A–Ciframento: $E_A(M) = M^{e_A} \bmod n_A$

B–Ciframento: $E_B(M) = M^{e_B} \bmod n_B$

A–Deciframento: $D_A(C) = C^{d_A} \bmod n_A$

B–Deciframento: $D_B(C) = C^{d_B} \bmod n_B$

Para o usuário B enviar uma mensagem M em sigilo para A basta aplicar a transformação pública de ciframento E_A a M obtendo C, ou seja, $C = E_A(M)$. Quando A recebe C, pode recuperar M aplicando a C a transformação D_A, pois $M = D_A(C)$.

Da mesma forma, A pode enviar uma mensagem M autenticada para B aplicando à M a transformação secreta D_A e o resultado disto será $C = D_A(M)$. Para B decifrar C ele aplica a transformação pública E_A de A ao criptograma C de modo que $M = E_A(C)$.

Uma dificuldade aparece quando se quer garantir ambos, sigilo e autenticidade, pois deverão ser aplicadas transformações de usuários diferentes e

que tem chaves com módulos diferentes. Assim se A quer enviar uma mensagem M a B, em sigilo e autenticada, ele pode aplicar sua transformação secreta de deciframento D_A à M e ao resultado parcial a transformação pública de ciframento E_B de B, obtendo $C = E_B(D_A(M))$. O usuário B aparentemente pode decifrar C, que contém M autenticada e em sigilo, aplicando inicialmente sua transformação secreta de deciframento ao criptograma C e ao resultado parcial assim obtido aplicar a transformação pública de ciframento E_A de A. Ou seja

$$\begin{aligned} E_A(D_B(C)) &= E_A(D_B(E_B(D_A(M)))) \\ &= E_A(D_A(M)) \\ &= M. \end{aligned}$$

Mas se $n_A > n_B$ pode acontecer de $D_A(M)$ também ser maior que n_B e o resultado final não estará correto.

Existem várias alternativas, até engenhosas, para resolver este problema. Uma, óbvia, seria fazer uma **reblocagem** de $D_A(M)$, que no entanto não é muito boa tendo em vista que neste caso seria necessária a aplicação de seis transformações de ciframento e deciframento e não as quatro usuais.

Outra alternativa, que foi sugerida pelos próprios criadores do RSA, consiste na utilização de número maior de transformações para cada usuário. Aqui, cada usuário tem um par de transformações, sendo que um par será utilizado para transações envolvendo sigilo e o outro para transações envolvendo autenticação. Por exemplo, o usuário A terá o par de transformações (E_{AS}, D_{AS}) utilizado em transações em sigilo além do par (E_{AA}, D_{AA}) para transações envolvendo autenticação. A diferença básica entre estes dois pares de transformação é que existe um valor limite h conhecido por todos os usuários e módulo n_{AA} do par de transformações envolvendo autenticidade é sempre menor que h, para todos os usuários, enquanto que o módulo n_{AS} das transformações que utilizadas em transações de sigilo é sempre maior que h.

Konfelder[KONF78] resolve o problema de forma muito simples e sutil. Ele aproveita a propriedade comutativa existente no RSA observando um fato óbvio. Ora se o cálculo de $C := E_B(D_A(M))$ apresenta problemas porque $n_A > n_B$, o cálculo de $C := D_A(E_B(M))$ vai funcionar corretamente. Agora sigilo e autenticidade pode ser garantidos da seguinte forma:

Se $n_A < n_B$: A calcula $C := E_B(D_A(M))$ B calcula $M := E_A(D_B(C))$

Se $n_A > n_B$: A calcula $C := D_A(E_B(M))$ B calcula $C := D_B(E_A(C))$.

1.4.2 A segurança do RSA

Não existem ferramentas que possam ser utilizadas para provar que um dado método criptográfico é seguro. A alternativa que resta é a certificação de esquemas criptográficos pelo trabalho meticuloso de técnicos experientes e que tentam de todas as formas encontrar uma maneira de quebrar o criptossistema.

O DES foi avaliado desta forma pela IBM, que utilizou um total de 17 homens.ano tentando, sem conseguir qualquer resultado positivo, quebrar o esquema. Isto na fase de projeto, uma vez que depois o DES foi analisado por centenas de pessoas, que aparentemente, de novo nada conseguiram.

No caso do RSA, muitas das maneiras óbvias que poderiam ser utilizadas para quebrá-lo equivalem a resolver o problema de fatoração de inteiros. Como este problema vem desafiando muitas gerações de matemáticos, sem que eles consigam uma solução razoável, então podemos talvez dizer que o RSA tem sido certificado indiretamente há séculos.

Vamos apresentar a seguir algumas das estratégias que podem ser utilizadas para quebrar o RSA e os argumentos que mostram a ineficácia das mesmas.

Fatoração de n A fatoração de n, que é público, permite a determinação da chave de deciframento, pois conhecendo-se p e q pode-se calcular $\phi(n) = (p-1)(q-1)$ e como também se conhece e, o cálculo de d é direto, correspondendo a resolver a equação $(ed) \bmod \phi(n) = 1$.

Muitos são os algoritmos de fatoração conhecidos, e Knuth[KNUT81] apresenta uma série deles. Dentre os algoritmos de fatoração já publicados o de Pollard[POLL74] é um dos mais eficientes e consegue fatorar um inteiro n em tempo $O(n^{1/4})$.

O algoritmo de fatoração mais eficiente conhecido, mas não publicado, é o criado por Richard Schroeppel que pode fatorar n com aproximadamente

$$n^{\sqrt{\frac{\ln(\ln(n))}{\ln(n)}}}$$

operações aritméticas.

Para fazer uma estimativa de tempo real de fatoração é conveniente considerar que cada operação aritmética possa ser realizada em 1 microssegundo. A Tabela 1.1 correlaciona o tempo de fatoração com o número de casas decimais de n e o número de operações aritméticas, usando-se o algoritmo de Schroeppel[RIVE78].

A estimativa dada na tabela 1.1 é muito otimista do lado do Algoritmo de Schroeppel e pessimista do lado da segurança do RSA. De qualquer forma, Rivest sugere a utilização de módulos de exponenciação n com 80 casas decimais, para proporcionar um grau moderado de segurança contra um ataque baseado em tecnologia atual. Entretanto a opção por um módulo de 200 casas decimais permite uma maior segurança contra o surgimento de novos e mais eficientes algoritmos de fatoração e avanço da tecnologia para construção de computadores mais rápidos.

Cálculo direto de $\phi(n)$ Se um criptoanalista consegue calcular $\phi(n)$ diretamente, mesmo sem conhecer a fatoração de n, então como no caso do parágrafo anterior, ele consegue quebrar o RSA. É possível mostrar, no entanto, que este problema é equivalente ao problema de fatoração de n, pois dado $\phi(n)$ é fácil fatorar n.

Pode-se fazer isto, da seguinte forma:

1. Calcula-se $(p+q)$ observando-se que é igual a $n - (\phi(n) + 1)$.

2. Calcula-se $(p-q)$ observando-se que é igual a $\sqrt{(p+q)^2 - 4n}$.

Digítos	*Número de Operações*	*Tempo*
50	$1,4 \times 10^{10}$	3,9 horas
75	$9,0 \times 10^{12}$	104 dias
100	$2,3 \times 10^{15}$	74 anos
200	$1,2 \times 10^{23}$	$3,8 \times 10^{9}$ anos
300	$1,5 \times 10^{29}$	$4,9 \times 10^{15}$ anos
500	$1,3 \times 10^{39}$	$4,2 \times 10^{25}$ anos

Tabela 1.1: A eficiência do Algoritmo de Schroeppel no Pior Caso

3. Calcula-se $q := \frac{(p+q)-(p-q)}{2}$ e $p := \frac{n}{q}$.

Determinação Direta de d Rivest, Shamir e Adleman mostram que qualquer criptoanalista que consiga calcular diretamente, por um meio qualquer, o valor de d conseguirá fatalmente obter a fatoração de n, de modo que este também é um problema equivalente ao da fatoração de inteiros.

Para mostrar isto, eles apresentam os seguintes argumentos. Se d é determinado então pode-se calcular $ed - 1$ que é um múltiplo de $\phi(n)$. Porém Miller[MILL75] mostrou que é possível fatorar rapidamente n se se conhece qualquer múltiplo de $\phi(n)$.

Cálculo de raízes módulo n O problema de cálculo de raízes módulo um número composto não é tão conhecido quanto o de fatoração de inteiros mas aparenta ser bastante difícil do ponto de vista computacional.

Embora não tenham conseguido mostrar a equivalência dos dois problemas, Rivest, Shamir e Adleman conjecturam que isto seja possível, de forma que assim fica demostrada a equivalência entre o problema de quebrar o RSA e fatoração de inteiros. Na verdade Rabin[RABI79,LUCC86] mostra que um algoritmo para calcular raízes quadradas módulo um número composto permite fatorá-lo.

Reciframento de mensagens Simmons e Norris[SIMM77] apresentam um esquema para quebrar o RSA sem a necessidade de saber a fatoração de n no caso de seus fatores não terem sido cuidadosamente escolhidos. Eles descobriram que para certas chaves poucos reciframentos sucessivos de algumas mensagens davam como resultado a própria mensagem. Neste caso, com o conhecimento do criptograma $C_0 := M^e \bmod n$ e da chave pública (e, n) o criptanalista poderia obter M da seguinte forma:

$$\begin{aligned} C_1 &:= C_0^e \bmod n \\ C_2 &:= C_1^e \bmod n \\ &\vdots \\ C_{k+1} &:= C_k^e \bmod n \\ \text{e } C_{k+1} &= C_0 \end{aligned}$$

e portanto certamente[5] $M = C_k$. Se o valor k não for muito grande para um número razoável de mensagens, este esquema pode ser desastroso para o RSA.

Rivest[RIV78b] mostrou que se $\phi(p)$ possui um fator primo p' grande e por sua vez $\phi(p')$ também possui um fator primo p'' grande, a probabilidade de sucesso do esquema de Simmons e Norris é muito pequena, algo em torno de 10^{-90}, se p é da ordem de 10^{90}.

Blakley e Blakley[BLAK78] e Blakley e Borosh[BLAK79] mostraram que para qualquer escolha de um produto $n = p_1 p_2 \cdots p_r$, de r primos ímpares distintos dois a dois, pelo menos 3^r mensagens são cifradas em si próprias, quando se usa um cifrador de exponenciação, que é o caso do RSA: é o caso de todas as soluções das congruências $x \equiv 0, \pm 1 \bmod p_i$. Ou seja, para qualquer escolha de n o RSA cifra pelo menos 9 mensagens em si mesmas.

A probabilidade de encontrar tais mensagens ao acaso é pequena, se considerarmos que n pode ser da ordem de 10^{200}, mas segundo os autores mencionados acima, existem casos extremos em que até 50% das mensagens são cifradas em si próprias. Para evitar este problema e também tornar o sistema mais resistente a certos algoritmos sofisticados de fatoração eles sugerem a utilização de primos seguros, que são os números primos da forma $2p' + 1$ onde p' também é primo.

[5] A função $f(x) := x^e \bmod n$ para x no intervalo $[0, n)$ é injetora

Capítulo 2

Algoritmos

Neste capítulo vamos apresentar os algoritmos mais importantes para a implementação do RSA. A maior parte destes algoritmos desempenha um papel muito importante na geração das chaves criptográficas. No resumo dos passos a executar para a geração de tais chaves, mostrado no capítulo anterior, foram utilizadas operações primitivas tais como: geração de primos, geração de números ao acaso, cálculo de mdc, cálculo do inverso multiplicativo módulo um número, etc. para as quais devem ser apresentados algoritmos eficientes que garantam a praticidade da implementação do RSA. Outra primitiva muito importante na operacionalidade do RSA é a exponenciação modular. Vamos apresentar um algoritmo eficiente para implementar esta operação também.

Para implementar as primitivas acima citadas vamos fazer uso das primitivas aritméticas básicas, que são soma, subtração, multiplicação e divisão inteira e resto de divisão inteira(modularização). Tais operações estão disponíveis aos programadores em todas as linguagens de programação de alto nível, que no entanto só trabalham com números relativamente pequenos. Assim é importante na implementação do RSA a realização de rotinas aritméticas que trabalhem com número de precisão múltipla, ou seja que comportam uma quantidade de dígitos superior à máxima permitida pelas linguagens de alto nível como operandos primitivos. A segunda parte deste capítulo tenta apresentar os algoritmos para a implementação de tais operações de múltipla precisão.

Os algoritmos apresentados aqui podem ser encontrados também nos trabalhos de Denning[DENN82], Knuth[KNUT81], Lucchesi[LUCC86]

dentre outros.

2.1 Geração de números primos

A geração de números primos grandes desempenha um papel fundamental na geração das chaves criptográficas do RSA e portanto na implementação do RSA.

Para gerar um número primo em um dado intervalo é utilizado um algoritmo probabilístico. A idéia é gerar números ao acaso dentro deste intervalo até que se encontre um primo. Suponhamos que queremos gerar um número primo dentro do intervalo $[l_{inferior}, l_{superior}]$: o algoritmo da figura 2.1 mostra como resolver isto. Na verdade para que o procedimento da figura 2.1 seja considerado um algoritmo é necessário que realmente existam primos dentro do intervalo considerado e que o gerador de aleatórios eventualmente forneça um tal primo.

função GeraPrimos($l_{inferior}, l_{superior}$):inteiro;

{ gera um número aleatório primo no intervalo $[l_{inferior}, l_{superior}]$ }

var p : inteiro;

início
  repita
    p := um número aleatório no intervalo $[l_{inferior}, l_{superior}]$
  até que p seja um primo;
  GeraPrimos := p;
fim;

Figura 2.1: Algoritmo para geração de um número primo em um dado intervalo

Consideremos um caso simples em que $l_{inferior} = 0$ e o $l_{superior} = N$, então o número de primos $\leq N$ é denotado pela função π como sendo $\pi(N)$. É difícil determinar o valor exato de $\pi(N)$ mas existem aproximações

bem razoáveis para esta função. Uma das aproximações mais simples é a seguinte:

$$\pi(N) \approx \frac{N}{\ln(N)},$$

que é boa para N grandes. De fato, $\pi(N)$ tende assintoticamente para a aproximação indicada[HARD59]; por exemplo, se N é um bilhão, o número de primos menores que N é exatamente 50.847.534[BEIL66], enquanto que o valor de $N/\ln N$ é 48.254.942,43, de modo que a diferença entre o valor real de $\pi(n)$ e a aproximação $N/\ln N$ é pouco maior que 5%.

Ou seja, no algoritmo acima, o número de candidatos até ser encontrado um primo é aproximadamente $\ln(N)$. No intervalo $[10^{100}, 10^{101}]$, aproximadamente 1/230 dos números são primos, pois $10^{101}/\ln 10^{101} - 10^{100}/\ln 10^{100}$ é aproximadamente a $1/230(10^{101} - 10^{100})$.

Falta agora considerar os problemas de geração de números aleatórios e de teste de primalidade.

2.1.1 Testes de primalidade

A verificação da primalidade de um número é um problema que vem desafiando os matemáticos há muito tempo, mas que hoje possui soluções bastante satisfatórias.

Partindo da definição de número primo, ou seja um número maior que 1 que só é divisível por 1 e por si mesmo, não se pode chegar muito longe. Alguma melhoria se consegue restringindo a primos os candidatos a serem utilizados em testes de primalidade baseados em testes de divisibilidade. Há uns 600 anos foi descoberto que é necessário apenas considerar os primos menores do que ou iguais à raiz quadrada do número a ser testado.

A utilização de teste de divisibilidade para provar a primalidade de um certo número n, no entanto, é uma idéia impraticável quando se considera números grandes. Além de necessitar de uma tabela enorme para armazenar os números primos menores que $\sqrt{n}$(sem considerar o tempo necessário para gerar esta tabela), o número de divisões a serem executadas quando n é primo seria da ordem de $2\sqrt{n}/\ln(n)$. Imaginando um número primo n com 200 casas decimais a tabela teria cerca de $2 \times 10^{98} \div \ln 10 = 8,69 \times 10^{98}$ entradas e o tempo gasto para conferir a primalidade de n chegaria a 10^{80} anos, supondo que uma divisão pudesse ser executada a cada 10^{-8} segundos[LUCC83].

Assim outras propriedades mais interessantes do ponto de vista computacional devem ser buscadas para tornar o teste de primalidade uma atividade viável. Esta propriedade existe e foi descoberta por Fermat. Baseado em extensões e generalizações da mesma é que foi possível criar muito d os teste de primalidade em uso atualmente.

O teorema de Fermat afirma que se n é primo então para qualquer a primo relativo com n

$$a^{n-1} \equiv 1 \bmod n \tag{2.1}$$

Por outro lado se

$$a^{n-1} \not\equiv 1 \bmod n$$

e a é primo relativo de n, então com certeza n é composto. O problema com o Teorema de Fermat deve-se ao fato de ainda que a congruência 2.1 se verifique nada pode ser dito sobre a natureza de n. Aliás existe uma família de números compostos, descobertos por Carmichael, que também possuem a propriedade de sempre satisfazerem a congruência 2.1[KNUT81,POME80].

Os algoritmos eficientes conhecidos atualmente são de dois tipos: determinísticos ou probabilísticos. Os algoritmos probabilísticos surgiram há mais tempo sendo que o primeiro foi inventado por Solovay e Strassen em 1974[SOLO77]. Rabin inventou a seguir um outro algoritmo probabilístico baseado fortemente no trabalho de Miller[MILL75] que inventou um algoritmo determinístico para teste de primalidade partindo do pressuposto que uma conjectura de Riemann seja verdadeira. Quanto aos algoritmos determinísticos eficientes de testes de primalidade, em geral, bem mais complexos do que os algoritmos probabilísticos, surgiram a partir do trabalho de Leonard M. Adleman[ADLE83] em 1980. H. Cohen e H. W. Lenstra Jr. utilizaram muito das idéias de Adleman para produzir um algoritmo mais simples[COHE84], mas ainda assim muito complexo.

Na nossa implementação adotamos como algoritmo de teste de primalidade o algoritmo de Rabin[RABI80,KNUT81,RABI77], pela facilidade de implementação e pela eficiência, que é superior ao do algoritmo de Solovay e Strassen na base de 2 para 1. A figura 2.2 apresenta o algoritmo de Rabin, numa versão criada por D.E.Knuth.

A idéia que está por trás do algoritmo de Rabin é que se $n = 1 + 2^k q$ é primo e $t^q \bmod n \neq 1$ a seqüência de potências quadráticas :

$$t^q \bmod n, t^{2q} \bmod n, \ldots, t^{2^k q} \bmod n$$

```
função TesteDeRabin(n : inteiro): booleano;

  { Aplica o teste de Rabin a n, inteiro ímpar, n > 1 }

  var q, y, t : inteiro;
      k, j : inteiro;

  início
      k := o expoente da maior potência de 2 que divide n − 1; { k ≥ 1 }
      q := (n − 1)/2^k; { n = 1 + 2^k q, q é ímpar }
      t := número aleatório escolhido no intervalo [2, n − 1];
      y := t^q mod n;
      se y = 1 então TesteDeRabin := Verdade { n pode ser primo }
      senão
          j := 1;
          enquanto (j < k) & (y ≠ n − 1) faça { y = t^(2^(j−1) q) }
                y, j := y^2 mod n, j + 1
          fim enquanto;
          TesteDeRabin := y = n − 1 { se falso então n não é primo
                                      e t é testemunha }
      fim se
  fim
```

Figura 2.2: O Algoritmo de Rabin para Teste de Primalidade

termina com uma sub-seqüência não vazia de 1's, e o valor que precede imediatamente o primeiro 1, deve ser $n-1$. Caso isto não aconteça n é composto e t é chamado de **testemunha** da não-primalidade de n.

Rabin demonstra que, no caso de n ser composto, a probabilidade de t ser uma testemunha da não-primalidade de n é maior que 75%!

Assim um algoritmo para testar se um dado n é primo ou não consistiria de um certo número de chamadas ao algoritmo da figura 2.2, de modo que quanto mais chamadas fossem feitas maior seria o grau de certeza quanto ao resultado. Chamamos de **sondagem** a realização de um teste de Rabin; o algoritmo da figura 2.3 verifica se n é primo aplicando nele até s sondagens. Assim se o resultado da chamada ao algoritmo for "verdade" a probabilidade de que a resposta esteja correta é maior que $1-(\frac{1}{4})^s$, onde $(\frac{1}{4})^s$ é o valor máximo da probabilidade de que a resposta esteja errada. Considerando $s = 100$ este valor é igual a 2^{-200}. Por outro lado, se o resultado da chamada do algoritmo for "falso", então, com certeza, n é composto.

```
função TestaSePrimo(n : inteiro; s : inteiro) : booleano;

  { aplica até s testes de Rabin a n }

  var j : inteiro;

  início
    j := 1;
    enquanto (j ≤ s) & TesteDeRabin(n) faça
        j := j + 1
    fim enquanto;
    TestaSePrimo := (j > s)
  fim;
```

Figura 2.3: Algoritmo para testar se um número é primo

Numa implementação prática o algoritmo da figura 2.3 teria uma chamada a uma função que testaria a divisibilidade de n por uma quantidade limitada de números primos pequenos. Isto serve para melhorar o

desempenho do algoritmo de teste de primalidade uma vez que a operação de divisão por um número pequeno é muito mais barata que o teste de primalidade de Rabin, que se baseia no cálculo de exponenciações modulares. Um aspecto interessante aqui é que não é necessário a utilização de um número muito grande de primos pequenos para conseguir uma melhoria significativa no teste de primalidade. O que acontece é que se de início um aumento do número de primos no teste de divisibilidade dá um aumento razoável no desempenho do algoritmo de teste de primalidade o mesmo não se pode dizer quando este número atinge valores maiores.

Geração de primos seguros O algoritmo da figura 2.3 pode ser facilmente estendido para testar se um número é um primo seguro. Nesse caso poderia ser passado ao algoritmo um outro parâmetro f que indicaria a força do primo, ou seja p é um primo seguro de nível f se p é primo e f vale 1 ou se $(p-1)$ div 2 é um primo seguro de nível $f-1$. Ou seja,

$$\text{segurança}(p) := \begin{cases} 1 + \text{segurança}(\frac{p-1}{2}) & \text{se } p \text{ é primo ímpar} \\ 0 & \text{caso contrário} \end{cases}$$

A figura 2.4 contém o algoritmo obtido diretamente destas afirmações.

Entretanto quando $f > 1$ podemos utilizar o conhecimento da fatoração de $p-1$ para construir um algoritmo mais eficiente uma vez que para provar que p é primo basta obter um dos geradores a de Z_p e que, se p é realmente primo, é um **não-resíduo quadrático** de Z_p diferente de $n-1$. Estas idéias são utilizadas na versão do algoritmo geral de teste de primalidade da figura 2.5.

2.2 Geração de números aleatórios

A utilização de números aleatórios é uma constante na área de computação. Aqui eles são utilizados em programas de simulação, análise numérica, teste de programas, jogos, etc. Na nossa implementação do RSA números aleatórios uniformemente distribuídos são utilizados no algoritmo probabilístico de teste de primalidade de Rabin, na geração de números primos aleatórios, e na geração das chaves criptográficas.

```
função TestaSePrimoSeguro(n : inteiro; s, f : inteiro) : booleano;

    { testa se n é um f-primo }

    var j : inteiro;

    início
      TestaSePrimoSeguro := falso;
      se n passa no crivo então
        se (f = 1) ou ((f > 1) & TestaSePrimoSeguro((n − 1) div 2, s, f − 1))
          então
            j := 1;
            enquanto (j ≤ s) & TesteDeRabin(n) faça
                j := j + 1
            fim enquanto;
            TestaSePrimoSeguro := j > s
        fim se
      fim se
    fim
```

Figura 2.4: Algoritmo para testar se n é primo com segurança f

```
função TestaSePrimoSeguro(n: inteiro; s, f: inteiro): booleano;

    { testa se n é um f-primo }
    var j: inteiro;
        a, y: inteiro;
    início
        TestaSePrimo := falso;
        se n passa no crivo dos primos menores que 1000 então
            se f = 1 então
                j := 1;
                enquanto (j ≤ s) & TesteDeRabin(n) faça
                    j := j + 1
                fim enquanto;
                TestaSePrimoSeguro := j > s
            senão
                se TestaSePrimoSeguro((n − 1) div 2, s, f − 1) então
                    { tenta obter um gerador a de Z_n }
                    j := 1;
                    repita
                        a := número aleatório escolhido no intervalo [2, n − 2];
                        y := a^((n−1)/2) mod n;
                        j := j + 1
                    até que (y seja diferente de 1) ou (j > s);
                    TestaSePrimoSeguro := (y = n − 1) ou (j > s)
                fim se
            fim se
        fim se
    fim
```

Figura 2.5: Algoritmo melhorado que testa se n é primo com segurança f

Este problema de gerar números aleatórios tem sido intensivamente estudado. Knuth[KNUT81] apresenta um estudo profundo dos aspectos teóricos e práticos da geração de números aleatórios.

Seqüências de números aleatórios podem ser utilizadas com finalidade criptográfica para simulação do **One Time Pad**. Neste caso os geradores de aleatórios devem apresentar, além de todas as propriedades estatísticas inerentes a um bom gerador, características adequadas a um ambiente criptográfico que garantam a segurança do sistema. Um gerador que aparentemente atende a tais exigências é o DES. Sloane[SLOA83] apresenta um esquema misto de geração de seqüências aleatórias no qual as saídas de geradores de números aleatórios com comprovadas propriedades estatísticas são combinadas com a saída de um gerador com boas características criptográficas que é o DES. O autor acredita que assim é possível obter um gerador com boas propriedades criptográficas e estatísticas.

Dentre os métodos de geração de números aleatórios uniformemente distribuídos o mais conhecido, estudado e utilizado é o método da congruência linear. O método exige a definição de 4 parâmetros básicos que são utilizados na geração de seqüências de números aleatórios uniformemente distribuídos, utilizando a equação de recorrência,

$$X_{i+1} := (aX_i + c) \bmod m; i \geq 0 \qquad (2.2)$$

onde,

m é o módulo; $m > 0$;

a é o multiplicador; $0 \leq a < m$;

c é o incremento; $0 \leq c < m$;

X_0 é o valor inicial da seqüência; $0 \leq X_0 < m$.

Assim $0 \leq X_i < m; i \geq 0$.

Para que um gerador de aleatórios construído segundo o método de congruência linear tenha as propriedades estatísticas necessárias a um bom gerador é necessário uma cuidadosa escolha dos parâmetros a e c.

Alguns valores para a e c podem ser desprezados sem exigir uma análise muito profunda. Por exemplo, os valores 0 e 1 para o parâmetro a. Neste caso a equação de recorrência 2.2 se deteriora para $X_{i+1} := (i+1)c \bmod m$

e $X_{i+1} := (X_i + (i+1)c) \bmod m$, respectivamente e a seqüência gerada não tem nada de aleatória. No caso do parâmetro c valer 0 podem acontecer duas coisas: 1) $X_i \neq 0, i \geq 0$ e a equação 2.2 simplifica para $X_{i+1} := aX_i \bmod m$ e nem todos os valores no intervalo $[0, m-1]$ serão gerados; 2) $X_i = 0, i \geq 0$ e nesse caso todos os valores restantes da seqüência serão iguais a 0.

Existem assim critérios para a escolha não só dos fatores a e c mas também do valor do módulo m. Quanto maior o valor de m maior o **período**[1] da seqüência a ser gerada. Entretanto quanto maior o valor de m mais caras são, computacionalmente, as operações aritméticas utilizadas. Ainda que consideremos valores de m de mesmo tamanho existem certos valores melhores do ponto de vista de eficiência computacional. Tais são as potências de 2, uma vez que que se o módulo é uma potência de 2 a operação módulo da equação 2.2 pode ser implementada sem precisar executar uma divisão.

Outro aspecto importante é que a escolha conveniente do valor de m pode facilitar as escolhas dos outros parâmetros a, c e X_0.

A desvantagem da escolha de m como uma potência de 2, segundo Knuth, é que os dígitos menos significativos de cada X_i são muito menos aleatórios que os dígitos mais significativos, pois se d divide m e

$$Y_i := X_i \bmod d$$

então

$$Y_{i+1} := (aY_i + c) \bmod d.$$

Assim, no caso de m ser potência de 2, os últimos k bits($k \geq 1$) de X_{i+1} dependem apenas dos últimos k bits de X_i. Portanto, o período máximo da seqüência formada pelo bit menos significativo é 2 e o da seqüência formada pelos dois bits menos significativos é 4, etc.

A escolha do parâmetro a é que vai determinar se o gerador produzirá uma seqüência de período máximo. Se fizermos $a = 1$ e $c = 1$ a seqüência gerada pela equação $X_{i+1} = (X_i + 1) \bmod m$, e tem um período de tamanho m. Tais valores de a e c, embora não produzam um gerador muito bom, servem pelo menos para mostrar que é possível obter valores para a e c em que a seqüência gerada tem um período máximo m. Knuth mostra que as

[1] O período de uma seqüência X_i é o menor t tal que $X_k = X_{k+t}$ para todo k maior que um certo k_0

seguintes condições são necessárias e suficientes para que os valores de a e c produzam um gerador de período máximo:

1. $\text{mdc}(c, m) = 1$;

2. $b := a - 1$ é um múltiplo de p, para todo primo p que divida m;

3. b é múltiplo de 4 se m é múltiplo de 4.

Concluindo, Knuth apresenta uma receita para a escolha dos parâmetros de um gerador de números aleatórios uniformemente distribuídos, que satisfaça os critérios apresentados acima, utilizando o método de congruência linear:

1. m é uma potência de 2;

2. $a \bmod 8 = 5$;

3. $\frac{m}{100} < a < \frac{99m}{100}$;

4. c deve ser ímpar;

5. X_0 pode ser qualquer.

Muitas vezes, no entanto, o valor de m é fornecido e o pior é que ele não é uma potência de 2. Nesse caso um pequeno truque serve para contornar o problema:

1. Calcule $m' :=$ menor potência de 2 maior que m;

2. Construa um gerador para gerar a seqüência $Y_{i+1} := (aY_i + c) \bmod m'$;

3. A seqüência X é obtida pela remoção da seqüência Y dos elementos maiores que m.

É fácil mostrar que se a seqüência Y é aleatória e seus elementos tem distribuição uniforme então a seqüência X, construída conforme acima, manterá estas propriedades também.

2.3 Exponenciação modular

A operação de exponenciação representa um papel muito importante na implementação do RSA uma vez que ela é utilizada para a realização das transformações de ciframento e decifrамento além de ser fundamental para a execução do teste de primalidade probabilístico de Rabin.

Esta operação consiste do cálculo de

$$a^e \bmod n \tag{2.3}$$

onde a,e e n são números inteiros positivos. Embora à primeira vista possa parecer que avaliação de 2.3 se dá em dois passos, com o cálculo de a^e precedendo a operação de modularização, não é isto o que acontece na prática, mesmo porque este seria um modo inviável computacionalmente de proceder. Felizmente a teoria dos números nos diz que no cálculo de 2.3 as operações de multiplicação podem ser intercaladas com as operações de modularização, o que torna esta operação viável.

Assim, para o calcular 2.3 basta acoplar aos algoritmos utilizados no cálculo de exponenciações uma operação de modularização após cada operação de multiplicação. A figura 2.6 apresenta uma versão recursiva de um algoritmo muito conhecido, que serve para calcular 2.3.

Um outro método que pode ser utilizado para o cálculo de exponenciação modular[2] depende do conhecimento da fatoração de e pois se $e = fg$ então $a^e \bmod n = (a^f \bmod n)(a^g \bmod n) \bmod n$. De acordo com Knuth[KNUT81] este método é melhor, na maioria dos casos, do que o método binário utilizado em 2.6, mas a melhoria obtida não é significativa e além disso o método tem o sério inconveniente de exigir a pré-computação dos fatores de e.

Depois de fazer uma séria análise do problema de determinar o número mínimo de multiplicações necessárias na avaliação de a^e, Knuth mostra que o algoritmo colocado na figura 2.6 é no pior caso apenas duas vezes mais lento do que um possível algoritmo ótimo, necessitando da avaliação de até $2\log_2(e)$ operações de multiplicação modular, enquanto que o mínimo teoricamente necessário seriam $\log_2(e)$ multiplicações modulares.

[2] e da exponenciação comum também

2.4 MDC e inverso multiplicativo

Determinar o mdc de dois números e calcular o inverso multiplicativo são operações utilizadas na geração de chaves criptográficas e portanto essenciais na implementação do RSA. Estas duas operações estão colocadas numa mesma seção porque tem implementações muito similares. Na verdade o algoritmo para cálculo de inverso multiplicativo é uma extensão do algoritmo para o cálculo do mdc.

A figura 2.7 mostra uma versão recursiva do algoritmo inventado por Euclides, para o cálculo do mdc.

Um algoritmo inventado por J. Stein em 1961[KNUT81] tem como características principais o fato de utilizar apenas operações do tipo subtração, divisão e multiplicação por 2, teste de paridade, etc. que o torna adequado para implementação em computadores.

A versão iterativa do algoritmo de J.Stein é apresentada na figura 2.8.

O algoritmo de Euclides pode ser estendido para possibilitar o cálculo de valores de x e y, de forma que,

$$ax + by = \text{mdc}(a, b).$$

O algoritmo estendido, mostrado na figura 2.9, pode então ser utilizado para o cálculo da solução da equação

$$ax \bmod n = 1$$

onde x é o chamado inverso multiplicativo de a módulo n, quando $\text{mdc}(a, n) = 1$. Para isto basta fornecer ao algoritmo de Euclides estendido os valores a e n.

Versões iterativas do algoritmo simples e estendido de Euclides podem ser encontradas nos trabalhos de Knuth e Lucchesi[KNUT81,LUCC86].

2.5 Algoritmos de precisão múltipla

Os algoritmos apresentados nas seções anteriores fazem uso de primitivas aritméticas tais como soma, subtração, multiplicação, divisão e modularização de uma forma bastante natural, de modo que tais algoritmos podem ser facilmente implementados em uma linguagem de programação de alto

```
função ExpMod(a, e, n: inteiro): inteiro;
{
      Calcula a^e mod n, a > 0, e ≥ 0, n > 2
}
início
      se e = 0 então ExpMod := 1;
      senão
      se e é ímpar então ExpMod := a Expmod(a, e − 1, n) mod n
      senão { e é par e > 0 }
         ExpMod := Expmod(a^2 mod n, e div 2, n)
      fim se
      fim se
fim
```

Figura 2.6: Algoritmo para o cálculo de $a^e \bmod n$

```
função mdc(a, b: inteiro):inteiro;
{
Calcula mdc(a, b), a, b ≥ 0
}
início
    se b = 0 então mdc := a
    senão
       mdc := mdc(b, a mod b)
    fim se
fim
```

Figura 2.7: O algoritmo de Euclides para o cálculo do mdc

```
função mdc(a, b : inteiro):inteiro;

    var r:inteiro;

início
    r := 0;
    enquanto (a mod 2 = 0) & (b mod 2 = 0) faça
        r, a, b := r + 1, a div 2, b div 2
    fim enquanto;
    se b mod 2 = 0 então a, b := b, a senão a := 2 × a fim se;
    enquanto a ≠ 0 faça { a par, b ímpar }
        repita a := a div 2 até que a mod 2 ≠ 0;
        se a < b então a, b := b, a fim se;
        a := a − b
    fim enquanto;
    mdc := a
fim
```

Figura 2.8: O Algoritmo de J.Stein para o cálculo do mdc

```
função EucEst(a, b:inteiro):inteiro,inteiro;
{
   Calcula x e y tais que ax + by = mdc(a, b); a, b ≥ 0;a + b > 0
}
    var x, y: inteiro;

início
    se b = 0 então EucEst := (1, 0) { mdc(a, b) = a }
    senão
        x, y := EucEst(b, a mod b);
        EucEst := (y, x − y(a div b)) { mdc(a, b) = mdc(b, a mod b) }
    fim se
fim
```

Figura 2.9: O algoritmo de Euclides estendido

nível como C e Pascal. Esta é uma maneira bastante conveniente de apresentar os algoritmos mas numa implementação prática as primitivas aritméticas devem ser traduzidas em rotinas que permitam a manipulação de números muito maiores do que as linguagens de programação em geral suportam. Assim muito do trabalho de implementação do RSA reside justamente na implementação de um pacote de rotinas aritméticas, que devem ter a máxima eficiência dentro dos limites das linguagens de programação. Nesta secção vamos apresentar os algoritmos mais importantes na manipulação de números grandes, os quais embora do ponto de vista teórico não sejam os mais eficientes possíveis, têm um desempenho muito bom, considerando a grandeza dos números a serem utilizados(da ordem de 100 a 200 casas decimais).

O trabalho de Knuth[KNUT81] é a fonte insubstituível destes e de muitos outros algoritmos semi-numéricos. Algumas outras referências de trabalhos de implementação de pacotes aritméticos são os trabalhos de Collins[COLL66] e Brent[BREN78].

Nos algoritmos a serem descritos estaremos trabalhando com números representados segundo o sistema de numeração posicional em uma base b, dessa forma o número inteiro positivo U pode ser representado pela seqüência de dígitos

$$U = \langle u_n, u_{n-1}, \ldots, u_2, u_1 \rangle$$

de modo que

$$U = \sum_{i=1}^{n} u_i b^{i-1}$$

e $0 \leq u_i < b$ para todo $1 \leq i \leq n$. Cada u_i será chamado de **dígito** de U. É conveniente que $u_n > 0$ e quando isto ocorrer diremos que U está **normalizado**. O número 0, seguindo esta notação pode ser representado pela seqüência de 0 elementos:

$$0 = \langle\rangle.$$

Números negativos não serão considerados aqui, mas os algoritmos e a representação apresentada podem ser facilmente estendidos para comportá-los. A finalidade de evitar os números negativos é diminuir certos detalhes que embora não apresentem dificuldade de serem tratados, servem

para obscurecer e atrapalhar a descrição dos pontos mais importantes do algoritmo.

Na descrição dos algoritmos, além de primitivas aritméticas para realização de operações com números menores que b^2, estaremos com muita liberdade fazendo uso de operações usualmente empregadas em manipulação de listas e seqüências, tais como concatenação, car, cdr, etc. que facilitam nossa tarefa. A operação de concatenação será denotado pelo símbolo $\circ$.

2.5.1 Soma

A operação de soma é uma das mais simples de serem implementadas e o algoritmo utilizado é ótimo uma vez que o número de operações primitivas realizadas é linearmente proporcional ao tamanho dos números a serem somados. A soma é realizada dígito a dígito da direita para a esquerda. Quando o resultado da soma de dois dígitos é maior do que a base b deve ser subtraído dela o valor b e o vai-um recebe o valor 1. Caso contrário o vai-um recebe o valor 0. Este algoritmo é mera mecanização do algoritmo utilizado pelas pessoas quando fazem soma usando apenas lápis e papel ou similar, com $b = 10$.

2.5.2 Subtração

O algoritmo para implementar a operação de subtração é muito similar ao algoritmo utilizado na operação de soma de números grandes. O caso em que o resultado é negativo é considerado erro. Ao contrário da soma, o resultado da subtração pode ter menos algarismos do que os operandos, e portanto deve ser tomado o cuidado de normalizar o resultado.

A figura 2.11 apresenta o algoritmo correspondente à operação de subtração.

2.5.3 Multiplicação

A operação de multiplicação é a mais importante de todo o pacote aritmético para manipulação de números grandes. De seu desempenho depende o desempenho do protótipo. Infelizmente os algoritmos mais eficientes para implementar a multiplicação, além de serem muito complicados, não

```
função soma(⟨u_n, ..., u_1⟩, ⟨v_m, ..., v_1⟩):⟨w_r, ..., w_1⟩;
{
    Calcula a soma de U e V
    O resultado terá r dígitos onde r = max(n, m) + (0 ∨ 1)
}
    var i : inteiro;
        c : inteiro; { Vai-um }

início
    se n < m então soma := soma(⟨v_m, ..., v_1⟩, ⟨u_n, ..., u_1⟩);
    senão
        W := V;
        para i := m + 1 até n faça
            W := 0 ∘ W
        fim para
        c := 0;
        para i := 1 até n faça
            w_i := w_i + u_i + c;
            se w_i ≥ b então
                        w_i, c := w_i − b, 1
            senão c := 0
            fim se
        fim para;
        se c = 1 então W := c ∘ W fim se;
        soma := W
    fim se
fim
```

Figura 2.10: Soma de números grandes

```
função subtração(⟨u_n,...,u_1⟩,⟨v_m,...,v_1⟩) : ⟨w_r,...,w_1⟩;
{
    Calcula U menos V
    r é o tamanho do resultado e 0 ≤ r ≤ n
}
    var i: inteiro;
        vaium,d: inteiro;

início
    se m > n então erro fim se;
    W,vaium := ⟨⟩,0;
    para i := 1 até n faça
        d := u_i − v_i − vaium;
        se d < 0 então d,vaium := d + b,1
                  senão vaium := 0
        fim se;
        W := d ∘ W
    fim para;
    se (m = n) & vaium então erro fim se
    para i := m + 1 até n faça
        d := u_i − vaium;
        se d < 0 então d,vaium := b − 1,1
                  senão vaium := 0
        fim se;
        W := d ∘ W
    fim para;
    enquanto W ≠ ⟨⟩ eentão (car(W) = 0) faça
        W := cdr(W)
    fim enquanto;
    subtração:= W
fim
```

Figura 2.11: Subtração de números grandes

possuem um desempenho muito bom para os números grandes com os quais o RSA trabalha e apenas quando manipulam números com milhares de casas decimais seu desempenho é superior ao algoritmo clássico que aprendemos na escola primária.

Mesmo quando se trabalha com o algoritmo clássico de multiplicação encontramos problemas na sua implementação por linguagens de alto nível, pois para evitar estouros na multiplicação de números pequenos, correspondentes aos dígitos dos números grandes, somos obrigados a subutilizar as primitivas de multiplicação.

Mas, ainda que uma série de obstáculos dificultem o desempenho do algoritmo de multiplicação, alguma coisa pode ser feita para evitar o pior. Assim ao se fazer divisões ou modularização pela base b e se esta for uma potência de 2, tais operações podem ser substituídas por deslocamentos e operações lógicas, respectivamente, que são muito mais eficientes.

A figura 2.12 apresenta o algoritmo de multiplicação.

2.5.4 Divisão e modularização

Dentre todas as rotinas de precisão múltipla a rotina de divisão e modularização se destaca por ser a mais complexa. Diferente dos algoritmos de precisão múltipla já citados nesta seção, que não passam de uma mecanização dos procedimentos aprendido nos bancos escolares, o algoritmo para as operações de divisão e modularização apresenta novidades.

Um computador não dispõe da capacidade intuitiva de um ser humano para escolher ao acaso os dígitos do quociente numa operação de divisão, assim faz-se necessário criar algum método mais mecânico para efetuar a divisão.

Felizmente esta possibilidade existe e consiste em obter uma estimativa para o dígito mais significativos do quociente baseado nos valores dos dois dígitos mais significativo do dividendo e no dígito mais significativo do divisor. A estimativa é testada e caso ela não seja igual ao valor esperado são feitas algumas iterações até que isto seja obtido.

Após ter sido o calculado o dígito mais significativo do quociente o processo se repete considerando agora como dividendo o resultado da subtração do dividendo pelo produto do dígito mais significativo do quociente e do divisor. E assim passo a passo são gerados os dígitos do quociente, além do que o valor do último dividendo é o resultado da operação de

```
função multiplicação(⟨u_n, ..., u_1⟩, ⟨v_m, ..., v_1⟩) : ⟨w_r, ..., w_1⟩;
{
    Calcula U vezes V
    m + n − 1 ≤ r ≤ m + n
}
    var i, j: inteiro;
        c, d: inteiro;

início
    se (n = 0) ∨ (m = 0) então multiplicação := ⟨⟩
    senão
        { Inicializa: }
        W := ⟨0_{n+m}, ..., 0_1⟩;
        { Multiplica }
        para i:= 1 até m faça
            c := 0;
            para j:= 1 até n faça
                d := w_{i+j−1} + u_j × v_i + c;
                w_{i+j−1}, c := d mod b, d div b;
            fim para;
            w_{i+n} := w_{i+n} + c
        fim para;
        { Normaliza }
        se w_{n+m} = 0 então W := cdr(W) fim se;
        multiplicação := W
    fim se
fim
```

Figura 2.12: O algoritmo clássico de multiplicação

modularização.

Para obter estimativas razoáveis do valor do dígito mais significativo do quociente é necessário apenas que o dígito mais significativo do divisor seja maior do que ou igual a $\lfloor \frac{b}{2} \rfloor$, onde b é o valor da base do sistema posicional utilizado [KNUT81]. Quando o dígito mais significativo do divisor é menor do que $\lfloor \frac{b}{2} \rfloor$ a solução é multiplicar, antes de começar efetivamente a divisão, tanto o divisor quanto o dividendo por um valor conveniente que torna o dígito mais significativo do novo divisor maior do que ou igual a $\lfloor \frac{b}{2} \rfloor$. O interessante que após ter sido feito esta multiplicação o resultado da divisão será o mesmo que o anterior e apenas o novo resto da divisão estará multiplicado pelo mesmo fator utilizado na multiplicação do divisor.

Seja $U = \langle u_n, \ldots, u_1 \rangle$ o dividendo e $V = \langle v_m, \ldots, v_1 \rangle$ divisor e $m < n$. Seja $\hat{q}$ uma estimativa para o dígito mais significativo do quociente $\lfloor \frac{U}{V} \rfloor$ e q o valor exato de tal dígito. Knuth mostra que se $v_m \geq \lfloor \frac{b}{2} \rfloor$ e

$$\hat{q} := \text{mínimo}(b-1, \lfloor \frac{u_n b + u_{n-1}}{v_m} \rfloor)$$

então:

$$q \leq \hat{q} \leq q + 2.$$

No caso de v_m ser menor que $\lfloor \frac{b}{2} \rfloor$, V e U devem ser multiplicados por d, onde d pode ser calculado de acordo com a seguinte fórmula:

$$d := \lfloor \frac{b}{1 + v_m} \rfloor.$$

A figura 2.13 apresenta o algoritmo para o cálculo de divisão e modularização.

2.6 Multiplicação por tábua

A operação de multiplicação de números pequenos é em geral executada por hardware ou microprograma na maioria dos computadores atuais. Isto não acontece em micros de 8 bits cujos processadores não possuem uma instrução de multiplicação. Dessa forma a operação de multiplicação deve ser implementada por software, através de algoritmos bastante eficientes mas o resultado não é muito bom.

```
função DivMod(⟨u_n, ..., u_1⟩, ⟨v_m, ..., v_1⟩)
                        : ⟨w_r, ..., w_1⟩, ⟨x_s, ..., x_1⟩;
{
    Calcula W = U div V e X = U mod V
}
    var i: inteiro;
        q̂, c, d: inteiro;
início
    se V = ⟨⟩ então erro("divisão por zero")
    senão
    se m > n então DivMod := (0,U)
    senão
        X, d := 0 ∘ U, b div (1 + v_m);
        X, V := multiplicação(X, ⟨d⟩), multiplicação(V, ⟨d⟩);
        W := ⟨⟩;
        para i := n + 1 descendo até n + 1 − m faça
            q̂ := min(b − 1, (x_i × b + x_{i−1}) div v_m);
            enquanto multiplicação(⟨q̂⟩, V) < ⟨x_i, ..., x_{i−m}⟩ faça
                q̂ := q̂ − 1
            fim enquanto;
            W := W ∘ q̂;
            ⟨x_i, ..., x_{i−m}⟩ := subtração(⟨x_i, ..., x_{i−m}⟩, multiplicação(⟨q̂⟩, V))
        fim para;
        { Divide X por d }
        c := 0;
        para i := m descendo até 1 faça
            x_i, c := (c × b + x_i) div d, (c × b + x_i) mod d
        fim para;
        { Normalização}
        enquanto (W ≠ ⟨⟩) eentão car(W) = 0 faça W := cdr(W) fim enquanto;
        enquanto (X ≠ ⟨⟩) eentão car(X) = 0 faça X := car(X) fim enquanto;
        DivMod := (W, X)
    fim se
    fim se
fim
```

Figura 2.13: Algoritmo de divisão e modularização

Uma maneira de contornar este problema é através do uso de tábuas. Porém uma tábua construída aos moldes daquela que aprendemos na escola teria o grave inconveniente de gastar uma grande quantidade de memória. Niven e Zuckerman[NIVE72] no entanto apresentam um esquema no qual o espaço utilizado é quase a raiz quadrada do espaço utilizado por uma tábua comum.

Se a base utilizada é b o espaço gasto é igual a $2b - 1$, sendo que o maior valor a ser armazenado é menor que b^2, como na tábua comum. Essa é uma tábua linear, inclusive seria melhor chamá-la de tabela.

Seja t uma tabela com índices variando de 0 a $2b-2$. Se inicializarmos esta tabela com os valores

$$t[i] := \lfloor \frac{i^2}{4} \rfloor$$

a operação de multiplicação poderá ser executada através da execução de uma soma, duas subtrações e duas consultas à tabela t. Sejam x e y dois números a serem multiplicados e $x \geq y$; então $z := x \times y$ pode ser calculado da seguinte forma:

$$z := t[x + y] - t[x - y].$$

A verificação desta afirmação não é muito complicada.

Um outro aspecto interessante neste método se deve ao fato de que a inicialização da tabela t não exige o cálculo de sequer uma multiplicação. Seus elementos podem ser obtidos através da soma dos elementos da seguinte seqüência,

$$S = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, \ldots \rangle$$

e $t[i]$ pode então ser calculado como:

$$t[i] := \sum_{j=0}^{i} S_j.$$

A utilização prática deste método de multiplicação por tábua apresentou resultados muito bons, tanto no micro de 8 bits quanto em um outro de 16, que inclusive, possui instrução de multiplicação por hardware.

Capítulo 3

O protótipo

3.1 Introdução

Neste capítulo vamos descrever alguns aspectos ligados à implementação do protótipo do RSA considerando detalhes da organização do programa, representação interna utilizada para números de precisão múltipla e testes de desempenho. Além disso será apresentada neste capítulo as conclusões a que chegamos.

Na implementação do protótipo do RSA foi adotado como estratégia de trabalho a implementação de um simulador de calculadora de inteiros de precisão múltipla enriquecida com certas primitivas de interesse criptográfico. A utilização desta estratégia permitiu a abstração de certos detalhes próprios de um sistema criptográfico de modo que isto tornou possível, como abordagem de implementação do protótipo, o método de programação por refinamentos sucessivos.

Uma outra vantagem que a implementação de um protótipo simulador de calculadora de precisão múltipla trouxe foi permitir a inserção de muitas outras primitivas de interesse criptográfico que não estão relacionadas diretamente com o sistema criptográfico de chave pública de Rivest, Shamir e Adleman. É possível que com estas e algumas outras primitivas seja possível tornar este protótipo em um núcleo de um ambiente adequado à criptografia de aplicações pelo menos didáticas.

Atualmente PRSA[1] é capaz de:

[1] de Protótipo do **RSA**

1. Executar as operações aritméticas usuais de soma, subtração, multiplicação, divisão e módulo em números com até 680 algarismos decimais.

2. Permitir ao usuário definir a base dos números utilizados(de entrada e saída e de representação interna).

3. Gerar números primos menores que certo limite.

4. Executar algumas operações não muito usuais tais como exponenciação, raiz quadrada, cálculo de logaritmos, mdc, etc.

5. Executar uma série de primitivas importantes na criptografia, tal como a geração das chaves do RSA.

6. Permitir monitoramento da execução de certos comandos, útil para a realização de testes.

7. Fornecer ao usuário a possibilidade de trabalhar com até 26 variáveis.

8. Permitir a entrada e saída de números na forma de cadeia de caracteres usando o código ASCII(ou, visto de outra forma, na base 128).

3.2 Representação de números de precisão múltipla

Uma das características interessantes do protótipo é a simplicidade das estruturas de dados utilizadas e toda complexidade que poderia existir na implementação reside nos algoritmos discutidos no capítulo 2.

A representação de números de precisão múltipla se dá através de vetores e para todo número de precisão múltipla utilizado em alguma parte do programa é alocada uma área de mesmo tamanho, para simplificar as rotinas de gerenciamento de espaço. Isto não causa maiores problemas uma vez que quase todos os números de precisão múltipla utilizados nos algoritmos são de mesmo tamanho.

Uma alternativa a esta solução seria a utilização de listas duplamente ligadas, as quais inclusive permitiriam que se trabalhasse com números

de precisão arbitrariamente grande de uma forma transparente para o usuário. No entanto a necessidade de um grande número de chamadas às rotinas de gerenciamento de memória durante a execução das várias rotinas aritméticas aliada ao aumento da complexidade da implementação de vários algoritmos prejudicou a opção por esta forma de representação de números de precisão múltipla. Como exemplo da utilização desta forma de representação existe o trabalho de Collins[COLL66] para a manipulação de polinômios.

Em troca de um incremento considerável na complexidade das rotinas de gerenciamento de memória é possível trabalhar com números de precisão múltipla arbitrária representados em vetores. Tal possibilidade foi implementada no aplicativo DC existente no sistema UNIX. Entretanto considerando o contexto em que o PRSA foi criado não existe a mínima necessidade da implementação de precisão múltipla arbitrária. De qualquer forma, o PRSA permite um pouco de flexibilidade, embora não transparente ao usuário, para definir o tamanho dos números a serem trabalhados. Brent[BREN78] também implementou um pacote aritmético para precisão múltipla usando a representação por vetores.

Na representação por vetores cada dígito do número de precisão múltipla é armazenado em uma posição. A base b de representação interna utilizada é dependente de máquina. É conveniente que b seja uma potência de dois, de modo que multiplicações, divisões e módulos por b sejam implementadas eficientemente através de operações de deslocamento e/ou lógicas.

Em geral a base b tem um expoente igual a um menos do que a metade do tamanho da palavra do computador utilizado. Por exemplo, em um VAX - 785 a palavra tem 32 bits e o expoente da base b é $15(b = 2^{15})$. A razão da utilização deste critério é evitar a ocorrência de estouros quando se multiplicam dois dígitos, por exemplo durante a multiplicação de dois números de precisão múltipla. A relevância principal desta limitação do tamanho da base é que o programa tem um desempenho efetivo até quatro vezes menor do que se poderia conseguir se fosse utilizada uma linguagem de montagem, ao invés de uma linguagem de alto nível, na sua implementação.

Além dos dígitos dos números inteiros de precisão múltipla armazenados no vetor é também reservada uma posição para armazenar o sinal do número e o seu tamanho, em número de dígitos(palavras).

As únicas estruturas de dados mais sofisticadas existentes no protótipo são duas pilhas, que inclusive compartilham a mesma área de mória. Uma delas é a pilha utilizada para a avaliação das expressões aritméticas e a outra é utilizada pelas rotinas de gerenciamento de memória, o que é feita através de rotinas próprias de PRSA.

Além destas existe uma tabela para armazenamento de alguns primos pequenos, e na implementação em máquinas de menor porte uma tabela usada para multiplicação de números pequenos.

3.3 Organização modular do protótipo

O PRSA consiste de 10 módulos de rotinas e 1 módulo para definição de variáveis, constantes, etc. Na sua última versão, em linguagem C, o programa era formado por cerca de 79 rotinas em 3691 linhas de comandos, comentários e linhas em branco. O código gerado no equipamento da Digirede, o D-8000, contém quase 45 Kbytes.

São os seguintes os nomes dos vários componentes do programa:

tpropri.c: é o módulo principal, e implementa um *driver* para a simulação da calculadora;

tdecl.h: Contém as definições das constantes, macros e variáveis usadas por um ou mais módulos do programa;

tentsai.c: Contém algumas rotinas de entrada e saída de números;

terro.c: Uma rotina para impressão de mensagens de erro;

tinicial.c: Uma rotina de inicialização e reinicialização de certas variáveis do programa;

tmemo.c: Rotinas de gerenciamento de memória e pilha aritmética;

tarit.c: É o núcleo do protótipo e onde estão as principais rotinas aritméticas;

ttempo.c: Algumas rotinas para tratamento de tempo;

taux.c: Algumas rotinas auxiliares usadas na redefinição de alguns parâmetros do protótipo;

trsa.c: Contém a rotina utilizada na geração das chaves do RSA;

tprimos.c: Rotinas utilizadas para a geração de números primos grandes.

O apêndice D contém uma descrição sucinta de cada um destes módulos, bem como uma passada rápida pelas rotinas que os compõem.

3.4 Testes de desempenho

Nesta seção serão apresentados alguns testes realizados para medir o desempenho do PRSA na execução de ciframentos(e portanto deciframentos) e na geração de números primos. Serão apresentados ainda relatos de algumas experiências realizadas para melhorar o nosso conhecimento do funcionamento, na prática, do algoritmo de Rabin.

3.4.1 Ciframentos e deciframentos

A realização de testes de desempenho do PRSA na execução de ciframentos e deciframentos serve para verificar a viabilidade ou não do programa para a operação criptográfica em tempo real. Assim foram feitas experiências de ciframento e deciframento em três máquinas considerando ordens de grandeza diferentes para o módulo de exponenciação.

Os resultados obtidos estão relacionados na tabela 3.1.

módulo	*PC*	VAX	*D-8000*
10^{50}	6,4	132,85	135,50
10^{100}	2,1	48,50	44,33
10^{200}	—	14,95	12,89

Tabela 3.1: Velocidade de ciframento do PRSA

As três colunas da direita mostram a taxa de ciframento do PRSA em bits/seg. O VAX utilizado foi o modelo 785 rodando sob o sistema operacional VMS, o PC utilizado foi produzido pela Itautec e possui um *clock* de 8 Mhz, sistema operacional SIM / DOS e o D - 8000, produzido pela Digirede, funciona com o processador M - 68.010 da Motorola com um *clock* de 12 Mhz; o sistema operacional, neste caso, é o sistema DIGIX.

Os programas que rodaram os testes nos diversos equipamentos foram escritos em C, com pequenas alterações devido a problemas de compatibilidade. No caso do PC foi utilizada multiplicação por tábua.

Um ganho considerável poderá ser conseguido ao se traduzir algumas rotinas que tratam da multiplicação e divisão de números grandes para linguagem de montagem. Nesse caso a base de representação interna poderá ter seu tamanho dobrado provocando até uma quadruplicação da taxa de ciframento.

Na implementação de Rivest, por hardware, do RSA obteve-se uma taxa de ciframento de 1200 bits/segundo com a utilização de módulos de 512 bits(10^{155}) [DENN82,TREL85,RIVE80], o que mostra, por um lado, um desempenho relativamente bom da implementação por software, mas que por outro demonstra as dificuldades inerentes ao próprio método de ciframento do RSA.

3.4.2 Geração de números primos

A geração de números primos, ao contrário das transformações de ciframento e deciframento que exigem taxas compatíveis com o processamento em tempo real, pode ser feita com mais calma, tendo em vista a natureza *off-line* da geração das chaves criptográficas, além do que é uma operação executada raras vezes. Entretanto qualquer melhoria conseguida na taxa de ciframento se reverte à geração de primos, uma vez que tanto uma quanto a outra tem como núcleo principal a operação de exponenciação modular.

A única medida a ser tomada no sentido de diminuir o número de exponenciações durante a geração de números primos é através da utilização de um crivo formado por números primos pequenos que serve para descartar um contingente razoável de candidatos ao teste de Rabin. Como quanto maior este crivo menor o número de testes de primalidade a serem executados e maior o número de testes de divisibilidade; foram feitos testes no sentido de determinar um compromisso entre estes dois fatores. Os resultados dos testes, realizados no D - 8000 estão nas tabelas 3.3 e 3.2.

A primeira coluna das tabelas 3.2 e 3.3 define o limite superior para os primos pequenos utilizados no crivo. Ao lado direito das mesmas estão quatro colunas, cada uma definindo uma ordem de grandeza(em casas decimais) diferente dos primos gerados. Para cada ordem de grandeza e

PRIMOS ATÉ	*ORDENS DE GRANDEZA*			
	10^{10}	10^{20}	10^{40}	10^{80}
2	8,47	58,12	437,42	3661,79
16	7,29	25,65	227,10	1884,44
64	5,37	24,20	143,05	1794,45
256	5,92	23,07	152,22	1082,34
1024	5,70	26,00	99,43	1008,04
4000	9,32	29,37	132,93	917,60
Melhor crivo	64	256	1024	4000

Tabela 3.2: Tempo de geração de primos

PRIMOS ATÉ	*ORDENS DE GRANDEZA*			
	10^{10}	10^{20}	10^{40}	10^{80}
2	89	255	494	788
16	58	62	215	360
64	19	52	104	335
256	24	41	109	165
1024	14	38	38	141
4000	16	31	56	107
Melhor crivo	1024	4000	1024	4000

Tabela 3.3: Candidatos que passaram pelo crivo

cada crivo foram gerados 10 primos, que tiveram de passar por 10 testes de Rabin antes de serem considerados primos(ainda assim seria melhor chamá-los de provavéis primos). A tabela 3.2 contém o tempo gasto, em segundos, para a geração dos primos, enquanto que a tabela 3.3 o contém o número de candidatos que passaram pelo crivo durante a geraç¿ ao dos primos.

A última linha de cada tabela informa, para cada limite, qual o crivo que proporcionou um melhor desempenho do programa na geração dos primos, na tabela 3.2 considerando o tempo de geração e na tabela 3.3 considerando o número de candidatos que passaram pelo crivo de primos pequenos. Os valores desta linha na tabela 3.3 servem apenas para lembrar que o gerador de números aleatórios não foi reinicializado para cada crivo, uma vez que se isto tivesse acontecido teria sido obtido em todas as colunas desta linha o valor 4000.

A tabela 3.2 mostra que para cada limite, mantendo outros fatores constantes, como número de sondagens, etc. , se um aumento do tamanho do crivo propicia, de início, um melhor desempenho na geração de números primos, a partir de um certo valor, dependente de cada limite, o desempenho tende a cair, ainda que, de acordo com a tabela 3.3 o número de candidatos que passam pelo crivo diminua, diminuindo o número de testes de Rabin a serem executados. A obtenção do ponto de ótimo desempenho do programa, de uma forma teórica, parece ser complicado, sendo mais atraente a possibilidade de consegui-lo experimentalmente.

3.4.3 Algumas experiências

Rabin[RABI80,KNUT81] mostra que para qualquer número composto n pelo menos 75% dos números menores que n são testemunhas da não-primalidade de n. Foram feitas duas experiências para verificar até que ponto era pessimista esta probabilidade.

Na primeira experiência foram gerados 1000 números com 10 algarismos que foram passados 100 vezes pelo teste de Rabin. Os resultados obtidos foram bastante extremos: ou um determinado número era primo e passava em todos os 100 testes de primalidade ou, quando era composto, não passava em nenhum, o que mostra que a probabilidade de apenas 75% dos números menores que n serem testemunhas da não-primalidade de n está muito longe do valor médio.

Na segunda experiência foram gerados 100 números com 100 algarismos que passavam por um teste de Rabin, a seguir tais números eram submetidos a 5 outros testes de Rabin e todos eles passaram em todos os testes.

Uma outra experiência teve a finalidade de determinar qual a menor testemunha de não-primalidade para números de Mersenne compostos[2]. Foram testados todos o números de Mersenne compostos com expoente menor que 300 e verificou-se que o número 3 é a menor testemunha de não-primalidade para tais números. Este valor é ótimo uma vez que o número 2 não pode testemunha de não-primalidade de um número de Mersenne composto[POME80].

Tais experiências parecem indicar a força do teste de Rabin e também quão confiável ele é.

3.5 Conclusão

Uma das primeiras decisões tomadas para a realização da implementação do protótipo para o RSA foi a de que este teria a forma de um simulador de calculadora de precisão múltipla. A notação polonesa foi utilizada porque facilitava a análise léxica e sintática das expressões de entrada. A seguir foram definidas a forma de representação dos números de precisão múltipla bem como a estrutura de dados. De início não nos preocupamos com os aspectos mais ligados à criptografia, tais como geração de primos, geração de aleatórios, etc. e pudemos dedicar-nos ao trabalho de implementação das rotinas aritméticas de precisão múltipla.

As primeiras rotinas a implementar foram as de entrada e saída, soma, subtração e multiplicação. Tais operações eram implementadas como procedimentos que trabalhavam diretamente com a pilha aritmética. Quando começou a implementação das operações de divisão e modularização descobriu-se que este esquema era inadequado e foram feitas as mudanças necessárias na estrutura de dados e nas rotinas já implementadas para garantir a transparência das mesmas com relação á estrutura de dados. Passou-se a ter duas pilhas, ao invés de apenas uma, que continham apontadores para uma área onde eram armazenados os números de precisão múltipla. Este esquema é utilizado até hoje.

[2]Os número de Mersenne são da forma $2^p - 1$, onde p é primo.

Após a implementação das rotinas aritméticas básicas foram implementadas as rotinas de exponenciação modular, geração de números aleatórios, máximo divisor comum, teste de primalidade(inicialmente o de Solovay e Strassen) e geração de números primos.

Até aí se trabalhava com um micro de 8 bits e a linguagem era Pascal. Foi testada a utilização de multiplicação por tábua que melhorou bastante o desempenho do programa. Entretanto ainda assim o programa era muito lento e optou-se por transportá-lo para um PC. A utilização de multiplicação por tábua também apresentou resultados positivos no PC, ainda que o processador desta máquina possua instrução de multiplicação por hardware(ou firmware).

Foi no PC que começaram a ser feitos os primeiros testes de desempenho do programa, facilitados pela possibilidade de acesso ao relógio da máquina e de redirecionamento de entrada/saída. Surgiram aí muitos dos comandos para medição de tempo e alteração de parâmetros internos do programa.

A lentidão do PC, por um lado, e a necessidade da execução de testes demorados, por outro, obrigou o transporte do programa para o VAX - 785. No VAX o programa era cerca de 30 vezes mais rápido do que no PC. O passo tomado a seguir foi traduzir o programa para a linguagem C, que com a existência de certos operadores especiais inexistentes no Pascal, permitiu a duplicação da velocidade de execução do programa.

O programa em C foi transportado para o PC e para o D-8000 da Digirede, que para surpresa de todos, teve um desempenho comparável ao do VAX. Além disto a existência de ferramentas mais adequadas ao trabalho de programação do sistema Unix motivou a implementação de um grande número de novas rotinas, como cálculo de raiz quadrada, logaritmo, raiz quadrada módulo um primo, solução de sistemas de congruência com o uso do teorema chinês do resto, etc. É interessante notar também que a relação de desempenho do programa no VAX e no PC caiu para cerca de 25 vezes(veja tabela 3.1).

A moral da história é que no afã de obter uma implementação eficiente do RSA primeiro lutamos contra as limitações inerentes à máquina, levando o programa de um micro de 8 bits até às máquinas maiores.

A seguir enfrentamos a limitações inerentes à linguagem de programação Pascal e convertemos o programa para a linguagem C. Continuando nesta linha o próximo passo seria a conversão de certas rotinas

para linguagem de montagem, mas que diminuiria a transportabilidade do programa.

Uma das possibilidades ainda não testadas é a utilização de algoritmos mais complexos e eficientes para a execução da operação de multiplicação. De acordo com a literatura[BREN78], tais algoritmos são indicados apenas quando se trabalha com números de milhares de casas decimais, entretanto talvez a utilização de estrutura de dados adequadas pudesse melhorar o desempenho destes algoritmos quando trabalhando com números de apenas centenas de casas decimais.

Assim a possibilidade da utilização prática da implementação em software do RSA não é muito atraente, e nem mesmo é prática a utilização da implementação por hardware. Entretanto é possível a utilização do criptossistema em transações que não envolvam um grande volume de mensagens. Uma aplicação típica, que recai neste caso, seria a troca de chaves para criptossistemas de chave secreta, e o RSA faria o papel de um canal lógico seguro, porém lento e oneroso, no sistema de comunicação.

Apêndice A

Demonstração do uso do PRSA

Vamos mostrar a seguir a listagem de um arquivo produzido pelo PRSA. Com isto esperamos que se tenha uma idéia melhor do funcionamento do programa.

```
< <O proposito deste arquivo e mostrar alguma coisa sobre o uso do PRSA.
< <Ele servira de entrada para o programa e por isto toda linha deve
< <comecar com um sinal de menor(<). Ou seja toda linha de comando que
< <comeca com um sinal de menor e tratada como comentario pelo PRSA.
< <Voce pode usar tambem o ponto(.) para inicar comentarios. Mas se usar
< <o sinal de maior(>) o programa terminara.
<
< <O PRSA e um programa que simula uma calculadora com capacidade de
< <trabalhar com numeros inteiros, positivos ou negativos, de precisao
< <multipla. Ele dispoe tambem de primitivas para utilizacao
< <criptografica, uma vez que o objetivo principal do programa e servir
< <como um prototipo para a implementacao do cifrador RSA.
<
< <O PRSA pode ser usado no modo interativo, bastando para isto digitar o
< <nome do programa: prsa, a partir dai o programa inica a execucao
< <recebendo comandos a partir do teclado e imprimindo os resultados na
< <tela. Todos os comandos que sao fornecidos ao PRSA tambem sao ecoados.
< <Se voce nao gosta disto digite o comando oe, e o eco termina.
<
< <O PRSA, integrado a filosofia do sistema UNIX, aceita redirecionamento
< <de entrada e saida. Este teste usou este esquema. Foi digitada a
< <linha de comando:prsa <demo.p >demo.out .
```

```
<
< <A calculadora implementada utiliza a notacao polonesa para
< <representacao de expressoes aritmeticas, a mesma usada pelas
< <calculadoras da HP. Aqui voce coloca primeiros os operandos e depois
< <o operador.
<
< <Para a entrada de numeros negativos voce tem que tomar cuidado pois o
< <sinal de menos e o sublinhado(_) e nao o - que e operador de
< <subtracao.
<
< <Voce pode fazer as operacoes aritmeticas usuais como soma(+),
< <subtracao(-), divisao(/) e multiplicacao(*). Como o programa so
< <trabalha com numeros inteiros o resultado da operacao de divisao e
< <truncado. Se voce quer saber quanto e o resto da divisao use o
< <operador da operacao resto(%).
<
< <Vamos dar um primeiro exemplo. Vamos mostrar como avaliar a expressao
< <(5 + 4)*3 :
<
< 5 4 + 3 * =
> 27.
<
< <ou
<
< 5 4+3*=
> 27.
<
< <Uma das caracteristicas da notacao polonesa reversa(posfixa) em
< <relacao ao modo usual(infixa) foi mostrada acima, que e desprezar
< <o uso de parenteses para determinar a ordem de avaliacao de uma
< <expressao.
<
< <O operador = mostra o resultado da avaliacao da expressao. Se voce
< <esta interessado em saber o resultado parcial na avaliacao de uma
< <expressao use o operador p. O operador = provoca uma impressao
< <destrutiva, enquanto que o operador p nao elimina o resultado
< <parcial da pilha aritmetica.
<
< 5 4 + 3 * p
> 27.
< 5 4 3 * + p + =
> 17.
> 44.
<
```

```
< <Logo apos o programa comecar voce pode trabalhar com numeros com ate
< <668 casas decimais. Mas voce pode mudar isto se quiser trabalhar com
< <numeros maiores. Basta usar o operador de redefinicao de tamanho(t).
< <Aqui voce define quantas palavras do computador um numero pode ocupar.
< <Inicialmente esta definido que este maximo e 300 e voce aumentar este
< <valor ate 1200 durante a execucao do programa.
<
< <A base em que os numeros sao escritos, que por default e 10 tambem
< <pode ser mudada usando o comando para mudanca de base(b). Exemplo:
<
< 2.000 45.567.345 + p
> 45.569.345.
< 16 b
< p
> 2.b75.541.
< 2 b
< p
> 10.101.101.110.101.010.101.000.001.
< 10.10 b      . todos os operandos devem ser fonecidos na base 2!
< =
> 45.569.345.
<
<
< <Agora podemos falar de outros operadores. Temos ainda exponenciacao
< <modular(^), mdc(|), raiz quadrada(rq), logaritmo(lg) e radiciacao
< <de modo geral(rz). A menos de rq que so tem um operando todos tem
< <dois. No caso de logaritmo voce primeiro entra com o numero a ser
< <"logaritmado" e depois com a base. Na radiciacao geral(rz) voce
< <entra inicialmente com o radicando e depois com o radiciador.
< <Exemplos:
<
< 10 20^ =            <exponenciacao modular
> 100.000.000.000.000.000.000.
< 714.234 321.456 | = .mdc
> 6..
< 234.567.879 rq =    .resultado truncado
> 15.315.
< 456.897.234 10 lg = .logaritmo
> 8.
< 231.937.231 3 rz =  .radiciacao
> 614.
<
< <Os resultados das operacoes que nao dao resultado inteiro sao sempre
< <truncados. A exponenciacao modular que sempre daria resultado inteiro
```

```
< <possui uma caracteristica propria: o resultado e sempre menor de que
< <um modulo. Nos casos em que o resultados da exponenciacao sao maiores
< <do que este modulo entao o programa divide-o pelo modulo e fornece
< <como saida o resto desta divisao. Interessante nao, esta operacao e
< <importantissima na implementacao do RSA e na geracao de numeros
< <primos.
< <Exemplos:
<
< 10 20^ =
> 100.000.000.000.000.000.000.
< 10 40^ =
> 10.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
< 10 80^ =
> 100.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.
< 10 160^ =
> 10.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.
< 10 320^ =
> 100.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.
< 10 640^ =
> 10.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.0
00.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.00
0.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.
< 10 1280^ =
> 49.143.595.261.396.964.689.352.745.379.920.764.970.783.785.512.738.
```

```
982.950.271.600.247.233.716.953.071.444.696.629.788.625.870.256.500.5
21.019.400.793.332.668.229.876.568.990.038.747.373.947.645.487.110.14
4.131.683.524.544.961.345.189.766.542.241.147.850.964.923.496.362.629
.416.506.072.845.808.635.210.355.950.652.236.307.274.778.157.950.399.
978.350.057.052.934.923.525.144.386.848.129.720.584.292.548.148.340.4
46.945.935.338.307.836.564.421.301.758.368.170.480.366.847.966.485.57
7.130.198.722.777.529.492.163.491.408.987.032.380.510.349.224.663.334
.908.214.022.362.197.000.093.019.156.714.089.303.634.876.814.650.525.
254.929.300.646.053.567.026.173.519.169.870.433.496.071.169.034.530.9
21.110.223.955.754.229.140.905.275.277.291.414.673.140.935.505.395.97
5.165.534.858.695.685.567.996.488.822.737.901.077.578.099.498.363.758
.009.150.250.795.354.247.842.350.106.085.905.035.157.760.376.832.
<
< <Para saber quanto vale o modulo use o comando maximo(mx).
<
< mx =
> 193.459.639.446.953.031.773.047.832.279.434.098.941.895.133.801.109
.228.768.168.668.886.279.166.083.475.564.052.795.534.481.737.552.937.
975.612.379.168.306.689.118.209.944.945.984.692.789.505.342.220.569.0
89.212.017.698.552.339.517.525.510.860.831.338.034.136.262.500.027.63
3.684.921.055.117.231.813.142.963.630.715.939.627.799.180.859.640.727
.652.297.538.303.667.266.768.025.447.405.385.543.026.356.287.783.691.
276.888.675.945.009.040.586.500.628.325.721.011.602.904.696.831.786.6
37.292.174.325.269.979.511.943.129.943.196.689.830.864.080.846.540.04
4.620.848.776.800.830.394.262.070.712.967.301.411.910.365.827.674.560
.604.328.220.729.891.631.617.831.601.699.590.002.504.485.370.256.021.
879.797.687.718.616.561.755.423.013.669.365.689.554.419.757.872.127.4
85.237.032.338.582.742.658.270.819.503.018.541.259.508.485.011.536.46
1.157.369.582.778.293.926.714.572.854.984.400.790.850.241.560.576.
<
< <O modulo de exponenciacao pode ser alterado usando-se o comando para
< <mudanca de modulo(m), que possui um operando. Toda vez que se usa o
< <comando de mudanca de tamanho(t) o modulo e aumentado simultaneamente
< <e seu valor e o maximo para o tamanho utilizado.
<
< <Para a avaliacao das expressoes aritmeticas o PRSA usa uma pilha onde
< <ficam armazenados resultados parciais. O comando p imprime o ultimo
< <resultado obtido assim como o comando =, a diferenca e que o comando =
< <tambem elimina este resultado do topo da pilha o que nao acontece com
< <o comando p.
<
< <Existem tres outros comandos para se trabalhar com a pilha. Sao eles:
< <duplica topo da pilha(d), troca ordem dos dois ultimos elementos do
< <topo da pilha(r) e desce topo da pilha(dt).
```

```
<
< <Uma das facilidades existentes e a disponibilidades de se usar
< <variaveis para armazenar resultados parciais. Entao voce pode
< <armazenar coisas nas variaveis usando o comando !x, onde o x indica
< <uma letra entre a e z, e tambem voce pode carregar o conteudo de
< <variaveis na pilha usando o comando ?x, onde x e o nome da variavel.
<
< <Se voce quer repetir um numero de vezes predeterminado certa sequencia
< <de comandos e so coloca-los entre parenteses, colchetes ou chaves
< <especificando a seguir o numero de vezes que a sequencia devera ser
< <executada, que pode variar de 1 a 16 vezes apenas. Ate 10 vezes voce
< <usa os digitos de 0 a 9 e de 11 a 16 voce deve usar as letras a ate f.
< <Exemplos:
<
< (5 5 + =)3
> 10.
> 10.
> 10.
> 10.
< 5 (5 + p)3 =
> 10.
> 15.
> 20.
> 25.
> 25.
< [(5 5 + =)3]a
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
```

```
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
> 10.
< 1([{0}4]4)4 =
> 100.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000
.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000.
000.000.000.000.000.000.000.000.
<
< <O PRSA permite a voce saber quanto tempo de cpu e gasto para a
< <realizacao de alguns calculos. Para isto voce usa o comando de zerar
< <tempo(zt) e o comando imprime tempo(it).
< <Exemplo:
<
< zt 5 {[(5 *)f]f}f = it
> 57.541.489.569.163.337.761.886.944.422.618.727.783.960.049.211.527.
461.249.567.364.168.788.736.896.015.945.161.408.158.398.499.434.947.6
20.523.633.151.993.370.183.810.275.551.417.776.246.518.903.580.251.01
8.291.045.011.573.625.193.619.567.688.186.031.824.293.267.974.058.002
.507.133.267.997.020.069.248.892.457.250.149.824.549.291.662.942.078.
821.830.800.868.509.406.354.753.111.486.917.048.717.371.074.865.083.2
23.336.991.120.443.975.189.672.286.012.620.264.952.906.732.362.488.49
```

```
0.146.298.831.615.873.141.591.434.320.735.760.120.463.019.030.524.807
.172.314.610.844.332.528.539.156.494.662.943.270.754.427.775.827.869.
782.959.000.971.042.769.098.480.123.382.995.245.824.647.436.400.997.2
61.567.903.523.480.122.881.753.832.335.532.080.195.114.923.033.187.58
6.045.448.160.939.180.364.795.614.557.200.152.222.222.268.568.601.374
.362.237.571.216.096.761.128.006.578.645.554.850.934.320.537.583.841.
746.519.663.267.071.440.346.302.403.942.430.447.564.594.330.196.605.0
77.259.898.672.201.730.542.573.292.125.778.081.301.087.912.504.033.07
1.837.368.815.800.036.647.245.180.324.555.787.899.633.363.832.890.974
.340.950.475.383.418.936.592.469.452.612.242.583.250.337.974.182.690.
618.617.100.793.419.263.286.772.950.868.385.707.553.180.143.292.797.9
39.921.166.429.747.569.378.648.235.412.504.667.682.873.901.969.879.52
4.598.276.131.870.820.061.692.655.647.969.504.995.695.513.963.451.675
.195.285.428.523.282.672.600.644.145.003.327.914.504.513.329.603.012.
276.127.660.170.309.936.099.000.447.492.970.371.711.710.530.311.894.2
89.853.484.757.763.202.058.389.854.657.559.647.851.803.809.139.094.63
2.083.976.051.177.775.626.001.559.419.128.346.969.401.961.033.045.414
.328.971.879.669.194.500.440.240.492.254.244.292.411.687.095.218.527.
013.007.370.310.022.861.782.146.021.648.573.929.077.852.320.965.896.3
35.441.925.
00:00:13.34
<
< <Voce pode tambem gerar sequencias de numeros aleatorios. Atraves do
< <comando de inicializacao do gerador de numeros aleatorios(ir) voce
< <determina o intervalo de valores em que a sequencia deve estar. Este
< <comando tem um operando, digamos n, e isto especifica que a sequencia
< <de valores a serem gerados esta no intervalo [0,n). Para gerar os
< <numeros voce deve usar o comando a. Exemplos:
<
< 10 100^ ir
< (a =)f
> 7.722.129.127.202.633.928.944.066.790.924.063.812.754.397.595.603.3
48.142.618.307.395.617.798.436.273.456.540.699.492.772.974.038.365.
> 4.314.960.392.827.087.908.929.393.932.157.858.433.104.614.898.938.9
16.672.565.907.333.777.158.160.149.796.381.257.874.698.648.969.235.
> 5.937.249.058.832.297.208.668.747.164.449.622.697.400.889.815.091.7
58.891.751.940.997.458.685.203.893.559.844.497.018.915.955.942.226.
> 5.802.430.202.162.905.364.135.729.495.785.444.232.007.851.756.700.7
83.391.994.650.937.603.200.547.185.407.793.382.729.733.037.240.367.
> 7.840.524.946.171.182.474.793.057.073.176.664.274.850.006.542.470.6
53.872.093.350.691.315.256.110.573.803.448.135.641.174.085.810.176.
> 3.411.314.014.688.630.310.050.092.011.204.384.521.559.485.062.744.1
28.354.814.711.879.661.617.560.454.096.455.163.763.308.763.647.115.
> 9.733.681.909.566.973.247.478.288.536.189.295.261.326.629.892.965.0
```

```
64.253.584.381.322.445.691.207.499.386.025.409.402.593.666.403.729.
> 7.050.352.507.947.609.732.453.791.473.596.891.581.785.856.190.967.0
17.710.948.487.119.679.619.628.310.634.245.470.041.950.719.735.274.
> 173.417.288.654.909.843.974.562.714.038.905.262.927.859.708.115.084
.709.480.267.607.585.260.576.972.505.808.280.253.271.326.421.208.
> 9.153.567.075.824.867.643.846.439.955.652.604.476.595.670.163.258.1
93.557.738.721.828.673.367.625.844.860.561.238.466.069.115.335.702.
> 2.667.212.350.579.196.321.960.791.251.679.664.308.737.779.481.937.5
80.101.729.916.280.506.681.552.155.499.792.633.212.527.564.724.739.
> 6.713.360.447.050.656.228.001.461.218.480.231.549.899.269.683.122.7
23.439.943.854.506.625.872.992.169.152.658.837.063.457.041.315.620.
> 8.619.743.618.978.039.374.724.395.687.781.345.781.848.360.689.724.9
65.569.843.373.648.635.869.713.943.881.058.897.299.742.343.865.929.
> 4.616.506.731.064.688.595.082.519.795.544.276.233.717.966.236.474.1
36.451.772.231.223.029.450.319.639.366.759.760.373.371.686.186.501.
> 3.944.234.081.040.900.915.489.315.155.633.035.441.380.912.831.185.7
41.722.555.751.223.452.116.369.214.191.179.912.582.918.719.460.220.
> 1.467.086.681.807.704.416.381.633.984.049.975.991.778.599.679.724.6
45.027.405.312.353.274.449.600.474.466.200.778.575.008.550.361.089.
<
< <Vamos colocar agora um trecho de exemplo mais completo. Vamos mandar o
< <PRSA calcular dez exponenciacoes modulares e ver quanto tempo ele
< <gasta para fazer isto:
<
< mx !m         <Salva modulo inicial na variavel m
< 10 100^ ir    <inicializa gerador de aleatorios
< zt (a a a m ^ dt)9 it < zerou tempo
00:01:17.10
<    <mandou executar dez vezes os comandos:
<    <          gera tres aleatorios
<    <          define um deles para ser o modulo de
<    <                exponenciacao
<    <          calcula a exponenciacao modular
<    <          elimina elemento do topo
<    <mandou imprimir tempo gasto
< ?m m          <recupera modulo inicial
<
< <Falta ainda falarmos dos comandos mais diretamente ligados a
< <criptografia.  O primeiro e para testar se um numero dado e primo ou
< <nao(p?). A seguir temos um comando para gerar primos grandes(pr).
< <Existem comandos para gerar as chaves do RSA(rs), calcular raiz
< <quadrada modulo primo(rp), achar a solucao da congruencia
< <ax mod n = b(rc) e tambem do sistema de congruencias: x = b1 mod n1
< <e x = b2 mod n2(:).
```

```
<
< <Antes de mostrar exemplos deste comandos e bom citar que o PRSA
< <trabalha tambem com strings que sao convertidos internamente para
< <numeros e portanto podem ser somados, subtraidos etc. Exemplo:
<
< "A Maria e bonita" p
> 2.647.046.989.647.517.060.408.388.022.336.097.
< 1 +
< ps                <imprime na forma de string
> "A Maria e bonitb"
< dt
<
< <Vamos a um exemplo que mostra as potencialidades criptograficas do
< <PRSA:
<
<
< 10 16^ 63- p?        < Este numero e primo (Knuth Vol 2)
> sim
<
< (10 20^ pr =)f       < Gera dezesseis numeros primos
ncup=1 neup=2
> 57.219.155.370.195.301.949.
ncup=5 neup=6
> 45.558.481.990.443.019.079.
ncup=2 neup=3
> 77.562.578.609.859.726.307.
ncup=7 neup=8
> 54.832.073.186.643.036.239.
ncup=3 neup=4
> 97.314.026.324.032.295.339.
ncup=13 neup=14
> 63.575.935.778.395.227.307.
ncup=14 neup=15
> 87.148.483.582.467.040.229.
ncup=12 neup=13
> 86.470.739.222.701.233.823.
ncup=1 neup=2
> 59.033.640.558.807.310.369.
ncup=1 neup=2
> 29.817.916.711.987.866.359.
ncup=2 neup=3
> 46.158.816.386.387.406.473.
ncup=1 neup=2
> 73.155.213.533.684.501.
```

```
ncup=2 neup=3
> 57.871.472.337.897.486.083.
ncup=1 neup=2
> 19.155.246.530.982.648.949.
ncup=4 neup=5
> 69.834.197.015.106.256.693.
ncup=2 neup=3
> 14.153.484.177.027.629.233.
<
< 10 40^ rs            < Gera chaves do RSA ( o modulo tera + ou - 80
<                      <                      casas)
ncup=3 neup=4
ncup=6 neup=7
< !e                   < armazena chave de ciframento na variavel e
< !d                   < armazena chave de deciframento na variavel d
< !n                   < armazena modulo na variavel n
<
< mx !m
<
< ?n m
< "A Maria e bonita" ?e ^     <Cifra a mensagem
< p
> 16.779.208.621.924.937.130.242.262.846.654.684.390.055.962.908.050.
934.853.350.881.402.347.930.513.761.488.
< ?d ^ ps =                   < Decifra a mensagem
> "A Maria e bonita"
> 2.647.046.989.647.517.060.408.388.022.336.097.
<
< <Calcula raiz quadrada modulo um primo:
<
< (64 257 rp =)5
> 8.
> 8.
> 249.
> 249.
> 249.
> 249.
<
< <Resolve congruencia ax mod n = b:
<
< 1000 31.991 1 rc  p
> 28.440.
<
< 1000 * 31.991 % p
```

```
> 1.
< 1 i?
> sim
<
< <Resolve congruencias x = b1 mod n1, x = b2 mod n2:
<
< 93 101 8 257:
< p              <imprime n1 * n2
> 25.957.
< r p            <imprime x
> 19.283.
< d 101 % =
> 93.
< 257 % =
> 8.
<
<
<
< <Existem alguns outros comandos, para testar se um numero e igual a
< <outro(i?) (usado acima), testar se um numero e maior que outro(m?)
< <sendo que e impresso 'nao' caso seja falso e 'sim' caso seja
< <verdade o teste.
<
< <O comando de definicao de numero de sondagens(s) determina quantas
< <vezes o teste de primalidade de Rabin e executado na geracao de
< <primos. Alem deste existem os comandos de zerar contador de candidatos
< <a primos gerados(zc) e para impressao deste numero(ic), zerar contador
< <de testes de Rabin executados(ze) e imprimi-lo(ie), definir numero de
< <bits da base interna de representacao dos numeros grandes(n).
<
< >
```

Apêndice B

Cara ou coroa por telefone

Neste capítulo serão mostradas algumas simulações do protocolo criado por Blum[BLUM81,DENN82] que permite jogar cara ou coroa por telefone.

O protocolo implementa um esquema em que Bob pode escolher entre CARA ou COROA e Alice joga a moeda de forma que cada um tenha 50% de chance de ganhar.

Foram executadas três simulações dadas nas seções a seguir.

B.1 Bob ganha

```
< . Cara ou coroa por telefone
< .
< . Sera testado o protocolo para jogar cara ou coroa por
< . telefone.
<
< . Biblio: Denn82.
<
< . Alice gera dois primos p e q e envia a Bob n := pq.
<
< .     Gera o primo p
<
<  10 50^ pr !p
ncup=10 neup=12
<
< .     Gera o primo q
```

```
<
<  10 50^ pr !q
ncup=1 neup=3
<
< .     Calcula n := pq e envia a Bob
<
<  ?p ?q * p !n
> 1.215.644.176.583.424.302.985.488.380.328.594.301.451.203.751.448.5
01.433.558.054.726.790.874.609.909.258.060.194.803.862.712.728.019.
<
< . Bob recebe n e gera ao acaso x em [0,n-1]
<
<  ?n ir a !x
<
< . Bob calcula y = sqr(x) mod n o envia a Alice
<
<  ?x ?x * ?n % p !y
> 868.867.292.952.642.704.136.720.436.791.672.324.793.256.063.903.479
.249.380.078.006.390.108.239.896.278.648.573.371.587.768.568.088.
<
< . Alice calcula uma das quatro raizes quadradas de y modulo n
<
< .       Calcula uma raiz quadrada de y modulo p
<
<  ?y ?p rp !w
<
< .       Calcula uma raiz quadrada de y modulo q
<
<  ?y ?q rp !z
<
< .       Finalmente, calcula a raiz quadrada modulo n, usando
< .       o teorema chines do resto
<
<  ?w ?p ?z ?q :
<
< .       Elimina n do topo da pilha e envia a raiz calculada de
< .       y a Bob
<
<  dt !t
<
< . Bob calcula o mdc de x+t e n.
< . Se o resultado for diferente de 1 e n Bob consegue fatorar n e ganha.
< . Caso contrario Alice ganha.
<
```

```
<  ?x ?t + ?n | p
> 70.557.038.001.338.417.503.018.382.395.958.682.179.896.382.869.541.

<  ?n p
> 1.215.644.176.583.424.302.985.488.380.328.594.301.451.203.751.448.5
01.433.558.054.726.790.874.609.909.258.060.194.803.862.712.728.019.
<
<  . Alice envia a Bob os valores p e q, para mostrar que ela
<  . nao trapaceou (Obviamente se ela tiver perdido isto nao e
<  . necessario).
<
<  ?p p
> 70.557.038.001.338.417.503.018.382.395.958.682.179.896.382.869.541.
<  ?q p
> 17.229.240.498.451.258.483.082.127.921.264.921.272.548.322.731.159.
< >
```

B.2 Alice ganha

```
<  . Cara ou coroa por telefone
<  .
<  . Sera testado o protocolo para jogar cara ou coroa por
<  . telefone.
<
<  . Biblio: Denn82.
<
<  . Alice gera dois primos p e q e envia a Bob n := pq.
<
<  .     Gera o primo p
<
<  10 50^ pr !p
ncup=1 neup=3
<
<  .  .  Gera o primo q
<
<  10 50^ pr !q
ncup=9 neup=11
<
<  .     Calcula n := pq e envia a Bob
<
<  ?p ?q * p !n
> 442.139.861.247.569.673.636.941.442.475.351.896.942.222.793.211.136
.545.392.830.134.507.004.801.425.501.805.490.267.259.676.094.081.
```

```
<
< . Bob recebe n e gera ao acaso x em [0,n-1]
<
<  ?n ir a !x
<
< . Bob calcula y = sqr(x) mod n o envia a Alice
<
<  ?x ?x * ?n % p !y
> 105.015.172.499.717.413.396.505.694.956.707.863.776.048.921.653.596
.848.634.531.528.504.858.362.593.253.252.036.951.226.157.790.098.
<
< . Alice calcula uma das quatro raizes quadradas de y modulo n
<
< .       Calcula uma raiz quadrada de y modulo p
<
<  ?y ?p rp !w
<
< .       Calcula uma raiz quadrada de y modulo q
<
<  ?y ?q rp !z
<
< .       Finalmente, calcula a raiz quadrada modulo n, usando
< .       o teorema chines do resto
<
<  ?w ?p ?z ?q :
<
< .       Elimina n do topo da pilha e envia a raiz calculada de
< .       y a Bob
<
<  dt !t
<
< . Bob calcula o mdc de x+t e n.
< . Se o resultado for diferente de 1 e n Bob consegue fatorar n e ganha.
< . Caso contrario Alice ganha.
<
<  ?x ?t + ?n | p
> 442.139.861.247.569.673.636.941.442.475.351.896.942.222.793.211.136
.545.392.830.134.507.004.801.425.501.805.490.267.259.676.094.081.
<  ?n p
> 442.139.861.247.569.673.636.941.442.475.351.896.942.222.793.211.136
.545.392.830.134.507.004.801.425.501.805.490.267.259.676.094.081.
<
< . Alice envia a Bob os valores p e q, para mostrar que ela
< . nao trapaceou (Obviamente se ela tiver perdido isto nao e
```

```
< . necessario).
<
<  ?p p
> 17.321.945.253.456.288.327.983.635.414.916.881.529.431.819.841.159.
<  ?q p
> 25.524.838.854.882.564.129.403.864.364.561.217.678.119.921.202.359.
< >
```

B.3 Bob ganha novamente

```
< . Cara ou coroa por telefone
< .
< . Sera testado o protocolo para jogar cara ou coroa por
< . telefone.
<
< . Biblio: Denn82.
<
< . Alice gera dois primos p e q e envia a Bob n := pq.
<
< .     Gera o primo p
<
<  10 50^ pr !p
ncup=5 neup=7
<
< .     Gera o primo q
<
<  10 50^ pr !q
ncup=10 neup=12
<
< .     Calcula n := pq e envia a Bob
<
<  ?p ?q * p !n
> 105.268.941.388.822.434.576.413.353.100.535.932.419.946.168.995.406
.049.149.455.747.331.038.269.056.736.370.512.617.092.005.707.003.
<
< . Bob recebe n e gera ao acaso x em [0,n-1]
<
<  ?n ir a !x
<
< . Bob calcula y = sqr(x) mod n o envia a Alice
<
<  ?x ?x * ?n % p !y
> 71.247.750.894.868.729.045.249.677.252.475.868.059.336.020.084.999.
```

```
781.581.440.510.623.890.955.637.763.681.819.713.256.054.283.332.
<
< . Alice calcula uma das quatro raizes quadradas de y modulo n
<
< .       Calcula uma raiz quadrada de y modulo p
<
<  ?y ?p rp !w
<
< .       Calcula uma raiz quadrada de y modulo q
<
<  ?y ?q rp !z
<
< .       Finalmente, calcula a raiz quadrada modulo n, usando
< .       o teorema chines do resto
<
<  ?w ?p ?z ?q :
<
< .       Elimina n do topo da pilha e envia a raiz calculada de
< .       y a Bob
<
<  dt !t
<
< . Bob calcula o mdc de x+t e n.
< . Se o resultado for diferente de 1 e n Bob consegue fatorar n e ganha.
< . Caso contrario Alice ganha.
<
<  ?x ?t + ?n | p
> 3.760.382.107.982.460.740.707.771.962.979.877.414.018.931.592.357.
<  ?n p
> 105.268.941.388.822.434.576.413.353.100.535.932.419.946.168.995.406
.049.149.455.747.331.038.269.056.736.370.512.617.092.005.707.003.
<
< . Alice envia a Bob os valores p e q, para mostrar que ela
< . nao trapaceou (Obviamente se ela tiver perdido isto nao e
< . necessario).
<
<  ?p p
> 3.760.382.107.982.460.740.707.771.962.979.877.414.018.931.592.357.
<  ?q p
> 27.994.213.983.031.064.978.714.618.103.492.442.476.152.905.224.479.
< >
```

Apêndice C

Uma tabela de números primos

As tabelas C.1, C.2, C.3 e C.4 mostram os primos mais próximos de potências de 10, com o expoente variando de 1 a 104. Tais primos podem ser muito úteis para teste de rotinas de teste de primalidade, inclusive alguns deles podem ser encontrados no trabalho de Knuth[KNUT81] e foram utilizados no teste do algoritmo implementado.

A leitura da tabela deve ser feita da seguinte forma: Na coluna do meio de cada linha estará a potência correspondente à linha, na coluna da esquerda a distância, com o sinal negativo do maior primo menor que a potência de 10 da linha e, finalmente, na coluna da direita a distância do menor primo maior que a potência de 10 respectiva.

Maior primo	*Potência de 10*	*Menor primo*
-3	10^1	1
-3	10^2	1
-3	10^3	9
-27	10^4	7
-9	10^5	3
-17	10^6	3
-9	10^7	19
-11	10^8	7
-63	10^9	7
-33	10^{10}	19
-23	10^{11}	3
-11	10^{12}	39
-29	10^{13}	37
-27	10^{14}	31
-11	10^{15}	37
-63	10^{16}	61
-3	10^{17}	3
-11	10^{18}	3
-39	10^{19}	51
-11	10^{20}	39
-101	10^{21}	117
-27	10^{22}	9
-23	10^{23}	117
-257	10^{24}	7
-123	10^{25}	13

Tabela C.1: Os números primos mais próximos de potências de 10 (10 até 10^{25})

Maior primo	*Potência de 10*	*Menor primo*
-141	10^{26}	67
-99	10^{27}	103
-209	10^{28}	331
-27	10^{29}	319
-11	10^{30}	57
-27	10^{31}	33
-21	10^{32}	49
-9	10^{33}	61
-411	10^{34}	193
-23	10^{35}	69
-159	10^{36}	67
-81	10^{37}	43
-59	10^{38}	133
-57	10^{39}	3
-17	10^{40}	121
-119	10^{41}	109
-83	10^{42}	63
-81	10^{43}	57
-53	10^{44}	31
-9	10^{45}	9
-33	10^{46}	121
-41	10^{47}	33
-33	10^{48}	193
-57	10^{49}	9
-57	10^{50}	151

Tabela C.2: Os números primos mais próximos de potências de 10 (10^{26} até 10^{50})

Maior primo	*Potência de 10*	*Menor primo*
-323	10^{51}	121
-231	10^{52}	327
-177	10^{53}	171
-291	10^{54}	31
-111	10^{55}	21
-593	10^{56}	3
-93	10^{57}	279
-149	10^{58}	159
-141	10^{59}	19
-161	10^{60}	7
-39	10^{61}	93
-83	10^{62}	447
-123	10^{63}	121
-51	10^{64}	57
-269	10^{65}	49
-621	10^{66}	49
-111	10^{67}	49
-393	10^{68}	99
-189	10^{69}	9
-93	10^{70}	33
-53	10^{71}	273
-117	10^{72}	39
-149	10^{73}	79
-47	10^{74}	207
-191	10^{75}	129

Tabela C.3: Os números primos mais próximos de potências de 10 (10^{51} até 10^{75})

Maior primo	*Potência de 10*	*Menor primo*
-363	10^{76}	133
-543	10^{77}	21
-63	10^{78}	93
-251	10^{79}	49
-11	10^{80}	129
-173	10^{81}	13
-363	10^{82}	391
-53	10^{83}	27
-399	10^{84}	261
-27	10^{85}	103
-17	10^{86}	151
-273	10^{87}	373
-41	10^{88}	181
-221	10^{89}	31
-143	10^{90}	289
-81	10^{91}	79
-783	10^{92}	399
-173	10^{93}	153
-567	10^{94}	97
-53	10^{95}	151
-269	10^{96}	127
-159	10^{97}	469
-71	10^{98}	49
-621	10^{99}	289
-797	10^{100}	267
-203	10^{101}	3
-39	10^{102}	117
-77	10^{103}	129
-449	10^{104}	267

Tabela C.4: Os números primos mais próximos de potências de 10 (10^{76} até 10^{104})

Apêndice D

Alguns Primos Seguros

O conceito de primo seguro foi definido no capítulo 1 e diz que se p é um primo seguro então $\frac{p-1}{2}$ também deve ser primo.

A tabela D.1 mostra os primos seguros mais próximos de algumas potências de 10. Para cada linha a coluna do meio contém o expoente da potência de 10 correspondente àquela linha, na coluna da esquerda estará a distância do maior primo seguro menor que a potência de 10 e finalmente na coluna da direita estará a distância do menor primo seguro maior que a potência de 10. As distâncias dos maiores primos seguros às respectivas potências de 10, contidas na coluna da esquerda, estão com o sinal negativo à frente.

Maior Primo Seguro	*Potência de 10*	*Menor Primo Seguro*
-1.217	10^{10}	259
-5.297	10^{20}	763
-2.657	10^{30}	1.783
-10.697	10^{40}	17.407
-2.093	10^{50}	4.483
-7.817	10^{60}	15.919
-47.033	10^{70}	5.707
-52.433	10^{80}	47.383
-60.461	10^{90}	36.307
-166.517	10^{100}	43.723

Tabela D.1: Os primos seguros mais próximos de algumas potências de 10

Apêndice E

Descrição sucinta dos módulos do PRSA

Neste capítulo serão descritos os módulos componentes do PRSA, através de uma passada pelas rotinas que compoem cada módulo.

E.1 Entrada e saída

As rotinas que tratam mais diretamente da entrada e saída pertencem ao módulo **Tentsai.c**, que é composto de 8 rotinas que estão listadas a seguir:

leia() Duas variáveis globais estão associadas a esta rotina e formam uma janela de dois caracteres sobre o dispositivo de entrada. A cada chamada a rotina atualiza estas variáveis. O dispositivo de entrada na verdade é lido linha a linha, que são armazenadas em um *buffer*.

le-numero(an) Esta rotina é chamada para ler um número de precisão múltipla que será armazenado na variável an.

imprime-numero(an) Rotina utilizada para a impressão de número de precisão múltipla. O número de precisão múltipla armazenado na variável an é convertido em uma cadeia de caracteres que é armazenada em um *buffer* que comporta até 3500 algarismos. A seguir a cadeia é impressa com um ponto decimal a cada grupo de 3 algarismos.

le-string(an) Lê uma cadeia de caracteres ASCII convertendo-a em um número de múltipla precisão que será armazenado na variável an.

escreva-string(an) Imprime o número de precisão múltipla armazenado em an como uma cadeia de caracteres ASCII.

lista-digitos(nan,an) Imprime a cadeia contida na variável nan e lista os dígitos do número de múltipla precisão contidos em an. É uma rotina utilizada para testes de funcionamento do programa.

tipo-parenteses(parenteses) A variável parenteses contém um dos caracteres (,), [,], { e } que são classificados em três categorias. A rotina devolve a categoria a qual o caracter pertence. É chamada pela rotina leia.

isgraph(ch) Esta é uma rotina que verifica se o caracter ASCII presente na variável ch pode ser impresso ou não. É chamada por escreva-string. Caracteres ASCII não gráficos serão impressos usando as convenções adotadas na linguagem C[1].

Ao contrário de um compilador, onde têm um peso importante no desempenho global do programa, as rotinas de entradas e saída afetam

---

1

- *fim de linha(NL,LF)*: \ n
- *tabulação horizontal(HT)*: \ t
- *tabulação vertical(VT)*: \ v
- *retrocesso(BS)*: \ b
- *volta do carro(CR)*: \ r
- *alimentação de folha(FF)*: \ f
- *barra invertida(\)*: \ \
- *quota(')*: \ '
- *nada*: \ espaço
- \ *ddd*

A seqüência \ *ddd* consiste de um *backslash* seguida de até três dígitos decimais, que especifica a posição do carácter no conjunto ASCII

muito pouco o tempo gasto pelo protótipo. Ainda assim a rotina imprime-numero mereceu cuidados especiais no sentido de minimizar o número de divisões necessárias para converter um número de precisão múltipla em uma cadeia de caracteres. A idéia utilizada é dividir tal número por uma potência da base de entrada e saída menor que a base de representação interna dos números de precisão múltipla e o resultado obtido será por sua vez dividido pela base de entrada e saída.

E.2 Tratamento de erros

Uma série de erros podem ser detectados durante a execução do programa. Além de tentar se recuperar destes erros o programa imprime uma mensagem concisa a seu respeito. Entretanto muito ainda deve ser feito para melhorar a robustez do programa.

Os erros mais comuns e detectáveis se devem à falta de espaço ou à falta de operandos na pilha para a execução de alguma operação. Erros devidos a estouros na execução de alguma operação não são detectados pelo programa.

Para evitar a impressão de uma listagem muito grande de erros PRSA automaticamente finaliza sua execução após a ocorrência de 50 erros. Quando for possível PRSA é reinicializado após a ocorrência de 10 erros.

O módulo que cuida da impressão das mensagens de erro e atualização das variáveis que determinam o *status* do PRSA neste sentido é o **terro.c** que possui apenas uma rotina: erro. Esta rotina é que é responsável por terminar a execução do programa.

E.3 Gerenciamento das pilhas

O módulo que trata do gerenciamento do espaço disponível e da pilha aritmética é o **tmemo.c**. Ele é composto de quatro rotinas:

forneca(an) Fornece um vetor para o eventual armazenamento de um número de precisão múltipla. Quando não existe espaço disponível relata o erro mas ainda devolve um vetor padrão contendo o número de precisão múltipla 0, para tentar evitar a ocorrência de outros erros.

libera(an) Recebe um vetor livre e o coloca na pilha de espaço disponível. Isto só é feito quando o vetor liberado não é o padrão. Esta rotina informa também quando se tenta liberar um vetor que eventualmente já tenha sido liberado, este seria um erro do próprio programa[2] e não do usuário.

push(an) Empilha um número grande na pilha aritmética.

pop(an) Desempilha um elemento da pilha aritmética se ela não está vazia. Da mesma forma que a rotina forneca, quando a pilha está vazia uma mensagem de erro é gerada e é devolvido um vetor padrão contendo o número de precisão múltipla 0.

As pilhas de espaço disponível e aritmética são implementadas utilizando uma mesma área e o topo de uma cresce contra o topo da outra. Isto é possível porque o tamanho máximo destas pilhas é um parâmetro definido em tempo de compilação. Devido a adoção desta política o número máximo de números de precisão múltipla em operação pelo protótipo em um dado instante é limitado. Esta limitação, digamos, vertical, definida em tempo de compilação pode ser deixada para ser determinada em tempo de execução sem muita dificuldade, mas isto implicaria numa mudança grande nas rotinas de gerenciamento das pilhas. Como esta não é uma limitação importante ela foi relegada a um segundo plano.

E.4 Inicialização

O módulo que cuida da inicialização das variáveis globais do PRSA é o **tinicial.c** com uma única rotina: finicializa.

A rotina **finicializa** é a primeira rotina a ser chamada quando começa a execução do programa e também quando o programa deve ser reinicializado. Isto pode ser feito tanto automaticamente pelo programa como pela ação do próprio usuário.

Na primeira vez em que é chamada, a rotina gera a tabela de primos pequenos, que não é feito durante as outras chamadas. As principais variáveis que são inicializadas aqui são as pilhas, os parâmetros de um gerador de números pequenos, o módulo da operação de exponenciação

[2] ou melhor, do programador

modular, os parâmetros do gerador de números de precisão múltipla e uma tabela associada as variáveis que o usuário poderá operar.

E.5 Tempo

Existe um módulo no PRSA que só tem rotinas para trabalhar com tempo: é o **ttempo.c**. Este módulo é composto das seguintes rotinas:

timer(t) A variável t é um registro com campos para armazenar hora, minuto e segundo. Esta rotina é utilizada para inicializar a variável t com o valor do tempo de cpu que o programa gastou desde o início de execução.

zera-tempo(t) Faz os campos de horas, minutos e segundos da variável t igual a zero.

soma-tempo(ta,tb,tc) Soma as variáveis ta e tb, que contém informações relativas a tempo, armazenando o resultado na variável tc.

subt-tempo(ta,tb,tc) Assim como a rotina soma-tempo, subt-tempo trabalha com variáveis que contém informações relativas a tempo. Em particular subt-tempo calcula a diferença entre as variáveis ta e tb, armazenando o resultado na variável tc.

tempo-imprime(t) Imprime o conteúdo da variável t.

O módulo **ttempo.c** é muito útil quando se quer medir o tempo de cálculo de certas rotinas e quando se quer comparar dois algoritmos.

E.6 Rotinas aritméticas

O módulo que implementa as rotinas aritméticas do PRSA é o **tarit.c**. Atualmente este é o maior módulo do programa, contendo cerca de 38 rotinas em 1340 linhas de texto.

As rotinas existentes no **tarit.c** podem ser agrupadas em três categorias:

1. Rotinas envolvidas apenas números pequenos: Nesta categoria estão as rotinas que trabalham apenas com números pequenos, em geral menores do que a base b de representação interna e positivos.

2. Rotinas mistas: São rotinas que trabalham com números inteiros pequenos e com números de precisão múltipla.

3. Rotinas aritméticas de precisão múltipla: Trabalham apenas com números de precisão múltipla.

O módulo **tarit.c** é o que forma o núcleo do PRSA e que determina o desempenho do mesmo. Qualquer melhoria no desempenho do programa implica na mudança de um ou vários algoritmos e estrutura de dados usados neste módulo. Em particular a rotina que implementa a operação de multiplicação é crucial para a eficiência do RSA. Infelizmente os algoritmos mais eficientes para a execução desta operação são mais complexos, não são melhores considerando a faixa de grandeza dos números de precisão múltipla utilizados e são mais difíceis de serem implementados de uma forma independente de máquina[BREN78].

E.6.1 Rotinas envolvendo apenas números pequenos

São as seguintes as rotinas que envolvem apenas números pequenos:

abs(x) Devolve o valor absoluto contido em x.

odd(x) Devolve TRUE se x é ímpar e FALSE caso contrário.

random() Devolve um número menor que a base b. Implementa um gerador de aleatórios segundo o método da congruência linear. Esta rotina é chamada durante a inicialização dos parâmetros dos geradores de números aleatórios de precisão múltipla. Para cada dígito de tais parâmetros é executado uma chamada a rotina random.

Em geral estas rotinas estão disponíveis na biblioteca de qualquer linguagem de programação, mas elas tiveram de ser implementadas por problemas de incompatibilidade de tipos, uma vez que a linguagem C dispõe de várias categorias de tipo inteiro.

E.6.2 Rotinas mistas

As rotinas mistas que trabalham com números pequenos e números de precisão múltipla facilitam a descrição de várias outras rotinas para as quais fica transparente o problema de conversão de números pequenos em números de precisão múltipla ou vice-versa, quando for o caso.

São as seguintes as rotinas mistas:

mult-pos(an,i,bn) A variável an contém um número de precisão múltipla, a variável i um número inteiro positivo menor que b, e a variável bn irá receber o resultado da multiplicação de an por i. Esta rotina é chamada pela rotina de divisão de números de precisão múltipla.

mod-pos(an,i,bn) Esta é uma função que devolve o resto da divisão do número de precisão múltipla contido em an pelo número pequeno positivo menor que b. Além disso a variável bn vai receber o número de precisão múltipla resultante da divisão inteira de an por i. Esta rotina é chamada pela rotinas de saída do módulo **tentsai.c**.

soma-int(an,i,bn) Converte o número inteiro contido em i em um número de precisão múltipla que será somado a an; o resultado será armazenado em bn.

subt-int(an,i,bn) Converte o número inteiro contido em i em um número de precisão múltipla que será subtraído de an. O resultado assim obtido é armazenado em bn.

mult-int(an,i,bn) Converte o número inteiro contido em i em um número de precisão múltipla que será multiplicado por an. O resultado obtido é armazenado em bn.

div-int(an,i,bn) Converte o número inteiro contido em i em um número de precisão múltipla. O resultado da divisão de an por tal número será armazenado em bn.

mod-int(an,i,j) As variáveis i e j são do tipo inteiro. Aqui o conteúdo da variável i é convertido em um número de precisão múltipla. O número de precisão múltipla obtido como o resto da divisão de an

pelo número de precisão múltipla correspondente a i é convertido em inteiro para ser armazenado na variável j.

copia-int(i,an) O número inteiro contido na variável i é convertido em um número de precisão múltipla e armazenado na variável an.

igual-int(an,i) O número inteiro contido na variável i é convertido em um número de precisão múltipla. A função devolve TRUE se tal número é igual a número de precisão múltipla contido em an e FALSE no caso contrário.

E.6.3 Rotinas aritméticas de precisão múltipla

As rotinas aritméticas de precisão múltipla presentes no módulo **tarit.c** são as rotinas necessárias à implementação das operações aritméticas clássicas de soma, subtração, multiplicação, divisão e módulo e por outras rotinas aritméticas de precisão múltipla importantes também.

As operações aritméticas de soma, subtração, multiplicação, divisão e módulo tem também como característica especial o fato de terem sido implementadas por duas rotinas cada e não uma. Assim para cada operação aritmética existe uma rotina que computa o resultado considerando operandos positivos e uma outra rotina, que chama a primeira, para tratar do problema do sinal.

São as seguintes as rotinas aritméticas de precisão múltipla:

soma(an,bn,cn) Calcula a soma de an e bn, colocando o resultado em cn. Conforme o sinal de an e bn chama a rotina sub-abs ou soma-abs.

subt(an,bn,cn) Subtrai bn de an armazenando o resultado em cn. Da mesma forma que a rotina soma, chama uma das rotinas sub-abs e soma-abs, conforme os sinais de an e bn.

soma-abs(an,bn,cn) Calcula a soma dos números inteiros positivos de precisão múltipla an e bn, colocando o resultado em cn.

sub-abs(an,bn,cn) Armazena em cn o resultado da subtração do números inteiros positivos contidos nas variáveis an e bn. Note que o resultado pode ser negativo.

mult(an,bn,cn) Calcula o produto de an e cn, colocando o resultado em cn. Esta rotina efetivamente só trata do problema do sinal, chamando a rotina mult-abs para efetuar os cálculos.

mult-abs(an,bn,cn) Utilizando o algoritmo clássico de multiplicação de números de precisão múltipla esta rotina calcula o produto dos números inteiros positivos de precisão múltipla contidos nas variáveis an e bn, e o resultado assim obtido é armazenado na variável cn.

divisao(an,bn,qn) Calcula o resultado da divisão inteira de an por bn e coloca o resultado em qn. Esta rotina trata apenas do problema do sinal, deixando à rotina divipos a execução dos cálculos.

modulo(an,bn,rn) Calcula o resto da divisão de an por bn, colocando o resultado em rn. O sinal de rn será igual ao sinal de an, independente do sinal de bn. Esta rotina trata apenas do problema do sinal e quem faz as contas é a rotina divipos.

divipos(an,bn,qn,rn) Calcula o quociente e o resto da divisão inteira dos números inteiros positivos de precisão múltipla colocando o resultado em qn e rn, respectivamente. Esta é a rotina que trouxe maiores dificuldade de implementação.

potencia(an,bn,cn,mn) Calcula an elevado a bn módulo mn. O resultado é armazenado em cn. A variável bn deve conter um número inteiro positivo de precisão múltipla. Esta é a rotina utilizada para implementar o ciframento e deciframento usando o RSA.

copia(an,bn) Copia o conteúdo da variável an na variável bn.

decrementa(an) Decrementa a variável an de 1, deixando o resultado na própria variável an.

incrementa(an) Semelhante a rotina decrementa. Incrementa a variável an de 1, deixando o resultado na própria variável an.

tam(an) Devolve o número de dígitos do número de múltipla precisão contido em an.

negativo(an) Devolve TRUE se o número de múltipla precisão contido em an.é negativo e FALSE no caso contrário.

odd-ng(an) Devolve TRUE se o número de múltipla precisão contido em an é ímpar e FALSE caso contrário.

maior(an,bn) Devolve TRUE se o número de múltipla precisão contido em an é maior do que o número de múltipla precisão contido em bn e FALSE caso contrário.

igual(an,bn) Devolve TRUE se o número de múltipla precisão múltipla em an é igual ao número de múltipla precisão contido em bn e FALSE no caso contrário.

mdc(an,bn,xn) Calcula o máximo divisor comum de an e bn, armazenando o resultado em xn. Utiliza o algoritmo de Euclides.

resolve-congruencia(an,bn,cn,xn) Resolve a equação

$$ax \bmod b = c$$

onde a, b e c são os conteúdos de an, bn e cn respectivamente. A solução x será armazenada na variável xn. Utiliza o algoritmo extendido de Euclides.

raiz-quadrada(an,bn) Calcula o maior número positivo inteiro menor que raiz quadrada de an. O resultado é armazenado em bn. Utiliza o método de Newton.

logaritmo(an,bn,ln) Calcula o maior número inteiro positivo menor que o logaritmo de an na base bn. O resultado é armazenado em ln. O método utilizado baseia-se na aplicação do cálculo do logaritmo na base 2.

raiz(an,bn,xn) Calcula o maior número inteiro positivo menor que a raiz bn-ésima de an. O resultado é armazenado em xn. Também utiliza o método de Newton.

gera-rand(xn,an,cn,mn,nn) Gera elementos sucessivos da seqüência de números aleatórios de acordo com o método da congruência linear, considerando como parâmetros os valores contidos em xn, an, cn e mn, respectivamente; até encontrar um elemento menor do que o valor de nn. Este ficará em xn. O valor de mn é uma potência de 2.

inicia-rand(xn,an,cn,mn,nn) Gera os parâmetros de um gerador congruência linear de modo que mn seja a menor potência de 2 maior que o valor de nn. Os outros valores são calculados de acordo com Knuth[KNUT81].

rand-simp(nn,an) Gera um número, dígito a dígito, usando a função random; menor do que nn. Ele será armazenado em an. O valor de nn é uma potência de 2.

E.7 Números primos

O módulo que trata da geração de números primos é o **tprimos.c** sendo composto de 8 rotinas. Além das rotinas para teste de primalidade e teste de divisibilidade onde o número a ser testado é passado por um crivo, algumas outras rotinas foram criadas para permitirem a realização de algumas experiências interessantes e também para gerar números primos interessantes.

São as seguintes as rotinas presentes no módulo **tprimos.c**:

obt-pr-aleatorio(nn,pn) Gera um primo aleatório menor que nn. Ele será armazenado em pn.

testa-se-primo(pn,s) Testa se o número contido em pn é um número primo ou não. A variável s diz quantas vezes o algoritmo deve ser executado para que pn seja considerado primo. Se pn passar em todos teste a função devolve TRUE e no caso contrário devolve FALSE.

rabin(nn,xn) Verifica se xn é uma testemunha da não-primalidade de nn. Se xn *não* é testemunha devolve TRUE e no contrário devolve FALSE.

testa-divisibilidade(an) Verifica se an é divisível pelos primos pequenos presentes na tabela gerada pelo programa. A função devolve TRUE se an é divisível por algum primo pequeno e FALSE caso contrário.

menor-primo(ln,pn,s) Calcula o menor número primo maior que ln. Este número deverá ser armazenado na variável pn e passar por uma quantidade de testes de Rabin de acordo com o valor de s.

maior-primo(ln,pn,s) Semelhante a rotina menor-primo. A diferença é que se deve calcular o maior primo menor que o valor de ln. O resultado deve ser armazenado em pn.

menor-testemunha(nn,tn) Calcula a menor testemunha da não-primalidade de nn e armazena em tn, se nn é composto. No caso contrário copia nn em tn.

menor-fator(nn,fn) Determina o menor fator do número contido em nn, através de testes de divisibilidade. Só considera os fatores primos pequenos armazenados na tabela de pequenos primos. O menor fator de nn será armazenado na variável fn. Se o menor fator de nn não estiver presente na tabela de primos pequenos será copiado em fn o valor de nn. A rotina altera também o valor de nn, dividindo-o por fn.

A rotina testa-se-primo na verdade testa se um número é um primo seguro. O *grau* de segurança é definido por uma variável global. Assim sendo as rotinas obt-pr-aleatorio, menor-primo e maior-primo geram primos seguros aleatórios, calcula o menor primo seguro maior que um dado limite e calcula o maior primo seguro menor que um dado limite, respectivamente.

Quanto ao teste de divisibilidade não é obrigatório o teste por todos os primos armazenados na tabela de pequenos primos. É possível definir um limite máximo para os primos pequenos a serem testados, ou seja é possível definir o tamanho do crivo, ao qual um candidato ao teste de primalidade deve passar antes de chegar ao teste de Rabin.

E.8 Geração de chaves criptográficas

O módulo de geração de chaves criptográficas, **trsa.c**, consiste de apenas uma rotina:

rsa(ln,nn,en,dn)

A rotina gera dois primos menores que ln que seram utilizados para a geração dos pares (en,nn) que é a chave pública de ciframento e (dn,nn) a chave secreta de deciframento.

E.9 Rotinas auxiliares

O módulo taux.c contém algumas rotinas auxiliares que são utilizadas na redefinição de certos parâmetros do PRSA.

As rotinas presentes em **taux.c** são:

muda-modulo(bn) Muda o módulo de exponenciação. O novo módulo passa a valer bn.

muda-base(an) Converte o valor de an em um número pequeno que será o valor da nova base de entrada e saída.

muda-tam(tam) Muda o tamanho da área reservada a cada número de precisão múltipla. O programa será reinicializado.

help() Permite um auxílio em tempo de interação ao usuário.

E.10 A comunicação com o usuário

O módulo principal do PRSA é o **tpropri.c** onde está o *driver* do simulador da calculadora com capacidade para trabalhar com números inteiros de precisão múltipla

A calculadora utiliza a notação polonesa reversa, o que facilitou mais ainda a programação deste módulo. Visto de cima o *driver* contém uma chamada a rotina de inicialização, e a seguir um comando de repetição e só. Dentro do comando de repetição está uma chamada a rotina de leitura de caracteres e um comando de seleção de múltiplas alternativas.

Quando o *driver* encontra um número este é empilhado na pilha aritmética. Quando encontra um operador, uma quantidade adequada de operandos é desempilhada e a operação executada com a possibilidade de se empilhar algum resultado.

Números são escritos da forma que estamos acostumados. Para facilitar o trabalho com números maiores é permitido colocar pontos onde o usuário achar mais conveniente. Um diferença importante é a utilização do caracter de sublinhado antes dos algarismos de um número negativo ao invés do sinal de menos.

Se o ponto é colocado fora de um número ele é interpretado como sendo o início de um comentário, de modo que o restante da linha não

será mais considerado. A possibilidade de colocar comentários junto a entrada do PRSA faz mais sentido quando consideramos a possibilidade de processamento em *batch*. Nesse caso o arquivo de saída conterá os comentários que podem conter informações utéis para a compreensão do resultado. Ao invés do ponto também pode ser o usado o sinal < para iniciar um comentário.

O programa permite releituras de partes de uma linha. Para conseguir isto basta cercar a cadeia em questão com pares de parenteses, colchetes ou chaves e a seguir colocar um dígito para definir quantas vezes a cadeia deve ser relida. É possível utilizar os dígitos decimais, de 0 a 9, além dos caracteres hexadecimais *a* a *f*, correspondendo a 0,9,10 e 15 relidas, respectivamente.

Os operadores consistem em geral de um símbolo especial como + para soma, − para subtração, etc., ou de uma, duas ou três letras, como *a* para gerar um número aleatório, *pr* para gerar um número primo e *map* para gerar o maior número primo menor que um dado valor.

Cadeias de caracteres como "maria" também pode ser utilizadas como operandos do programa. Caracteres não gráficos podem ser representados usando a convenção usada pela linguagem C, com uma pequena diferença: ao invés de colocar o código dos caracteres em octal deve se utilizar a base 10.

É a seguinte a lista de operadores/comandos disponíveis:

- **Operações Aritméticas**
  - **Unárias**

    pr Geração de primos: Gera um primo aleatório menor que o operando. Empilha o número gerado e imprime número de candidatos que passaram no crivo e também número de testes de Rabin executados.

    rs Geração das chaves do RSA:Gera as chaves do RSA, escolhendo dois primos que são menores que o argumento fornecido. Empilha o produto dos dois primos, a chave de deciframento e a chave de ciframento.

    rq Raiz quadrada: Calcula o maior inteiro menor que a raiz quadrada do argumento. O resultado é empilhado.

map Maior primo: Gera o maior primo menor que o argumento.

mep Menor primo: Gera o menor primo maior que o argumento.

mt Menor testemunha: Gera a menor testemunha da não-primalidade do argumento, se ele for um número composto.

mf Menor fator: Gera o menor fator de um número composto, se ele pertence a tabela de primos pequenos utilizada pelo programa. Empilha também o argumento dividido pelo fator.

- **Binárias**

  \+ Soma

  \- Subtração

  * Multiplicação

  / Divisão

  % Módularização

  ↑ Exponenciação

  | Mdc

  rz Radiciação: Seja x o valor que está no topo da pilha e y o que está abaixo do topo. Então será calculada a raiz x-ésima de y, ou melhor será calculado o maior inteiro menor que a raiz x-ésima de y.

  lg Logaritmo: Usando a mesma notação acima, será calculado o maior inteiro que é menor que o logaritmo de y na base x.

- **Ternárias**

  rc Resolve congruência: Seja b o valor do topo da pilha, n está abaixo de b e a abaixo de n. É resolvida a equação $ax \bmod n = b$, se o $\mathrm{mdc}(a, n) = b$ e a solução é empilhada.

- **Sem operandos**

  a Gera número aleatório: É gerado um elemento menor que o módulo do gerador de aleatórios, que deve ter sido previamente definido.

mx Carrega módulo: Empilha o módulo de exponenciação modular.

- **Operações lógicas**
  - **Unárias**

    p? Testa se primo: Verifica se o topo da pilha é um número primo ou não.
  - **Binárias**

    m? Testa se maior: Verifica se o elemento abaixo do topo da pilha é maior que o elemento do topo.

    i? Testa se igual: Verifica se os dois elementos do topo da pilha são iguais.

- **Saída de resultados**

  p Imprime apenas: Imprime o topo da pilha, sem destruir. Um argumento.

  = Impressão destrutiva: Imprime o topo da pilha e elimina. Um argumento.

  ps Imprime cadeia: Imprime topo da pilha na forma de de cadeia de caracteres. Impressão não destrutiva. Um argumento.

  ic Imprime candidatos: Imprime número de candidatos que passaram pelo crivo de primos pequenos.

  ie Imprime testes: Imprime número de testes de Rabin já executados.

  it Imprime tempo: Imprime tempo de cpu gasto desde a inicialização do cronômetro.

- **Manipulação da pilha aritmética**

  r Troca: Troca a posição dos dois elementos do topo da pilha. Dois argumentos.

  d Duplica: Duplica elemento do topo da pilha. Um argumento

  dt Elimina: Decrementa topo da pilha, eliminando último elemento. Um argumento.

- **(Re)Definição de parâmetros**

  b Base de entrada/saída: Define nova base para entrada e saída dos números de precisão múltipla. Um argumento.

  n Base interna: Define número de bits da base de representação interna. O programa é reinicializado. Um argumento.

  t Tamanho: Define tamanho dos números de precisão múltipla, com relação ao número de palavras ocupadas. O PRSA será reinicializado. Um argumento.

  s Número de sondagens: Define número máximo de testes de Rabin a serem efetuados para considerar um número como primo. Um argumento.

  m Módulo de exponenciação: Define módulo de exponenciação. Um argumento.

  f Força dos primos: Define a força dos primos. Todo primo tem força igual a 1. Os primos seguros tem força igual a 2, etc. Um argumento.

  ir Inicialização gerador de aleatórios: Define módulo gerador de aleatórios e gera parâmetros para o método de congruência linear. Um argumento.

  ze Zera contador de testes: Zera o contador de número de testes de rabin já efetuados.

  zc Zera contador de candidatos: Zera o contador de número de candidatos que passam pelo crivo de primos pequenos.

  zt Zera cronômetro: Zera o marcador de tempo de cpu gasto.

  i Reinicializa: Reinicializa as variáveis globais do programa.

- **Variáveis e outros**

  !x Armazena na variável: Coloca o conteúdo do topo da pilha na variável x, onde x é uma letra entre a e z. Um argumento.

  ?x Carrega variável: Empilha conteúdo da variável x, onde x é uma letra entre a e z.

  st Mostra estado: Mostra o conteúdo de algumas variaveís do programa.

oe (Des)Liga eco: Normalmente toda linha de entrada é ecoada na tela pelo PRSA. Para evitar isto existe este comando que chaveia de um modo para o outro.

> Termina: Termina a execução do programa.

Muitos dos comandos/operadores utilizados tem como finalidade facilitar a atividade de teste do programa. Outros permitem que se tenha uma *ponta de prova* em cada rotina mais importante do programa.

Bibliografia

[ADLE83] L.M.Adleman, C.Pomerance e R.S.Rumely, On Distinguishing Prime Numbers from Composite Numbers, *Annals of Mathemathics*, Vol. 117, 173–206, 1983.

[ALEX84] Alexi, Chor e Goldreich, RSA bit is $\frac{1}{2} + \frac{1}{\log^c N}$ secure, *IEEE Focs*, 1984.

[BEIL66] A.H.Beiler, *Recreations in the Theory of Numbers*, Segunda edição, Dover Publications Inc., New York, 1966.

[BLAK78] B.Blakley e G.R.Blakley, Security of Number Theoretic Public Key Cryptosystem Against Random Attack, *Cryptologia*, Vol. 2, 305–321, 1978, Vol. 3, 29–42, 1979, Vol. 3, 105–118, 1979.

[BLAK79] G.R.Blakley e I.Borosh, Rivest-Shamir-Adleman Public Key Cryptosystems Do Not Always Conceal Messages, *Computers & Mathematics with Applications*, Vol.5, 169-178, 1979.

[BLUM81] Blum, M., *Three Applications of the Oblivious Transfer: 1. Coin Flipping by Telephone, 2. How to Exchange Secrets, 3. How to Send Certified Electronic Mail*, Dept. EECS, Univ. of California, Berkeley, California, 1981.

[BREN78] R.P.Brent, A Fortran Multiple-Precision Arithmetic Package, *ACM Transactions on Mathematical Software*, Vol.4, 57–70, 1978.

[COHE84] H.Cohen e H.W.Lenstra Jr., Primality Tests and Jacobi Sums, *Mathemathics of Computation*, Vol. 42, 297–330, 1984.

[COLL66] G.E.Collins, PM, A System for Polynomial Manipulation, *Communications of the ACM*. Vol.9, 578-589,1966.

[DAHA84] R.Dahab, *Alguns Aspectos da Criptografia Computacional*, Tese de Mestrado, UNICAMP, IMECC, DCC, 1984.

[DENN82] D.E.R.Denning, *Criptography and Data Security*, Addison-Wesley, 1982.

[DIFF77] W.Diffie e M.Hellman, New Direction in Criptography, *IEEE Transactions on Information Theory*, Vol. 22, 397–427, 1977.

[FIAT86] A.Fiat e A.Shamir, *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*, Resumo Estendido, 1986.

[GARD77] M.Gardner, Mathematical Games, *Scientific American* Vol. 237(2), 120-124, 1977.

[HARD59] G.H.Hardy e E.M.Wright, *An Introduction to the Theory of Numbers*, Oxford University Press, Inglaterra, 1959.

[KANH67] D.Kahn, *The Codebreakers*, MacMillan, 1967.

[KNUT81] D.E.Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, Reading, Mass., 1981.

[KONF78] L.M.Konfelder, On the Signture Reblocking Problem in Public-Key Cryptosystems, *Communications of ACM*, Vol. 21, 179, 1978.

[KONH81] A.G.Konheim, *Cryptography: a Primer*, Willey, 1981.

[LAKS83] S.Lakshmivarahan, Algorithms for Public Key Cryptosystems: Theory and Application, *Advances in Computers*, Vol. 22, 45–108, 1983.

[LEMP79] A.Lempel, Cryptology in Transition, *Computing Surveys*, Vol. 11, 285–303, 1979.

[LS3K79] C.L.Lucchesi, I.Simon, I.Simon, J.Simon e T.Kowaltowski, *Aspectos Teóricos da Computação*, Coleção Euclides, IMPA-CNPq, 1979.

[LUCC83] C.L.Lucchesi, Fatoração de Inteiros e de Polinômios Racionais, *Noticiário da SBM*, 1983.

[LUCC86] C.L.Lucchesi, *Introdução à Criptografia Computacional*, Editora Papirus, Campinas, SP, 1986.

[LUCC87] C.L.Lucchesi e M.J.C.Euzébio, Descrição da Implementação de um protótipo do RSA, *Anais do VII Congresso da Sociedade Brasileira de Computação*, Salvador, Bahia, 1987.

[MCEL78] A Public Key Cryptosystem Based in Algebraic Coding Theory, *DSN Report*, Jet Propulsion Lab., Californica Institute of Technology, Pasadena, 1978.

[MERK78] R.C.Merkle e M.E.Hellman, Hiding Information and Signatures in Trapdoor Knapsacks, *IEEE Transactions on Information Theory*, IT-24, 525–530, 1978.

[MILL75] G.I.Miller, Riemann's Hypothesis and Tests for Primality, *Proc. ACM Symp. Theory Comput.*, Vol.7, 234–239, 1975.

[NBS77] Data Encryption Standard, *FIPS PUB 46*, National Bureau of Standards, Washington, D.C., 1977.

[NIVE72] I.Niven e H.A.Zuckerman, *An Introduction to the Theory of Numbers*, Willey, New York, N.Y., 1972.

[POHL78] S.Pohlig e M.Hellman, An Improved Algorithm for Computing Logarithms over **GF**(p) and its Cryptography Significance, *IEEE Transactions on Information Theory*, Vol. IT-24, 106–110, 1978.

[POLL74] J.M.Pollard, Theorems on Factorization and Primality Testing, *Proc. Cambridge Philos. Soc.*, Vol.76, 521–528, 1974.

[POME80] C.Pomerance, J.L.Selfridge e S.S.Wagstaff Jr., The Pseudoprimes to 25.10^9, *Mathematics of Computation*, Vol. 35, 1.003–1.026, 1980.

[RABI76] M.O.Rabin, Probalistic Algorithms, 21–39, in J.F.Traub, *Algorithms and Complexity, New Directions and Recent Results*, Academic Press, 1976.

[RABI77] M.O.Rabin, Complexity of Computations, *Communications of ACM*, Vol. 20, 625–633, 1977.

[RABI79] M.O.Rabin, *Digitalized Signatures and Public-Key Function as Intractable as Factorization*, MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

[RABI80] M.O.Rabin, Probabilistic Algorithms for Testing Primality, *Journal of Number Theory*, Vol. 12, 128–138, 1980.

[RIVE78] R.L.Rivest, A.Shamir e L.Adleman, A Method for Obtaining Digital Signatures and Public-Key Criptosystems, *Communications of ACM*, Vol. 21, 120–126, 1978.

[RIV78b] R.L.Rivest, Remarks on a Proposed Cryptanalytic Attack of the MIT Public-Key Cryptosystem, *Cryptologia*, Vol. 2, 62–65, 1978.

[RIVE80] R.L.Rivest, A Description Of a Single-Chip Implementation of the RSA Cipher, *Lambda*, 4th Quartely, 14–18, 1980.

[SHAM82] A.Shamir, A Polynomial Time Algorithm for Breaking the Merkle-Hellman Cryptosystem, *Proc. 23rd. IEEE Symposium on Foundations of Computer Science*, 145–152, 1982.

[SIMM77] G.J.Simmons e J.N.Norris, Preliminary Comments on the M.I.T. Public Key Cryptosystem, *Cryptologia*, Vol.1, 406–414, 1977.

[SLOA83] N.J.A.Sloane, Encryption by Random Rotations, *Lecture Notes in Computer Science*, 71–128, 1983.

[SMIT83] J.Smith, Public Key Criptography, *BYTE*, 198–218, 1983.

[SOLO77] R.Solovay e V.Strassen, A Fast Monte-Carlo Test for Primality, *SIAM Journal Computing*, Vol. 6, 84–85, 1980.

[TERA86] R.Terada, *Soluções Práticas dos Problemas de Identificação e Assinatura*. 1986.

[TREL85] P.C.TreLeaven e S.A.Mansi, VLSI Architecture, *Anais do V Congresso da Sociedade Brasileira de Computação*, Vol. 2, Porto Alegre, RS, 1985.

[VINO77] I.M.Vinogradov, *Fundamentos de la Teoria de los Numeros*, MIR, Moscou, 1977.

Índice