Projeto e implementação de um protocolo para transmissão multidestinatária segura

UFRJ

Projeto submetido para a obtenção do título de

Engenheiro Eletrônico

ao Departamento de Eletrônica

da Escola de Engenharia da UFRJ

por

Eric Ricardo Anton

Projeto e implementação de um protocolo para transmissão multidestinatária segura

Autor:

Eric Ricardo Anton

Orientador:

Prof. Otto Carlos Muniz Bandeira Duarte, Dr.

Examinadores:

Prof. José Ferreira de Rezende, Dr.

Prof. Mauros Campello Queiroz, M.Sc.

Departamento de Eletrônica - Escola de Engenharia - UFRJ

Rio de Janeiro, RJ - Brasil

Março de 2001

A Ana Carolina.

Agradecimentos

À minha família, principalmente meus pais por todo o amor, orientação e apoio ao longo da minha vida.

À minha querida Ana Carolina e sua família pela maravilhosa presença na minha vida.

A toda a equipe do GTA, em particular aos professores Otto e Rezende, por toda a orientação e conselhos, Aloysio, pelo ensino das técnicas de modelagem de protocolos, e Leão, pela ajuda no aprendizado da linguagem Java.

Aos professores do Departamento de Eletrônica, principalmente ao professor Mauros, por todos os conhecimentos adquiridos e pela experiência obtida com a superação das dificuldades enfrentadas.

Resumo do Projeto Final apresentado ao DEL/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletrônico.

Projeto e implementação de um protocolo para transmissão multidestinatária segura

Eric Ricardo Anton

Março de 2001

Orientador: Otto Carlos Muniz Bandeira Duarte
Departamento: Engenharia Eletrônica

Em comunicações via *Internet*, a utilização da técnica de transmissão multidestinatária (*multicast*) torna possível a transmissão do tipo um-para-muitos. Esta técnica é utilizada por exemplo na transmissão de palestras, seminários, músicas e vídeos. Devido à forma como este tipo de transmissão foi implementado, é difícil a restrição dos dados transmitidos a um conjunto específico de destinatários.

Este trabalho propõe e implementa um protocolo para a transmissão segura de dados utilizando transmissão multidestinatária. É utilizada criptografia como forma de restringir o acesso aos dados. Dados são encriptados por meio da sua combinação com uma chave criptográfica, gerando um conjunto de dados incompreensível, que somente pode ser restaurado ao seu estado original com o conhecimento da chave criptográfica utilizada.

Palavras Chave

Rede de computador

Internet

Segurança

Criptografia

Multidestinatário

Sumário

SUMÁRIO

SUMÁRIO

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Capítulo I

Introdução

I.1 Motivação

Em comunicações via *Internet*, a utilização de endereços IP do tipo multidestinatário torna possível a transmissão do tipo um-para-muitos, onde o destino é um endereço multidestinatário, também chamado de grupo multidestinatário. Assim, qualquer entidade que se associe a um grupo multidestinatário recebe todos os dados enviados a este endereço, de forma semelhante à sintonia de um rádio. Esta técnica é utilizada na transmissão via *Internet* de palestras, seminários, músicas e vídeos. Devido à forma como o roteamento multidestinatário foi implementado na *Internet*, que se serve de um modelo totalmente aberto, é difícil a restrição dos dados transmitidos a um conjunto específico de destinatários, uma vez que qualquer usuário pode se associar ao endereço multidestinatário.

Em sistemas de difusão como rádio e televisão, os sinais são enviados em todas as direções, sendo limitados pelo alcance da antena transmissora. Todos os receptores localizados na área de alcance e sintonizados de forma apropriada são capazes de receber a transmissão. A transmissão multidestinatária na *Internet* ocorre sobre uma infra-estrutura em forma de malha. Os receptores interessados em receber dados destinados a um endereço multidestinatário simplesmente associam-se a este endereço. Uma forma de transmissão multidestinatária em redes com topologia em malha é a por inundação, onde dados são transmitidos em todos os enlaces, cau-

sando desperdício de banda-passante. Este problema foi solucionado com a implementação de algoritmos que inibem a transmissão de dados destinados a um endereço multidestinatário em ramos da rede que não possuem receptores inscritos neste endereço. Estes algoritmos não restringem, no entanto, os receptores a somente um conjunto pré-estabelecido, uma vez que qualquer entidade pode associar-se a qualquer grupo multidestinatário.

A estratégia adotada neste trabalho para restringir o conjunto de receptores consiste na transmissão de dados encriptados por meio da sua combinação com uma chave criptográfica, gerando um conjunto de dados incompreensível. Os dados originais somente podem ser recuperados com o conhecimento de uma chave criptográfica que pode ser igual ou não à utilizada para encriptar.

A utilização de criptografia em transmissões multidestinatárias não é usual devido à necessidade de distribuição de uma chave criptográfica para todos os destinatários. Esta transmissão deve receber especial atenção, pois a exposição indevida da chave criptográfica expõe os dados, colocando em risco a segurança da transmissão.

Este projeto desenvolveu um protocolo que permite a transmissão multidestinatária a um conjunto pré-estabelecido de receptores, sem a necessidade de compartilhamento prévio de uma única chave criptográfica. Cada destinatário deve estar cadastrado em uma base de dados do emissor, tendo obtido um identificador e uma chave criptográfica por meio da qual lhe é possível obter acesso à chave criptográfica que protege os dados. O emissor, antes de iniciar a transmissão dos dados, envia no endereço multidestinatário pacotes contendo um identificador de usuário e a chave utilizada na transmissão de dados encriptada com a chave compartilhada com o usuário. Cada receptor ao receber o pacote que lhe corresponde utiliza sua chave, obtendo conhecimento da chave utilizada na proteção dos dados transmitidos. O protocolo permite a alteração periódica da chave, mesmo durante a transmissão de dados, por meio do reenvio dos supracitados pacotes.

Algumas possíveis aplicações para este protocolo são a distribuição de vídeos e músicas para assinantes de um serviço, a transmissão de palestras e seminários para participantes de congressos ou cursos a distância, a troca de dados sigilosos entre departamentos de uma empresa

através de sua *intranet*, ou ainda o envio de memorandos internos a diversas filiais de uma empresa.

Capítulo II

Segurança

O termo segurança aplicado a redes de computadores possui um sentido bastante amplo, podendo referir-se a:

- segurança na execução de códigos, que pode ser ameaçada por programas como vírus ou cavalos-de-tróia;
- segurança de servidores, que podem sofrer ataques levando ao interrompimento da prestação dos serviços;
- segurança na transmissão ou armazenamento de mensagens ou dados.

Dada a proposta deste trabalho, serão apresentados neste capítulo os principais aspectos relacionados à segurança na transmissão de mensagens e armazenamento de dados, que envolvem alguns requisitos e algumas técnicas adotadas para prover esta segurança.

II.1 Requisitos de segurança

Segurança pode ser provida a uma comunicação em três aspectos:

- autenticação — garante que o emissor de uma mensagem é realmente a entidade que se identifica como tal;
- integridade — garante que os dados recebidos em uma mensagem correspondem aos enviados, sem sofrer quaisquer alterações indevidas;
- privacidade — garante que os dados enviados em uma mensagem somente podem ser lidos pelo destinatário.

São apresentados a seguir os principais conceitos e algoritmos associados a criptografia, como estes podem ser adotados para oferecer suporte aos requisitos acima expostos, e os principais tipos de ataque conhecidos para burlar estes algoritmos.

II.2 Criptografia

Criptografia é a ciência que estuda técnicas de codificação de dados de modo a torná-los incompreensíveis. Um texto (*plaintext* ou *cleartext*) é transformado em um texto cifrado (*ciphertext*) ou criptograma, que deve ser incompreensível. O processo de transformação do texto em criptograma é denominado encriptação ou cifragem e o processo inverso decriptação ou decifragem, conforme ilustrado na Figura II.1.

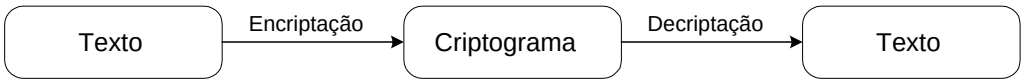

Figura II.1: Criptografia e decriptografia.

Um dos primeiros e mais simples algoritmos criptográficos, desenvolvido por Julius Cæsar, é conhecido como Cifra de Cæsar. A encriptação consistia na substituição de cada caractere de uma mensagem pelo caractere três posições adiante no alfabeto, retornando-se ao início do alfabeto quando necessário. Desta forma, a letra ‘A’ era substituída por ‘D’, ‘B’ por ‘E’, ‘C’ por ‘F’, e assim por diante. A operação de decriptação consistia no processo inverso.

Diversos outros algoritmos criptográficos foram desenvolvidos e em todos eles, a segurança da informação dependia do sigilo do algoritmo. Dada a grande dificuldade em se criar novos algoritmo e o grande inconveniente em compartilhá-los, estabeleceu-se o conceito de que a segurança do algoritmo não deveria depender do seu conhecimento, mas sim do conhecimento de algum segredo compartilhado.

Desenvolveu-se então o conceito de chave criptográfica. Em algoritmos que fazem uso de chaves criptográficas, a segurança dos dados deve depender apenas do conhecimento da chave ou chaves utilizadas , e não do algoritmo. Esta filosofia é análoga ao funcionamento de uma fechadura: embora o funcionamento interno (algoritmo) de uma fechadura seja de conhecimento público, só é possível trancar ou destrancá-la com a utilização da chave apropriada.

A segurança de um sistema de criptografia deve depender portanto apenas do conhecimento das chaves, e não do algoritmo utilizado, que pode ser de livre conhecimento, devendo apenas a chave ser mantida em segredo.

Matematicamente, um sistema criptográfico pode ser definido por uma mensagem M a ser protegida, uma transformação de cifragem E que gera um criptograma C a partir de M ($C = E(M)$), e uma transformada de decifração D que recupera M a partir de C ($M = D(C) = D(E(M))$).

Um fato interessante e de grande importância, conforme apresentado adiante, é a possibilidade de gerar um criptograma a partir da aplicação de uma função de decriptação a um dado, e a possibilidade de recuperação do dado por meio da aplicação da função de encriptação ao criptograma, ou seja, a relação $E(D(M)) = M$ deve permanecer verdadeira. Com esta técnica é possível "encriptar" decriptando e "decriptar" encriptando.

Uma vez que uma dada informação esteja protegida por criptografia, esta só pode ser acessada com o conhecimento da chave criptográfica que a protege. Assim, se a chave for ameaçada, a informação também o estará, e se a chave for perdida, a informação não poderá ser acessada.

Existem dois tipos criptografia que fazem uso de chaves:

- criptografia simétrica ou de chave secreta;

- criptografia assimétrica ou de chave pública.

Estes tipos de criptografia juntamente com seus principais algoritmos são apresentados a seguir.

II.2.1 Criptografia Simétrica ou de Chave Secreta

Este tipo de criptografia faz uso de uma única chave criptográfica para realizar os processos de encriptação e decriptação, conforme ilustrado pela Figura II.2. O termo simétrico refere-se ao fato de uma mesma chave ser utilizada nas duas operações. Para prover segurança, esta chave deve ser de conhecimento apenas das entidades que desejem trocar informações, sendo mantida secreta de todas as demais.

Algumas aplicações para este tipo de criptografia são a proteção de dados transmitidos por um canal inseguro ou armazenados em um meio também inseguro, a autenticação de entidades, entre outros.

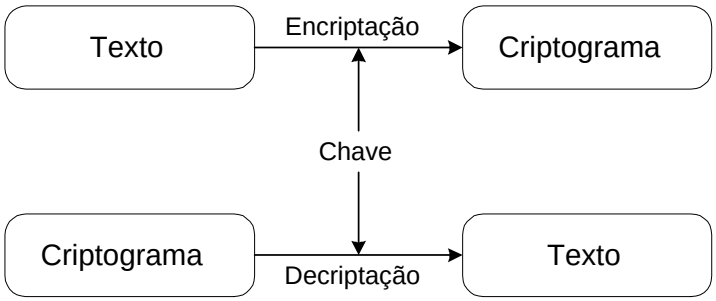

Figura II.2: Criptografia simétrica ou de chave secreta.

Um algoritmo simples capaz de gerar um criptograma indecifrável consiste na escolha de uma seqüência randômica de bits do mesmo tamanho da mensagem a ser encriptada. Cada bit desta seqüência, que poderia ser, por exemplo, a representação ASCII dos caracteres formadores de uma frase, deve sofrer a operação de OU-EXCLUSIVO (XOR) com cada bit da mensagem, gerando um criptograma que não fornece qualquer informação sobre a chave ou a mensagem.

Para que o criptograma gerado por este algoritmo permaneça indecifrável, é importante que cada chave seja utilizada uma única vez, razão pela qual este algoritmo é chamado de Bloco de Uma Só Utilização (*One-Time Pad*). A utilização prática deste algoritmo é inviável devido a necessidade de compartilhamento de uma chave secreta muito grande, do tamanho dos dados a serem encriptados, o que é impraticável. Com a finalidade de evitar este tipo de inconveniente, foram desenvolvidos algoritmos criptográficos que fazem uso de chaves de tamanho limitado, conforme exposto a seguir.

Encriptação de blocos

Um algoritmo criptográfico converte um bloco de texto em um criptograma. A utilização de chaves criptográficas muito pequenas não fornece segurança, dada a facilidade de se testar todos seus valores possíveis. Similarmente, a utilização de blocos de texto muito pequenos torna possível o mapeamento entre todos os blocos e seus respectivos criptogramas, comprometendo a segurança do algoritmo. Em contrapartida, a utilização de blocos muito longos pode tornar o algoritmo desnecessariamente complexo, acarretando em problemas de desempenho. Um tamanho de bloco ideal deve ser capaz de balancear estes dois fatores (segurança $\times$ complexidade).

O modo mais genérico de encriptar um bloco de k bits consiste em executar um mapeamento do tipo um-para-um entre os 2^k valores possíveis. Este mapeamento deve ser do tipo um-para-um de modo a permitir a operação inversa: a decriptação dos dados. Para o mapeamento de blocos de 64 bits é necessária a construção de uma tabela de mapeamento com 2^{64} entradas, exigindo 2^{70} bits ($2^{64} + 64$), o que corresponde a 134.2E6 TB de informação. Este mapeamento atuaria como o segredo compartilhado entre as entidades. Percebe-se que o compartilhamento de tal informação é inviável.

Algoritmos criptográficos devem ser desenvolvidos de modo a utilizar chaves criptográficas de tamanhos razoáveis e funções que executem mapeamentos um-para-um que pareçam ser completamente aleatórios, tornando impossível o estabelecimento de qualquer correlação entre os dados de entrada e saída do algoritmo. A alteração de um único bit na entrada da função de encriptação deve permitir a geração de uma saída totalmente independente. Cada bit de entrada

deve ser capaz de alterar cada bit da saída com um probabilidade de 50%.

Existem dois tipos de operação que são efetuadas em blocos de dados: substituição e permutação. Uma substituição especifica, para um bloco de k bits, o valor correspondente na saída a cada um dos 2^k valores de entrada. A especificação completa de todas as substituições necessita de $k \times 2^k$ bits. Como já foi apresentado, isto é inviável para grandes blocos, sendo no entanto perfeitamente viável para blocos pequenos, na ordem de 8 bits.

Uma permutação especifica para cada um dos k bits de entrada sua posição na saída. Um exemplo é a movimentação do 1^o bit para a 10^a posição, do 2^o bit para a 3^a posição, e assim por diante. A especificação de uma permutação necessita de $k \times log_2 k$ bits. Uma permutação é um caso particular de substituição em que cada bit da saída é influenciado por apenas um bit da entrada. Para blocos de mesmo tamanho, o número de possíveis permutações é menor que o de substituições, permitindo sua aplicação em blocos maiores, na ordem de 64 bits.

As operações de permutação e substituição podem ser implementadas em circuitos simples denominados respectivamente caixa-P (*P-box*) e caixa-S (*S-box*). A Figura II.3 ilustra o funcionamento desses circuitos. Mesmo em implementações de algoritmos em *software*, os termos caixa-P e caixa-S são utilizados para representar essas operações.

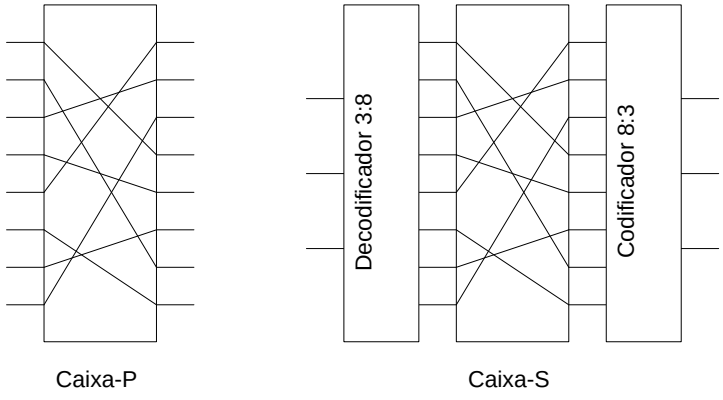

Figura II.3: Caixas P e S.

Uma técnica utilizada na construção de algoritmos criptográficos consiste na divisão do texto em blocos menores, a execução de uma substituição em cada bloco, a união dos blocos, e

a execução de uma permutação utilizando um permutador do mesmo tamanho da entrada. Este processo é ilustrado na Figura II.4 para uma entrada de 64 bits dividida em blocos de 8 bits. Cada execução é chamada de rodada, podendo diversas rodadas serem executadas de modo a aumentar a segurança do algoritmo.

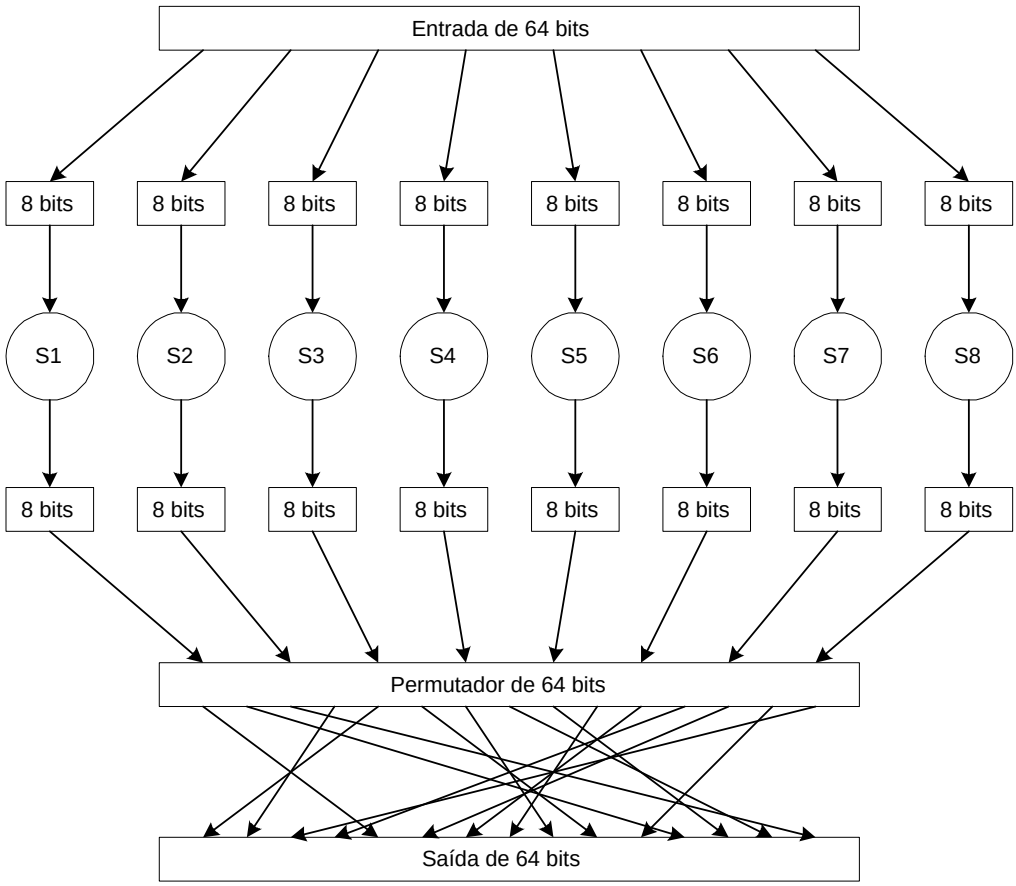

Figura II.4: Encriptação de blocos.

Neste algoritmo, em uma única rodada cada bit de entrada pode influenciar oito bits de saída, pois participa de apenas uma substituição. Em uma segunda rodada, os oito bits afetados pela primeira podem, devido à permutação, participar de diferentes substituições, tornando possível a influência sobre todos os bits da saída. Por segurança, deve-se executar o maior número de rodadas possível, e por eficiência, não mais que o necessário. Existe para cada algoritmo um número ótimo de rodadas que deve ser determinado de modo a oferecer um equilíbrio entre segurança e eficiência.

Um algoritmo de criptografia deve ainda ser eficiente no sentido inverso: o de decriptografia. Percebe-se que na técnica exposta acima as operações são eficientes em ambos os sentidos.

DES

O DES (*Data Encryption Standard* ou Padrão para Encriptação de Dados) foi publicado em 1977 pelo *National Bureau of Standards* (Escritório Nacional de Padrões dos Estados Unidos) para uso em aplicações comerciais e não-classificadas do governo americano. A chave criptográfica utilizada possui 56 bits, embora aparente ter 64, pois um bit de cada byte é utilizado para paridade impar. São utilizados blocos de entrada de 64 bits que são mapeados em blocos de saída de igual tamanho.

Na época de seu desenvolvimento, o DES era eficiente para implementações em *hardware*, mas relativamente lento para implementações em *software*. Embora na época esta característica limitasse sua utilização a organizações capazes de arcar com o custo de *hardware*, atualmente, com os avanços tecnológicos, sua implementação em *software* tornou-se perfeitamente viável e bastante utilizada.

Um dos aspectos mais controversos do DES é a utilização de uma chave de 56 bits, o que torna o algoritmo menos seguro contra ataques exaustivos do que se fossem utilizados 64 bits. Em contrapartida, o DES apresenta a vantagem, segundo seus desenvolvedores, de permitir a detecção de chaves inválidas. O uso de paridade embora dificulte a utilização de uma chave inválida não a impede, pois chaves inválidas podem ser erroneamente tomadas como válidas.

O funcionamento dos DES é apresentado na Figura II.5 e consiste das seguintes etapas:

- permutação dos dados de entrada;
- geração de 16 chaves de 48 bits a partir da chave criptográfica;
- em cada rodada, é utilizada como entrada a saída fornecida pela rodada anterior, sendo cada rodada comandada por uma das 16 chaves de 48 bits;

- após a 16^a rodada, a saída sofre uma permutação que corresponde ao inverso da primeira permutação.

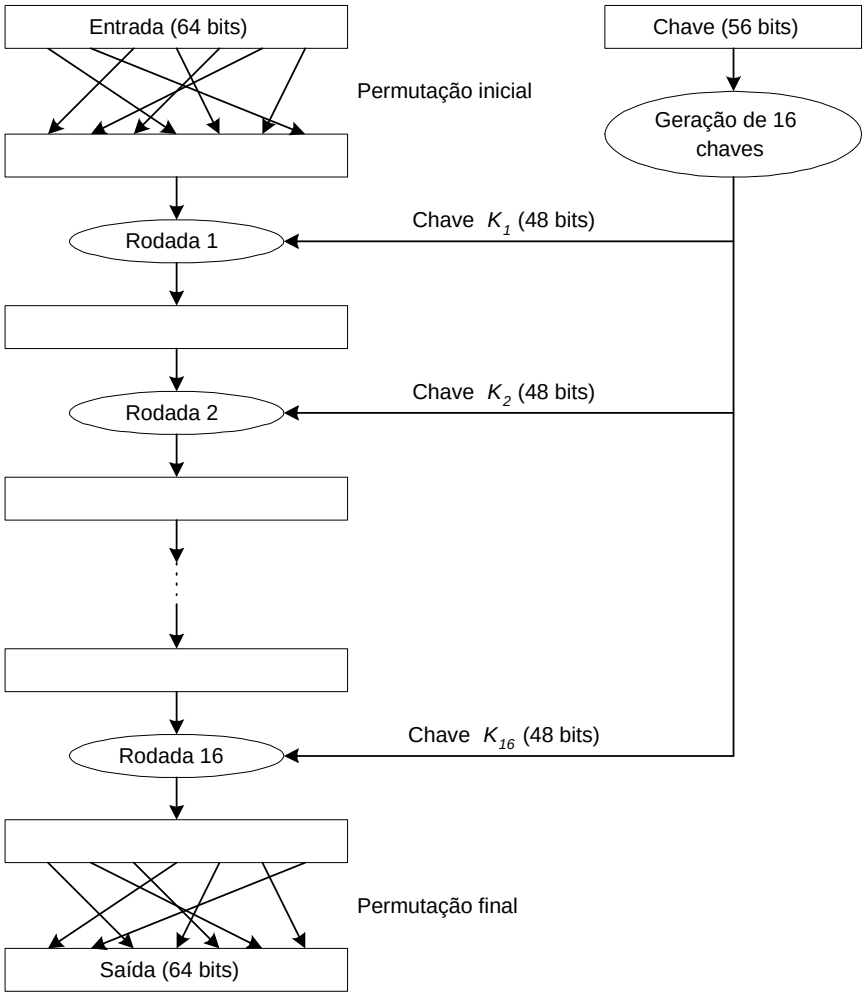

Figura II.5: Funcionamento do DES.

As permutações inicial e final não aumentam a segurança do algoritmo. Especula-se que tenham sido adicionadas com a finalidade de tornar o algoritmo menos eficiente em implementações por *software*.

Para a geração das chaves utilizadas em cada rodada, a chave de 56 bits sofre uma permutação inicial, sendo o valor gerado dividido em dois valores de 28 bits: C_0 (a metade superior) e D_0 (a metade inferior). O processo de geração de cada chave se dá conforme ilustrado na Figura II.6.

C_{i-1} e D_{i-1} gerados na rodada anterior sofrem um desvio rotacionado para a esquerda de 1 bit nas rodadas 1, 2, 9 e 16, e de 2 bits nas demais, gerando C_i e D_i, que sofrem permutações e são concatenados gerando a chave da rodada K_i. Durante essas duas permutações finais, 4 bits são descartados em cada uma, fazendo com que a chave secreta de 56 bits gere chaves de rodada de 48 bits.

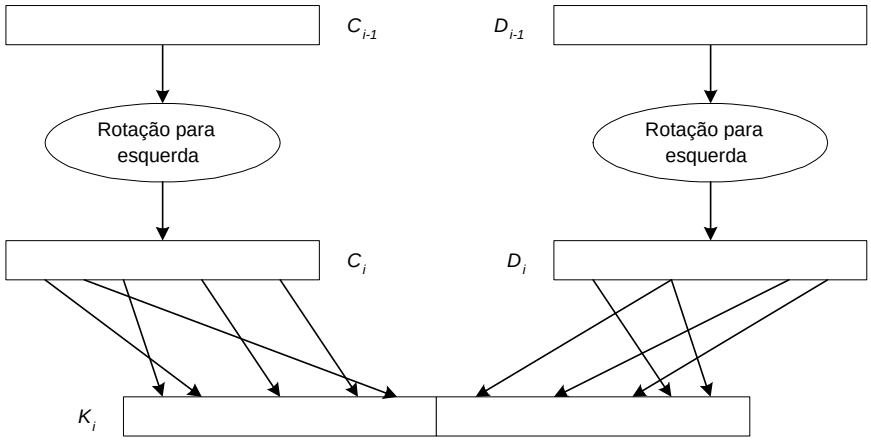

Figura II.6: Geração das chaves de rodada utilizadas pelo DES.

Uma rodada do algoritmo é ilustrada na Figura II.7. As etapas envolvidas na decriptografia dos dados correspondem ao caminho inverso, exceto pela função $f(R_i, K_i)$ que é sempre executada no seu sentido natural. Esta função divide os 32 bits de R_i em 8 blocos de 4 bits, que são expandidos concatenando-se um bit adjacente a cada lado, obtendo-se 8 blocos de 6 bits. Estes blocos sofrem então uma operação de OU-EXCLUSIVO (XOR) com a chave K_i. O resultado dessa operação sofre uma substituição, que é diferente para cada rodada e mapeia os 6 bits de cada bloco em 4 bits, possuindo o valor de saída dessa função 32 bits. Todas as seqüências de permutação são predefinidas, constituindo parte do algoritmo.

Profundas análises deste algoritmo concluíram tratar-se de um algoritmo matematicamente são.

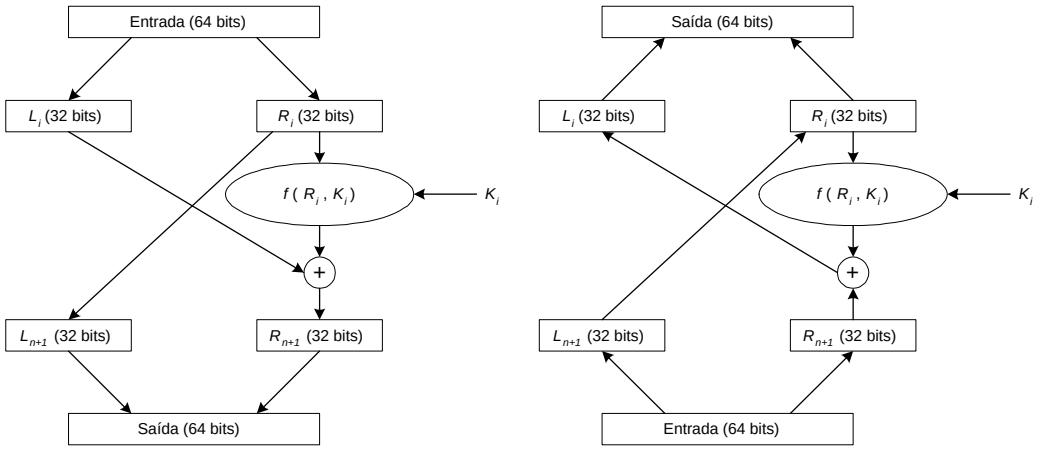

Figura II.7: Algoritmo de uma rodada do DES: encriptação e decriptação.

IDEA

O IDEA (*International Data Encryption Algorithm* ou Algoritmo Internacional para Encriptação de Dados), foi desenvolvido em 1991 por Xuejia Lai e James L. Massey, e projetado de modo a ser eficiente para implementação em *software*. É utilizada uma chave criptográfica de 128 bits para criptografar um bloco de 64 bits em um criptograma de mesmo tamanho.

Similarmente ao DES, este algoritmo, ilustrado na Figura II.8, opera com várias rodadas e possui uma função que não necessita ser efetuada no sentido inverso para executar a decriptografia.

A chave de 128 bits é expandida em 52 chaves de 16 bits $K_1, K_2, \ldots, K_{52}$, sendo a expansão diferente para os processos de encriptação e decriptação. Isto ocorre de forma que o mesmo algoritmo seja executado para as duas operações. A expansão das chaves para encriptação consiste dos seguintes passos:

- dividir a chave de 128 bits, a partir do primeiro bit, em 8 partes de 16 bits, que são as 8 primeiras chaves;
- executar a mesma operação, iniciando no 25^o bits e percorrendo de volta ao início, obtendo as 8 chaves seguintes;

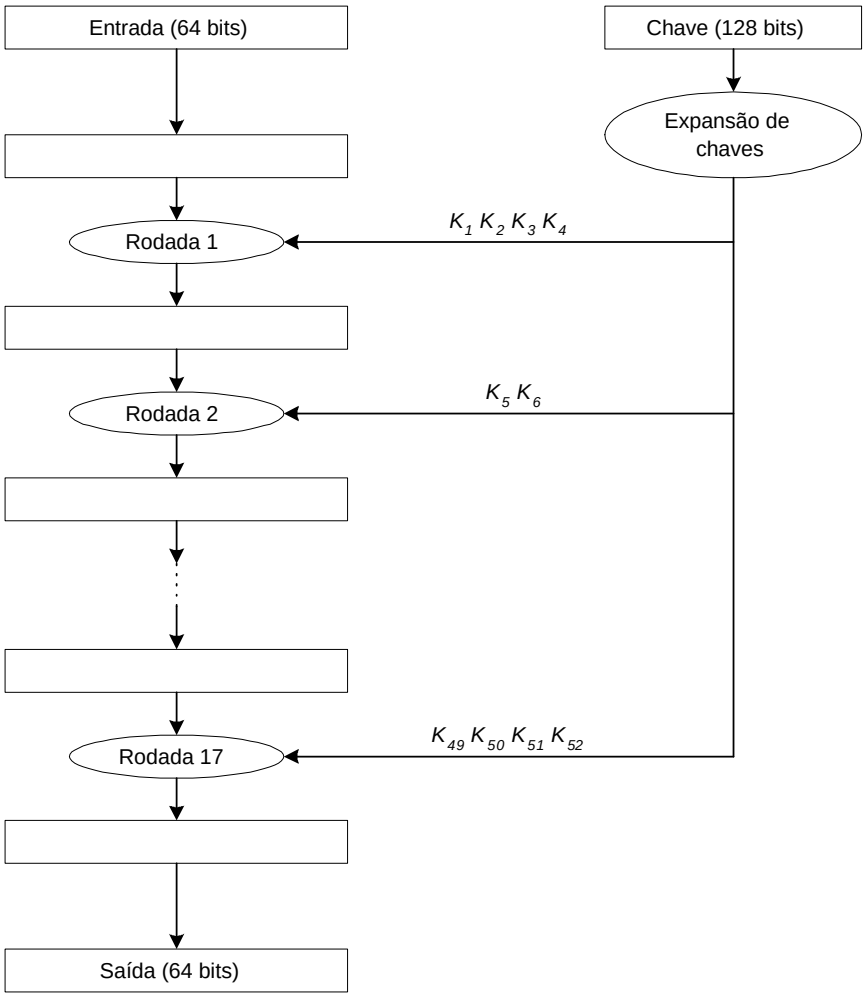

Figura II.8: Esquema funcional do IDEA.

- repetir o passo anterior, sempre iniciando a extração a partir do bit 25 posições subseqüentes até que sejam obtidas 52 chaves;
- trocar as chaves K_{50} e K_{51}.

O algoritmo consiste de 17 rodadas, que são diferentes para rodadas pares e impares, as impares utilizando 4 chaves (K_a, K_b, K_c e K_d) e as pares 2 (X_e e X_f). Cada rodada recebe um valor de 64 bits que é tratado como 4 valores de 16 bits (X_a, X_b, X_c e X_d). As operações realizadas em cada rodada são:

- Rodadas Impares

  $X_a \leftarrow X_a \times K_a$

  $X_b \leftarrow X_c + K_c$

  $X_c \leftarrow X_b + K_b$

  $X_d \leftarrow X_d \times K_d$

- Rodadas Pares

  Cada rodada utiliza quatro valores Y_i, Z_i, Y_o, Z_o definidos como

  $Y_i = X_a \oplus X_b$

  $Z_i = X_c \oplus X_d$

  $Y_o = ((K_e \times Y_i) + Z_i) \times K_f$

  $Z_o = (K_e \times Y_i) + Y_o$

  para modificar os valores X_a, X_b, X_c e X_d da seguinte forma:

  $X_a \leftarrow X_a + Y_o$

  $X_b \leftarrow X_b + Y_o$

  $X_c \leftarrow X_c + Z_o$

  $X_d \leftarrow X_d + Z_o$,

onde na adição o eventual bit excedente é desconsiderado e a multiplicação é módulo $2^{16} + 1$ (ver Apêndice A).

As rodadas impares apresentam a característica de executarem a operação inversa quando fornecidas as chaves correspondentes ao inverso matemático das chaves originais, revertendo a operação realizada pela encriptação. As chaves inversas para as chaves K_a e K_d consistem do inverso da operação de multiplicação, e para as chaves K_b e K_c do inverso da operação de adição.

As rodadas pares apresentam a característica de serem suas próprias inversas, de modo que o fornecimento dos blocos de saída da encriptação como entrada para a decriptação, com o uso das mesmas chaves, apresenta como saída os blocos fornecidos para encriptação.

Para a decriptação são, portanto, utilizados os mesmos conjuntos de chaves, em ordem contrária (primeiro $K_{49}, \ldots, K_{52}$, seguidas por K_{47} e K_{48}, até $K_1, \ldots, K_4$), sendo as chaves fornecidas às rodadas impares as correspondentes ao inverso da operação realizada na encriptação: ${K_a}^{-1}$, $-K_b$, $-K_c$ e ${K_d}^{-1}$.

A simetria do algoritmo permite que este seja tão eficiente para encriptação como para decriptação. As operações envolvidas foram escolhidas de modo a facilitar a implementação em *software*.

3DES

Uma crítica feita ao DES refere-se ao pequeno tamanho da chave utilizada, tornando-o inseguro contra ataques exaustivos. Visando resolver este problema, desenvolveu-se o conceito de encriptação múltipla, que consiste na encriptação e/ou decriptação em cadeia da informação a ser protegida, utilizando uma ou mais chaves. Este conceito pode ser aplicado não somente ao DES mas a qualquer algoritmo criptográfico. Como já foi comentado na SeçãoII.2, a operação de decriptação também pode ser utilizada para a encriptação de dados.

Com o objetivo de aumentar a segurança proporcionada pelo DES foi padronizado o 3DES (*Triple DES* ou DES Triplo), que consiste na aplicação de três operações criptográficas com a utilização de duas chaves, na forma encriptação-decriptação-encriptação (EDE) conforme apresentado na Figura II.9.

Diversas combinações de encadeamento de operações criptográficas foram consideradas antes da padronização do 3DES. A encriptação do tipo EE com duas chaves K_1 e K_2 revelou-se vulnerável a um ataque conhecido como Encontrar no Meio (*Meet-in-the-Middle*), que consiste em, para pares mensagem-criptograma conhecidos, encriptar a mensagem com todas as possível chaves K_1 e decriptar o criptograma com todas as possível chaves K_2, verificando quais resultados dessas operações são iguais, correspondendo a candidatos para o par $\langle K_1, K_2 \rangle$. Esta vulnerabilidade levou a tentativas com o encadeamento de três operações.

A decisão de se adotar duas chaves ao invés de três advém do fato de o uso de duas ter

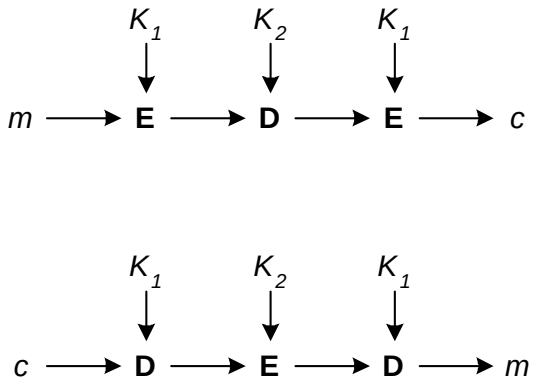

Figura II.9: Encriptação e decriptação com 3DES.

sido considerado seguro o suficiente para inibir ataque do tipo Força Bruta (ver Seção II.7.2). A análise as opções de ordenação das operações criptográficas concluiu que todas apresentariam resultados semelhantes a nível de segurança. A utilização do 3DES na maneira como foi proposto apresenta a vantagem de manter compatibilidade de interação com sistemas DES, por meio da utilização de chaves K_1 e K_2 iguais. A utilização de três blocos revelou-se suficiente para os padrões de segurança estabelecidos.

Blowfish

Blowfish é um algoritmo de criptografia simétrica desenvolvido por Bruce Schneier em 1993 como uma alternativa ao DES e ao IDEA. Este algoritmo atua sobre blocos de dados de 64 bits e utiliza chaves criptográficas de qualquer tamanho menor que 448 bits. Embora haja uma fase de inicialização bastante complexa, os processos de encriptação e decriptação são bastante eficientes quando implementados em processadores de 32 bits.

O algoritmo consiste em duas etapas:

- expansão da chave criptográfica, que é convertida em diversas sub-chaves totalizando 4168 bytes;
- encriptação dos dados, executada por meio de 16 rodadas, cada uma consistindo de uma

permutação dependente da chave e substituição dependente da chave e do dado.

As sub-chaves são computadas e armazenadas nas seguintes estruturas:

- um conjunto de 18 variáveis P de 32 bits:
  $P_1, P_2, \ldots, P_{18}$;
- 4 caixas-S de 32 bits com 256 posições cada:
  $S_{1,0}, S_{1,1}, \ldots, S_{1,255}$
  $S_{2,0}, S_{2,1}, \ldots, S_{2,255}$
  $S_{3,0}, S_{3,1}, \ldots, S_{3,255}$
  $S_{4,0}, S_{4,1}, \ldots, S_{4,255}$.

O método utilizado para o cálculo das sub-chaves será apresentado mais adiante.

A encriptação se dá por meio de 16 rodadas que utilizam como entrada um bloco X de 64 bits. Cada rodada é constituída por:

- dividir X em duas partes de 32 bits: X_L e X_R;
- para $i = 1, 2, \ldots, 16$:
  - $X_L \leftarrow X_L \oplus \pi$
  - $X_R \leftarrow F(X_L) \oplus X_R$
  - Alternar s valores de X_L e X_R
  - Alternar s valores de X_L e X_R (desfazendo a última alternação)
  - $X_R \leftarrow X_R \oplus P_{17}$
  - $X_L \leftarrow X_L \oplus P_{18}$
- recombinar X_L e X_R.

A função F é definida dividindo-se X_L em quatro partes a, b, c, d de 8 bits e aplicando-se $F(X_L) = \Big(\Big(((S_{1,a} + S_{2,b}) \bmod 2^{32}) \oplus S_{3,c}\Big) + S_{4,d}\Big) \bmod 2^{32}$.

O processo de decriptação ocorre da mesma forma que o de encriptação exceto pelas variáveis $P_1, P_2, \ldots, P_{18}$, que são utilizadas em ordem reversa.

O cálculo das sub-chaves se dá, utilizando o próprio algoritmo, conforme os seguintes passos:

- inicializar $P_1, P_2, \ldots, P_{18}$ e as quatro caixas-S, em ordem, com o conjunto de bits correspondente à representação em hexadecimal dos dígitos de π, exceto pelo número 3 inicial;
- executar operações de OU-EXCLUSIVO (XOR) entre P_1 e os primeiros 32 bits da chave, P_2 e os 32 bits seguintes, e assim repetidamente, voltando ao início da chave quando necessário, até que todos os bits das variáveis $P_1, P_2, \ldots, P_{18}$ tenham sofrido operações com os bits da chave;
- encriptar uma seqüência de bits 0 com o algoritmo, armazenando o resultado em P_1 e P_2;
- encriptar a saída do passo anterior, armazenando o resultado em P_3 e P_4;
- repetir este processo até substituir todos os valores das variáveis $P_1, P_2, \ldots, P_{18}$ e das caixas-S.

São executadas um total de 521 interações para gerar todas as sub-chaves. Em algumas aplicações estas podem ser armazenadas evitando a necessidade de nova geração.

Todas as operações executadas pelo algoritmo são OU-EXCLUSIVO (XOR) ou adição de números de 32 bits, tornando sua execução bastante eficiente.

II.2.2 Criptografia Assimétrica ou de Chave Pública

Este tipo de criptografia faz uso de duas chaves criptográficas, associadas a uma certa entidade: uma privada que deve ser mantida em sigilo, e uma pública que pode ser livremente distribuída.

Dados encriptados com uma chave somente podem ser decriptados com a outra. Utiliza-se portanto a chave publica na criptografia de dados que devem permanecer em sigilo e somente poderão ser acessados pelo detentor da chave privada. Uma vez que dados encriptados com a chave privada podem ser acessados com o uso da chave pública, que é de conhecimento geral, esta operação não permite o suporte a privacidade, mas sim a autenticação, pois somente o detentor da chave privada pode gerar um criptograma válido. Devido a este fato, é comum referir-se a operação de encriptação com chave privada como assinatura da mensagem e a decriptação como verificação desta assinatura. O relacionamento entre estas chaves é ilustrado na Figura II.10.

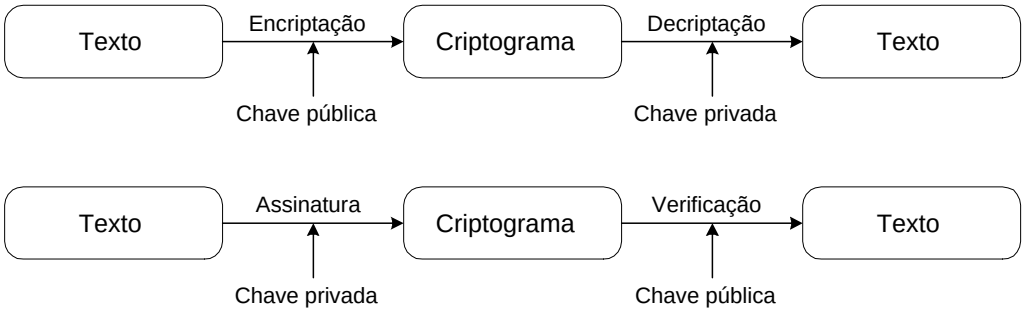

Figura II.10: Criptografia assimétrica ou de chave pública.

Duas notações, apresentadas na Tabela II.1, são usualmente utilizadas na representação das operações realizadas por este tipo de algoritmo.

Tabela II.1: Notações para operações de criptografia assimétrica.

Notação 1	Notação 2	Operação
$C = E(M)$	$C = E_{Kpubl}(M)$	Encriptação com chave pública
$M = D(C)$	$M = D_{Kpriv}(C)$	Decriptação com chave privada
$S = D(M)$	$S = E_{Kpriv}(M)$	Assinatura com chave privada
$M = E(S)$	$M = D_{Kpubl}(S)$	Verificação de assinatura com chave privada

Os algoritmos deste tipo de criptografia, devido às operações que os compoem, demandam mais tempo para serem processados que os de criptografia simétrica, tornando ineficiente

seu uso na encriptação de longas seqüências de dados. Uma técnica muito utilizada para contornar esta deficiência é a encriptação dos dados com um algoritmo de criptografia simétrica, seguida pela encriptação da chave utilizada com um algoritmo de criptografia assimétrica. Os dois criptogramas atuam então como se fossem apenas um. Esta solução une a maior velocidade de processamento da criptografia simétrica à flexibilidade proporcionada pela criptografia assimétrica.

Ambos criptografia e autenticação ocorrem sem o compartilhamento de chaves privadas. Uma entidade utiliza somente sua própria chave privada e a chave pública da outra entidade. Qualquer entidade pode enviar mensagens encriptadas ou verificar a assinatura de uma mensagem recebida utilizando apenas chaves públicas. Apenas entidades que possuam a chave privada podem decriptar ou assinar estas mensagens.

RSA

O RSA (Rivest-Shamir-Adleman), inventado por Ron Rivest, Adi Shamir e Leonard Adleman em 1977, apresenta a vantagem de permitir a utilização de chaves criptográfica de tamanho variável, permitindo a utilização de chaves grandes, favorecendo a segurança, ou pequenas, favorecendo a eficiência, sendo 256 bits o tamanho geralmente adotado. O tamanho do bloco de dados a ser encriptado é também variável, desde que seu tamanho seja menor que o da chave utilizada. O criptograma gerado é sempre do mesmo tamanho que a chave.

Para a geração das chaves criptográficas são escolhidos dois números primos p e q (geralmente com tamanho em torno de 256 bits cada). Estes números devem ser mantidos em sigilo, sendo utilizados na geração de um número n que é o produto entre p e q. Uma vez conhecido n, a obtenção de p e q é muito difícil de ser calculada, pois envolve a fatoração de um número muito grande em números primos, uma atividade muito difícil de ser realizada com a atual tecnologia.

Para a geração da chave pública, é escolhido em número e tal que e e $\phi(n)$ são primos entre si, onde, conforme apresentado no Apêndice A, $\phi(n) = (p-1) \cdot (q-1)$. A chave pública é então $\langle e, n \rangle$.

A chave privada é $\langle d, n\rangle$, onde d é o inverso multiplicativo de e mod $\phi(n)$.

A encriptação de um texto $M(< n)$, é realizada utilizando a chave pública para calcular o criptograma $C = M^e$ mod n, que só pode ser decriptado com o uso da chave privada, obtendo-se $M = C^d$ mod n.

A assinatura é realizada de forma similar, utilizando-se a chave privada e obtendo-se a assinatura $S = M^d$ mod n, que pode ser verificada pela recuperação do texto $M = S^e$ mod n.

O funcionamento deste algoritmo pode ser comprovado observando as operações matemáticas mod n envolvidas. Sendo $n = p \cdot q$ e $\phi(n) = (p-1) \cdot (q-1)$ conhecidos, foram escolhidos d e e de modo que $d \cdot e = 1$ mod $\phi(n)$. Assim, para qualquer x, decorre que $x^{d \cdot e} = x$ mod n. Uma operação de encriptação seguida pela decriptação consiste portanto de $(M^e)^d = M^{e \cdot d} = M$, obtendo-se o texto original. Operação similar ocorre na assinatura seguida pela sua verificação: $(M^d)^e = M^{d \cdot e} = M$.

A segurança deste algoritmo baseia-se no princípio de que a fatoração de números muito grandes, na ordem de 512 bits, é uma operação difícil que demanda alto tempo de processamento. Como já foi mencionado, a segurança pode ser aumentada com o aumento do tamanho das chaves criptográficas utilizadas.

A capacidade de fatoração rápida tornaria simples a tarefa de decifrar criptogramas RSA sem o conhecimento das chaves criptográficas utilizadas. Dada uma chave pública $\langle e, n\rangle$, tem-se que d é o inverso multiplicativo de e mod $\phi(n)$, conforme apresentado. Deve-se então descobrir $\phi(n) = (p-1) \cdot (q-1)$. A dificuldade consiste em identificar p e q a partir de n, o que envolve a fatoração de n. É valido lembrar que quando se dá a geração das chaves, p e q são escolhidos e multiplicados para obter n, o que consiste em uma operação muito mais simples.

Diffie-Hellman

Uma característica curiosa deste algoritmo, proposto por W. Diffie e M. E. Hellman em 1976, é o fato de não serem realizadas operações de encriptação e assinatura de mensagens. Este algoritmo torna possível a determinação de uma chave compartilhada entre duas entidades

que desejem trocar dados de forma segura mas que só possam trocar dados por meio de um canal público. Sua função é portanto o estabelecimento de uma chave secreta. Uma vez que esta tenha sido estabelecida, algum algoritmo de criptografia simétrica pode ser utilizado para promover a troca segura de dados.

Para o estabelecimento de uma chave compartilhada entre as entidades A e B são seguidos os seguintes passos:

- as entidades devem concordar[1] na utilização de dois números escolhidos randomicamente p e g, tais que p é primo, geralmente de 512 bits, e $g < p$;
- cada entidade escolhe um número aleatório, em geral com cerca de 512 bits, que deve permanecer em sigilo: S_A e S_B;
- as entidades calculam $T_A = g^{S_A} \bmod p$ e $T_B = g^{S_B} \bmod p$;
- A envia T_A para B e B envia T_B para A;
- A calcula ${T_B}^{S_A} \bmod p$ e B calcula ${T_A}^{S_B} \bmod p$.

Os resultados destas duas operações são iguais, pois ${T_B}^{S_A} = (g^{S_B})^{S_A} = g^{S_B \cdot S_A} = g^{S_A \cdot S_B} = (g^{S_A})^{S_B} = {T_A}^{S_B}$, correspondendo à chave a ser compartilhada.

A segurança deste algoritmo reside na dificuldade do cálculo de S_A e S_B a partir de T_A e T_B, necessários à obtenção de $g^{S_A \cdot S_B}$. O cálculo de S a partir de $g^S \bmod p$ é denominado logaritmo discreto, sendo de difícil computação.

Embora não seja um algoritmo de criptografia, e sim para o estabelecimento de uma chave secreta, este algoritmo costuma ser categorizado como tal devido ao fato de S atuar como uma chave secreta e T como uma chave pública, sendo alteradas a cada estabelecimento de chave compartilhada.

Este algoritmo apresenta a desvantagem de não prover autenticação, tornando possível o estabelecimento de uma chave compartilhada com uma entidade que esteja pretendendo ser

[1] Todas as informações trocadas entre as entidades fazem uso do meio inseguro, sendo portanto de conhecimento público

outra. Esta deficiência torna possível um ataque conhecido como Homem Intermediário (*Men-In-The-Middle Attack*), apresentado na Seção II.7.4.

II.2.3 Enciptação Encadeada

ECB

Observando os algoritmos criptográficos apresentados é possível perceber a limitação imposta ao tamanho dos dados de entrada quando estes são mensagens muito extensas ou correspondem a um fluxo contínuo de dados. A solução mais intuitiva é denominada ECB (*Electronic Code Book* ou Livro de Código Eletrônico) e corresponde à divisão da mensagem em diversos blocos de igual tamanho que são encriptados de forma independente e transmitidos. O receptor concatena o resultado da decriptação de cada criptograma, obtendo a mensagem, conforme ilustrado pela Figura II.11.

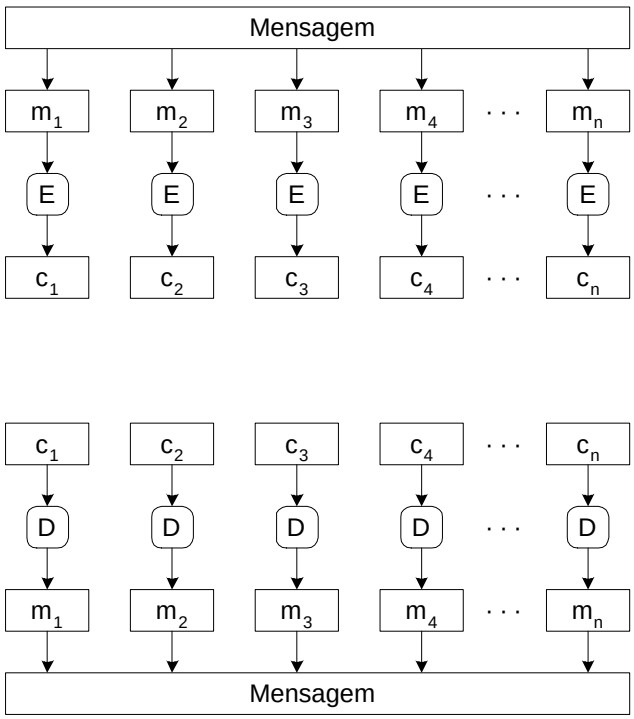

Figura II.11: Encriptação e decriptação com ECB.

Esta solução apresenta uma vulnerabilidade devido ao fato de os blocos de entrada iguais apresentarem criptogramas também iguais, o que pode fornecer informações a um atacante sobre os dados transmitidos. É possível ainda a substituição indevida de blocos. Esta vulnerabilidade torna possível, por exemplo, um ataque no qual o atacante descobre os blocos responsáveis por armazenar os valores dos salários dos funcionários de uma empresa e troca dois blocos entre si, trocando os salários de dois funcionários, ou substitui todos os blocos por um só, alterando todos os salários para um só valor. A mensagem recebida seria válida, dada a independência entre os blocos, e estas alterações só seriam detectadas caso o conteúdo da mensagem fosse analisado, por exemplo, pelo contador da empresa.

CBC

Essa vulnerabilidade do ECB, descrita na seção precedente, é eliminada com a utilização de uma técnica conhecida como CBC (*Cipher Block Chaining* ou Encadeamento de Blocos de Cifra), que consiste na geração de criptogramas de modo que estes dependam não somente do bloco correspondente da mensagem, mas também do criptograma obtido com o bloco anterior, que são combinados por meio de uma operação de OU-EXCLUSIVO (XOR). Como o primeiro bloco não possui um bloco anterior, é utilizado no seu lugar um vetor de inicialização VI com conteúdo gerado randomicamente, que é enviado ao receptor junto com os criptogramas. A Figura II.12 apresenta como são realizadas as operações de encriptação e decriptação com CBC.

Havendo a dependência entre o criptograma de um bloco e os anteriores, a substituição de qualquer bloco é mais facilmente detectada. A escolha de valores randômicos para VI impede ainda ataques do tipo Mensagem Escolhida (*chosen plaintext*), apresentado na Seção II.7, pois a encriptação de uma mesma mensagem mais de uma vez gera criptogramas diferentes.

Vale notar que a menos que se utilize alguma técnica de detecção de erros, é sempre possível a um atacante alterar o conteúdo de uma mensagem por meio de alteração de seu criptograma. É importante no entanto que ele não possua controle sobre estas alterações, sempre alterando o criptograma sem saber os efeitos que isto exerce sobre a mensagem. Informações sobre suporte a integridade são apresentados na Seção II.5.

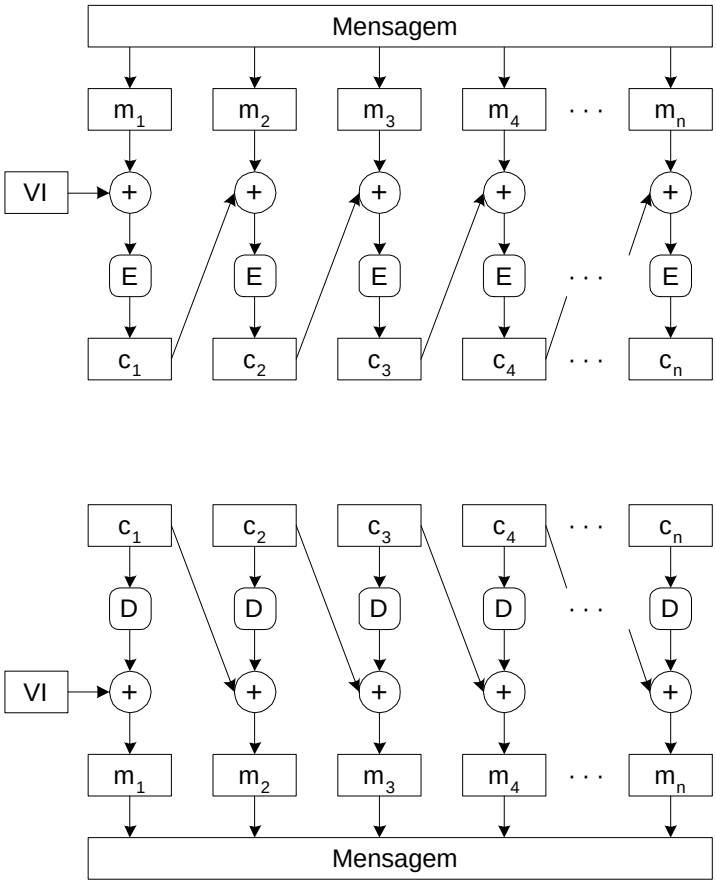

Figura II.12: Encriptação e decriptação com CBC.

II.3 *Hash* ou *Message Digest*

Os termos *hash* e *message digest* são utilizados para identificar funções com características tais que:

- mapeiem um valor de entrada em um valor de saída;
- não haja qualquer correlação entre os valores de entrada e saída;
- recebam valores de entrada de tamanhos arbitrários e mapeiem-nos em valores de saída de tamanho fixo.

Uma função *hash* é dita segura se for computacionalmente muito difícil:

1. descobrir a entrada correspondente a uma dada saída;
2. descobrir alguma entrada que seja mapeada em uma dada saída;
3. descobrir duas entradas que sejam mapeadas em uma mesma saída.

Este tipo de função atua portanto sobre mensagens de tamanho arbitrário gerando valores de tamanho fixo que são denominados valores *hash* ou *message digests*.

Dadas estas características, várias mensagens são mapeadas em valores *hash* iguais. É importante que os itens 2 e 3 apresentados anteriormente sejam atendidos. Dada uma função *hash* que gere saídas de k bits, há uma probabilidade de aproximadamente 0.5^k de se encontrar uma mensagem que satisfaça ao item 2, e uma probabilidade de aproximadamente $0.5^{k/2}$ de se encontrar duas mensagens que satisfaçam ao item 3. O grau de segurança da função é portanto dependente do tamanho da saída, sendo esta mais segura a medida que sejam utilizados mais bits na representação dos valores *hash*.

Algumas aplicações que utilizam este tipo de função são:

- geração e armazenamento em local seguro de valores *hash* de programas e arquivos de modo a detectar alterações indevidas a estes programas e arquivos;
- autenticação por senha — em sistemas que autentiquem o acesso a usuário por meio de senhas, muitas vezes não é desejado o armazenamento dessas senhas. Funções *hash* são utilizadas de modo que somente o valor *hash* é armazenado. Para autenticar um usuário o sistema recebe a senha por ele digitada, calcula o valor *hash* correspondente e o compara ao valor armazenado autorizando o usuário a acessar o sistema caso este valores sejam iguais (ver Seção II.3.5);
- geração de códigos de integridade de mensagem (ver Seção II.5);
- encriptação — funções *hash* podem ser utilizadas para gerar um Bloco de Uma Só Utilização (*One-Time Pad*) que é utilizado para encriptação e decriptação (ver Seção II.2.1). A geração deste bloco, constituído por diversas partes b_i, se dá com base em uma chave K

compartilhada entre as entidades, com a seguinte regra de formação: $b_i = \begin{cases} MD(K|VI) & , i = 1 \\ MD(K|b_{i-1}) & , i \geq 2 \end{cases}$
.

O desenvolvimento do RSA tornou possível a assinatura digital de mensagens. No entanto, devido à lentidão deste algoritmo quando aplicado a mensagens longas, surgiu a idéia de gerar uma versão compactada da mensagem a ser assinada, aplicando-se então o RSA sobre esta. A geração desta versão compactada foi chamada de *message digest*[2]. Os mais conhecidos algoritmos de função *hash* ou *message digest* são:

- MD — algoritmo proprietário, nunca foi publicado;
- MD2 — documentado na RCF 1319;
- MD3 — tornou-se defasado pelo MD4 antes mesmo de ser publicado;
- MD4 — documentado na RCF 1320;
- MD5 — documentado na RCF 1321;
- SHS — similar ao MD5, sendo no entanto mais seguro.

Serão apresentados a seguir os algoritmos MD2, MD4, MD5 e SHS.

II.3.1 MD2

Este algoritmo recebe como entrada uma mensagem de tamanho arbitrário múltipla de 8 bits e produz como saída valores de 128 bits. O algoritmo é constituído por três etapas:

1. enchimento —

[2]Esta expressão na língua inglesa tem o significado de “uma versão mais resumida”.

A mensagem de entrada sofre uma concatenação com bits de enchimento de modo a torná-la múltipla de 16 bytes. Isto ocorre mesmo para mensagens que já atendam a esta condição, situação na qual recebem 16 bytes de enchimento. O enchimento consiste de bytes contendo um valor r, $1 \leq r \leq 16$, que é o número de bytes adicionados;

2. *checksum* —

   A mensagem com enchimento, possuindo $k \times 16$ bytes, é utilizada para gerar um valor de *checksum* de 16 bytes que é concatenado a essa. Seu valor inicial é 0 e seu cálculo envolve operações com um byte por vez. Cada byte da mensagem com enchimento é lido seqüencialmente e responsável pela alteração do $(n \text{ mod } 16)$-ésimo byte do *checksum*, sendo cada byte deste atualizado k vezes. Cada atualização obedece à relação

   $$C_{n \text{ mod } 16} \leftarrow C_{n \text{ mod } 16} \oplus f_s(C_{n-1 \text{ mod } 16} \oplus M_n) \text{ ,}$$

   onde C_i é o i-ésimo byte do *checksum*, M_i é o i-ésimo byte da mensagem com enchimento e f_s é uma função de substituição;

3. cálculo do valor *hash* —

   Este cálculo é executado em $k+1$ estágios que processam a mensagem com *checksum* em blocos de 16 bytes. Cada bloco é utilizado para gerar um novo bloco de 48 bytes segundo a relação

   $$b_i = \begin{cases} d_i & , \ 0 \leq i \leq 15 \\ m_{i-16} & , \ 16 \leq i \leq 31 \\ d_{i-32} \oplus m_{i-32} & , \ 32 \leq i \leq 47 \end{cases} ,$$

   para $i = 0, 1, \ldots, 47$, onde

   - b_i é o i-ésimo byte do novo bloco;
   - m_i é o i-ésimo byte da mensagem com enchimento e *checksum*;
   - d_i é o i-ésimo byte do valor *hash* em processamento, inicializado em 0.

   São executados 18 passos sobre o bloco de 48 bits constituídos por

$$b_i = b_i \oplus f_s(b_{n-1}),$$

para $i = 0, 1, \ldots, 47$, onde um byte extra B_{-1} inicializado em 0 é alterado ao fim de cada passo, recebendo o valor $(b_{47} + N_p)$ mod 256, onde N_p é o número do passo.

O valor *hash* obtido ao final de cada estágio é dado por $b_0, b_1, \ldots, b_{15}$, sendo estes valores atribuídos a $d_0, d_1, \ldots, d_{15}$. O valor *hash* final é aquele obtido ao final do último estágio.

II.3.2 MD4

Criado por Ron Rivest, este algoritmo foi projetado de modo a executar operações com palavras de 32 bits, otimizando sua execução em processadores de 32 bits. As mensagens de entrada podem possuir qualquer tamanho, sendo as saídas sempre de 128 bits (4 palavras).

A mensagem de entrada é concatenada a bits de enchimento de modo a torná-la múltipla de 512 bits. Este enchimento é constituído por um bit '1' seguido por um número suficiente de bits 0, e por 64 bits que representam o tamanho em número de bits mod 2^{64} da mensagem original.

A mensagem é então processada em blocos de 512 bits (16 palavras), sendo o valor de saída inicializado em um valor fixo e modificado a cada estágio. Cada estágio consiste de três rodadas sobre o bloco utilizado. O valor final de cada estágio é igual ao valor final da terceira rodada somado ao valor inicial no estágio. As palavras do valor *hash* são denominadas d_0, d_1, d_2, d_3 (inicializadas respectivamente como 01234567_{16}, $89ABCDEF_{16}$, $FEDCBA98_{16}$ e 76543210_{16}) e as do bloco tratado $m_0, m_1, \ldots, m_{15}$.

As modificações sofridas em cada rodada são:

- 1^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + F(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_i) \lrcorner S_1(i \wedge 3)$

- 2^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + G(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_{X(i)} + K_1) \lrcorner S_2(i \wedge 3)$

- 3^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + H(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_{R(i)} + K_2) \lrcorner S_3(i \wedge 3)$,

para $i = 0, 1, \ldots, 15$, onde:

- $F(x, y, z) = (x \wedge y) \vee (\overline{x} \wedge z)$;
- $G(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$;
- $H(x, y, z) = x \oplus y \oplus z$;
- $S_1(i) = 3 + 4i$;
- $S_2(0) = 3,\ S_2(1) = 5,\ S_2(2) = 9,\ S_2(3) = 13$;
- $S_3(0) = 3,\ S_3(1) = 9,\ S_3(2) = 11,\ S_3(3) = 15$;
- $K_1 = \lfloor 2^{30}\sqrt{2} \rfloor$: $5a827999_{16}$;
- $K_2 = \lfloor 2^{30}\sqrt{3} \rfloor$: $6ed9eba1_{16}$;
- $X(i) = 4i - 15\lfloor i/4 \rfloor$;
- $R(i) = 8i - 12\lfloor i/2 \rfloor - 6\lfloor i/4 \rfloor - 3\lfloor i/8 \rfloor$;
- $\lfloor x \rfloor$ corresponde ao maior inteiro não maior que x
- $\overline{x}$ corresponde à operação NÃO (NOT) aplicada a x;
- $x \wedge y$ corresponde à operação E (AND) binária entre x e y;
- $x \vee y$ corresponde à operação OU (OR) binária entre x e y;
- $x \oplus y$ corresponde à operação OU-EXCLUSIVO (XOR) binária entre x e y;
- $x+y$ corresponde à operação de soma entre x e y com o descarte do eventual bit excedente da soma (vai um);
- $x \lrcorner y$ corresponde à operação de desvio rotacionado a esquerda por y posições dos bits de x.

II.3.3 MD5

Também criado por Ron Rivest, este algoritmo é similar ao MD4, apresentando no entanto maior segurança e desempenho inferior.

Suas principais diferenças são a execução de quatro rodadas a cada estágio e a utilização de 64 constantes $T_1, T_2, \ldots, T_{64}$ dadas por $T_i = \lfloor 2^{32} |sen\ i| \rfloor$, ao invés das duas utilizadas pelo MD4.

As modificações sofridas em cada rodada são:

- 1^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + F(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_i + T_{i+1}) \lrcorner S_1(i \wedge 3)$

- 2^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + G(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_{(5i+1)\wedge 15} + T_{i+17}) \lrcorner S_2(i \wedge 3)$

- 3^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + H(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_{(3i+5)\wedge 15} + T_{i+33}) \lrcorner S_3(i \wedge 3)$

- 4^a rodada:

  $d_{(-i)\wedge 3} \leftarrow (d_{(-i)\wedge 3} + I(d_{(1-i)\wedge 3}, d_{(2-i)\wedge 3}, d_{(3-i)\wedge 3}) + m_{(7i)\wedge 15} + T_{i+49}) \lrcorner S_4(i \wedge 3)$,

para $i = 0, 1, \ldots, 15$, onde:

- $F(x, y, z) = (x \wedge y) \vee (\overline{x} \wedge z)$;

- $G(x, y, z) = (x \wedge z) \vee (y \wedge \overline{z})$;

- $H(x, y, z) = x \oplus y \oplus z$;

- $I(x, y, z) = y \oplus (x \vee \overline{z})$;

- $S_1(i) = 7 + 5i$;

- $S_2(i) = i(i+7)/2 + 5$;

- $S_3(0) = 4,\ S_3(1) = 11,\ S_3(2) = 16,\ S_3(3) = 23$;

- $S_4(i) = (i+3)(i+4)/2$.

II.3.4 SHS

O SHS (*Secure Hash Standard* ou Padrão de *Hash* Seguro), também conhecido como SHA (*Secure Hash Algorithm* ou Algoritmo de *Hash* Seguro) foi proposto pelo NIST (*National Institute of Standards and Technology* ou Instituto Nacional de Padrões e Tecnologia dos Estados Unidos) e gera uma saída de 160 bits, sendo definido para mensagens de qualquer tamanho até 2^{64} bits. Esta aparente limitação não se revela em um problema, por se tratar de uma quantidade de dados extremamente grande para os padrões atuais[3].

Este algoritmo é semelhante ao MD4 e ao MD5, sofrendo inclusive o mesmo tipo de enchimento de bits. São realizadas cinco rodadas a cada bloco de dados de 512 bits, o que o torna mais seguro e em contrapartida mais lento.

A saída de 160 bits é composta por cinco palavras de 32 bits $d_0, d_1, \ldots, d_4$, que apresentam inicialmente valores predefinidos (respectivamente 67452301_{16}, EFCDAB89_{16}, 98BADCFE_{16}, 10325476_{16} e C3D2E1F0_{16}).

Ao início de cada estágio, o bloco de 512 bits é utilizado para gerar um novo bloco de 5×512 bits, correspondendo a 80 palavras. Este bloco é construído obedecendo à relação

$$W_i = \begin{cases} m_i & ,\ 0 \leq i \leq 15 \\ m_{i-3} \oplus m_{i-8} \oplus m_{i-14} \oplus m_{i-16}\ ^{\lll 4} & ,\ 16 \leq i \leq 79 \end{cases},$$

para $i = 0, \ldots, 79$.

As modificações sofridas em cada rodada, baseadas no novo bloco de 80 palavras, são:

- $d_0 \leftarrow d_4 + (d_0 \lll 5) + W_i + K_i + f(i, d_1, d_2, d_3)$

[3] 2^{64} bits = 2.1E6 TB.

- $d_1 \leftarrow d_0$
- $d_2 \leftarrow d_1 \lrcorner 30$
- $d_3 \leftarrow d_2$
- $d_4 \leftarrow d_3$,

para $i = 0, \ldots, 79$, onde:

- $K_t = \begin{cases} \lfloor 2^{30}\sqrt{2} \rfloor = \text{5a827999}_{16} & , \ 0 \leq i \leq 19 \\ \lfloor 2^{30}\sqrt{3} \rfloor = \text{6ed9eba1}_{16} & , \ 20 \leq i \leq 39 \\ \lfloor 2^{30}\sqrt{5} \rfloor = \text{8f1bbcdc}_{16} & , \ 40 \leq i \leq 59 \\ \lfloor 2^{30}\sqrt{10} \rfloor = \text{ca62c1d6}_{16} & , \ 60 \leq i \leq 79 \end{cases}$;

- $f(t, d_1, d_2, d_3) = \begin{cases} (d_1 \wedge d_2) \vee (\overline{d_1} \wedge d_3) & , \ 0 \leq i \leq 19 \\ d_1 \oplus d_2 \oplus d_3 & , \ 20 \leq i \leq 39 \\ (d_1 \wedge d_2) \vee (d_1 \wedge d_3) \vee (d_2 \wedge d_3) & , \ 40 \leq i \leq 59 \\ d_1 \oplus d_2 \oplus d_3 & , \ 60 \leq i \leq 79 \end{cases}$.

II.3.5 *Hash* por meio de criptografia simétrica

É possível a implementação de funções *hash* por meio da utilização de algoritmos de criptografia simétrica, o que consiste, conforme ilustrado pela Figura II.13 na encriptação de um conjunto de dados de entrada pré-determinado sob o comando de uma chave criptográfica obtida a partir dos dados dos quais se deseja obter o valor *hash*. Esta técnica é utilizada, por exemplo, nos sistemas operacionais do tipo UNIX para geração dos valores *hash* das senhas dos usuários.

Sistema de autenticação do UNIX

Os sistemas do tipo UNIX utilizam uma versão modificada do DES para calcular os valores *hash* das senhas dos usuários. A razão pela qual se utiliza esta versão modificada do DES é dificultar que equipamentos de *hardware* desenvolvidos para reverter o DES sejam utilizados

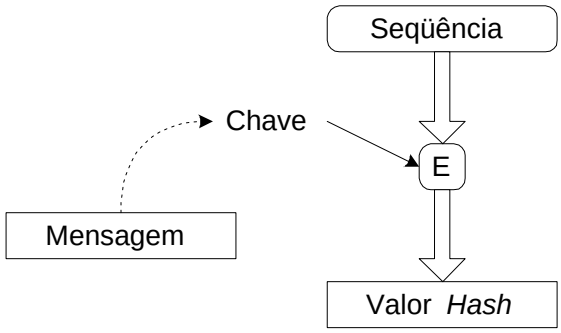

Figura II.13: *Hash* com criptografia simétrica.

na tentativa de obter a senha a partir do valor *hash*. A chave do algoritmo é obtida utilizando-se a representação em ASCII de 7 bits dos primeiros 8 caracteres da senha. Os demais caracteres são ignorados, não tendo qualquer validade. Esta chave é utilizada para encriptar um bloco de dados com todos os bits em 0 gerando o valor *hash* da senha do usuário.

O funcionamento do algoritmo (ver Seção II.2.1) é influenciado por um valor de 12 bits denominado sal (*salt*) que modifica a função $f(R_i, K_i)$, estabelecendo quais bits devem ser duplicados na expanção de R_i de 32 para 48 bits. Este valor é armazenado em arquivo junto com o valor *hash* obtido e consultado quando é necessário autenticar um usuário.

II.3.6 *Hash* de mensagens longas

Em sistemas nos quais se deseje que a função *hash* atue sobre todo o conteúdo de uma mensagem com tamanho maior que o da chave de k bits, pode-se dividir a mensagem em blocos de k bits, que são utilizados como chaves para sucessivas encriptações do conjunto de dados pré-determinado. O resultado final é o valor *hash* correspondente.

Caso o número de bits do valor *hash* obtido seja considerado pequeno, é possível gerar valores *hash* com número de bits n vezes maior executando-se o algoritmo n vezes, cada uma atuando sobre um valor pré-determinado diferente, e concatenando-se os n resultados.

II.4 Privacidade

O termo privacidade refere-se ao impedimento do acesso por parte de entidades não autorizadas às informações trocadas em uma comunicação.

O uso de criptografia simétrica permite o suporte a privacidade entre as entidades que compartilhem uma chave criptográfica. Com o uso de criptografia assimétrica, a pricacidade é garantida pelo envio de mensagens encriptadas com a chave pública do destinatário, pois somente este é capaz de decriptá-la. A encriptação de mensagens é normalmente suficiente para garantir sua privacidade, desde que as chaves utilizadas sejam mantidas em sigilo. A privacidade é ameaçada por ataques de criptoanálise, por força-bruta, ou do tipo Homem Intermediário. Este ataques são apresentados na Seção II.7.

II.5 Integridade

O termo integridade refere-se à garantia de que uma mensagem recebida corresponda exatamente à que foi enviada, sem sofrer quaisquer alterações indevidas.

A simples utilização de algoritmos criptográficos não garante a integridade de mensagens devido à possibilidade do criptograma sofrer alterações, intencionais ou não. Analisando os algoritmos atualmente utilizados, é possível perceber que a qualquer conjunto de bits de tamanho apropriado pode ser aplicada uma função de decriptação, podendo a mensagem obtida ser ou não válida. Em situações nas quais a integridade das mensagens recebidas seja necessária, devem ser adotadas medidas que garantam esta integridade.

Uma possibilidade é a utilização de códigos de detecção e/ou correção de erros, como por exemplo bits de paridade, aplicados a mensagens antes da encriptação e após a decriptação.

Outra possibilidade é a geração de uma MIC (*Message Integrity Code* ou Código de Integridade de Mensagem), que consiste em um bloco de dados capaz de atestar a integridade de uma mensagem. Uma técnica aplicada a mensagens constituídas por n blocos e que devem ser transmitidas sem criptografia consiste em computar o CBC da mensagem (vide Seção II.2.3),

de modo a obter o último bloco c_n que é então denominado Resíduo CBC. Este bloco é enviado junto com a mensagem, e a integridade desta pode ser verificada repetindo-se o mesmo procedimento e verificando se o resíduo CBC obtido confere com o que foi recebido.

É intuitivo pensar que o envio do criptograma junto com o resíduo CBC garanta a privacidade e a integridade da mensagem. Isto no entanto não se comprova, pois uma vez que o criptograma seja decriptado pelo receptor, o resíduo CBC obtido é sempre igual ao último bloco do criptorama recebido. Logo, qualquer alteração indevida ao criptograma não seria descoberta desde estas informações fossem iguais.

A adição do resíduo CBC à mensagem antes da encriptação também não fornece suporta a integridade pois a última etapa da encriptação consistiria no cálculo de $c_{n+1} = c_n \oplus residuo$, que é sempre igual a 0, pois $residuo = c_n$. Uma forma de contornar este problema é a utilização de duas chaves criptográficas: uma para calcular o resíduo e outra para encriptar a mensagem concatenada com o resíduo. Uma das chaves pode ser gerada a partir da outra, através de alguma regra de formação, eliminando a necessidade de gerenciamento de duas chaves.

Outra solução consiste em concatenar à mensagem, antes da encriptação, o seu valor *hash*. O receptor após a decriptação calcula este valor e compara com o recebido verificando a integridade da mensagem.

II.6 Autenticação

O termo autenticação refere-se à garantia de que uma entidade seja realmente aquela que afirma ser. O processo de autenticação geralmente ocorre por meio de protocolos de comunicação que fazem uso de algoritmos criptográficos ou de funções *hash* de modo que ao fim do protocolo as partes envolvidas possuam conhecimento sobre a identidade das demais.

São apresentados a seguir os principais tipos de protocolos de autenticação e o conceito de certificado ou assinatura digital.

II.6.1 Autenticação baseada em criptografia simétrica

Para este tipo de autenticação é necessário que as duas entidades a serem mutuamente autenticadas A e B compartilhem uma chave secreta K_{AB}. O protocolo conhecido como desafio-resposta (*challenge-response*) baseia-se na idéia de uma entidade enviar uma mensagem M (o desafio) à outra, que deve responder com $K_{AB}(M)$. Se este criptograma estiver correto, a entidade que o enviou terá provado sua identidade.

Dois exemplos de protocolos baseados neste conceito são ilustrados na Figura II.14, sendo o segundo uma versão mais compacta, e, conforme apresentado na Seção II.7.3, vulnerável a um ataque denominado Ataque por Reflexão (*Reflection Attack*). Como pode ser observado, a autenticação baseia-se no conhecimento da chave secreta K_{AB}, devendo esta ser conhecida apenas por A e B.

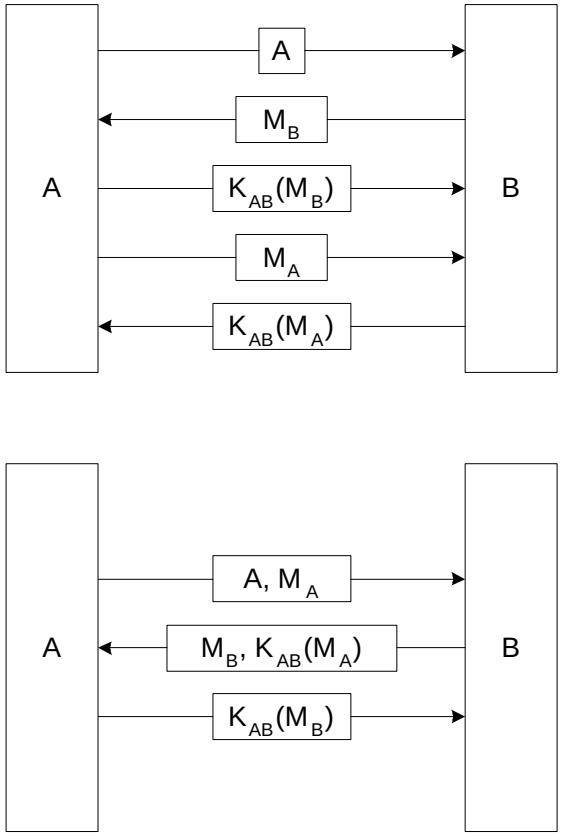

Figura II.14: Protocolos de autenticação baseados em criptografia simétrica.

II.6.2 Autenticação baseada em criptografia assimétrica

Criptografia assimétrica pode ser utilizada para autenticação mútua entre duas entidade A e B conforme ilustrado pela Figura II.15. Assumindo que ambas as entidade conhecem a chave pública da outra, A utiliza a chave pública de B para enviar-lhe sua identidade e uma mensagem de desafio M_A. A entidade B utiliza então a chave pública de A para enviar-lhe A mensagem M_A, uma mensagem de desafio M_B e uma proposta para a chave de seção K_S. Ao decriptar esta mensagem e verificar que M_A está correta, A tem a certeza da identidade de B, pois somente B pode ter obtido acesso a M_A. A utiliza então a chave K_S para enviar M_B para B. Ao receber corretamente esta informação, B tem a certeza da identidade de A, pois somente A pode ter obtido acesso a K_S.

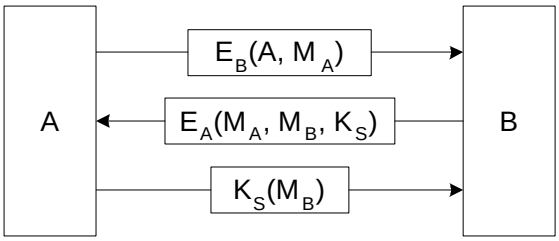

Figura II.15: Protocolo de autenticação baseado em criptografia assimétrica.

Como depende do conhecimento da chaves públicas, este protocolo é vulnerável à falsificação dessas chaves. É importante que haja uma forma segura de torná-las conhecidas. A simples troca de chaves públicas no início do protocolo torna-o vulnerável a ataques do tipo Homem Intermediário (*Men-in-the-Middle*), apresentado na Seção II.7.4.

Este tipo de autenticação é geralmente utilizado para que duas entidade se autentiquem e concordem em uma chave secreta K_S, que é então utilizada em conjunto com algum algoritmo de criptografia simétrica para a troca de mensagens.

II.6.3 Autenticação baseada em um Centro de Distribuição de Chaves

Este tipo de autenticação apresenta como vantagem o fato de não ser necessário o compartilhamento de informações com cada entidade que se deseja estbelecer comunicação. Esta abordagem introduz o conceito de KDC (*Key Distribution Center* ou Centro de Distribuição de Chave), que é uma entidade confiável capaz de atestar a identidade de seus membros. Deste modo, para estabelecer comunicação com n entidades não é mais necessário o conhecimento de n chaves, mas de apenas de uma compartilhada com o KDC.

A Figura II.16 ilustra o estabelecimento de uma comunicação segura iniciada por A entre A e B. A envia ao KDC uma mensagem contendo sua identidade, e um criptograma protegido com sua chave contendo a identidade de B e a chave de seção K_S que deseja utilizar. O KDC envia então para B uma mensagem encriptada contendo a identidade de A e a chave a ser utilizada na comunicação. Assumindo-se o KDC tratar-se de uma entidade confiável, A e B podem ser mutuamente autenticados. A autenticação de A perante o KDC é implícita, pois somente A pode gerar um criptograma válido, o mesmo ocorrendo entre o KDC e B.

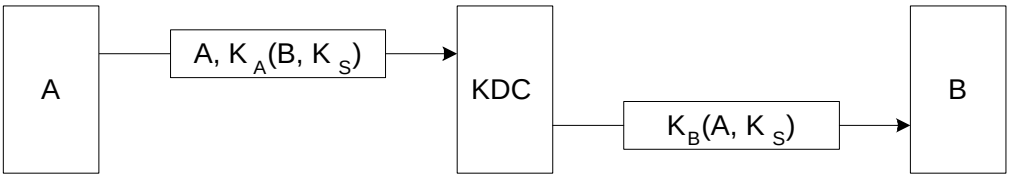

Figura II.16: Autenticação com KDC.

Este protocolo é vulnerável a um ataque conhecido como Ataque por Reenvio (*Replay Attack*), apresentado na seção II.7.5.

II.6.4 Assinatura Digital ou Certificado Digital

O conceito de assinatura ou certificado digital surgiu da necessidade de se reproduzir em documentos digitais a autenticação tradicionalmente garantida pela assinatura de documentos em papel. Assinaturas Digitais devem portanto atender às seguintes condições:

1. o receptor deve ser capaz de verificar a identidade do emissor e a autenticidade da mensagem;
2. o emissor não deve ser capaz de negar o envio e a autenticidade da mensagem;
3. não deve ser possível a geração de uma assinatura válida por outra entidade.

Assinaturas com criptografia simétrica

Assinaturas digitais podem ser geradas com a utilização de criptografia simétrica desde que haja uma entidade confiável CA (*Certification Authority* ou Autoridade de Certificação) que compartilhe uma chave secreta com cada entidade a ser certificada.

A assinatura de uma mensagem M enviada de A para B se dá conforme ilustrado pela Figura II.17, onde R_A e t são respectivamente uma mensagem randomicamente gerada e o dia e hora (*timestamp*) da assinatura.

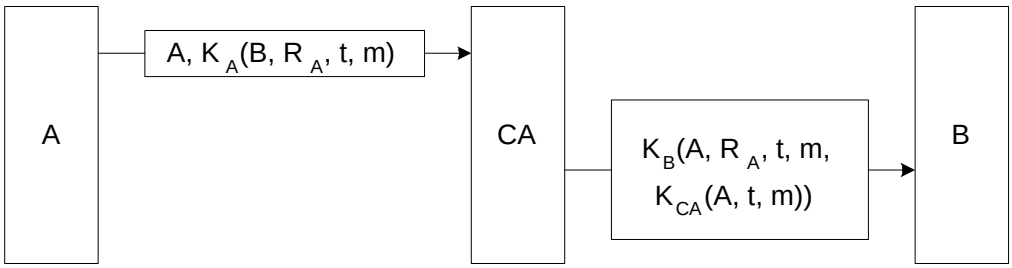

Figura II.17: Assinatura digital com criptografia simétrica.

A se autentica perante CA pelo envio de um criptograma que utiliza a chave K_A compartilhada entre eles. Similarmente, CA se autentica perante B pela utilização da chave K_B. Quando contestada por A a autenticidade da assinatura exibida por B, CA deve intervir decriptando $K_{CA}(A, t, M)$, o que comprova o conteúdo, o emissor da mensagem e o momento da assinatura.

Neste protocolo foram incluídos R_A e t com a finalidade de impedir ataque por reenvio (ver Seção II.7.5). Mensagens recebidas que apresentem um t mais antigo que um certo valor são

desconsideradas; para valores recentes de t, R_A é verificado, sendo desconsideradas mensagens com valores de R_A repetidos.

Uma desvantagem deste tipo de assinatura é o fato de a autoridade de certificação possuir acesso aos dados transmitidos, o que em algumas situações é indesejável. Esta deficiência pode ser contornada com a utilização de assinaturas com criptografia assimétrica.

Assinaturas com criptografia assimétrica

A geração de assinaturas digitais com uso de criptografia assimétrica faz uso das propriedades apresentadas na Seção II.2.2, conforme ilustrado pela Figura II.18.

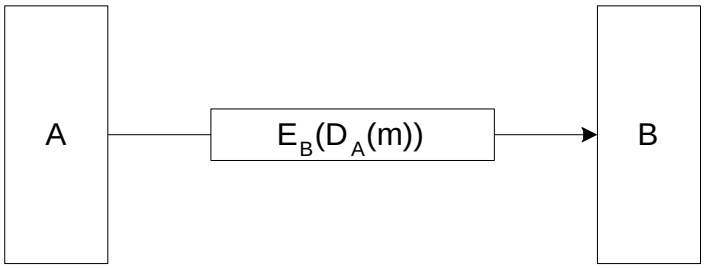

Figura II.18: Assinatura digital com criptografia assimétrica.

Para enviar uma mensagem M a B, A inicialmente a assina utilizando a função de decriptação e sua chave privada, obtendo $D_A(M)$. Este valor é então encriptado com a chave pública de B, obtendo-se $E_B(D_A(M))$, que é transmitido a B. Ao receber esta mensagem, B realiza as operações de decriptação com sua chave privada seguida pela verificação da assinatura com a função de encriptação e a chave pública de A, o que lhe permite ler a mensagem.

Este tipo de assinatura garante tanto a autenticidade do emissor (garantida por $D_A(M)$, que só pode ter sido encriptada por A), quanto a privacidade do conteúdo (garantida por $E_B(D_A(M))$, que só pode ser decriptada por B).

El Gamal

Este algoritmo, proposto por T. El Gamal, baseia-se no algoritmo Diffie-Hellman para tornar possível a assinatura digital de mensagens. As chaves utilizadas são as mesmas do algoritmo Diffie-Hellman. Para cada mensagem M é gerado um número randômico S_M e computado $T_M = g^{S_M} \bmod p$. É então utilizada alguma função de *hash* que seja de conhecimento geral para obter o valor *hash* d_M de M concatenada a T_M. Calcula-se então a assinatura $s = S_M + d_M \cdot S \bmod (p-1)$ e transmite-se $\langle M, T_M, s \rangle$.

Para a verificação da assinatura, o receptor calcula d_M e verifica se $g^s = T_M \cdot T^{d_M} \bmod p$. Isto se deve pois $g^s = g^{S_M + d_M \cdot S} = g^{S_M} \cdot g^{d_M \cdot S} = T_M \cdot T^{d_M} \bmod p$.

DSS

O DSS (*Digital Signature Standard* ou Padrão para Assinatura Digital), também conhecido como DSA (*Digital Signature Algorithm* ou Algoritmo para Assinatura Digital), foi proposto pelo NIST (*National Institute of Standards and Technology* ou Instituto Nacional de Padrões e Tecnologia dos Estados Unidos) para permitir a assinatura digital de mensagens. Este algoritmo baseia-se no algoritmo El Gamal, sendo otimizado para apresentar melhor performance. Ao invés de todas as operações serem realizadas mod p, sendo p um número primo de 512 bits, algumas são realizadas mod q, sendo q um número primo de 160 bits que divide $p-1$, tornando o processo de assinatura cerca de 3 vezes mais rápido.

A assinatura de uma mensagem M é realizada segundo os seguintes passos:

- gerar p (512 bits) e q (160 bits) primos tais que $p = k \cdot q + 1$, que serão de conhecimento público;
- gerar g tal que $g^q = 1 \bmod p$, que será de conhecimento público;
- obter um par de chaves pública e privada $\langle T, S \rangle$, tal que $S < q$ e $T = g^S \bmod p$;
- obter um par de chaves pública e privada $\langle T_M, S_M \rangle$ específico para a mensagem a ser assinada, tal que $T_M = (g^{S_M} \bmod p) \bmod q$ e calcular ${S_M}^{-1} \bmod q$;

- calcular o valor *hash* d_M da mensagem utilizando SHS (ver Seção II.3.4);
- calcular a assinatura $s = {S_M}^{-1}(d_M + S \cdot T_M)$;
- transmitir $\langle M, T_M, s \rangle$ (assume-se que T, p, q e g são livremente conhecidos).

Para a verificação da assinatura, são executados os seguintes passos:

- calcular o inverso mod q da assinatura (s^{-1});
- calcular d_M;
- calcular $x = d_M \cdot s^{-1}$ mod q;
- calcular $y = T_M \cdot X^{-1}$ mod q;
- calcular $z = (g^x \cdot T^y \bmod p) \bmod q$.

Se $z = T_M$, então a assinatura é válida. Segue a prova desta propriedade:

$$\left[\begin{array}{l} \text{Seja } v = (d_M + S \cdot T_M)^{-1} \bmod q \\ s^{-1} = ({S_M}^{-1}(d_M + S \cdot T_M))^{-1} = S_M(d_M + S \cdot T_M)^{-1} = S_M \cdot v \bmod q \\ x = d_M \cdot s^{-1} = d_M \cdot S_M \cdot v \bmod q \\ y = T_M \cdot X^{-1} = T_M \cdot S_M \cdot v \bmod q \\ z = g^x \cdot T^y = g^{d_M \cdot S_M \cdot v} \cdot g^{S \cdot T_M \cdot S_M \cdot v} = g^{(d_M + S \cdot T_M) S_M \cdot v} = g^{S_M} = T_M \; mod p \bmod q \end{array}\right].$$

Este algoritmo apresenta a desvantagem de necessitar de um par de chaves por mensagem, o que o torna vulnerável caso a assinatura de mais de uma mensagem seja realizada com um mesmo par de chaves $\langle T_M, S_M \rangle$. Como a assinatura de uma mensagem é dada por $s = {S_M}^{-1}(d_M + S \cdot T_M) \bmod q$, com o conhecimento de duas mensagens M e M' que compartilhem o mesmo par de chaves é possível calcular $(s_M - s_{M'})^{-1}(d_M - d_{M'}) \bmod q = S_M \bmod q$, descobrindo-se S_M. Uma vez que este valor seja conhecido, S pode ser descoberto por $(s_M \cdot S_M - d_M){T_M}^{-1} \bmod q = S \bmod q$, permitindo a assinatura indevida de mensagens.

II.7 Tipos de Ataque

São considerados ataques quaisquer tentativas em se burlar a segurança proporcionada pelos conceitos de privacidade, autenticação e integridade. O executor dessas tentativas é denominado atacante, podendo atuar de forma ativa ou passiva, conforme ilustrado pela Figura II.19. O atacante passivo apenas "escuta" as mensagens transmitidas (*eavesdrop*), buscando obter conhecimento sobre o seu conteúdo, enquanto o ativo as intercepta, altera seu conteúdo e retransmite, ou mesmo gera novas mensagens, pretendendo ser outra entidade.

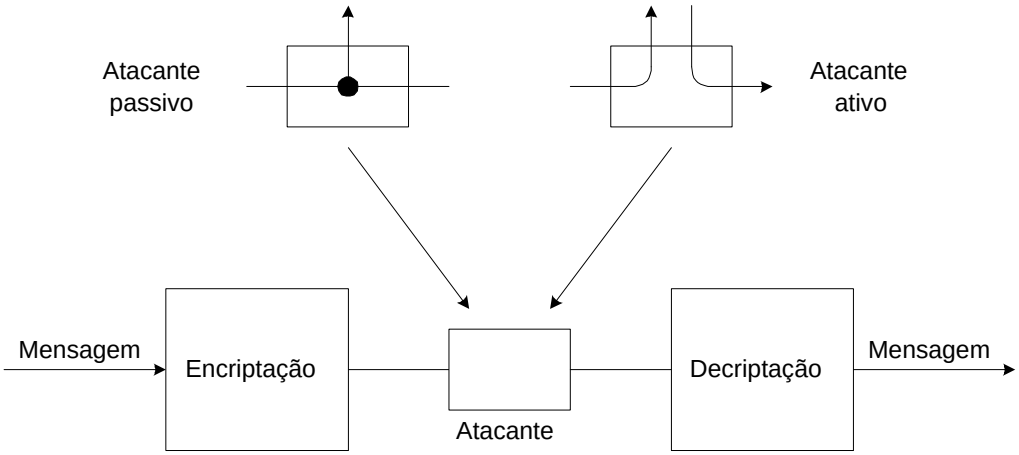

Figura II.19: Formas de ataque.

II.7.1 Ataques por criptoanálise

Criptoanálise é a ciência que abrange os princípios e métodos para se chegar à decriptação de um criptograma, sem o prévio conhecimento do algoritmo ou chaves criptográficas empregadas na sua geração.

Um atacante pode tentar obter acesso a dados encriptados por duas formas: obtendo a chave secreta, ou tentando deduzir a mensagem a partir do criptograma. Esta última faz uso de métodos de criptoanálise

Há basicamente três métodos de ataque para se decifrar um criptograma: utilizando apenas

o criptograma, conhecendo-se a mensagem, ou escolhendo-se a mensagem.

Em ataques utilizando apenas o criptograma (*ciphertext-only*), o criptoanalista deve determinar a chave a partir de criptogramas interceptados. Em alguns casos o método de criptografia, o idioma, o assunto ou as prováveis palavras utilizadas nas mensagens podem ser conhecidos facilitando o ataque.

Em ataques nos quais a mensagem é conhecida (*known-plaintext*), o criptoanalista possui acesso a pares mensagem-criptograma e procura reconhecer padrões entre diferentes pares.

Em ataques do tipo mensagem escolhida (*chosen-plaintext*), o criptoanalista possui acesso a algum mecanismo que lhe fornece o criptograma referente a uma mensagem de sua escolha, como por exemplo o segundo protocolo da Figura II.14. É então possível obter os criptogramas correspondentes a mensagens convenientemente selecionadas de modo a facilitar a criptoanálise.

Um algoritmo criptográfico é considerado incondicionavelmente seguro se não existe análise sistemática capaz de obter acesso a mensagens, independente da quantidade de criptogramas aos quais se tenha acesso.

II.7.2 Ataque por Força Bruta (*Brutal Force Attack*)

Este ataque consiste na experimentação de várias chaves na tentativa de decriptar um criptograma. As chaves experimentadas podem ser selecionadas seqüencialmente, ou com base em padrões observados. Havendo tempo suficiente, este ataque sempre obteria sucesso. O melhor modo de evitá-lo é portanto a utilização de chaves criptográficas com um número de bits tal que o teste de todas as possibilidades necessite, mesmo com o uso da mais avançada tecnologia, de uma quantidade de tempo inviável. Com a utilização, por exemplo, de uma chave criptográfica de 256 bits, existem $2^{256} = 10^{77}$ possibilidades de chaves. Há no entanto o risco de o atacante ter sucesso com uma das primeiras chaves testadas.

II.7.3 Ataque por Reflexão (*Reflection Attack*)

Utilizado para burlar autenticações baseadas em criptografia simétrica (ver Seção II.6.1), este tipo de ataque pode ocorrer quando há a possibilidade de estabelecimento de múltiplas sessões entre as entidades. A forma como ocorre este ataque é ilustrada pela Figura II.20. Utilizando o segundo protocolo exibido na Figura II.14, uma entidade X, que deseja se passar por A, envia um desafio a B, recebendo uma resposta. Ao receber o desafio de B, A inicia uma segunda sessão, enviando como desafio a mensagem que havia recebido. A entidade A responde então ao desafio da primeira sessão com o criptograma recebido na segunda.

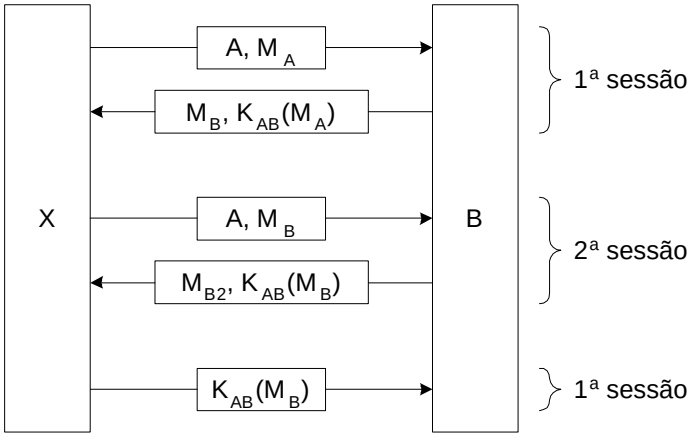

Figura II.20: Ataque por Reflexão.

Algumas regras que podem ser impostas ao protocolo de modo a evitar este tipo de ataque são:

- o iniciador deve ser autenticado antes do respondedor, evitando que o respondedor forneça informações valiosas desnecessariamente;
- utilização de duas chaves distintas K_{AB} e K_{BA}, cada uma utilizada na comunicação em um sentido;
- iniciador e respondedor devem escolher mensagens de desafio de conjuntos distintos (um escolhe somente números pares e o outro impares, por exemplo).

Vale notar que na primeira versão do protocolo apresentado na Figura II.14, o iniciador é autentidado antes do respondedor, sendo este protocolo imune a esse tipo de ataque.

II.7.4 Ataque Homem Intermediário (*Men-In-The-Middle Attack*)

Neste tipo de ataque, também conhecido como Ataque Brigada de Incêndio (*Bucket Brigade Attack*) e ilustrado na Figura II.21, duas entidades A e B desejam estabelecer uma comunicação segura por meio de criptografia assimétrica. No entanto, um atacante X intercepta e altera de forma conveniente as informações trocadas entre A e B, de modo a impedir a comunicação direta entre eles.

Ao interceptar a chave pública de A enviada por A para B, X envia para B sua própria chave pública, pretendendo ser A. O mesmo ocorre para a chave pública de B enviada para A. Deste modo, X consegue estabelecer comunicações seguras com A e B, fazendo estas duas crerem estar se comunicando diretamente.

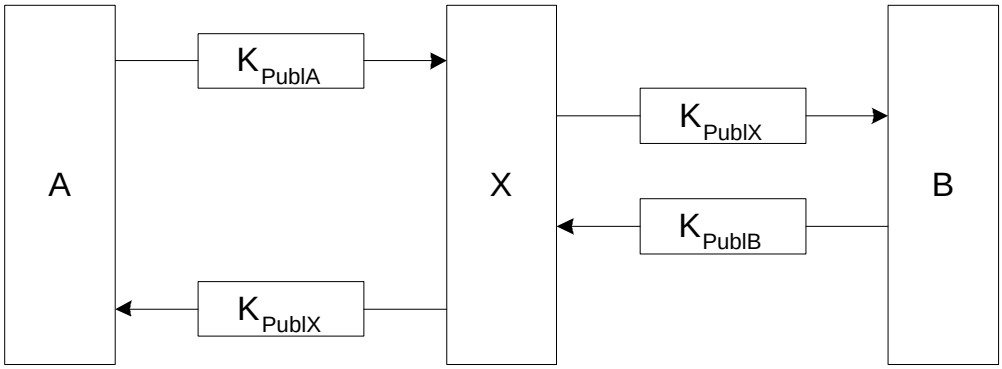

Figura II.21: Ataque Homem Intermediário.

Ataques ao algoritmo de Diffie-Hellman podem ser dificultados com a adoção de valores fixos de S e T para cada entidade, como ocorre com as chaves privada e pública dos demais algoritmos de criptografia assimétrica.

Para impedir este tipo de ataque é importante que as chaves públicas envolvidas sejam fornecidas por alguma entidade confiável.

II.7.5 Ataque por Reenvio (*Replay Attack*)

Este tipo de ataque consiste na retransmissão indevida por parte do atacante de alguma mensagem anteriormente interceptada. Como exemplo, pode-se imaginar um cliente (A) que deve solicitar ao banco (B) a transferência de um certo valor para um empregado (X). X intercepta então a mensagem (encriptada) com a solicitação, e mesmo sem compreender seu conteúdo, mas sabendo que esta lhe favorece, reenvia, possivelmente diversas vezes, a mesma mensagem para B. Algumas formas de se contornar esta vulnerabilidade, são:

- a anexação da data e hora de envio, devidamente encriptados, de modo que pedidos mais antigos sejam desconsiderados. Mensagens reenviadas dentro desse espaço de tempo seriam no entanto consideradas válidas. É necessário sincronismo entre os relógios internos dos equipamentos envolvidos, o que muitas vezes não é uma tarefa simples;
- a anexação de números de seqüência à mensagens, também encriptados, de modo a rejeitar mensagens com números de seqüência repetidos;
- o desenvolvimento de protocolos de autenticação mais complexos.

Capítulo III

Protocolo proposto

III.1 Funcionamento

Este protocolo visa permitir a distribuição segura de dados, por um servidor, utilizando uma transmissão do tipo multidestinatário, baseando-se na troca periódica da chave criptográfica que protege os dados enviados pelo servidor.

De modo a garantir privacidade e autenticação, a transmissão dos dados deve utilizar criptografia simétrica, conforme apresentado nas Seções II.4 e II.6.1. A integridade dos dados é garantida por meio da encriptação da mensagem concatenada ao seu valor *hash*, conforme apresentado na Seção II.5.

O uso do protocolo deverá:

1. permitir o acesso de novos usuários às informações transmitidas;
2. impedir o acesso a estas informações de usuários não-autorizados (inclusive aqueles que foram autorizados anteriormente);
3. restringir o acesso de atacantes que obtenham conhecimento da chave K_D ao mínimo de informação possível.

Com a finalidade de atender a estas condições o protocolo executa a troca e o reenvio da chave criptográfica K_D utilizada na encriptação dos dados transmitidos. Cada usuário é cadastrado em um banco de dados do servidor, possuindo um identificador ID e uma chave criptográfica K_{ID} compartilhada com o servidor. Ao realizar a troca da chave utilizada na proteção os dados o servidor envia pelo próprio grupo multidestinatário um pacote denominado KDP (*Key Distribution Packet* ou Pacote de Distribuição de Chave) para cada usuário contendo seu identificador e a nova chave K_D concatenada ao seu valor *hash* encriptada com a chave compartilhada.

Cada usuário ao receber este pacote verifica se o ID corresponde ao seu identificador, situação na qual decripta o restante do pacote e verifica a validade do valor *hash*, obtendo conhecimento da nova chave K_D. A partir deste momento, este cliente está apto a receber os dados enviados pelo servidor. Caso o usuário tenha sido descadastrado do servidor, seu identificador não constará em nenhum KDP, tornando-o incapaz de compreender os novos dados, o mesmo ocorrendo com algum atacante que tenha obtido acesso indevido a K_D.

Todos os termos utilizados na denominação dos serviços, parâmetros de mensagens e nomes das variáveis utilizadas neste protocolo encontram-se na língua inglesa de modo a tornar sua especificação e código gerado universalmente compreensíveis.

III.2 Especificação

III.2.1 Modelo em camadas

Tendo-se como referência a arquitetura TCP/IP, este protocolo atua como uma subcamada da camada Aplicação, situando-se entre esta camada e a camada Transporte. O protocolo é constituído por dois módulos: um servidor, responsável pela transmissão de dados, e um cliente, responsável pela recepção dos dados transmitidos.

O modelo em camadas utilizado é representado pelas seguintes partes:

- camada Aplicação, constituída por um módulo cliente ou servidor responsável por interagir com o usuário, recebendo comandos e exibindo dados;
- subcamada Aplicação, denominada *Secure Multicast*, constituída por um módulo cliente ou servidor, onde é implementado o protocolo proposto;
- camada Transporte, responsável pela transmissão dos dados; engloba os protocolos TCP e a transmissão multidestinatária.

São apresentados a seguir os serviços prestados por cada camada à camada superior, assim como as mensagens trocadas entre elas e as respectivas funções desses serviços e mensagens.

Serviços prestados pelo módulo servidor à camada Aplicação

- `Transmission_Request(addr, port)` — Solicita o início da atividade de transmissão de dados no grupo multidestinatário passado como parâmetro (endereço e porta);
- `Change_Key_Request` — Solicita a alteração da chave criptográfica utilizada na encriptação dos dados;
- `Resend_Key_Request` — Solicita o reenvio da chave criptográfica utilizada na encriptação dos dados;
- `Data_Request(data)` — Solicita o envio do dado passado como parâmetro;
- `End_Of_Transmition_Request` — Solicita o fim da atividade de transmissão de dados.

Serviços prestados pelo módulo cliente à camada Aplicação

- `Receive_Request(addr, port)` — Solicita o início da atividade de recepção de dados do grupo multidestinatário passado como parâmetro (endereço e porta);
- `Data_Indication(data)` — Fornece como parâmetro o dado recebido do servidor;

- `End_Of_Reception_Request` — Solicita o fim da atividade de recepção de dados;
- `End_Of_Transmition_Indication` — Informa o fim da atividade de transmissão de dados por parte do servidor.

Serviços prestados pela camada Transporte

- `Data_Request(data)` — Solicita o envio do dado passado como parâmetro;
- `Data_Indication(data)` — Fornece como parâmetro o dado recebido;
- `Join(addr, port)` — Solicita o ingresso no grupo multidestinatário passado como parâmetro (endereço e porta);
- `Leave(addr, port)` — Solicita o abandono do grupo multidestinatário passado como parâmetro (endereço e porta).

Dicionário de termos e siglas utilizados

- `KDP` — Pacote de Distribuição de Chave;
- `EOT` — *End Of Transmission* ou Fim de Transmissão; indica o encerramento de uma transmissão;
- `S[]` — Simboliza que a informação entre colchetes está protegida por criptografia.

Cenários de uso

Os cenários de uso especificados para o protocolo são ilustrados na Figura III.1 e correspondem a:

1. início da atividade de transmissão pelo servidor;
2. início da atividade de recepção pelo cliente;

3. encerramento da atividade de transmissão pelo servidor;
4. encerramento da atividade de recepção pelo cliente;
5. transmissão de dado pelo servidor;
6. alteração ou reenvio da chave criptográfica utilizada na proteção dos dados.

III.2.2 Especificação em Rede de Petri

A Rede de Petri é uma ferramenta utilizada na modelagem de algoritmos visando a detecção, antes da implementação, de possíveis falhas como bloqueios ou estouro do número de estados possíveis. Informações adicionais sobre esta ferramenta podem ser obtidas em [14] .

A Figura III.2 apresenta a modelagem em Rede de Petri do protocolo proposto. Foram especificados o servidor e o cliente, sendo o meio utilizado para a transmissão das mensagens (os protocolos multidestinatário e UDP) representado de uma forma simplificada. Em situações práticas, pode ocorrer acúmulo de mensagens nos *buffers* utilizados pelo receptor e pelos roteadores responsáveis pelo encaminhamento das mensagens, sendo o número de mensagens limitado pela capacidade destes *buffers*. Mensagens podem ainda ser perdidas ou sofrer desordenamento, sendo sempre enviadas em série.

A perda e o desordenamento de mensagens apresentam alguma influência sobre o protocolo, podendo levar alguns clientes a momentaneamente não compreender dados recebidos. Isto pode ocorrer caso o KDP seja perdido ou recebido após um pacote de dados já encriptado com a nova chave.

Esta especificação foi analisada com uso do programa *Analisador de Redes de Petri ARP*[1] versão 2.3, com o intuito de verificar seu correto funcionamento antes da implementação.

Os resultados obtidos indicam que a rede não é limitada, sendo os lugares `Chave` e `Dado` responsáveis pelo acúmulo de fichas. Isto se deve à simplificação adotada na modelagem do

[1]Desenvolvido pelo Laboratório de Controle e Microinformática (LCMI) do Departamento de Engenharia Elétrica (DEEL) da Universidade Federal de Santa Catarina (UFSC).

III.2 Especificação

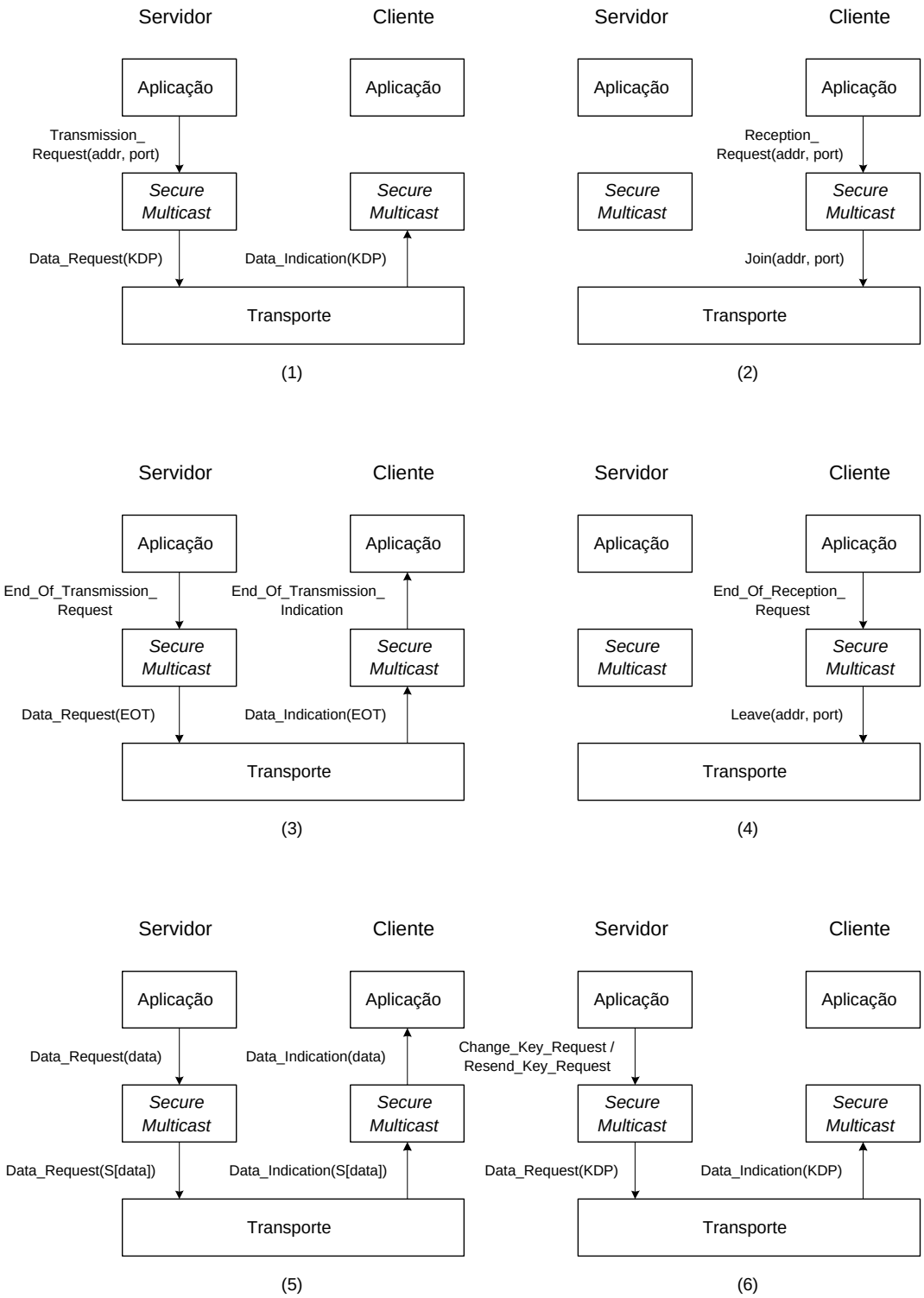

Figura III.1: Casos de uso especificados para o protocolo.

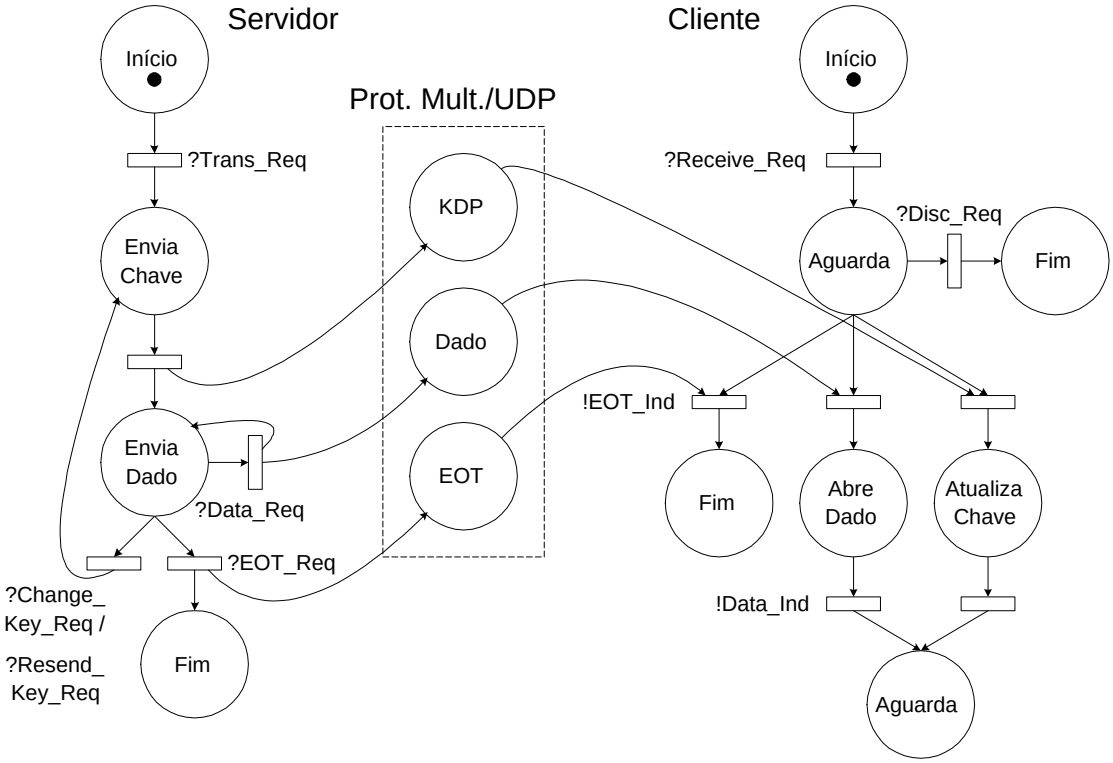

Figura III.2: Especificação do protocolo em Rede de Petri.

meio, não constituindo uma falha do protocolo. Todos os demais lugares alternam-se entre possuir zero ou uma ficha, conforme esperado.

Foram detectadas seqüências de disparo que levam a bloqueios. Em todas estas seqüências o bloqueio somente ocorre após o disparo de uma das duas transições que levam ao lugar `Fim` do cliente, correspondendo ao recebimento de mensagens após a finalização do cliente. Desta forma, não é esperada a ocorrência de qualquer bloqueio prático, pois o programa cliente já estará finalizado, sendo estas mensagens descartadas pela camada Transporte.

A análise destes resultados permitiu verificar a ausência que qualquer característica grave que pudesse impedir o funcionamento do protocolo.

Capítulo IV

Implementação

IV.1 Ambiente de desenvolvimento

O protocolo proposto foi implementado na linguagem de programação Java. Os principais fatores que levaram à escolha desta linguagem foram:

- esta ser uma linguagem multiplataforma, permitindo a execução da implementação em diversas plataformas;
- esta ser uma linguagem de alto nível, permitindo a concentração do trabalho no algoritmo;
- a grande expansão da utilização desta linguagem no mercado de trabalho[1].

O ambiente Java utilizado foi o *Java 2 SDK Standard Edition* versão 1.3, desenvolvido pela *Sun Microsystems Inc.* e que consiste em um pacote de *software* de uso gratuito que inclui diversas classes Java assim como ferramentas para compilação e execução das aplicações geradas. Maiores informações sobre esta linguagem e o ambiente utilizado podem ser obtidas em [15] .

[1] A importância deste item se deve à capacitação de trabalho com esta linguagem, adquirida com sua utilização nesta implementação.

Foi também utilizado o pacote `cryptix`, de uso gratuito, desenvolvido pela *The Cryptix Foundation Limited* [16] e composto por classes que implementam serviços e algoritmos de criptografia. O algoritmo de criptografia simétrica adotado foi o Blowfish, apresentado na Seção II.2.1, com chaves de 256 bits. A escolha da utilização deste algoritmo ocorreu por este ser otimizado para execução em computadores de 32 bits, favorecendo o desempenho. É utilizada a função *hash* MD5 de modo a garantir a integridade dos dados transmitidos, conforme apresentado na Seção II.5.

IV.2 Estrutura do sistema

A implementação do protocolo foi composta por duas classes que implementam respectivamente o servidor e os clientes: `SecMultiServer` e `SecMultiClient`. Como este protocolo constitui uma subcamada da camada Aplicação, foram desenvolvidas classes que implementam as aplicações servidor e cliente que desejam realizar transmissão segura. São elas: `Server`, `ServerInput` e `ServerApp` para o servidor e `Client`, `ClientInput` e `ClientApp` para o cliente. Estas classes foram desenvolvidas com o intuito de testar o funcionamento do protocolo, utilizando os serviços oferecidos pelas classes `SecMultiServer` e `SecMultiClient`. Dada a estrutura em camadas, estas aplicações podem ser substituídas por outras.

Foram desenvolvidas algumas classes auxiliares: `BlowFish`, `HashMD5`, `CharFile` e `GenerateUsers`. O Apêndice B apresenta as listagens dos códigos-fonte das classes desenvolvidas. A seguir são apresentados e descritos os métodos disponíveis e o funcionamento de cada classe.

IV.2.1 Classes que Implementam as Aplicações

Server

A execução da aplicação servidora é iniciada por meio de uma chamada via linha de comando e esta classe. Ao usuário é possível especificar, pela utilização de parâmetros, o endereço do grupo multidestinatário e a porta utilizados na transmissão, o valor TTL[2] (*Time-To-Live*) a ser atribuído aos pacotes, o arquivo contendo os dados a serem enviados, e o intervalo de tempo entre o envio de dois dados, a probabilidade com que pacotes a serem enviados devem ser propositalmente perdidos, de modo a permitir a simulação de perdas durante os testes do protocolo, e quantos devem ser perdidos em seqüência, de modo a simular perdas em rajada, a freqüência com que a chave K_D deve ser automaticamente trocada, e a freqüência com que esta deve ser reenviada. A sintaxe deste comando é `java Server [-G group] [-P port] [-T ttl] [-F file] [-D delay] [-L loss] [-B burst] [-K change key] [-R resend key]`. Para os parâmetros não fornecidos são utilizados valores padrão, respectivamente '230.0.0.1', '4446', '1', 'one-liners.txt', '1', '0', '1', '0' e '0'.

Esta classe instancia[3] duas classes: `ServerInput` e `ServerApp`, repassando à segunda alguns dos parâmetros recebidos e sendo finalizada em seguida.

ServerInput

Esta classe consiste de uma *thread*[4] cuja função consiste em receber comandos emitidos pelo usuário via teclado. Estes comandos correspondem ao pressionamento das teclas `K`, `R` ou `Q` seguido pelo pressionamento da tecla `ENTER`. As atividades desencadeadas pela recepção desses comandos são:

[2]Este valor especifica o número máximo de roteadores que podem ser percorridos por cada pacote, sendo útil na restrição do alcance destes.

[3]Instanciar consiste no ato de criar um objeto a partir de uma classe, podendo o objeto receber um nome que permita identificá-lo ou não.

[4]Parte do programa executada independentemente. Sua utilização permite neste caso às aplicações aguardar comandos via teclado enquanto realizam outras tarefas.

- K — troca da chave K_D por meio da execução do método `changeKey` do objeto `serverApp` (instanciação da classe `ServerApp`);
- R — reenvio da chave K_D por meio da execução do método `sendKey` do objeto `serverApp`;
- Q — finalização do aplicativo por meio da execução dos métodos `finish` do objeto `serverApp` e seu próprio.

ServerApp

Esta classe consiste de uma *thread* que instancia a classe `SecMultiServer`, criando um objeto denominado `secServ`, e passa a solicitar periodicamente a este, por meio do método `sendData`, o envio de informações obtidas com o método `getNextData` até que seja solicitada a execução de seu método `finish`. É então executa uma chamada ao método `finish` do objeto `secServ`.

Métodos:

- `getNextData` —

  Executa a leitura de uma linha do arquivo de mensagens, retornando seu valor. Caso o arquivo não seja encontrado, são retornadas a data e a hora correntes na forma de texto. Quando todas as mensagens tiverem sido enviadas, é executado o método `finish`;

- `changeKey` —

  Executa o método `changeKey` do objeto `secServ`, responsável pela troca da chave criptográfica utilizada;

- `sendKey` —

  Executa o método `sendKey` do objeto `secServ`, responsável pelo envio da chave criptográfica utilizada;

- `finish` —

  Determina o fim da execução desta *thread*.

Client

A execução da aplicação cliente é iniciada por meio de uma chamada via linha de comando e esta classe. Ao usuário é obrigatória a especificação pela utilização de parâmetros do identificador do usuário, sendo opcional a especificação do grupo multidestinatário e da porta utilizados na recepção dos dados. Caso um desses dois parâmetros não seja especificado, são utilizados valores padrão, respectivamente '230.0.0.1' e '4446'. A sintaxe deste comando é `java Client userID [-G group] [-P port]`.

Esta classe instancia duas classes: `ClientInput` e `ClientApp`, repassando à segunda os parâmetros recebidos e sendo finalizada em seguida.

ClientInput

Esta classe consiste de uma *thread* cuja função é receber comandos emitidos pelo usuário via teclado. O único comando disponível corresponde ao pressionamento da tecla `Q` seguido pelo pressionamento da tecla `ENTER`. A atividade desencadeada pela recepção desse comando é a finalização do aplicativo por meio da execução do método `finish` do objeto `clientApp`.

ClientApp

Esta classe instancia a classe `SecMultiClient`, criando um objeto denominado `secCli`, cria um canal de comunicação com este, e passa a aguardar o recebimento de dados entregues por este objeto por meio de chamadas ao seu método `readData`.

Métodos:

- `readData` —

  Envia pelo canal de dados estabelecido com o objeto `secCli` a informação recebida do servidor, já decriptada, e exibe-a na tela;

- `finish` —

Executa os métodos `finish` dos objetos `secCli` e `clientInput` com o objetivo de finalizá-los.

IV.2.2 Classes que Implementam o Protocolo

É mantido pelo servidor um registro do número de pacotes enviados[5] e perdidos a cada sessão. Por cada cliente é mantido um registro com o número de pacotes recebidos a cada sessão. A partir do número de pacotes transmitidos pelo servidor e recebidos pelo cliente, é possível calcular o número de pacotes perdidos no total, inclusive aqueles que foram recebidos após um KDP perdido, tornando-se incompreensíveis.

SecMultiServer

Ao ser instanciada, esta classe:

- cria um *socket* UDP responsável pela transmissão dos dados encriptados com o valor de TTL a ser utilizado;
- executa o método `readUsers`, responsável pela leitura do cadastro de usuários;
- obtém a chave K_D por meio do método `readString` da classe `CharFile`;
- envia esta chave aos clientes por meio do método `sendKey`.

Todas as transmissões realizadas pelo servidor possuem algum dado encriptado, ou com a chave de transmissão de dados ou a compartilhada com um cliente, garantindo sua autenticação perante os clientes.

Métodos:

[5]Todos os pacotes de dados recebidos para transmissão são contabilizados como transmitidos, pois a perda proposital simula uma característica do meio.

- `readUsers` —

  Executa a leitura, a partir de arquivo, dos dados dos clientes. Estes dados consistem, para cada cliente, de um identificador ID e de uma chave criptográfica compartilhada K_{ID};

- `changeKey` —

  Gera um valor randômico para a chave K_D, salva-o em arquivo, e executa o método `sendKey`, passando K_D como parâmetro;

- `sendKey` —

  Recebe como parâmetro a chave K_D a ser transmitida e calcula $MD(K_D)$[6]. Para cada cliente ID calcula $E_{K_{ID}}(MD(K_D) \mid K_D)$[7], solicitando ao método `transmit` o envio da informação [KDP | ID | $E_{K_{ID}}(MD(K_D) \mid K_D)$], onde KDP é um código que identifica o pacote como um Pacote de Distribuição de Chave;

- `sendData` —

  Recebe como parâmetro a informação M a ser enviada, calcula $MD(M)$ e $E_{K_D}(MD(M) \mid M)$, solicitando o envio da informação [DATA | $E_{K_D}(MD(M) \mid M)$], onde DATA é um código que identifica o pacote como um pacote de dados. É atualizado o registro de pacotes transmitidos e perdidos e verificada a necessidade de troca ou reenvio da chave, baseando-se nas freqüências especificadas. Caso necessário são executados respectivamente os métodos `changeKey` e `sendKey`;

- `transmit` —

  Responsável pela criação de um pacote IP com a informação recebida como parâmetro e seu envio pelo *socket*. É causada perda proposital dos pacotes a serem enviados segundo os parâmetros especificados;

- `finish` —

  Executa a encriptação com a chave K_D de uma seqüência de 7 caracteres '1' e solicita ao método `transmit` o envio da informação [EOT | $E_{K_D}(1111111)$], onde EOT é um

[6] Por meio do método *hash* da classe *HashMD5*.

[7] Por meio do método *encrypt* da classe *BlowFish*.

código que identifica o pacote como um pacote de fim de transmissão. Esta seqüência de caracteres encriptada tem a função de garantir a autenticidade do servidor.

SecMultiClient

Ao ser instanciada, esta classe:

- executa a leitura, a partir de arquivo, de sua chave K_{ID} por meio do método `readString` da classe `CharFile`;
- cria um *socket* associado endereço IP e porta especificados, associando-se ao grupo multidestinatário e passando a receber os dados transmitidos neste grupo;
- executa a leitura, a partir de arquivo, por meio do método `readString` da classe `CharFile` da chave K_D que estava sendo utilizada na última sessão;
- cria um canal de comunicação com a aplicação (objeto clientApp), por onde são enviados os dados decriptados.

Esta classe passa então a aguardar o recebimento de pacotes pelo *socket*, verificando o código no início de cada pacote. Caso este código não corresponda a KEY, DATA ou EOT, o pacote é descartado. Caso corresponda é tratado da seguinte forma:

- KDP — é executado o método `readKey` passando-se como parâmetro o restante do pacote;
- DATA — é executado o método `readData` passando-se como parâmetro o restante do pacote;
- EOT — o restante do pacote é decriptado com K_D por meio do método `decrypt` da classe `BlowFish`, sendo o conjunto de dados obtido comparado com um conjunto de 7 caracteres ‘1’. Se esta condição for verdadeira, ocorre uma chamada ao método `finish`, sendo descartado caso contrário. A verificação dessa condição permite a autenticação do servidor.

Este processo é repetido até que o método `finish` seja executado.

Métodos:

- readKey —

  Executa a leitura do identificador do usuário. Caso corresponda ao próprio usuário, decripta o restante do pacote obtendo a nova chave K_D cuja integridade é verificada pela comparação entre os valores *hash* recebido e calculado;

- readData —

  O restante do pacote é decriptado[8] com K_D, sendo recuperados o valor *hash* e o conjunto de dados enviado, cujo valor *hash* é então calculado e comparado com o recebido, garantindo a integridade da informação. Informações válidas são enviadas à aplicação por meio da execução do método `readData` do objeto `clientApp` (instanciação da classe ClientApp). É atualizado o registro de pacotes recebidos caso este conjunto de dados seja válido;

- finish —

  Abandona o grupo multidestinatário, finaliza o *socket* e executa o método `finish` do objeto `clientApp`.

IV.2.3 Classes Auxiliares

BlowFish

Esta classe é responsável por exercer a interface com a classe `Blowfish` do pacote `cryptix` que implementa o algoritmo Blowfish. Os métodos disponíveis são:

- `encrypt` —

[8] Por meio do método *decrypt* da classe *BlowFish*.

Recebe como parâmetros, ambos na forma de um conjunto de caracteres, a informação a ser encriptada e a chave criptográfica a ser utilizada, representada em hexadecimal. O bloco de dados recebido sofre um enchimento de modo a tornar seu tamanho múltiplo de 64 bits. Este enchimento é similar àquele utilizado pelas funções *hash* MD4, MD5 e SHS, sendo no entanto realizado a nível de caractere e constituído por uma quantidade suficiente de caracteres de espaço seguida pelo caractere numérico que representa o tamanho total do enchimento (um máximo de 8). O conjunto de caracteres obtido é transformado em uma seqüência de bytes que é fornecida ao algoritmo de criptografia. A seqüência de bytes retornada por este é convertida em um conjunto de caracteres que é o criptograma retornado pelo método;

- `decrypt` —

  Executa a seqüência inversa ao método `encrypt`, sendo o enchimento verificado quanto a sua validade. Caso não corresponda ao esperado, o que indica corrupção dos dados, é retornado um conjunto vazio de caracteres.

HashMD5

Esta classe é responsável por exercer a interface com a classe `MD5` do pacote `cryptix` que implementa a função de *hash* MD5. O único método disponível é denominado `hash` e recebe como parâmetro um conjunto de caracteres que é convertido em uma seqüência de bytes e fornecido à classe `MD5`. Esta fornece um conjunto de bytes que é convertido em caracteres sendo retornado pelo método.

CharFile

Responsável pela leitura e escrita de valores numéricos inteiros e conjuntos de caracteres em arquivos. Os métodos disponíveis são:

- `writeInt` —

  Escreve um inteiro;

- `readInt` —

  Lê um inteiro;

- `writeString` —

  Escreve um conjunto de caracteres;

- `readInt` —

  Lê um conjunto de caracteres.

GenerateUsers

Utilizada na criação dos dados relativos a um conjunto de usuários, esta classe não é utilizada por qualquer uma das outras. É executada via linha de comando recebendo como parâmetro o número de usuários a serem criados. Para cada usuário são especificados um identificador ID e uma chave criptográfica K_{ID} gerada randomicamente. É também criada randomicamente a chave K_D a ser inicialmente utilizada na criptografia dos dados. Todos estes dados são gravados em arquivos apropriados com o uso dos métodos `writeInt` e `writeString` da classe `CharFile`.

IV.3 Testes realizados

Os testes realizados consistiram na transmissão de dados na rede local do laboratório do GTA (Grupo de Teleinformática e Automação). Este rede possui uma topologia em estrela com enlaces de 100 Mbps. Em todos os testes os fenômenos de perda e desordenamento de pacotes não foram verificados, sendo portanto simulados por meio de parâmetros passados à aplicação servidora que causam o descarte proposital de pacotes, tanto os de dados quanto dos pacotes KDP e EOT. A ocorrência de desordenamento entre um pacote de dados um KDP pode ser tratada como a perda do pacote de dados, dada utilização da nova chave K_D. Para KDPs que estejam sendo reenviados e já sejam de conhecimento dos clientes este fenômeno não representa um problema.

Erros adicionais em relação àqueles causados pela transmissão tradicional são portanto devidos a:

- aplicações recém iniciadas que ainda não possuem a nova chave K_D;
- perda de um KDPs com a nova chave K_D que ainda não é de conhecimento de alguns clientes.

Espera-se que erros devidos a este protocolo sejam portanto transitórios, sendo interrompidos após o recebimento do KDP. Pelo controle do período de reenvio do KDP, deve ser possível o controle sobre as perdas. Estes erros devem ainda ser tão menos freqüentes quanto menos freqüente for a troca da chaves K_D.

Os testes realizados envolveram a transferência de 520 KB de informação, formatada em um arquivo com 10.000 linhas, cada linha correspondendo a um conjunto de dados a ser enviado. O servidor possuía 100 usuários cadastrados, e a troca da chave K_D ocorreu a cada 100 pacotes, sendo variados os valores da probabilidade de perda do meio e de reenvio do KDP, visando verificar a influência destes parâmetros sobre o protocolo. Foram realizados dois conjuntos de testes que se diferenciam pela forma como ocorrem as perdas. No primeiro cada pacote era perdido independentemente, e no segundo os pacotes eram perdidos em rajadas de tamanho configurável de modo a simular perdas devido a sobrecarga de *buffer*. Em todos os conjuntos de testes os clientes já se encontravam em modo de recepção quando a transmissão era iniciada, de modo a permitir o cálculo de pacotes perdidos.

Os dados apresentados nas Tabelas IV.1 e IV.2 correspondem a proporção entre pacotes perdidos por ação deste protocolo e os perdidos pelo meio, dada por $\frac{\text{enviados}-\text{recebidos}}{\text{perdidos}}$. As linhas correspondem à probabilidade de perda do meio e as colunas à freqüência de reenvio do KDP.

A análise destes resultados permite observar que a diminuição do período de reenvio de pacotes reduz a proporção de erro, permitindo a obtenção de resultados próximos àqueles obtidos com a utilização tradicional de transmissões multidestinatárias.

Foram realizados ainda testes nos quais a chave K_D não era alterada, sendo apenas periodicamente reenviada. Em todos estes testes a relação entre as perda foi igual a 1, comprovando

Tabela IV.1: Proporção entre as perdas com e sem a utilização deste protocolo para perdas independentes.

	5	10	50	100
1%	1.08	1.11	1.50	1.45
2%	1.07	1.23	1.25	1.50
5%	1.03	1.08	1.27	1.76
10%	1.05	1.14	1.61	1.47
20%	1.05	1.08	1.60	1.77

Tabela IV.2: Proporção entre as perdas com e sem a utilização deste protocolo para perdas em rajada.

	5	10	50	100
1%	1.04	1.22	1.42	1.00
2%	1.06	1.27	1.24	2.16
5%	1.03	1.14	1.31	1.26
10%	1.02	1.09	1.67	1.58
20%	1.03	1.10	1.25	1.67

que perdas adicionais somente podem ser causadas no momento da troca da chave e que a perda de um KDP com uma senha já conhecida não possui influência sobre o protocolo.

Capítulo V

Conclusão

O protocolo proposto fornece suporte a autenticação e privacidade, por meio do uso de criptografia simétrica, e a integridade, pelo uso de função *hash*.

A análise dos resultados obtidos permitiu comprovar que perdas adicionais de pacotes são causadas pela perda de KDPs com senha ainda desconhecida pelo cliente. Valores adequados para a freqüência de reenvio do KDP permitem o controle das taxas de erro obtidas, até o limite dos erros causados pela utilização tradicional do protocolo multidestinatário.

Em situações de uso prático a troca da chave K_D pode ser realizada apenas quando necessário descadastrar clientes, ou quando seu sigilo for posto em risco. Quanto menos freqüente for realizada esta troca, mais baixas são as perdas de pacotes. Dada a característica transitória dessas perdas, pode ser interessante o reenvio mais freqüente do KDP nos momentos que se seguem à troca da chave.

O período de reenvio deve ser determinado com base na freqüência com que usuários costumam iniciar a recepção e na freqüência com que ocorre a troca da chave, de modo a minimizar o consumo dos recursos da rede.

A utilização deste protocolo causa o envio de pacotes adicionais à transmissão, além do aumento do tamanho dos pacotes devido à concatenação do valor *hash* à mensagem. Esta carga adicional é no entanto tão relativamente menor quanto maiores sejam as mensagens enviadas.

A transmissão de KDPs para um número muito elevado de usuários pode causar problema de sobrecarga da rede, o que pode ser evitado por meio da limitação do número de usuários cadastrados junto ao servidor.

O maior consumo de recursos é compensado pela segurança provida às transmissões. Como pode ser observado no Capítulo II, todas as técnicas disponíveis para prover segurança requerem um maior consumo de recursos.

Neste trabalho foi proposto, implementado e validado um protocolo eficaz na realização de transmissões multidestinatárias seguras.

Referências Bibliográficas

[1] C. Kaufman, R. Perlman e M. Speciner, *Network Security: Private Communication in a Public World*. Englewood Cliffs, NJ: PTR Prentice Hall, 1995.

[2] W. Stallings, *Network and Internetwork Security: Principles and Practice*. Englewood Cliffs, NJ: Prentice Hall, Inc; IEEE Press, 1995.

[3] R. E. Smith, *Internet Cryptography*. Addison-Wesley Longman, Inc., 1997.

[4] A. S. Tanenbaum, *Computer Networks*. Upper Saddle River, NJ: Prentice Hall PTR, third ed., 1996.

[5] S. Brown, *Implementing Virtual Private Networks*. McGraw-Hill, 1999.

[6] D. J. Stang e S. Moon, *Network Security Secrets*. San Mateo, CA: IDG Books Worldwide, Inc., 1993.

[7] S. L. Shaffer e A. R. Simon, *Network Security*. Cambridge, MA: AP Professional, 1994.

[8] V. Ahuja, *Network & Internet Security*. Chestnut Hill, MA: AP Professional, 1996.

[9] B. Pfaffenberger, *Protect Yor Privacy on the Internet*. Wiley Computer Publishing; Jonh Wiley & Sons, Inc., 1997.

[10] D. E. R. Denning, *Cryptography and Data Security*. Addison-Wesley Publishing Company, Inc., 1983.

[11] A. B. de Holanda Ferreira, *Novo Dicionário da Língua Portuguesa*. Editora Nova Fronteira S.A., segunda ed., 1986.

REFERÊNCIAS BIBLIOGRÁFICAS

[12] F. Fernandes, C. P. Luft e F. M. Guimarães, *Dicionário Brasileiro da Língua Portuguesa*. Editora Globo S.A., 30 ed., 1993.

[13] B. Schneier, “Description of a new variable-length key, 64-bit block cipher (blowfish)”, *First Fast Software Encryption Workshop*, 1994. Cambridge, UK.

[14] J. L. Peterson, “Petri nets”, *Computing Surveys*, vol. 9, no. 3, pp. 223–252, setembro 1977.

[15] “java.sun.com - the source for java(tm) technology”. http://www.java.sun.com.

[16] “Cryptix - cryptix 3”. http://www.cryptix.org/products/cryptix31.

[17] “Uol - michaelis - moderno dicionário da língua portuguesa”. http://www.uol.com.br/michaelis.

[18] “Frees/wan project: Home page”. http://www.xs4all.nl/ freeswan/.

Apêndice A

Aritmética modular

Diversas operações realizadas em algoritmos criptográficos, particularmente de criptografia assimétrica, envolvem o uso de aritmética modular. A principal diferença entre a aritmética tradicional e a modular é que esta última representa números utilizando somente inteiros não-negativos menores que um inteiro positivo n. Diz-se então que estes números são representados em aritmética módulo n. As operações aritméticas são realizadas normalmente, sendo o resultado obtido dividido por n e o resto desta divisão correspondendo ao resultado final. A notação "x mod n" é então utilizada para representar o resto da divisão de x por n. Esta notação torna-se implícita, caso tenha sido explicitado a utilização de aritmética modular.

São apresentadas a seguir algumas operações em aritmética modular e suas respectivas operações inversas. Para cada uma é apresentada a tabela correspondente a sua operação em módulo 10 com a finalidade de ilustrar a operação.

A.1 Adição modular

O inverso desta operação é equivalente à subtração de x, que pode ser realizada somando-se $-x$. Esta operação é definida então como o valor que, somado a x, iguala 0. Observando a Tabela A.1, percebe-se, por exemplo, que o inverso aditivo de 4 é 6, pois $4 + 6 = 0$. A adição

modular pode ser utilizada como um algoritmo muito simples de criptografia, pois observando a Tabela A.1, nota-se que para cada coluna, os valores das linhas mapeiam para valores únicos. Este algoritmo é conhecido como Cifra de Cæsar, e a chave consiste da coluna a ser utilizada. Ainda observando a tabela em questão, dados encriptados com uma chave, por exemplo 4, são decriptados com sua inversa aditiva, no caso 6.

Tabela A.1: Adição módulo 10.

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

A.2 Multiplicação modular

Para a multiplicação, nem todas as colunas podem ser utilizadas para encriptação. No caso de multiplicação módulo 10, observando a Tabela A.2, nota-se que apenas os valores 1, 3, 7 e 9 podem ser utilizados para tal fim, pois realizam substituições do tipo um-para-um. Para decriptação, é então necessária a execução da operação inversa, que na aritmética tradicional corresponde a $1/x$. No entanto, para valores de x diferentes de 1, seu inverso corresponde a um número fracionário, que está fora do domínio da aritmética modular. A operação inversa da multiplicação na aritmética modular, denominada x^{-1} é definida como o número que se deve

multiplicar x para obter 1.

Para os números apresentados anteriormente, o inverso da multiplicação por 1 é 1, por 3 é 7, por 7 é 3 e por 9 é 9.

A razão pela qual estes são os únicos números em módulo 10 com inversos multiplicativos é o fato de cada um deles e o número 10 serem primos entre si. Dois ou mais números são ditos primos entre si se e somente se o maior número que divide a ambos é 1. De uma forma mais geral, pode-se dizer que em aritmética módulo n para todo número x tal que x e n são primos entre si, x possui um inverso multiplicativo.

Tabela A.2: Multiplicação módulo 10.

$\times$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

Uma função de grande utilidade é denominada Função Totiente de Euler, representada por $\phi(n)$, e informa a quantidade de números m inteiros menores que n tais que m e n são primos entre si. Sendo n primo, todos os inteiros não negativos menores que n são primos entre si e n, sendo portanto $\phi(n) = n - 1$. Se n for o produto entre dois números primos p e q, segue que $\phi(n) = (p - 1) \cdot (q - 1)$, o que pode ser comprovado excluindo-se aqueles números que são múltiplos de p ou de q. Existem p múltiplos de q menores que n e q múltiplos de p menores que n, existindo portanto $p + q - 1$ números menores que n a serem excluídos (excluindo o 0

somente uma vez). Obtém-se então $\phi(n) = p \cdot q - (p + q - 1) = (p - 1) \cdot (q - 1)$. Na módulo 10, tem-se $p = 2$ e $q = 5$, logo, $\phi(n) = 4$.

A.3 Exponenciação modular

Observando a Tabela A.3, percebe-se que as colunas 1 e 5, assim como 2 e 6, e 3 e 7 são idênticas. É possível provar que $x^y \bmod n = x^{y \bmod \phi(n)} \bmod n$. No caso de aritmética módulo 10, já foi apresentado que $\phi(n) = 4$, sendo portanto a i-ésima coluna sempre igual à $(i + 4)$-ésima[1].

Outra relação de grande importância ocorre quando $y = 1 \bmod \phi(n)$. Tem-se então que para qualquer número x, $x^y = x \bmod n$.

Tabela A.3: Exponenciação módulo 10.

x^y	0	1	2	3	4	5	6	7	8	9
0	-	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	8	6	2	4	8	6	2
3	1	3	9	7	1	3	9	7	1	3
4	1	4	6	4	6	4	6	4	6	4
5	1	5	5	5	5	5	5	5	5	5
6	1	6	6	6	6	6	6	6	6	6
7	1	7	9	3	1	7	9	3	1	7
8	1	8	4	2	6	8	4	2	6	8
9	1	9	1	9	1	9	1	9	1	9

O inverso desta operação é denominado logaritmo discreto e consiste em, a partir do conhecimento de $z = x^y \bmod n$, x e n, calcular y. Não há um equacionamento para este cálculo,

[1]Esta relação somente não é verdadeira para valores de n que contém um fator p^2, para p primo. Tais números não são no entanto utilizados em algoritmos criptográficos.

sendo este de difícil realização para valores primos grandes. Os algoritmos de criptografia assimétrica geralmente se baseiam nesta dificuldade de cálculo para prover a segurança.

Apêndice B

Códigos-Fonte

B.1 Protocolo

B.1.1 SecMultiServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import cryptix.util.core.Hex;


public class SecMultiServer {

  static final char KDP  = '0';
  static final char DATA = '1';
  static final char EOT  = '2';

  public InetAddress group = null;
  public int port;
  public double lossProb;
  public int keyChange;
  public int keyResend;
  public int burst;

  public MulticastSocket socket;
  boolean finish = false;
```

```
public String KEY_FILE = "data_key.dat";
public String USERS_FILE = "user_keys.dat";
public int countSent = 0;
public int countLost = 0;
public int burstCount = Integer.MAX_VALUE;

public int users;
public int[] userID;
public String[] userKey;
public String key;


public SecMultiServer(InetAddress group, int port, int ttl,
double lossProb, int burst, int keyChange, int keyResend)
throws IOException {

  this.group = group;
  this.port = port;
  this.lossProb = lossProb;
  this.keyChange = keyChange;
  this.keyResend = keyResend;
  this.burst = burst;

  socket = new MulticastSocket(4445);
  socket.setTimeToLive(ttl);

  readUsers();

  FileReader keyFile = new FileReader(KEY_FILE);
  key = CharFile.readString(keyFile);
  keyFile.close();

  System.out.println("Using key: '" + key + "'");

  sendKey(key);

}




public void readUsers() throws IOException {

  FileReader inFile = new FileReader(USERS_FILE);
```

```
    users = CharFile.readInt(inFile);

    userID = new int[users];
    userKey = new String[users];

    for (int i = 0 ; i < users ; i++) {
      userID[i] = CharFile.readInt(inFile);
      userKey[i] = CharFile.readString(inFile);
    }

    inFile.close();

  }




  public void changeKey() {

    key = Hex.longToString( (long)(Long.MAX_VALUE * Math.random()) );

    try {
      FileWriter keyFile = new FileWriter(KEY_FILE);
      CharFile.writeString(keyFile, key, ';');
      keyFile.close();
    } catch (IOException e) {
      System.err.println("IOException: " + e);
    }

    sendKey(key);

  }




  public void sendKey() {

    sendKey(key);

  }
```

```
public void sendKey(String key) {

  String hashValue = HashMD5.hash(key);

  for (int i = 0 ; i < users ; i++) {

    StringBuffer outData = new StringBuffer(256);
    outData = outData.append(userID[i]).append(',');
    outData = outData.append(BlowFish.encrypt((hashValue + key),
userKey[i]));

    String data = KDP + outData.toString();

    if (transmit(data)) {
      System.out.println("Sent key for user: '" + i + "'");
    } else {
      System.out.println("Lost key for user: '" + i + "'");
    }

  }

  System.out.println("Sent key: '" + key + "' for all users");

}




public void sendData(String plainText) {

  String hashValue = HashMD5.hash(plainText);
  String cipherText = BlowFish.encrypt((hashValue + plainText), key);
  String data = DATA + cipherText;

  if (transmit(data)) {

    System.out.println("Sent data: '" + plainText + "'");

  } else {

    System.out.println("Lost data: '" + plainText + "'");

    countLost++;
    try {
      FileWriter fileCountLost = new FileWriter("count_lost.dat");
```

```
        CharFile.writeInt(fileCountLost, countLost, ';');
        fileCountLost.close();
      } catch (IOException e) {
        System.err.println("IOException: " + e);
      }

    }

    countSent++;
    try {
      FileWriter fileCountSent = new FileWriter("count_sent.dat");
      CharFile.writeInt(fileCountSent, countSent, ';');
      fileCountSent.close();
    } catch (IOException e) {
      System.err.println("IOException: " + e);
    }

    if ((keyChange != 0) && ((countSent % keyChange) == 0)) {
      changeKey();
    }

    if ( ((keyResend != 0) && ((countSent % keyResend) == 0)) &&
!((keyChange != 0) && ((countSent % keyChange) == 0)) ) {
      sendKey(key);
    }



  }




  public boolean transmit(String data) {

    byte[] buf = new byte[256];
    buf = data.getBytes();

    DatagramPacket packet = new DatagramPacket(buf, buf.length,
group, port);

    boolean returnValue;

    if (burstCount < burst) {
```

```
      burstCount++;
      returnValue = false;

    } else {

      if (Math.random() < 1-(lossProb/burst)) {

        try {
          socket.send(packet);
        } catch (IOException e) {
          System.err.println("IOException: " + e);
          returnValue = false;
        }
        returnValue = true;

      } else {

        burstCount = 1;
        returnValue = false;

      }

    }

    return returnValue;

  }




  public void finish() {

    String data = EOT + BlowFish.encrypt("1111111", key);

    if (transmit(data)) {
      System.out.println("Sent EOT notification");
    } else {
      System.out.println("Lost EOT notification");
    }

    socket.close();
    System.out.println("Closed the socket");

    System.out.println("Sent " + countSent + " packets and lost "
```

```
+ countLost);

  }
```

B.1.2 SecMultiClient.java

```
import java.io.*;
import java.net.*;
import java.util.*;


class SecMultiClient extends Thread {

  static final char KDP  = '0';
  static final char DATA = '1';
  static final char EOT  = '2';

  public MulticastSocket socket = null;
  public InetAddress group = null;
  public ClientApp clientApp;
  public PipedOutputStream pout;
  public BufferedWriter bout;

  public int userID;
  public String userKey;
  public String key;

  public boolean finish = false;
  public String key_file;
  public int countReceived = 0;


  public SecMultiClient(ClientApp clientApp, int userID,
  InetAddress group, int port) throws IOException {

    this.userID = userID;
    this.group = group;
    this.clientApp = clientApp;

    FileReader fileUserKey = new FileReader("user_key_" + userID + ".da
    userKey = CharFile.readString(fileUserKey);
```

```
    fileUserKey.close();


    socket = new MulticastSocket(port);
    try {
      socket.joinGroup(group);
    } catch (IOException e) {
      System.err.println("IOException: " + e);
      clientApp.finish();
      return;
    }

    System.out.println("Client '" + userID + "' joined the multicast gr

    key_file = "data_key_" + userID + ".dat";

    FileReader fileDataKey = new FileReader(key_file);
    key = CharFile.readString(fileDataKey);
    fileDataKey.close();

    System.out.println("Using key: " + key);

    pout = new PipedOutputStream(clientApp.pin);
    bout = new BufferedWriter(new OutputStreamWriter(pout));

  }




  public void run() {

    while (!finish) {

      byte[] buf = new byte[256];
      DatagramPacket packet = new DatagramPacket(buf, buf.length);

      try {
        socket.receive(packet);
      } catch (SocketException e) {
        return;
      } catch (IOException e) {
        System.err.println("IOException: " + e);
      }
```

```
      String received = new String(packet.getData());
      received = received.substring(0, packet.getLength());

      char code = received.charAt(0);
      String data = received.substring(1);


      switch (code) {

        case KDP:
          readKey(data);
          break;


        case DATA:
          readData(data);
          break;


        case EOT:
          String plainText = BlowFish.decrypt(data, key);
          if ((plainText != null) && (plainText.equals("1111111"))) {
            System.out.println("Received EOT notification");
            clientApp.finish();
          }
          break;


        default:
          System.out.println("Received an invalid code");
          break;

      }

    }

  }



  public void readKey(String data) {

    byte[] dataByte = data.getBytes();
```

```
char[] buf = new char[0];
int position = 0;
char temp = (char)dataByte[position];

while (temp != ':' && temp != ';' && temp != ',' && temp != ' ') {

  char[] bufNew = new char[position+1];
  for (int j = 0 ; j < position ; j++) {
    bufNew[j] = buf[j];
  }
  buf = bufNew;
  buf[position] = temp;
  position++;
  temp = (char)dataByte[position];

}


int ID = Integer.decode(new String(buf).substring(0,position)).intV


if (ID == userID) {

  position++;

  byte[] cipherByte = new byte[500];

  int i;
  for (i = 0 ; i < dataByte.length-position ; i++) {
    cipherByte[i] = dataByte[i+position];
  }

  String cipherText = new String(cipherByte).substring(0,i);
  String plainText = BlowFish.decrypt(cipherText, userKey);

  if (plainText != null) {

    String hashReceived = plainText.substring(0, 32);
    String tempKey = plainText.substring(32);

    String hashValue = HashMD5.hash(tempKey);

    if (hashValue.equals(hashReceived)) {

      key = tempKey;
```

```
          try  {
            FileWriter fileDataKey = new FileWriter(key_file);
            CharFile.writeString(fileDataKey, key, ';');
            fileDataKey.close();
          } catch (IOException e) {
            System.err.println("IOException: " + e);
          }

          System.out.println("Received key: " + key);

        }

      }

    }

  }
```

```
  public void readData(String data) {

    String cipherText = data;
    String plainText = BlowFish.decrypt(cipherText, key);

    if (plainText != null) {

      String hashReceived = plainText.substring(0, 32);
      plainText = plainText.substring(32);

      String hashValue = HashMD5.hash(plainText);

      if (!hashValue.equals(hashReceived)) {
        plainText = null;
      }

    }

    String dataUp = plainText + "\n";

    try {
      bout.write(dataUp, 0, dataUp.length());
      bout.flush();
```

```
      clientApp.readData();
    } catch (IOException e) {
      System.err.println("SecMultiClient: " + e);
    }

    if (plainText != null) {
      countReceived++;
      try {
        FileWriter fileCountReceived =
new FileWriter("count_received_" + userID + ".dat");
        CharFile.writeInt(fileCountReceived, countReceived, ';');
        fileCountReceived.close();
      } catch (IOException e) {
        System.err.println("IOException: " + e);
      }
    }

  }



  public void finish() {

    finish = true;

    try {
      socket.leaveGroup(group);
      System.out.println("Left the multicast group");
    } catch (SocketException e) {
    } catch (IOException e) {
      System.err.println("IOException: " + e);
    }
    socket.close();
    System.out.println("Closed the socket");

    System.out.println("Received " + countReceived + " packets");

  }


}
```

B.2 Aplicação Servidora

B.2.1 Server.java

```
import java.io.*;
import java.net.*;
import java.util.*;


public class Server {

  public static void main(String[] args) throws java.io.IOException {


    String usage = "Usage: java Server [-G group] [-P port] [-T ttl]
[-F file] [-D delay] [-P loss] [-B burst] [-K change key]
[-R resend key]";


    String threadName = "serverApp";
    String address = "230.0.0.1";
    int port = 4446;
    int ttl = 1;
    double lossProb = 0;
    int keyChange = 0;
    int keyResend = 0;
    String inputFile = "one-liners.txt";
    float delay = 1;
    int burst = 1;


    if ((args.length % 2) == 0) {

      for (int i = 0 ; i < (int)(args.length / 2) ; i++) {

        try {

          if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-G"))
  address = args[(int)(2*i+1)];
          if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-P"))
  port = Integer.decode(args[(int)(2*i+1)]).intValue();
          if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-T"))
  ttl = Integer.decode(args[(int)(2*i+1)]).intValue();
```

```
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-L"))
  lossProb = Double.valueOf(args[(int)(2*i+1)]).doubleValue();
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-K"))
  keyChange = Integer.decode(args[(int)(2*i+1)]).intValue();
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-R"))
  keyResend = Integer.decode(args[(int)(2*i+1)]).intValue();
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-F"))
  inputFile = args[(int)(2*i+1)];
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-D"))
  delay = Float.valueOf(args[(int)(2*i+1)]).floatValue();
            if (args[(int)(2*i)].toUpperCase(Locale.US).equals("-B"))
  burst = Integer.decode(args[(int)(2*i+1)]).intValue();

          } catch (NumberFormatException e) {
            System.out.println(usage);
            return;
          }

        }

        ServerApp serverApp = new ServerApp(threadName, address, port,
  ttl, lossProb, burst, keyChange, keyResend, inputFile, delay);
        serverApp.start();
        new ServerInput(serverApp).start();

      } else {

        System.out.println(usage);

      }

  }

}
```

B.2.2 ServerInput.java

```
class ServerInput extends Thread {


  public ServerApp serverApp = null;
```

```
  public ServerInput(ServerApp serverApp) {
    this.serverApp = serverApp;
  }


  public void run() {

    boolean finish = false;

    while (!finish) {

      char tecla;
      try {
        tecla = Character.toUpperCase((char)System.in.read());
      } catch (IOException e) {
        System.err.println("IOException: " + e);
        break;
      }

      if (tecla == 'Q') {
        finish = true;
        serverApp.finish();
      } else {
        if (tecla == 'K') {
          serverApp.changeKey();
        } else {
          if (tecla == 'R') {
            serverApp.sendKey();
          }
        }

      }

    }

  }

}
```

B.2.3 ServerApp.java

```
class ServerApp extends Thread {

  public int ONE_SECOND = 1000;
  public float delay;
  public BufferedReader in = null;
  public boolean finish = false;
  public SecMultiServer secServ = null;


  public ServerApp(String threadName, String address, int port,
  int ttl, double lossProb, int burst, int keyChange, int keyResend,
  String inputFile, float delay) throws IOException {

    super(threadName);
    InetAddress group = InetAddress.getByName(address);
    this.delay = delay;


    try {
      in = new BufferedReader(new FileReader(inputFile));
    } catch (FileNotFoundException e) {
      System.err.println("Could not open quote file. Serving time inste
    }

    System.out.println("Transmitting at multicast group " + group + ":"
port + " with TTL = " + ttl);
    System.out.println("Transmitting data from file '" + inputFile +
"' with a delay of " + delay + " seconds");
    System.out.println("Loss probability = " + lossProb*100 +
"% with a burst size of " + burst + " packets");
    System.out.println("Changing key every " + keyChange +
" packets and resending it every " + keyResend + " packets.");

    try {
      secServ = new SecMultiServer(group, port, ttl, lossProb, burst,
  keyChange, keyResend);
    } catch (IOException e) {
      System.err.println("IOException: " + e);
      return;
    }
```

```
}



public void run() {

  while (!finish) {

    secServ.sendData(getNextData());

    try {
      sleep((int)(delay * ONE_SECOND));
    } catch (InterruptedException e) { }

  }

  secServ.finish();

}



public String getNextData() {

  String returnValue = null;
  if (in == null) {
    returnValue = new Date().toString();
  } else {
    try {
      returnValue = in.readLine();
      if (returnValue == null) {
        in.close();
        finish();
        returnValue = "No more data.";
      }
    } catch (IOException e) {
      returnValue = "IOException occurred in server.";
    }
  }
  return returnValue;

}
```

```
  public void changeKey() {
    secServ.changeKey();
  }

  public void sendKey() {
    secServ.sendKey();
  }

  public void finish() {
    finish = true;
  }

}
```

B.3 Aplicação Cliente

B.3.1 Client.java

```
import java.io.*;
import java.net.*;
import java.util.*;



public class Client {

  public static void main(String[] args) throws java.io.IOException {

    String usage = "Usage: java Client userID [-G group] [-P port]";

    int userID = 0;
    String threadName = "ClientApp";
    String group = "230.0.0.1";
    int port = 4446;


    if ( (((args.length-1) % 2) == 0) & (args.length != 0) ) {

      userID = Integer.decode(args[0]).intValue();
```

```
      for (int i = 0 ; i < (int)(args.length / 2) ; i++) {

        try {
          if (args[(int)(1+2*i)].toUpperCase(Locale.US).equals("-G"))
  group = args[(int)(1+2*i+1)];
          if (args[(int)(1+2*i)].toUpperCase(Locale.US).equals("-P"))
  port = Integer.decode(args[(int)(1+2*i+1)]).intValue();
        } catch (NumberFormatException e) {
          System.out.println(usage);
          return;
        }

      }

      ClientApp clientApp = new ClientApp(threadName, userID, group, po
      ClientInput clientInput = new ClientInput(clientApp);
      clientInput.start();

      clientApp.clientInput = clientInput;

    } else {

      System.out.println(usage);

    }

  }

}
```

B.3.2 ClientInput.java

```
class ClientInput extends Thread {

  public boolean finish = false;
  public ClientApp clientApp;


  public ClientInput (ClientApp clientApp) {
    this.clientApp = clientApp;
  }
```

```
  public void run() {

    while (!finish) {

      char tecla;
      try {
        tecla = Character.toUpperCase((char)System.in.read());
      } catch (IOException e) {
        System.err.println("IOException: " + e);
        break;
      }

      if (tecla == 'Q') {
        clientApp.finish();
      }

    }

  }


  public void finish() {
    finish = true;
  }


}
```

B.3.3 ClientApp.java

```
class ClientApp {

  public boolean finish = false;
  public BufferedReader bin;
  public PipedInputStream pin;
  public ClientInput clientInput;
  public SecMultiClient secCli;
  public boolean finished = false;
```

```
  public ClientApp(String name, int userID, String address, int port)
  throws IOException {

    InetAddress group = InetAddress.getByName(address);


    pin = new PipedInputStream();
    bin = new BufferedReader(new InputStreamReader(pin));

    System.out.println("Listenning to multicast group " + group +
":" + port + ".");

    secCli = new SecMultiClient(this, userID, group, port);
    secCli.start();

  }




  public void readData() {

      String data = new String();
      try {
        data = bin.readLine();
      } catch (IOException e) {
        System.err.println(e);
      }

      System.out.println("Received data: '" + data + "'");

  }


  public void finish() {
    if (!finished) {
      clientInput.finish();
      secCli.finish();
      finished = true;
    }
  }
```

```
}
```

B.4 Classes auxiliares

B.4.1 BlowFish.java

```
import cryptix.provider.key.RawSecretKey;
import cryptix.util.core.Hex;
import xjava.security.Cipher;

class BlowFish {

  public static String encrypt(String plaintext, String keyString) {

    Cipher alg;
    try {
      alg = Cipher.getInstance("Blowfish", "Cryptix");
    } catch (java.security.NoSuchAlgorithmException e) {
      System.err.println("NoSuchAlgorithmException: " + e);
      return null;
    } catch (java.security.NoSuchProviderException e) {
      System.err.println("NoSuchProviderException: " + e);
      return null;
    }

    RawSecretKey key = new RawSecretKey("Blowfish", Hex.fromString(keyS



    byte pad = (byte)(8 - (plaintext.length() % 8));
    for (int i = 0 ; i < pad-1 ; i++) {
      plaintext = plaintext.concat(" ");
    }
    plaintext = plaintext.concat(Byte.toString(pad));



    try {
      alg.initEncrypt(key);
```

```
    } catch (java.security.KeyException e) {
      System.err.println("KeyException: " + e);
      return null;
    } catch (Exception e) {
      System.err.println("Exception: " + e);
      return null;
    }


    byte[] plainbyte = plaintext.getBytes();
    byte[] cipherbyte = alg.crypt(plainbyte);
    String ciphertext = new String(cipherbyte);


    return ciphertext;

  }






  public static String decrypt(String ciphertext, String keyString) {

    String plaintext;
    byte[] plainbyte, cipherbyte;

    Cipher alg;
    try {
      alg = Cipher.getInstance("Blowfish", "Cryptix");
    } catch (java.security.NoSuchAlgorithmException e) {
      System.err.println("NoSuchAlgorithmException: " + e);
      return null;
    } catch (java.security.NoSuchProviderException e) {
      System.err.println("NoSuchProviderException: " + e);
      return null;
    }

    RawSecretKey key = new RawSecretKey("Blowfish", Hex.fromString(keyS



    try {
      alg.initDecrypt(key);
```

```
    } catch (java.security.KeyException e) {
      System.err.println("KeyException: " + e);
      return null;
    } catch (Exception e) {
      System.err.println("Exception: " + e);
      return null;
    }


    cipherbyte = ciphertext.getBytes();
    plainbyte = alg.crypt(cipherbyte);
    plaintext = new String(plainbyte);


    try {
      byte pad = new Byte(plaintext.substring
  (plaintext.length()-1,plaintext.length())).byteValue();

      boolean goodPlaintext = true;
      for (int i = 1 ; i < pad  ; i++) {
        if ( !plaintext.substring
(plaintext.length()-i-1,plaintext.length()-i).equals(" ") ) {
          goodPlaintext = false;
        }
      }

      if (goodPlaintext) {
        plaintext = plaintext.substring(0, plaintext.length() - pad);
      } else {
        plaintext = null;
      }
    } catch (Exception e) {
      plaintext = null;
    }


    return plaintext;

  }

}
```

B.4.2 HashMD5.java

```
import cryptix.util.core.Hex;
import java.security.MessageDigest;

class HashMD5 {

  public static String hash(String plaintext) {

    MessageDigest alg;
    try {
      alg = MessageDigest.getInstance("MD5", "Cryptix");
    } catch (java.security.NoSuchAlgorithmException e) {
      System.err.println("NoSuchAlgorithmException: " + e);
      return null;
    } catch (java.security.NoSuchProviderException e) {
      System.err.println("NoSuchProviderException: " + e);
      return null;
    }

    alg.reset();

    byte[] plainbyte = plaintext.getBytes();
    byte[] x = alg.digest(plainbyte);
    String hashValue = Hex.toString(x);


    return hashValue;

  }

}
```

B.4.3 CharFile.java

```
import java.io.*;


public class CharFile {
```

```
    public static void writeInt(FileWriter outFile, int number, char ma
throws IOException {

    outFile.write(Integer.toString(number) + mark);

  }



  public static int readInt(FileReader inFile) throws IOException {

    char[] buf = new char[0];
    char temp = (char)inFile.read();
    int i = 0;

    while (temp != ':' && temp != ';' && temp != ',' && temp != ' ') {

      char[] bufNew = new char[i+1];
      for (int j = 0 ; j < i ; j++) {
        bufNew[j] = buf[j];
      }
      buf = bufNew;
      buf[i] = temp;
      temp = (char)inFile.read();
      i++;

    }

    return(Integer.decode(new String(buf).substring(0,i)).intValue());

  }



  public static void writeString(FileWriter outFile, String string,
  char mark) throws IOException {

    outFile.write(string + mark);

  }
```

```
  public static String readString(FileReader inFile) throws IOException

    char[] buf = new char[0];
    char temp = (char)inFile.read();
    int i = 0;

    while (temp != ':' && temp != ';' && temp != ',' && temp != ' ') {

      char[] bufNew = new char[i+1];
      for (int j = 0 ; j < i ; j++) {
        bufNew[j] = buf[j];
      }
      buf = bufNew;
      buf[i] = temp;
      temp = (char)inFile.read();
      i++;

    }

    return(new String(buf).substring(0,i));

  }

}
```

B.4.4 GenerateUsers.java

```
import java.io.*;
import cryptix.util.core.Hex;


public class GenerateUsers {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) {
      System.err.println("Usage: java GenerateUsers <NumberOfUsers>");
      return;
    }
```

```
    String dataKey = Hex.longToString
( (long)(Long.MAX_VALUE * Math.random()) );


    int users = Integer.decode(args[0]).intValue();
    int[] userID = new int[users];
    String[] userKey = new String[users];

    FileWriter fileServerUserKeys = new FileWriter("user_keys.dat");

    CharFile.writeInt(fileServerUserKeys, users, ':');

    for (int i = 0 ; i < users ; i++) {

      userID[i] = i;
      userKey[i] = Hex.longToString( (long)(Long.MAX_VALUE * Math.rando

      CharFile.writeInt(fileServerUserKeys, userID[i], ',');
      CharFile.writeString(fileServerUserKeys, userKey[i], ';');

      FileWriter fileUserKey = new FileWriter("user_key_"+userID[i]+".d
      CharFile.writeString(fileUserKey, userKey[i], ';');
      fileUserKey.close();

      FileWriter userDataKey = new FileWriter("data_key_"+userID[i]+".d
      CharFile.writeString(userDataKey, dataKey, ';');
      userDataKey.close();

    }

    fileServerUserKeys.close();

    FileWriter serverDataKey = new FileWriter("data_key.dat");
    CharFile.writeString(serverDataKey, dataKey, ';');
    serverDataKey.close();

  }

}
```