Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Este trabalho tem por objetivo comparar e estudar quatro algoritmos de criptografia de chave simétrica conhecidos, publicados e padronizados: DES (*Data Encryption Standard*) [DES, 1977], 3DES (*Triple Data Encryption Standard*) [3DES, 1996], AES (*Advanced Encryption Standard*) [TANENBAUM, 2003] e Blowfish [SCHNEIER, 1994]. Como ambiente para realização dos experimentos utilizou-se uma aplicação de rede cliente-servidor de fluxo de vídeo contínuo. Esta aplicação utiliza uma estrutura de redes de computadores do tipo *Ethernet* e transfere através desta rede *frames* de vídeo a partir de um servidor até um cliente.

Além destes algoritmos, outros dois relevantes algoritmos de chave pública são apresentados: Diffie-Hellman [DIFFIE, 1976] e RSA [RIVEST, 1978] [RSA, 2011]. No primeiro é apresentado um exemplo, através de um ataque conhecido, uma possível quebra deste protocolo, mas que com uma ligeira modificação produz-se um protocolo aparentemente inatacável [TERADA, 2000]. No segundo algoritmo demonstra-se como são feitas a criptografia e a decriptografia, como o par de chaves deve ser calculado de maneira eficiente e por que, baseado no problema da fatoração de números grandes em primos [RIESEL,1994], não existe nenhum algoritmo que fatore as chaves deste protocolo em tempo polinomial [COUTINHO, 2003].

Mesmo antes do surgimento dos computadores, o homem sempre se preocupou em proteger informações sigilosas para que pessoas não autorizadas não tivessem acessos a tais dados. Amores proibidos, comunicações em tempo de guerra, transações financeiras e declarações de imposto de renda são algumas motivações para desenvolver e aperfeiçoar a segurança na troca de mensagens [KAHN, 1966].

Com a proliferação dos computadores e dos sistemas de comunicação nos anos 60 [MENEZES, 1996] era crescente a demanda para proteger as informações em meios digitais e prover um serviço de comunicação segura.

No começo da década de 70, Feistel, trabalhando para a IBM, publica um artigo [FEISTEL, 1975], que culmina para a base do algoritmo DES, e em seguida é adotado pelo governo norte americano para a criptografia de informações não-classificadas [FIPS, 1977].

Outro marco importante ocorre em 1976, quando o artigo [DIFFIE, 1976] introduz um novo conceito de criptografia: a criptografia de chaves públicas. A segurança por trás deste método baseia-se na intratabilidade do problema do logaritmo discreto [TERADA, 2000] [STALLINGS, 2003].

Em [MENEZES, 1996] propõe-se uma taxionomia para classificar os diferentes tipos de algoritmos de criptografia.

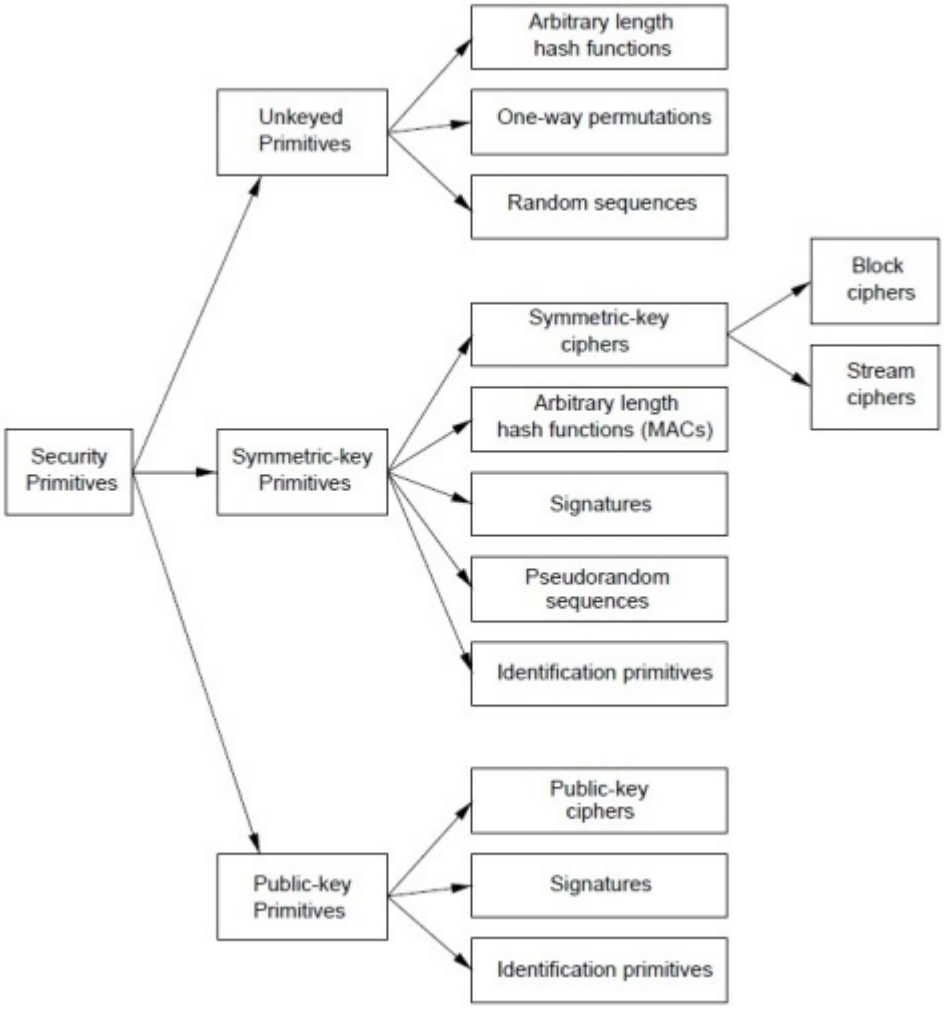

Figura 1 - Taxonomia de algoritmos criptográficos [MENEZES, 1996]

Na Figura 1 apresenta-se esta taxonomia. Nela os algoritmos criptográficos são divididos em três classes primitivas: de chaves simétricas, de chaves públicas e sem chaves. Um segundo nível de classes é proposto: neste nível os algoritmos sem chaves são subdivididos em funções *hash* de tamanho arbitrário, permutações *one-way* e sequencias randômicas. Já os

de chaves públicas subdividem-se em cifras de chaves públicas, assinaturas ou identificações primitivas. Por fim temos a subdivisão dos algoritmos de chaves simétricas: cifras de chaves simétricas, funções *hash* de tamanho arbitrário, assinaturas, sequencias pseudo-randômicas e identificações primitivas. Um terceiro nível de subdivisão é aplicado às cifras de chaves simétricas: cifras de blocos e de fluxo.

As cifras de bloco são as bases para o desenvolvimento dos algoritmos de criptografia de chave simétrica. Uma cifra de bloco é uma função definida por E: $\{0,1\}^k$ x $\{0,1\}^n \rightarrow \{0,1\}^n$, onde esta notação, significa que E recebe duas entradas: a primeira é um *string* de *k-bit* e a segunda um *string* de *n-bit*. A primeira entrada é chave e a segunda é o texto claro. A saída é conhecida como texto cifrado [GOLDWASSER, 2008].

Com a popularização da internet pelo mundo, a demanda por transmissões de vídeo vêm aumentando e se popularizando também. Transmissões em tempo real, sob demanda, vídeo conferências, *peer-to-peer*, são alguns dos exemplos do seu uso. Essas transmissões podem conter informações sigilosas ou possuir um conteúdo que não deve ser público. Então faz-se cada vez mais necessário o controle para sua distribuição através da Internet.

Através do uso de algoritmos de criptografia é possível controlar a distribuição dessas transmissões de vídeo garantindo que apenas pessoas autorizadas sejam capazes de acessar tais informações (isto é, possuam a chave correta para decriptografar corretamente os pacotes) mesmo que esta transmissão utilize a Internet como ambiente de distribuição.

Os algoritmos de criptografia implementados no experimento apresentado neste trabalho localizam-se na camada de aplicação da pilha de protocolos da Internet e são escritos na linguagem Java, usando métodos do *framework* JCE (*Java Cryptography Extension*). Para a análise de desempenho de cada algoritmo, utilizou-se o método *ThreadMXBean* do pacote *java.lang.management* do Java 5 [JAVA, 2011].

A aplicação multimídia de testes cliente-servidor utilizada é um exercício de programação do capítulo 7 do livro “Redes de Computadores e a Internet” [KUROSE, 2006] e seu código é parcialmente implementado e disponível para *download* no site do autor. Esta aplicação utiliza-se da visão de camadas de protocolos. O cliente usa o protocolo RTSP (*Real-Time Streaming Protocol*) para controlar ações sob o servidor. O servidor usa o protocolo RTP (*Real-Time Transfer Protocol*) para empacotar e enviar o vídeo em datagramas. Todos os pacotes trocados trafegam sob o protocolo de transporte UDP (*User Datagram Protocol*).

1.1. Organização deste trabalho

No Capítulo 2 apresentar-se-á um breve histórico da evolução da criptografia, as classes de cifras criptográficas e os modos de operação, que basicamente são substituições monoalfabéticas utilizando-se de caracteres grandes, são explicados. Os algoritmos de chave pública, RSA e Diffie-Hellman, também são apresentados neste Capítulo. Os algoritmos de chave simétrica DES, 3DES, AES e Blowfish são apresentados detalhadamente no Capítulo 3.

Uma visão geral da pilha da camada de protocolos da Internet será apresentada no Capítulo 4. Como os protocolos estudados são implementados nas camadas de aplicação e transporte, somente estas camadas são apresentadas com mais detalhe neste Capítulo.

A aplicação apresentada foi desenvolvida utilizando-se a linguagem de programação JAVA e para facilitar a implementação dos algoritmos de criptografia utilizou-se o *framework* JCE. A descrição técnica desta implementação e os desempenhos de cada algoritmo apresentados serão detalhados nos Capítulos 5 e 6.

Por fim, no Capítulo 7, apresenta-se possíveis trabalhos futuros para continuação deste trabalho e uma conclusão acerca de tudo o que foi apresentado e desenvolvido.

2. Criptografia

O uso da criptografia sempre esteve associado à proteção de segredos nacionais e estratégias. Sua história data desde o Antigo Egito, passando por duas grandes guerras mundiais e aplicando-se nos dias atuais às redes de computadores e os sistemas de comunicação, por exemplo, a Internet. Neste Capítulo apresentar-se-á um breve histórico da criptografia e como operam dois importantes algoritmos de chaves públicas: Diffie-Hellman e RSA.

2.1. Evolução

A palavra criptografia possui origem grega que significa "escrita secreta" e possui uma história de milhares de anos. Em [KAHN, 1966] cobre-se os aspectos da criptografia utilizada desde os Egípcios, 4.000 anos atrás, até o Século XX, quando teve papel importante durante as duas guerras mundiais.

É importante fazer uma distinção entre cifras e códigos. Uma cifra é uma transformação de caractere por caractere ou de *bit* por *bit*, sem levar em conta a estrutura linguística da mensagem. Em contraste, um código substitui uma palavra por outra palavra ou símbolo. Atualmente não faz-se mais sentido o uso de códigos [TANENBAUM, 2003].

Técnicas criptográficas permitem que um remetente embaralhe os dados de modo que um intruso não consiga obter nenhuma informação dos dados interceptados. O destinatário, é claro, deve estar habilitado a recuperar os dados originais a partir dos dados embaralhados [KUROSE, 2006].

As mensagens a serem criptografadas, conhecidas como texto claro (*plain text*), são transformadas por uma função que recebe uma chave. Após este processo gera-se uma saída, conhecida como texto cifrado (*chiper text*) e esta mensagem é transmitida. Pressupõe-se que o inimigo, ou intruso, consiga interceptar essa mensagem cifrada. No entanto, ao contrário do destinatário pretendido, ele não conhece a chave para decriptografar o texto e, portanto, não pode fazê-lo com muita facilidade [TANENBAUM, 2003].

Até o fim da década de 70, todos os algoritmos criptográficos eram secretos, principalmente aqueles utilizados pela diplomacia e pelas forças armadas de cada país. A famosa máquina de criptografia ENIGMA [TERADA, 2000], usada pelos alemães até a Segunda Guerra Mundial era totalmente secreta. A sua chave era definida através da posição de rotores.

Com a proliferação dos computadores e a crescente demanda por proteção das informações digitais, em 1977 adota-se um padrão de criptografia para informações não-classificadas [FIPS, 1977]. Este algoritmo de criptografia tornou-se bastante conhecido e utilizado em transações comerciais e financeiras por todo o mundo [MENEZES, 1996].

Em 1976 dois autores publicam um artigo que sugere um novo conceito para algoritmos criptográficos [DIFFIE, 1976]. A segurança deste método é baseado no problema do logaritmo discreto.

Até hoje não se conhece nenhum método matemático para provar que um algoritmo criptográfico é seguro. A maneira mais próxima deste ideal que se conhece é publicá-lo em conferências ou na Internet e tê-lo analisado pelos pesquisadores especializados que conheçam os métodos mais sofisticados para atacá-lo. Se o algoritmo passa por tal avaliação, a comunidade aceita-o como seguro em relação ao seu estado-da-arte [TERADA, 2000].

De forma geral, todos os algoritmos criptográficos, funcionam aplicando-se uma função matemática em um texto claro, tornando-o um texto cifrado. Para decifrar este texto basta-se aplicar a inversa desta função. Deve-se ressaltar que a função de codificação é conhecida, padronizada, publicada e disponível na Internet. Então o que garante a sergurança de um algoritmo é a sua chave secreta compartilhada apenas entre as duas entidades envolvidas.

A Figura 2 apresenta os componentes usados em uma criptografia simétrica. Aplicando-se uma função de criptografia $E\ (\ )$, que possui como entrada uma chave K e o texto claro P, gera-se um texto cifrado $C = E_K(P)$. Este texto cifrado C, submetido à função matemática de decriptografia $D\ (\ )$, com o parâmetro K, gera novamente o texto claro inicial, ou seja, $D_K(E_K(P)) = P$.

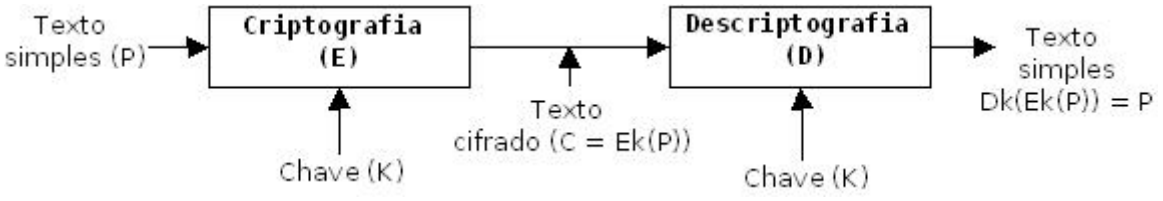

Figura 2 - O modelo de criptografia, utilizando chaves simétricas [TANENBAUM, 2003]

Os métodos de criptografia de chave simétrica são subdivididos em duas categorias: as cifras de substituição e as cifras de transposição. Cada uma dessas técnicas serviu de base para a criptografia moderna.

2.2. Cifras de substituição e transposição

Em uma cifra de substituição, cada letra ou grupo de letras é substituído por outra letra ou grupo de letras, de modo a criar um "disfarce". Um exemplo prático é a cifra de César.

Esta cifra funciona substituindo-se cada letra da mensagem pela *k*-ésima letra sucessiva do alfabeto (a letra "*z*" é seguida pela letra "*a*"). Por exemplo, se $k = 3$, então "*a*" deve ser substituída por "*d*" e "*b*" transforma-se em "*e*", e assim por diante. Por exemplo a mensagem "*estudar é importante*" tornar-se-á "*hvxgdu h lpsruxdqxh*". Embora o texto cifrado pareça não ter nexo, não levar-se-ia muito tempo para quebrar o código se soubesse que foi usada a cifra de César, pois há somente 25 valores possíveis para as chaves. Existe uma série de variações a partir deste modelo, mas todos são facilmente quebrados através do método da força bruta e da análise estatística da linguagem (ou contagem de frequência) do texto claro, por exemplo, na língua portuguesa a frequência das letras A, E, (geralmente mais frequentes), e Y, W (tipicamente menos frequentes) são particularmente distintas.

As cifras de substituição preservam a ordem dos símbolos no texto claro, mas disfarçam esses símbolos. Por outro lado, as cifras de transposição reordenam as letras, mas não as disfarçam. A Tabela 1 mostra uma cifra de transposição muito comum, a transposição de colunas.

A cifra se baseia em uma chave que é uma palavra ou frase que não contém letras repetidas. Nesse exemplo, "*hamlet*" é a chave. O objetivo da chave é numerar as colunas de modo que a coluna 1 fique abaixo da letra da chave mais próxima do início do alfabeto e assim por diante. O texto claro "*estudar é importante*" é escrito horizontalmente, em linhas. O texto cifrado é lido em colunas, a partir da coluna cuja letra da chave seja a mais baixa. Obtêm-se assim o texto cifrado "*set dpt err umn tia aoe*".

Tabela 1 - Aplicação de cifras de transposição em uma mensagem

H	**A**	**M**	**L**	**E**	**T**
3	1	5	4	2	6
e	s	t	u	d	a
r	e	i	m	p	o
r	t	a	n	t	e

Embora a criptografia moderna utilize as mesmas ideias básicas da criptografia tradicional (transposição e substituição), sua ênfase é diferente. Tradicionalmente os algoritmos eram simples. Hoje em dia, acontece o inverso: o objetivo é tornar o algoritmo de criptografia tão complexo e emaranhado, mesmo que a mensagem cifrada seja interceptada totalmente, sem a chave não será capaz de captar qualquer sentido na mensagem.

Os algoritmos de criptografia podem ser implementados em *hardware* (obtendo-se maior velocidade) ou em *software* (aumentando sua flexibilidade) [TANENBAUM, 2003].

Além das cifras de transposição e substituição, temos outra classe na qual as cifras podem ser classificadas: cifras de fluxo ou de bloco.

2.3. Cifras de fluxo e de bloco

Cifras de fluxo formam uma importante classe dos algoritmos de criptografia. Estas cifras criptografam caracteres individualmente (em geral dígitos binários) de um texto claro, um de cada vez, usado uma transformação criptográfica que varia com o tempo. Por contraste as cifras de bloco, tendem a criptografar simultaneamente um grupo de caracteres de um texto claro, usando uma transformação criptográfica fixa. As cifras de fluxo são mais apropriadas, quando o armazenamento (*buffering*) é limitado ou quando os caracteres necessitam serem processados individualmente à medida que são recebidos. Como possuem um nível baixo de erros de propagação, possuem vantagens em situações onde há grandes possibilidades de erros de transmissão. Um exemplo do seu uso é em aplicações de telecomunicações. Estas cifras podem ser com algoritmos de chave simétrica ou pública [MENEZES, 1996].

As cifras de bloco realizam basicamente uma cifra de substituição monoalfabética que utiliza uma cadeia grande de caracteres (128 *bits*, por exemplo). Sempre que o mesmo bloco de texto claro é processado o mesmo bloco de texto cifrado é gerado. Caso codifique-se o mesmo texto, usando a mesma chave, o mesmo texto cifrado será obtido [TANENBAUM, 2003].

A seguir são apresentados cinco modos de operação para criptografar um bloco de texto claro [FIPS, 1980]:

- Modo ECB – *Eletronic Code Book Mode*
- Modo CBC – *Cipher Block Chaining Mode*
- Modo CFB – *Cipher Feedback Mode*
- Modo OFB – *Output Feedback Mode*
- Modo CTR – *Counter Mode*

Estes modos podem ser utilizados em qualquer algoritmo de chave simétrica, incluindo o 3DES e o AES.

2.3.1. Modo ECB (*Eletronic Code Book Mode*)

O modo ECB (*Eletronic Code Book Mode*) é o mais natural dos quarto modos. Inicialmente divide-se o texto claro em blocos de comprimento fixo *t* (tipicamente de 64 *bits*) da entrada do algoritmo, obtendo-se *n* blocos, $x_1, x_2,...,x_n$, sendo o último bloco completado com brancos, se necessário, em seguida é aplicada a criptografia em cada bloco x_j separadamente.

Nestas condições, se houverem blocos repetidos no texto claro, haverão blocos correspondentes criptografados idênticos. Os blocos de texto criptografados podem ser interceptados e sua ordem de transmissão alterada, por exemplo, se dois blocos criptografados repetidos correspondem ao salário de duas pessoas, pode-se substituir um dos blocos por outro bloco ilegível que talvez corresponda a um salário maior ou menor.

Vulnerabilidades deste tipo tornam este modo, dentre os quatro apresentados, o menos usado.

2.3.2. Modo CBC (*Cipher Block Chaining Mode*)

Este modo de cifras, permite evitar o problema apresentado no modo anterior, pois mesmo que ocorram blocos repetidos no texto claro, os bloco criptografados não serão idênticos.

Na Figura 3 (a), apresenta-se um valor inicial *VI* (pode ser sempre fixo) que é submetido a um *XOR* () com o primeiro bloco claro P_0. Em seguida o bloco cifrado $C_0 = E(P_0 \oplus VI)$ é calculado e enviado ao destinatário e assim por diante.

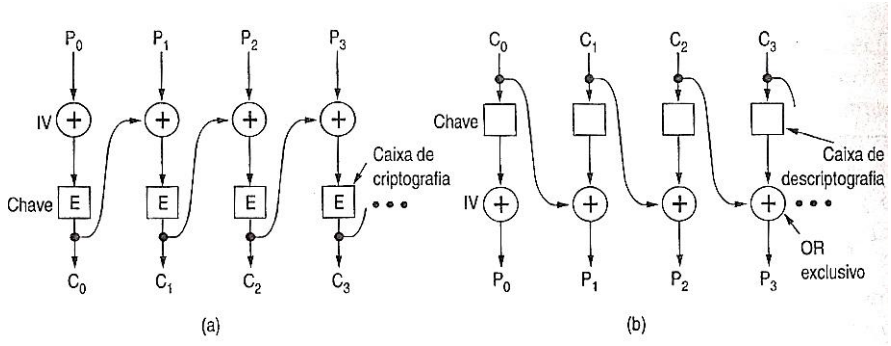

Figura 3 - Modo CBC. Codificação (a). Decodificação (b) [TANENBAUM, 2003]

Percebe-se que caso algum bloco seja alterado, digamos P_2, para P'_2, então C_2 será C'_2. Portanto uma alteração mesmo que pequena de pelo um *bit* em C_2 acarreta alterações nos valores dos blocos C_2, C_3, ...

A decriptografia é feita de maneira análoga como se mostra na Figura 3 (b).

O encadeamento de blocos possui uma vantagem em relação ao modo ECB: o mesmo bloco de texto claro não resultará no mesmo bloco de texto criptografado. Assim, a criptoanálise será difícil. De fato essa é a principal razão para o seu uso [TANENBAUM, 2003].

2.3.3. Modo CFB (*Cipher Feedback Mode*)

O modo CFB (*Cipher Feedback Mode)* permite criptografar blocos de texto claro de comprimento relativamente menor que no modo CBC. É possível definir tamanhos de blocos de um *byte* [TERADA, 2000]. Este é modo é utilizado em terminais interativos, por exemplo, onde menos de oito caracteres podem ser enviadas.

Na Figura 4 (a), o estado da máquina de criptografia é mostrado após os *bytes* 0 a 9 terem sido criptografados e enviados. Ao chegar o *byte* 10 do texto claro, o algoritmo opera sobre o registrador de deslocamento de 64 *bits* para gerar um texto criptografado de 64 *bits*. O

byte mais à esquerda desse texto criptografado é extraído e submetido a uma operação *XOR*

com P_{10}. O registrador de deslocamento (*shift register*) é deslocado 8 *bits* à esquerda, fazendo C_2 ficar fora da extremidade esquerda, e C_{10} é inserido na posição que acabou de ficar vaga na extremidade direita, logo depois de C_9. O conteúdo do registrador de deslocamento depende de todo o histórico anterior do texto claro; assim, um padrão que se repetir várias vezes no

texto claro será criptografado de maneira diferente do texto cifrado a cada repetição. Como ocorre no encadeamento de blocos de cifras, é necessário um vetor de inicialização para dar início ao processo [TANENBAUM, 2003].

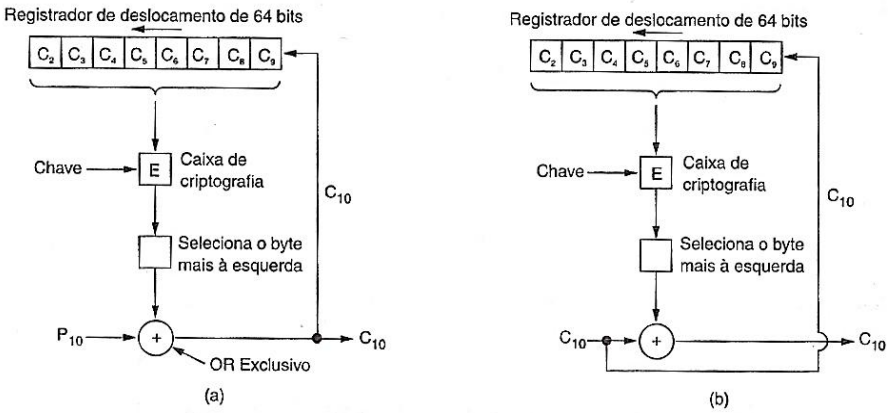

Figura 4 - Modo CFB. Codificação (a). Decodificação (b) [TANENBAUM, 2003]

A decriptografia funciona exatamente como a criptografia como se exibe na Figura 4 (b).

Uma vantagem do modo CFB em relação ao CBC é que poucos *bits*, *s bits*, são cifrados de cada vez. Porém uma desvantagem deste modo é caso um *bit* do texto cifrado for invertido acidentalmente, os 8 *bytes* decodificados enquanto o *byte* defeituoso estiver na função de deslocamento serão danificados. Após as rodadas com este *byte* defeituoso termina, o texto claro será corretamente gerado [TANENBAUM, 2003].

2.3.4. Modo OFB (*Output Feedback Mode*)

Existem aplicações em que um erro de transmissão de 1 *bit* alterando 64 *bits* de texto claro provoca um grande impacto. Para essas aplicações, há o modo OFB (*Output Feedback Mode*), que possui uma operação muito parecida com o modo CFB. Ele funciona criptografando um vetor de inicialização, com uma chave para obter um bloco de saída. O bloco de saída é então criptografado, usando-se a chave para obter-se um segundo bloco de saída. Em seguida, esse bloco é criptografado para obtendo-se um terceiro bloco e assim por diante. A sequência (arbitrariamente grande) de blocos de saída, chamada fluxo de chaves, é

tratada como uma chave única e submetida a uma operação *XOR* com o texto claro para se

obter o texto cifrado, como mostra a Figura 5 (a). O vetor de inicialização só é utilizado na primeira etapa. O fluxo de chaves é independente dos dados e, portanto pode ser calculado com antecedência, se necessário, e é insensível a erros de transmissão.

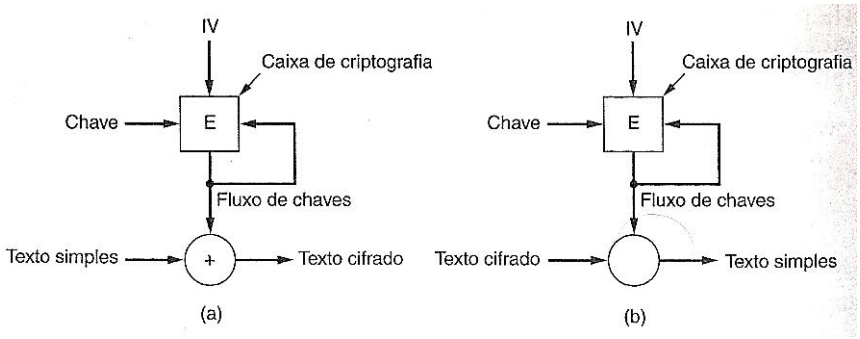

Figura 5 - Modo OFB. Codificação (a). Decodificação (b) [TANENBAUM, 2003]

A decriptografia ocorre gerando-se o mesmo fluxo de chaves no lado receptor, Figura 5 (b). Como o fluxo de chaves só depende do vetor de inicialização e da chave, ele não é afetado por erros de transmissão no texto cifrado. Desse modo, um erro de 1 *bit* no texto cifrado transmitido gera apenas um erro de 1 *bit* no texto claro decifrado.

2.3.5. Modo CTR (*Counter Mode*)

Neste modo de operação cada bloco de texto claro é submetido uma operação *XOR* com um contador de tamanho igual ao tamanho do bloco de texto. O requerimento exigido é que o valor deste contador seja diferente para cada bloco que será criptografado. Tipicamente, o contador é inicializado com algum valor e é incrementado em 1 para cada bloco subsequente (mod 2^b, onde b é o tamanho do bloco). Para criptografar, o contador é criptografado e uma operação *XOR* com o bloco de texto claro é processada, produzindo um bloco de texto cifrado. Não é gerado nenhum encadeamento entre os blocos seguintes. Para decriptografar, a mesma sequência de contadores é usada, e novamente uma operação *XOR* é aplicada entre os contadores criptografados e os blocos cifrados correspondentes. A saída desta operação é o texto claro original

2.4. Criptografia de Chave Pública

Até a década de 70, a comunicação cifrada exigia que as duas partes comunicantes compartilhassem a chave simétrica para criptar e decriptar. Para haver o compartilhamento dessa chave era preciso uma comunicação, mas caso essa comunicação fosse interceptada a chave não seria mais secreta.

2.4.1. Diffie-Hellman

Em 1976 dois autores Diffie, W. e Hellman, M.E. publicam um artigo [DIFFIE, 1976] que possui uma abordagem segura e radicalmente diferente dos algoritmos de chave simétrica.

Neste artigo foi proposto um modelo de criptografia chamado "Modelo de Chave Pública" em que cada usuário possui um par de chaves (*S, P*) sendo *S* a sua chave particular (secreta e guardada de forma segura) e *P* pública (facilmente acessada e disponibilizada publicamente). Estas chaves são relacionadas matematicamente de tal forma que:

- Se *x* denota um texto claro, e *S( )* a aplicação da chave *S*, que transforma *x* em $S(x) = y$ então $P(y) = x$, onde *P( )* denota a aplicação da chave *P*. Ou seja *S* é a chave inversa da chave $P - P(S(x)) = x$;
- O cálculo do par de chaves *(S, P)* é de tempo polinomial, ou seja, computacionalmente fácil;
- Para calcular *S* a partir do conhecimento de *P*, é computacionalmente difícil, ou seja, não se conhece um algoritmo para o cálculo em tempo polinomial;
- Os cálculos de *P( )* e *S( )* são computacionalmente fáceis para quem conhece as chaves;
- É computacionalmente difícil calcular *S( )* sem conhecer a chave *S*.

A seguir lista-se os passos deste protocolo para que seja possível combinar-se uma chave secreta K_{CS} entre um cliente e um servidor. Previamente ambos conhecem publicamente dois inteiros *g* e *p*, sendo *p* um primo longo e *g* tal que $0 < g < p$ gerador de Z^*_p [TERADA, 2000].

1. O cliente escolhe um número aleatório S_C, $1 \leq S_C \leq p - 2$;
2. O servidor escolhe um número aleatório S_S, $1 \leq S_S \leq p - 2$;
3. O cliente calcula $t_C = g^{Sc} \bmod p$ e o envia para o servidor;
4. O servidor calcula $t_S = g^{Ss} \bmod p$ e o envia para o cliente;
5. O cliente calcula $(t_S)^{Sc} \bmod p = K_{CS}$. Nota-se que $(t_S)^{Sc} = [g^{Ss} \bmod p]^{Sc} = g^{SsSc} \bmod p$.
6. O servidor calcula $(t_C)^{Ss} \bmod p = K_{CS}$. Nota-se que $(t_C)^{Ss} = [g^{Sc} \bmod p]^{Ss} = g^{ScSs} \bmod p$.

Este protocolo é baseado na dificuldade computacional do Problema do Logaritmo Discreto, que é "Dado um primo *p* e inteiros *g*, *t* : $0 < g, t < p$, calcular um inteiro *s* tal que $t = g^s \bmod p$." [TERADA, 2000].

Entretanto, este protocolo não garante a autenticidade nem do cliente e nem do

servidor, portanto pode ser quebrado por um intruso ativo (ataque conhecido como *man-in-the-middle*), como apresenta-se a seguir [TERADA, 2000]:

1. Quando o cliente envia t_C, o intruso bloqueia t_C e envia para o servidor outro inteiro $t_I = g^{Si}$ mod;
2. Cliente → t_C → Intruso → t_I → Servidor
3. O servidor recebe t_I como se fosse t_C e faz os cálculos necessários de acordo com o protocolo, estabelecendo uma chave K_{SI} com o Intruso, pensando falsamente que esta é a chave entre ele e o cliente;
4. O intruso bloqueia t_C e envia outro inteiro no seu lugar que pode ser o t_I usado antes;
5. O cliente recebe t_I como se fosse t_S e faz os cálculos estabelecendo uma chave K_{CI} com o Intruso, pensando falsamente que esta é a chave entre ele e o servidor.
6. A partir deste ponto tem-se a seguinte interceptação de pacotes por parte do Intruso
7. Cliente ⇔ K_{CI} ⇔ Intruso ⇔ K_{SI} ⇔ Servidor.

Porém uma ligeira modificação produz um protocolo útil e aparentemente inatacável [TERADA, 2000].

2.4.2. Diffie-Hellman modificado

Com esta modificação evita-se o ataque *man-in-the-middle*, para tanto, deve-se incluir um terceiro valor inteiro à chave pública, tornando-se (p, g, T) onde $T = g^S$ mod p [TERADA, 2000].

Portanto, o cliente publica previamente a sua chave pública (p_C, g_C, T_C) e mantém sua chave secreta S_C tal que $T_C = (g_C)^{Sc}$ mod p_C (analogamente o servidor faz o mesmo). O algoritmo modificado fica como a seguir, quando o cliente desejar enviar uma mensagem x para o servidor [TERADA, 2000]:

1. O cliente escolhe S_C e calcula $(g_S)^{Sc}$ mod $p_S = U_C$ e $K_{CS} = (T_S)^{Sc}$ mod p_S; usa K_{CS} para cifrar: $K_{CS}(x) = y$ e envia y e U_C para o servidor:
2. Cliente → $U_C = (g_S)^{Sc}$ mod p_S, $y = K_{CS}(x)$ → Servidor
3. O servidor calcula $[U_C]^{Ss}$ mod $p_S = [(g_S)^{Sc}$ mod $p_S]^{Ss}$ mod $p_S = K_{CS}$ e usa K_{CS} para decifrar y e obter x.

Desta forma não há mais necessidade de um diálogo entre cliente e servidor [TERADA, 2000].

2.4.3. RSA

O algoritmo RSA, publicado em 1978 e cujo os inventores são Ron Rivest, Adi Shamir, Leonard Adlemar. O RSA envolve um par de chaves, uma chave pública que pode ser conhecida por todos e uma chave privada que deve ser mantida em sigilo. Toda mensagem criptografada usando uma chave pública só pode ser decriptografada usando a respectiva chave privada. A seguir apresenta-se as propriedades que as suas chaves públicas e particulares devem possuir, como é realizada a criptografia e decriptografia e a sua segurança.

Inicialmente cliente e servidor devem calcular um par de chaves da seguinte maneira [TERADA, 2000] [RSA, 1978]:

1. Calcular dois números inteiros primos e longos (centenas de *bits*) chamados q e r; e calcular o seu produto $n = q * r$. Recomenda-se que o comprimento de q seja próximo do comprimento de r para tornar inviável a fatoração rápida de n em primos;
2. Calcular um terceiro número chamado s relativamente primo a $(q - 1) * (r - 1)$ (que é igual a $\Phi(n) = \Phi(q) * \Phi(r)$) e calcular um inteiro p que satisfaça $p * s = 1 \bmod (q - 1) * (r - 1)$ *i.e.*, $p = s^{-1} \bmod n$, através do Algoritmo de Euclides estendido;
3. A chave secreta $S = (s, n)$ é guardada com cuidado e a chave pública $P = (p, n)$ é publicada.

Para realizar-se a criptografia de um texto claro m expresso como um número inteiro $0 <= m <= n - 1$, usa-se a seguinte função: $c = m^p \bmod n$, onde c é o texto cifrado. A decriptografia é feito o seguinte processamento: $m = c^s \bmod n$, onde m é o texto claro original.

Este algoritmo é baseado na dificuldade computacional de fatorar um número inteiro em primos. Caso seja possível fatorar o número n em primos q e r, então ele pode-se calcular s que satisfaça $p * s = 1 \bmod (q - 1) * (r - 1)$ como efetuado no cálculo de um par de chaves. É por isso que recomenda-se que q e r tenham tamanhos próximos para dificultar a fatoração de $n = q * r$. Até hoje não se conhece um algoritmo em tempo polinomial para fatorar um número n "longo" em primos [TERADA, 2000].

Para ilustrar a dificuldade de fatorar um inteiro, quando o número n possui 129 algarismos decimais gastaria-se 5 mil MIPS-anos, um MIPS-ano significando usar um computador por um ano executando um milhão de instruções por segundo, utilizando um dos algoritmos mais rápido que se conhece, chamado QS (*Quadratic Sieve*) [POMERANCE, 1985]. O algoritmo QS possui tempo de execução proporcional a $e^{\sqrt{(\ln[n])(\ln\ln[n])}}$ [TERADA, 2000].

Outro algoritmo para fatoração chamado NFS (*Number Field Sieve*) [LENSTRA, 1993] com tempo de execução proporcional a $e^{1.92\,(\ln[n])^{\frac{1}{3}}(\ln\ln[n])^{\frac{2}{3}}}$ é mais rápido que QS para números com mais de 350 *bits*, por ser mais rápido assintoticamente que QS.

$$\lim_{n\to\infty}\left(\frac{e^{1.92\,(\ln[n])^{\frac{1}{3}}(\ln\ln[n])^{\frac{2}{3}}}}{e^{\sqrt{(\ln[n])(\ln\ln[n])}}}\right)^{n} = 0$$

Em 1999, Adi Shamir apresentou um computador específico para fatoração baseado em dispositivos opto-eletrônicos. Com uma implementação do algoritmo QS (ou do NFS) o TWINKLE [SHAMIR, 1999] consegue analisar 100 milhões de inteiros e determinar em menos de 10 milissegundos quais destes são fatoráveis completamente sobre uma base dos primeiros 200 mil primos. Shamir afirma que assim torna viável a fatoração de inteiros de comprimento entre 565 a 665 *bits*; isso torna vulneráveis os sistemas que utilizam RSA com chaves de 512 *bits*.

Em 1996, 512 *bits* para o módulo *n* do RSA era considerado razoavelmente seguro. Mas devido aos eventos mencionados e devido aos eventos mencionados e aos algoritmos QS e NFS, hoje se recomenda no mínimo 768 *bits*. Para longo prazo, recomenda-se desde já adotar 1024 *bits*. É importante também mencionar que até hoje os pesquisadores não conseguiram descobrir qualquer outro ponto fraco neste sistema de criptografia.

A exponenciação exigida é um processo que consome um tempo considerável. Como comparação, o DES, ao contrário, no mínimo, cem vezes mais veloz em software e entre mil e dez mil vezes mais veloz em hardware [RSA FAST, 2011] [KUSORE, 2006].

Portanto o RSA é frequentemente usado em combinação com o DES ou AES. Quando deseja-se transferir uma grande quantidade de *bits*, alta velocidade, por exemplo, uma transmissão de vídeo, primeiramente o servidor escolhe uma chave simétrica aleatória e denominada chave de sessão. Esta chave é cifrada usando um algoritmo de chave pública e é enviado ao cliente, este decifra a mensagem e obtém a chave de sessão. A partir desse momento a troca de dados passa a ser feita usando um algoritmo de chave simétrica, utilizando-se a chave de sessão, para cifrar e decifrar os pacotes.

Estes algoritmos criptográficos de chaves simétricas, suas vulnerabilidades e seus conceitos técnicos serão apresentados no Capítulo seguinte.

3. Criptografia de Chave Simétrica

Quando uma chave secreta é compartilhada entre dois sistemas e ambos os sistemas utilizam um mesmo algoritmo de criptografia, configura-se um cenário de criptografia de chave simétrica. Neste Capítulo serão apresentados os algoritmos XOR, DES, 3DES, AES e Blowfish que fazem parte deste grupo de algoritmos de chave simétrica ou chave compartilhada.

3.1. XOR

Ou-exclusivo chamada também "disjunção exclusiva", conhecido geralmente por *XOR*, é uma operação lógica em dois operandos que resulta em um valor lógico verdadeiro, se e somente se, exatamente um dos operandos tem um valor verdadeiro.

Em criptografia, esta operação pode ser aplicada em um texto claro, a fim de criar uma versão cifrada. Através de um algoritmo básico:

$$(textoClaro \oplus chave) = textoCifrado$$

Pode-se dizer que o *XOR* realiza uma operação reversível, pois se aplicarmos $A \oplus B$ e reaplicarmos o *XOR* no resultado com o mesmo B, teremos A, como vemos a seguir:

$$(A \oplus B) \oplus B = A$$

É baseado na reversibilidade da operação *XOR* que a técnica é usada em algoritmos criptográficos. Esta é uma técnica extremamente usada como um componente, ou uma etapa, em algoritmos de criptografia mais complexos. Usada isoladamente, com a mesma chave repetidas as vezes, torna-se vulnerável a ataques do tipo texto claro conhecido, já que *textoClaro* $\oplus$ *textoCifrado* = chave. A criptografia *XOR* é um tipo de algoritmo usado quando a segurança não é crucial para a aplicação. Porém, caso use-se uma chave randômica, que nunca se repita e com mesmo tamanho da mensagem, esta criptografia torna-se mais segura. O algortimo O*ne-time pad* [SHANNON, 1949] é um exemplo. Este algoritmo é derivado da cifra de Vernam. O sistema de Vernam é uma cifra que combina uma mensagem com uma chave lida de um laço da fita adesiva de papel. Em seu formulário original, o sistema de Vernam não é inquebrável porque a chave poder ser reutilizada. Em criptografia, O*ne-time pad* (OTP), ou cifra de chave única, é um algoritmo de criptografia onde o texto claro é combinado com uma chave aleatória ou uma "*pad*" que seja tão grande quanto o texto claro e é usado somente uma vez. Uma adição modular (por exemplo *XOR*) é usada para combinar o texto claro com a *pad*. Se a chave for verdadeiramente aleatória, nunca reutilizada, e mantida em segredo, a *one-time pad* pode ser inquebrável [KAHN, 1966]. Também provou-se que toda a cifra teórica inquebrável deve usar chaves com as mesmas exigências que chaves de OTP. Não foi atendida como padrão de mercado porque é trabalhoso gerar sempre chaves aleatórias grandes e manter em sigilo.

3.2. DES

O DES foi um padrão de criptografia de chaves simétricas desenvolvido em 1977 pelo *US Nation Bureau of Standards*. Este algortimo codifica um texto claro em porções de 64 *bits*, usando uma chave de 64 *bits*. Porém, para cada grupo de 8 *bits* há um *bit* de paridade, sendo assim a chave tem efetivamente 56 *bits* de comprimento, e assim é citado o tamanho da sua chave [KUROSE, 2006].

Em [FIPS, 1977] esboça-se o algoritmo DES da seguinte forma: Seja *M* a representação binária de uma mensagem em texto claro, de tamanho 64 *bits* e *K* uma chave de 56 *bits* aos quais são adicionados os *bits* de paridade, de forma que *K* tenha 64 *bits*. Inicialmente a chave é permutada de acordo com a Tabela 2.

Tabela 2 - Permutação a ser aplicada na chave *K* [FIPS, 1977]

PC-1						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Nesta tabela os *bits* de paridade foram excluídos (os *bits* 8, 16, 24, 32, 40, 48, 56 e 64 estão ausentes), portanto, esta operação só é efetuada depois da verificação de integridade da chave. Como a primeira entrada da tabela é "57", isto significa que o 57º *bit* da chave original *K* torna-se o primeiro *bit* da chave permutada. O 49º *bit* da chave original transforma-se no segundo *bit* da chave permutada. O 4º *bit* da chave original é o último *bit* da chave permutada. Apenas 56 *bits* da chave original aparecem na chave permutada. Obtém-se então a chave *K+*, resultado da permutação, que possui 56 *bits*.

3.2.1. Formação das subchaves

A chave *K+* é dividida em duas metades, esquerda C_0 e direita D_0, de 28 *bits*. Com C_0 e D_0 definidas criam-se 16 blocos C_n e D_n , para $1 <= n <= 16$, onde cada par é formado pelo par anterior C_{n-1} e D_{n-1} usando o esquema de deslocamentos a esquerda da Tabela 3.

Tabela 3 - Sub-chaves obtidas com deslocamentos à esquerda [FIPS, 1977]

Número da iteração	Número de deslocamentos à esquerda
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Tabela 4 - Permutação PC-2 aplicada aos blocos que foram deslocados [FIPS, 1977]

PC - 2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

As 16 sub-chaves K_n, para $1<=n<=16$, são obtidas a partir de cada bloco C_nD_n de acordo com a Tabela 4.

3.2.2. Criptografar bloco de 64 bits

A partir da representação em 64 *bits* *M*, da mensagem em texto claro, obtém-se a permutação *M'* segundo a Tabela 5. Então *M'* é dividida em 2 blocos de 32 *bits*, L_0 e R_0. Para cada par L_nR_n, utiliza-se a sub-chave K_n e a função *f ( )* para aplicar 16 iterações até obter o bloco cifrado $L_{16}R_{16}$ de acordo com a fórmula:

$$L_n = R_{n-1}$$
$$R_n = L_{n-1} + f(R_{n-1},K_n)$$

Para calcular *f ( )*, primeiramente expande-se cada bloco R_{n-1} de 32 para 48 *bits*, para se ajustar ao tamanho das sub-chaves. Isto é feito de acordo com a Tabela 6, uma tabela de

seleção que repete alguns dos *bits* em R_{n-1}. Sendo $E\ (\ )$ a operação de expansão, denotará-se por $E(R)$ o bloco R original expandido em 48 *bits*.

Tabela 5 - Permutação aplicada ao bloco de 64 *bits M* para se obter *M'* [FIPS, 1977]

IP - *Initial Permutation*							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Tabela 6 - Regra usada para expandir um bloco de 32 *bits* para 48 *bits* [FIPS, 1977]

Expansion (E)					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

A seguir, no cálculo de $f\ (\ )$, fazemos um *XOR* na saída $E\ (R_{n-1})$ com a chave K_n. O motivo de se utilizar o *XOR* lógico é porque este é reversível. Se $A \oplus B = C$, então $A \oplus C = B$ e $B \oplus C = A$. A reversibilidade é importante para poder reverter-se o processo quando ocorrer a decodificação da mensagem cifrada. A operação *XOR* resulta em um bloco de 48 *bits* que será quebrado em 8 blocos de 6 *bits* cada. Cada grupo de 6 *bits* será usado como endereço em tabelas denominadas "*S boxes"*. Os *bits* de 1 a 6 são B_1, *bits* de 7 a 12 são B_2, e assim por diante com *bits* de 43 a 48 sendo B_8, como descrito abaixo:

$$K_n + E(R_{n-1}) = B_1B_2B_3B_4B_5B_6B_7B_8$$

Tabela 7 - *S-Boxes* [FIPS, 1977]

S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Seja S_n é a função definida de acordo com a Tabela 7 e B é um bloco de 6 *bits*, então $S_n\ (B)$ é determinada da seguinte maneira: O primeiro e o último *bit* de B representam um número na base 2 com valor decimal entre 0 e 3 (ou binário 00 a 11). Que este número seja i.

Os 4 *bits* centrais de *B* representam um número na base 2 com valor decimal entre 0 e 15 (ou binário de 0000 a 1111). Que este número seja *j*. Procura-se o número na tabela localizado na *j*-ésima coluna e na *i*-ésima linha. Esse número varia de 0 a 15 e é unicamente representado por um bloco de 4 *bits*. Este bloco é a saída $S_n(B)$ de S_n para a entrada *B*.

Finalmente obtém-se *f ( )* pela permutação da concatenação da saída das *S-boxes* tendo os blocos B_n como entrada. A permutação é dada pela Tabela 8.

$$f = P(S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8))$$

O passo final do algoritmo DES é realizar a permutação IP^{-1}, como na Tabela 9, sobre o bloco $L_{16}R_{16}$ para obter o bloco cifrado.

Tabela 8 - Permutação final para obter a função *f ( )* [FIPS, 1977]

Permutação P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Tabela 9 - Permutação que resulta no bloco cifrado [FIPS, 1977]

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	52	18	58	26
33	1	41	9	49	14	57	25

O projeto do DES é inspirado em outro algoritmo da IBM chamado LUCIFER, que possui entrada e saída também de 64 *bits*, porém com uma chave mais longa: 128 *bits*. Acredita-se que o encurtamento da chave tenha sido proposital, a pedido do *National Security Agency* (NSA) dos EUA. Antes da sua adoção como padrão, DES foi alvo de fortes críticas que persistem até hoje. A primeira crítica é o decréscimo do comprimento da chave de 128

bits para 56 *bits*: isso torna muito mais economicamente viável o cálculo da chave secreta, mesmo por força bruta, i.e., enumerando as 2^{56} chaves. A segunda crítica é o segredo mantido quanto aos critérios de projeto da estrutura interna, os *S-boxes*. Os usuários do DES não possuem qualquer garantia de ausência, de pontos vulneráveis nos *S-boxes* que permita à NSA decriptografar sem conhecer chave em uso.

Em 1977, dois pesquisadores de criptografia de Stanford, Diffie, W. e Hellman, M.E., projetaram uma máquina para decifrar o DES e estimaram que ela poderia ser montada por um custo de 20 milhões de dólares. Com base em um pequeno trecho de texto simples e no texto cifrado correspondente, essa máquina poderia descobrir a chave através de uma pesquisa exaustiva do espaço de chaves de 2^{56} entradas em menos de um dia. Atualmente, essa máquina custaria bem menos de um milhão de dólares.

Em Julho de 1998, a *Eletronic Frontier Foundation* (EFF), anunciou a quebra de uma encriptação DES usando uma máquina "DES Cracker" construída com menos de U$ 250.000. O ataque durou menos de três dias. Em seguida a EFF publicou a descrição desta máquina, possibilitando a qualquer pessoa criar a sua própria máquina [EFF, 98].

Até 1999 o governo norte-americano exigia que o DES usado internacionalmente fosse restrito ao uso de 40 *bits* na chave. Mas uma chave de 40 *bits* corresponde a cerca de 1 trilhão de chaves possíveis, e em 18 minutos à razão de 1 bilhão de tentativas de chaves por segundo, pode-se calcular a chave correta de 40 *bits*.

3.3. 3DES

Se um algoritmo de criptografia é suscetível a uma quebra por busca de chaves (tamanho de chave inadequado), criptografar novamente o mesmo bloco de texto cifrado, mais de uma vez, pode-se aumentar a segurança do bloco. Muitas técnicas de múltiplas critptografias de blocos de n-*bits* são conhecidas. Um exemplo dessa técnica é o 3DES [MENEZES, 1996] [3DES, 1996].

Dada a potencial vulnerabilidade do DES à um ataque de força bruta, havia a necessidade de se encontrar uma alternativa. Uma abordagem adotada, ao invés de se criar um novo algoritmo e para preservar o investimento já feito em *software* e *hardware*, é realizar várias etapas de criptografia usando o DES com múltiplas chaves.

Acreditava-se inicialmente que seria suficiente aplicar duas etapas de criptografia, como na equação abaixo (sendo P o texto claro K_1 e K_2 duas chaves distintas e C o texto cifrado):

$$C = E_{K2}[E_{K1}[P]]$$

Para decriptografar a mensagem, é necessário que as chaves sejam aplicadas na ordem inversa:

$$P = D_{K1}[D_{K2}[C]]$$

Para validar essa nova abordagem, havia a necessidade de se garantir que essa múltipla criptografia não seria equivalente a alguma criptografia simples do DES. Foi demonstrado que para o conjunto de todas as chaves possíveis de 56-*bits*, dadas duas chaves K_1 e K_2, seria impossível achar uma chave K_3 de forma a satisfazer a equação abaixo [CAMP, 1992]:

$$E_{K2}[E_{K1}[P]] = E_{K3}[P]$$

No entanto, existe uma vulnerabilidade a essa abordagem que não depende de nenhuma particularidade do DES, e que funcionaria com qualquer bloco criptografado por outro método [STALLINGS, 2003], que será descrita na próxima seção.

3.3.1. Ataque Man-in-the-middle

O algoritmo conhecido como ataque *Man-in-the-Middle*, foi apresentado pela primeira vez em [DIFFIE, 1976]. É baseado na premissa de que se tivermos

$$C = E_{K2}[E_{K1}[P]]$$

Então

$$X = E_{K1}[P] = D_{K2}[C]$$

Dado um par conhecido de texto claro/texto cifrado, (*P, C*) o ataque procede como a seguir. Primeiramente, criptografa-se *P* para todos os 2^{56} valores possíveis da chave K_1. Esses valores são armazenados em uma tabela e ordenados pelo valor de *X*. A seguir, decriptografa-se *C* usando todos os 2^{56} valores possíveis para a chave K_2. À proporção que cada resultado da decriptografia é produzido o valor é comparado na tabela. Se o valor for encontrado, testa-se as duas chaves com um novo par texto claro/texto cifrado conhecido. Se as duas chaves produzirem o texto cifrado correto, elas são aceitas como as chaves certas.

Para qualquer texto claro *P*, existem 2^{64} possibilidades de texto cifrado que podem ser produzidas aplicando o DES duas vezes (2DES). Essa abordagem usa de fato chaves de 112-*bits*, portanto existem 2^{112} possibilidades de chave. Logo, em média, dado um texto claro *P*, o numero de chaves diferentes de 112 *bits* que produzirão um dado texto cifrado C é $2^{112}/2^{64} = 2^{48}$. Assim, o procedimento acima produzirá aproximadamente 2^{48} alarmes falsos para o primeiro par (*P, C*). Um argumento similar indica que com 64 *bits* adicionais de texto claro/texto cifrado conhecido, a taxa de alarmes falsos será reduzida para $2^{48-64} = 2^{-16}$.

Se o ataque *man-in-the-middle* for realizado sobre dois blocos de um conhecido par texto claro/texto cifrado, a probabilidade de se encontrar as chaves corretas é de $1 - 2^{-16}$. O resultado é que com texto claro conhecido o ataque será bem sucedido contra o 2DES com chaves de 112 *bits*, com um esforço de 2^{56}, não muito maior que os 2^{55} necessários para o DES simples [STALLINGS, 2003].

3.3.2. 3DES com duas chaves

Um contra ataque óbvio ao *man-in-the-middle* é usar três fases de criptografia, com três chaves diferentes. Isso eleva o esforço para um ataque com texto claro conhecido ser bem sucedido para 2^{112}. No entanto isso levaria a necessidade de as chaves serem de 56 x 3 = 168 *bits*.

Como alternativa, Tuchman propôs uma tripla criptografia usando apenas duas chaves [TUCHMAN, 1979]. O método consiste em cifrar o texto na sequência criptofra-decriptogra-criptografa, como a seguir:

$$C = E_{K1}[D_{K2}[E_{K1}[P]]]$$

A etapa de decriptografia é usada somente para usuários do 3DES poderem decifrar mensagens cifradas com DES simples e não possui qualquer significado criptográfico adicional.

Atualmente não existe ataque criptanalítico prático ao 3DES. Coppersmith [COPP, 1994] ressalta que o custo de uma pesquisa pela chave usando força-bruta no 3DES é da ordem de $2^{112} \sim (5 \times 10^{33})$ e estima que o custo cresce exponencialmente, comparável ao DES simples, excedendo 10^{52}.

3.3.3. 3DES com três chaves

Apesar dos ataques se demonstrarem não práticos, qualquer um que use o 3DES pode se sentir um pouco inquieto. Assim, já considera-se usar o 3DES com três chaves distintas como uma abordagem aprimorada [KALI, 1996a]. Usando três chaves distintas, o tamanho da chave passa a ser efetivamente 168 bits e o método de criptografia é como a seguir:

$$C = E_{K3}[D_{K2}[E_{K1}[P]]]$$

A compatibilidade retroativa com o DES é alcançada fazendo $K_3 = K_2$ ou $K_1 = K_2$.

3.4. AES

Em 1998, o *National Institute of Standards and Technology (NIST/USA)*, anunciou

uma “competição” para um nova cifra de bloco. Esta nova cifra iria substituir o DES. O problema do tamanho relativamente pequeno do espaço de chaves (2^{56}) do DES motivou esse esforço. Com o desenvolvimento de um novo algoritmo, haver-se-ia a possibilidade de um algoritmo com processamento mais rápido (em comparação ao DES) e tamanho de blocos de 128 *bits*, ao invés de apenas 64 *bits*. No verão de 2001, o NIST anunciou a sua escolha: o algoritmo chamado Rijndael fora o escolhido. Este algoritmo foi desenvolvido pelos belgas Joan Daemen e Vincent Rijmen.

O tamanho de bloco do AES é de n = 128 *bits* e sua chave, de tamanho k, é variável. Ela pode ser de 128, 192 ou 256 *bits*. O AES baseia-se em quatro funções para a sua execução, são elas: *Expand*, *SubBytes*, *Shift-rows*, e *Mix-columnss.* A seguir exibe-se o algoritmo AES [TANENBAUM, 2003] e também como é a implementação da função *Expand*.

```
função AES_K(M)
      (K_0,...,K_10) ← expand(K)
      state ← M XOR K_0
      para r = 1 até 10 faça
            state ← SubBytes(state)
            state ← shift-rows(state)
            se r <= 9 então state ← mix-columns(state) fim se
            state ← state XOR K_r
      fim para
retorne state

função expand(byte key[16],word w[44])
      word temp
      para i = 0 até i < 4 faça
            w[i] = (key[4*i],key[4*i+1],key[4*i+2],key[4*i+3])
      fim para
      para i = 4 até i = 44 faça
            temp = w[i -1]
            se (i mod(4) = 0) então
                  temp = SubWord(RotWord(temp))xor Rcon[i/4]
            w[i] = w[i – 4] xor temp
      fim para
```

A função *Expand* recebe um *string* de 128 *bits* e produz um vetor de onze chaves (K_0, ..., K_{10}). A chave K_0 será utilizada no início do cálculo e as outras chaves serão usadas durante

dez rodadas. Antes de inicializar as rodadas a chave K_0 é submetida a uma operação XOR em *state byte* por *byte*. O *array state* é responsável por guardar o status atual do dado e o seu tamanho é igual ao tamanho de entrada do texto claro. As etapas do AES são descritas a seguir.

SubBytes: a etapa denominada *SubBytes* é uma simples procura em uma tabela. AES define uma matriz quadrada de *bytes* de ordem 16, chamada *S-box* (Tabela 10), que contém a permutação de todos os 256 possíveis valores de 8-*bits*. Diferentemente do DES, que tem diversas caixas *S-boxes*, o AES possui apenas uma caixa *S-box*. Cada *byte* do *array State* é mapeado em um novo *byte* da seguinte maneira: os 4 *bits* mais significativos do *byte* são usados como valor para uma linha da *S-box* e os 4 *bits* menos significativos são usados como valor para uma coluna. Esses valores de linha e coluna são usados como índices na *S-box* para selecionar um único valor de saída de 8-*bits*.

Tabela 10 - AES S-*box*. Todos os valores estão em hexadecimal [GOLDWASSER, 2008]

AES S-boxes															
63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
e0	32	3a	0a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8
ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
70	3e	b5	66	48	3	f6	0e	61	35	57	b9	86	c1	1d	9e
e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Shift-rows: esta etapa consiste em uma simples permutação entre os *bytes* de cada linha do *array State*. A primeira linha não é alterada. Para a segunda linha é feito um deslocamento de 1-*byte* à esquerda. Para a terceira linha, deslocamento à esquerda de 2-*bytes* e para a quarta linha o deslocamento é de 3-*bytes*.

MixColumns: esta etapa opera em cada coluna separadamente. Cada *byte* de uma

coluna é mapeado em um novo *byte* em função do valor dos 4 *bytes* da coluna. A transformação é definida pela multiplicação do *array State* por uma matriz constante, sendo a multiplicação feita com o campo finito de Galois, GF(2^8) [TANENBAUM, 2003]. A matriz é definida, na Tabela 11.

AddRoundKey: é realizado um *XOR* dos 128-*bits* de *State* com os 128-*bits* da chave da rodada, coluna a coluna. A complexidade da expansão da chave da rodada mais a complexidade das outras etapas do AES garantem a segurança do algoritmo [STALLINGS, 2003].

Tabela 11 - Matriz utilizada na função *MixColumns* [TANENBAUM, 2003]

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Tendo em vista que cada etapa é reversível, a decriptografia pode ser feita executando-se o algoritmo no sentido inverso. Porém, também existe um artifício, pelo qual a decriptografia poder ser utilizada executando-se o algoritmo de criptografia com tabelas diferentes [TANENBAUM, 2003].

3.5. Blowfish

O algortimo Blowfish [SCHNEIER, 1994], possui uma etapa de 16 rodadas assim como o DES, blocos de 64 *bits* e chaves com tamanhos de até 448 *bits*. Uma etapa para a expansão das chaves cria 18 sub-chaves de 32 *bits* e 8 *S-boxes* de 32 *bits* derivados da chave de entrada [MENEZES, 1996].

O algoritmo Blowfish é uma cifra de bloco simétrica desenvolvida por Bruce Schneier [SCHNEIER, 1994]. Blowfish criptografa blocos de 64-*bits* de texto claro em blocos de 64-*bits* de texto cifrado e, foi desenvolvido para ter as seguintes características [STALLINGS, 2003]:

Rapidez: os dados são criptografados, em um processador de 32-*bits*, a uma taxa de 18 ciclos de relógio por *byte*;

Compacto: O Blowfish pode rodar em menos de 5K de memória;

Simplicidade: a simples estrutura do Blowfish é fácil de implementar e facilita a

determinação da força do algoritmo;

Flexibilidade: o tamanho da chave é variável e pode ser de até 448-*bits*. Isso permite flexibilidade entre maior segurança e maior rapidez.

O algoritmo consiste de duas partes, sendo elas a expansão da chave e a criptografia dos dados. A primeira consiste na transformação da chave em sub-chaves, totalizando 4168 *bits*. A segunda consiste de 16 fases, sendo que, em cada uma dessas, é feita uma permutação dependente da chave e uma substituição dependente da chave e dos dados.

O Blowfish usa chaves que variam de 32 a 448 *bits* (ou seja, de uma a quatorze palavras de 32-*bits*). Essa chave é usada para gerar 18 sub-chaves de 32 *bits* e 4 matrizes *S-box* 8 x 32 contendo um total de 1024 entradas de 32-*bits*.

As chaves são armazenadas no *array K*:

$$K_1, K_2, \ldots, K_J \qquad 1 \leq j \leq 14$$

As sub-chaves são armazenadas no *array P*:

$$P_1, P_2, \ldots, P_{18}$$

São 4 *S-boxes*, cada uma com 256 entradas de 32-*bits*:

$$S_{1,0}, S_{1,1}, \ldots, S_{1,255}$$
$$S_{2,0}, S_{2,1}, \ldots, S_{2,255}$$
$$S_{3,0}, S_{3,1}, \ldots, S_{3,255}$$
$$S_{4,0}, S_{4,1}, \ldots, S_{4,255}$$

Para gerar *P* e as *S-boxes*, são realizadas as seguintes etapas:

1. Inicializa-se *P* e as quatro S-boxes usando os bits da parte fracionária da constante π. Assim, os 32 *bits* mais significativos da parte fracionária de π se tornam P_1, em seguida os próximos 32 *bits* se tornam P_2 e assim por diante até $S_{4,255}$.
2. É realizado um *XOR bit*-a-*bit* entre *P* e *K*, re-usando palavras de *K* quando necessário. Por exemplo, para uma chave de tamanho máximo (14 palavras de 32 *bits*), $P_1 = P_1 \oplus K_1$, $P_2 = P_2 \oplus K_2$, ..., $P_{14} = P_{14} \oplus K_{14}$, $P_{15} = P_{15} \oplus K_1$, ...,

$P_{18} = P_{18} \oplus K_4$.

3. Criptografa um bloco de 64-*bits* contendo somente zeros usando os valores atuais de *P* e *S*; substitui P_1 e P_2 com a saída dessa criptografia.
4. Criptografa a saída da etapa 3 usando *P* e *S* e substitui P_3 e P_4 com a saída do texto cifrado.
5. O processo continua até que todos os elementos de *P* e *S* tenham sido atualizados, usando em cada etapa a saída do algoritmo Blowfish.

O processo de atualização de *P* e *S* pode ser resumido como abaixo:

$P_1,P_2 = E_{P,S}[0]$

$P_3,P_4 = E_{P,S}[P_1,P_2]$

...

$P_{17},P_{18} = E_{P,S}[P_{15},P_{16}]$

$S_{1,0}, S_{1,1} = E_{P,S}[P_{17},P_{18}]$

...

$S_{4,254}, S_{4,255} = E_{P,S}[S_{4,252}, S_{4,253}]$

Para criptografar, o Blowfish divide o texto claro em duas metades de 32-*bits LE0* e *RE0*. Esta etapa é descrita no pseudo código abaixo:

Para i =1 até 16 **faça**

$RE_i = LE_{i-1} \oplus P_i$

$LE_i = F(RE_i) \oplus RE_{i-1}$

Fim para

$$LE_{17} = RE_{16} \oplus P_{18}$$

$$RE_{17} = LE_{16} \oplus P_{17}$$

O texto crifrado está nas variáveis LE_{17} e RE_{17}. A entrada da função *F* é dividida em 4 *bytes a, b, c* e *d*, e a função é definida como $F[a,b,c,d] = ((S_{1,a} + S_{2,b}) \oplus S_{3,c}) + S_{4,d}$

Para decriptogrfar o algoritmo é análogo ao de criptografia, sendo a entrada o texto crifado de 64 *bits*.

Os algoritmos apresentados neste Capítulo podem ser implementados sobre qualquer tráfego de dados. A Internet é formada por uma pilha de protocolos que trafegam dados entre si, desta maneira os algoritmos de criptografia podem ser implementados em qualquer uma dessas camadas da pilha de protocolos da Internet. A descrição dos protocolos usados serão alvo de estudo do Capitulo seguinte.

4. A Pilha de Protocolos da Internet

Cada protocolo da internet possui uma especificação própria, formando uma pilha onde cada um só se comunica com o protocolo imediatamente acima ou abaixo. Neste trabalho somente as camadas de transporte e de aplicação são apresentadas.

4.1. Arquitetura da Internet

A Internet é um sistema que possui muitos componentes: inúmeras aplicações e protocolos, vários tipos de sistemas finais, conexões e meios físicos de enlace. Visando uma simplificação deste sistema, a Internet é dividida em camadas, modularizando e simplificando a implementação do serviço oferecido em cada camada. Cada camada contém seus protocolos padronizados e desenvolvidos para atender as necessidades específicas de cada nível. A Figura 6 apresenta as cinco camadas da pilha de protocolos da Internet: aplicação, transporte, rede, enlace e física.

Figura 6 - A pilha de protocolos da Internet [KUROSE, 2006]

O escopo deste trabalho concentra-se nas camadas de aplicação e transporte, portanto somente os protocolos destas camadas serão apresentados.

4.2. Camada de Transporte

A camada de transporte disponibiliza dois protocolos distintos para as camadas de aplicação e rede. O primeiro é o UDP (*User Datagram Protocol*) que provê um serviço não

orientado para conexão e não garante a entrega dos pacotes enviados. O segundo desses protocolos é o TCP (*Transmission Control Protocol*) que garante à aplicação solicitante um serviço confiável e orientado para conexão.

Ao projetar uma aplicação de rede, um desses dois protocolos deve ser especificado como o protocolo para a camada de transportes. Tipicamente aplicações de correio eletrônico, acesso à terminal remoto, web e transferência de arquivos são implementadas utilizando-se o TCP, enquanto serviços de tradução de nomes, protocolos de roteamento, telefonia por Internet (*VoIP*) e transmissões de multimídia utilizam o UDP. Não existe nenhuma norma ou padrão que defina qual protocolo deve ser utilizado, a escolha do protocolo dependerá diretamente do tipo de aplicação a ser desenvolvida.

Utilizando controle de fluxo, números de sequência, reconhecimentos e temporizadores, o protocolo TCP assegura que os dados sejam entregues corretamente e em ordem. Além disso, ele também provê o controle de congestionamento evitando assim, uma quantidade excessiva de tráfego na rede de computadores.

4.2.1. Protocolo UDP (*User Datagram Protocol*)

O UDP (*User Datagram Protocol*) é um protocolo da camada de transporte não orientado a conexões, ou seja, oferece um meio para as aplicações enviarem datagramas encapsulados sem que seja necessário estabelecer uma conexão. Implementado na camada de transporte, este protocolo exige que os datagramas sejam encapsulados, na camada de rede, usando o protocolo IP.

Na Figura 7, é apresentado o cabeçalho do UDP. Formado por quatro campos, cada um consistindo de 2 *bytes*, são eles: um campo para informar o número da porta de origem e outro para a porta de destino, um campo que informa a soma de verificação ou *checksum* e o campo de comprimento da mensagem, que especifica o comprimento do segmento UDP, incluindo o cabeçalho, em *bytes*. Os dados da aplicação ocupam o campo de dados do segmento UDP.

O protocolo UDP, não provê nenhuma garantia de entrega ou duplicidade de pacotes até o seu destino. Tudo isso cabe aos processos do usuário. Os dados são transmitidos apenas uma vez e a integridade é verificada pelo sistema de CRC (*Cyclic redundancy check*) de 16 *bits* (informação presente no campo *checksum*). Os pacotes corrompidos simplesmente são descartados sem que o transmissor tenha conhecimento. Portanto ele é recomendado para transmissão de dados pouco sensíveis, tais como fluxos de áudio e vídeo, serviços de tradução nome-número (DNS) ou transferência de arquivos simples (protocolo TFTP) [INS, 1979],

[RFC, 1350], [RFC, 768].

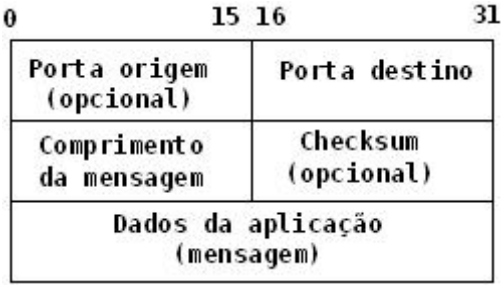

Figura 7 - Estrutura do protocolo UDP [RFC, 768]

Comparando-se com o protocolo TCP (*Transmission Control Protocol*), que também é implementado nesta mesma camada de rede, porém orientado para conexão, seu tempo de latência de rede é menor. Facilmente notado já que não é necessário garantir retransmissão, detecção e correção de pacotes corrompidos, congestionamento do fluxo ou espera de dados. O UDP provê suporte a *broadcasting* (transmisão de pacotes, onde o mesmo pacote é enviado para todos receptores ao mesmo tempo) e *multicasting* (ou *broadcast* multiplexado, porém usando a estratégia mais eficiente onde os dados só passam por um link uma única vez e somente são duplicadas quando o link para os destinatários se divide em duas direções). Além disto não é necessário estabelecer uma conexão ponto-a-ponto otimizando dessa forma a troca de pacotes.

4.3. Camada de Aplicação

É nesta camada que ocorre a interação homem-máquina, esta camada, localizada no topo da pilha de protocolos da Internet, comunica-se com a camada de transporte. Os protocolos presentes na camada de aplicação definem como processos de uma aplicação, que funciona em sistemas finais diferentes, passam mensagens entre si [KUROSE, 2006].

Alguns protocolos da camada de aplicação estão padronizados e são de domínio público. Por exemplo a camada de aplicação da Web, HTTP (HyperText Transfer Protocol) [RFC 2616], outros protocolos comum nesta camada são SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol), POP3 (Post Office Protocol - Version 3).

4.3.1. Protocolo RTP (*Real-time Protocol*)

O protocolo RTP (*real-time protocol*), é um padrão de domínio público para o encapsulamento de dados de áudio e vídeo. Este protocolo possui campos de cabeçalho extremamente úteis para aplicações deste tipo como: números de sequência, marcas de tempo e campos para dados de áudio e vídeo [RFC, 1889].

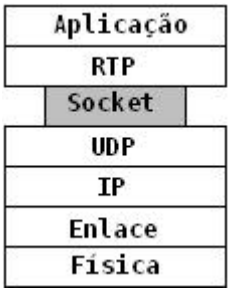

Figura 8 - Posição do RTP na pilha de protocolos [KUROSE, 2006]

O RTP opera da maneira descrita a seguir: a aplicação de multimídia consiste em vários fluxos de áudio, vídeo, texto e possivelmente outros fluxos. Esses fluxos são armazenados na biblioteca RTP, que se encontra no espaço do usuário, juntamente com a aplicação. Essa biblioteca efetua a multiplexação dos fluxos e os codifica em pacotes RTP, que são então colocados em um soquete. Na outra extremidade do soquete (no núcleo do sistema operacional), os pacotes UDP são gerados e incorporados a pacotes IP. Se o computador estiver em uma rede *Ethernet*, os pacotes IP serão inseridos em quadros *Ethernet* para transmissão. A pilha de protocolos para essa situação é mostrada na Figura 8 [TANENBAUM, 2003].

Como conseqüência dessa estrutura, é um pouco difícil dizer em que camada o RTP está. Como ele funciona no espaço do usuário e está vinculado ao programa aplicativo, certamente parece ser um protocolo de aplicação. Por outro lado ele é um protocolo genérico e independente das aplicações que apenas fornecem recursos de transporte, e assim também é semelhante a um protocolo de transporte. Talvez a melhor descrição do RTP seja como um protocolo de transporte implementado na camada de aplicação [TANENBAUM, 2003].

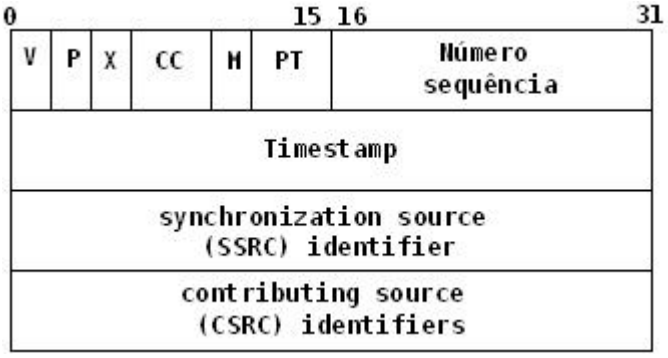

Figura 9 - Campos de cabeçalho do pacote RTP [RFC, 1889].

A função básica do RTP é multiplexar diversos fluxos de dados de tempo real sobre um único fluxo de pacotes UDP. O fluxo UDP pode senviado a um único destino (unidifusão)

ou a vários destinos (multidifusão) [TANENBAUM, 2003].

Cada pacote enviado em um fluxo recebe um número uma unidade maior que seu predecessor. Essa numeração permite ao destino descobrir se algum pacote está faltando. Como conseqüência, o RTP não tem nenhum controle de fluxo, nenhum controle de erros, nenhuma confirmação e nenhum mecanismo para solicitar retransmissões [TANENBAUM, 2003].

O cabeçalho do RTP é ilustrado na Figura 9. Ele consiste em três palavras de 32 *bits* e, potencialmente, algumas extensões. A primeira palavra contém o campo v*ersion* (*V*), que atualmente já está em 2. O *bit P* indica que o pacote foi completado até chegar a um múltiplo de 4 *bytes*, o último *byte* de preenchimento informa quantos *bytes* foram acrescentados. O *bit X* indica que um cabeçalho de extensão está presente. O formato e o significado do cabeçalho de extensão não são definidos. O único detalhe definido é que a primeira palavra da extensão fornece o comprimento. Essa é uma válvula de escape para quaisquer exigências imprevistas. O campo *CC* informa quantas origens de contribuição estão presentes, de 0 a 15. O *bit M* é um marcador específico da aplicação, ele pode ser usado para marcar o começo de um quadro de vídeo, o começo de uma palavra em um canal de áudio ou qualquer outro elemento que a aplicação reconheça. O campo *PT* (*payload type*) informa que algoritmo de codificação foi usado (por exemplo, áudio não-compactado de 8 *bits*, MP3, etc). Tendo em vista que todo pacote apresenta esse campo, a codificação pode mudar durante a transmissão. O campo número de sequência é apenas um contador incrementado em cada pacote RTP enviado. Ele é usado para detectar pacotes perdidos.

O timbre de hora (*timestamp*) é produzido pela origem do fluxo para anotar quando a primeira amostra no pacote foi realizada. Esse valor pode ajudar a reduzir a flutuação no receptor, desacoplando a reprodução no momento da chegada do pacote. O *synchronization source identifier* informa a que fluxo o pacote pertence. Esse é o método usado para multiplexar e demultiplexar vários fluxos de dados em um único fluxo de pacotes UDP. Finalmente, os campos c*ontributing source identifiers*, se estiverem presentes, serão usados quando houver misturadores (*mixers*) de áudio no estúdio. Nesse caso, o misturador será a origem de sincronização, e os fluxos que estão sendo mixados serão listados nesse campo [RFC, 1889].

Diferentemente dos protocolos HTTP e FTP, o RTP não realiza um *download* completo de todos os dados para o cliente. Ao invés disso, ele realiza um fluxo contínuo de transmissão de áudio e vídeo para o cliente. Por essas razões utilizou-se estes protocolos nesta aplicação cliente-servidor.

4.3.2. Protocolo RTSP (*Real-Time Streaming Protocol*)

Para permitir que seja possível controlar o fluxo contínuo em tempo real, o cliente e o servidor precisam de um protocolo para que ocorra esta troca de informações de controle. O RTSP (*Real-Time Streaming Protocol*), definido em [RFC, 2326], é este protocolo.

É importante que faça-se uma distinção entre os protocolos RTP e RTSP. O RTSP permite uma comunicação bidirecional, isto é, o cliente pode comunicar-se com o servidor de mídia e realizar pedidos de iniciar, pausar, avançar ou parar uma reprodução. O servidor recebe estas requisições e trata cada uma devolvendo uma resposta ao cliente. Já o RTP é um protocolo utilizado para envio de fluxo contínuo do servidor para o cliente. Desta forma é possível enumerar o que o RTSP não faz:

1. Não define esquemas de compressão para áudio e vídeo;
2. Não define como áudio e vídeo são encapsulados em pacotes para uma transmissão por uma rede; o encapsulamento pode ser fornecido por RTP ou por um protocolo proprietário;
3. Não restringe o modo como a mídia de fluxo é transportada; pode ser por UDP ou TCP;
4. Não restringe o modo como o cliente armazena o áudio e vídeo. Ele pode ser reproduzido logo que começar a chegar, após um atraso de alguns segundos ou pode ser descarregado integralmente antes de ser reproduzido.

O RTSP é um protocolo “fora da banda”. Em particular, as mensagens RTSP são enviadas “fora da banda”, ao passo que o fluxo de mídia, cuja estrutura não é definida pelo RTSP, é considerado “dentro da banda”. Mensagens RTSP utilizam a porta 544 e o fluxo de mídia utiliza um número diferente [KUROSE, 2006].

Devido à especificação do RTSP permitir que suas mensagens sejam enviadas por TCP ou UDP, neste trabalho as mensagens são enviadas através do UDP.

A Figura 10, apresenta um exemplo simples de uma interação entre cliente/servidor. Como ilustra a figura o cliente e o servidor enviam um ao outro uma série de mensagens RTSP. O cliente envia uma requisição RTSP *setup* e o servidor responde com uma mensagem RTSP *ok*. O cliente envia uma requisição RTSP *play* e recebe uma resposta com uma mensagem RTSP *ok*. Nesse ponto, o servidor de fluxo contínuo bombeia vídeo para dentro de seu próprio canal “dentro da banda” e a execução do vídeo inicia-se. Mais tarde, o cliente envia uma requisição RTSP *pause* e o servidor responde com uma mensagem RTSP *ok*. Quando o usuário termina, o cliente envia uma requisição RTSP *teardown* e o servidor

confirma com uma resposta RTSP *ok*.

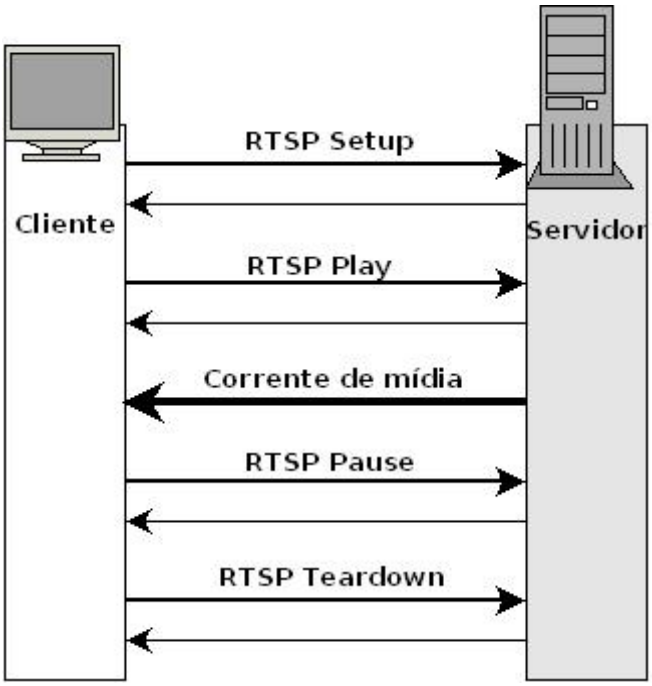

Figura 10 - Integração entre cliente-servidor usando RTSP [KUROSE, 2006]

Agora apresenta-se as mensagens RTSP propriamente ditas entre um cliente (C:) e um servidor (S:).

```
C: SETUP movie.Mjpeg RTSP/1.0
C: CSeq: 1
C: Transport: RTP/UDP; client_port= 25000; mode=PLAY

S: RTSP/1.0 200 OK
S: CSeq: 1
S: Session: 123456

C: PLAY movie.Mjpeg RTSP/1.0
C: CSeq: 2
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 2
S: Session: 123456

C: PAUSE movie.Mjpeg RTSP/1.0
C: CSeq: 3
C: Session: 123456
```

```
S: RTSP/1.0 200 OK
S: CSeq: 3
S: Session: 123456

C: TEARDOWN movie.Mjpeg RTSP/1.0
C: CSeq: 5
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 5
S: Session: 123456
```

Todas as mensagens de requisição e resposta são em texto ASCII, o cliente emprega métodos padronizados (*setup*, *play*, *pause* e assim por diante) e o servidor responde com códigos padronizados de resposta. O servidor RTSP monitora o estado do cliente para cada sessão em curso. Por exemplo, o servidor monitora se o cliente está em um estado de inicialização (*init*), de reprodução (*play*) ou em um estado de pausa (*pause*). Os números de sessão e de sequência, que fazem parte de cada requisição e resposta, ajudam o servidor a monitorar o estado da sessão. O número da sessão é fixo durante toda a comunicação, o cliente incrementa o número de sequência cada vez que envia uma nova mensagem e o servidor devolve um eco com a sessão e o número de sequência corrente.

Figura 11 - Diagrama de estados para um cliente usando o RTSP [KUROSE, 2006]

Como demonstra-se o cliente inicia a sessão com a requisição *setup*, fornecendo a *url* do arquivo que deverá ser transmitido e a versão do RTSP. A mensagem de estabelecimento (*setup*) inclui o número de porta do cliente para o qual a mídia deve ser enviada. Essa mensagem também indica que a mídia deve ser enviada por UDP usando o protocolo de empacotamento RTP.

Portanto outra característica do RTSP é que ele é capaz de manter um estado para cada

sessão criada. O cliente é capaz de mudar seu estado, quando recebe do servidor uma resposta à sua requisição, de acordo com o diagrama de estados da Figura 11.

5. Aplicação de teste cliente-servidor

Neste capítulo apresenta-se a aplicação cliente-servidor que será utilizada nos experimentos de análise dos algoritmos de criptografia. Uma descrição geral das classes também é apresentada.

5.1. Visão geral

A Internet comporta uma grande variedade de aplicações de multimídia interessantes – vídeo em tempo real, telefonia IP (*Internet Protocol*), rádio por Internet, teleconferência, ensino à distância – são somente alguns exemplos dessas aplicações. Aplicações de multimídia são altamente sensíveis a atraso, porém em sua maioria são tolerantes à perda. Perdas ocasionais causam somente pequenas perturbações na recepção de áudio e vídeo, e essas perdas podem ser parcialmente ou totalmente encobertas pela aplicação [KUROSE, 2006].

Devido a grande variedade de aplicações multimídia presentes na Internet, pode-se classificar em três classes essas aplicações: áudio e vídeo de fluxo contínuo armazenado, áudio e vídeo de fluxo contínuo ao vivo e áudio e vídeo interativos em tempo real [KUROSE, 2006].

A aplicação apresentada neste trabalho classifica-se na primeira classe, por este motivo menciona-se a seguir três características fundamentais desta classe:

Mídia armazenada: o conteúdo multimídia está armazenado completamente em um servidor. Por esta razão um usuário que tenha acesso a esse servidor pode voltar, avançar ou parar a transmissão;

Fluxo contínuo (streaming): numa aplicação cliente-servidor armazenado, o cliente inicia a reprodução do áudio e vídeo alguns segundos após começar a receber o arquivo do servidor. Desta forma o arquivo não necessita ser descarregado inteiro, para começar a ser reproduzido;

Reprodução contínua: ao iniciar-se a reprodução do áudio e vídeo, ela deve prosseguir de acordo com a cadência original da gravação. Isso impõe sérias restrições à variação de atraso na entrega de dados.

Tomando-se como base o exercício prático do Capítulo 7, do Livro "Redes de Computadores e a Internet" [KUROSE, 2006], implementou-se alguns métodos que não estavam presentes nas classes JAVA que simulam um servidor e um cliente, numa aplicação de fluxo contínuo de vídeo. Essas mesmas classes foram modificadas para permitirem o uso de algoritmos de criptografia, dessa forma simulando no servidor a cifragem do pacote de dados e no cliente a decifragem deste pacote.

A chave de criptografia a ser utilizada por cada algoritmo está presente no próprio código do algoritmo. Dessa forma no código do cliente e no código do servidor existe uma chave armazenada.

O *framework* utilizado possui disponível métodos de chaves públicas que podem serem utilizados para criar uma chave de sessão entre o cliente e o servidor evitando que seja necessário pré-estabelecer essa chave no código da aplicação. Porém por uma limitação de tempo optamos por não desenvolver a criação desta chave de sessão através de algoritmos de chaves públicas.

Das quatro classes que serão apresentadas a seguir, as classes Cliente e Servidor, foram modificadas para permitirem o uso dos algoritmos de criptografia.

5.2. Cliente

Esta classe representa a interface do usuário. Exibe o vídeo e permite controlar o fluxo de dados, através do envio de comandos RTSP ao servidor.

Esta classe deve ser invocada da seguinte maneira:

```
java Client [host] [porta RTSP] [arquivo vídeo] [algoritmo criptografia]
```

Os parâmetros, respectivamente, são: endereço de *host* do servidor de mídia, porta RTSP na qual o servidor irá realizar a comunicação, o nome do arquivo de vídeo no servidor e qual algoritmo de criptografia será utilizado.

No momento em que o cliente é inicializado, cria-se um *socket*, através de uma porta para comunicação com o servidor. É através deste *socket* que os dados de comunicação RTSP serão enviados e recebidos.

O cliente reserva uma área de memória de 15.000 *bytes* para armazenar cada pacote recebido do servidor. Essa área armazena temporariamente cada pacote UDP que é recebido através do *socket* de conexão com o servidor. Implementou-se nessa classe as ações do

cliente, quando os botões de controle são acionados. Estes botões, seus tratamentos e ações são descritos a seguir:

Setup

1. cria um *socket* para receber os dados RTP e possui um tempo de expiração no *socket* para 5 milisegundos. Caso o pacote não chegue nesse tempo um aviso é levantado, mas a execução continua, quando o pacote for recebido. Quando os dados não são criptografados este tempo é suficiente, porém aplicando-se algoritmos de criptografia, necessitou-se aumentar esse tempo para 1.000 milisegundos, já que este foi o maior tempo registrado para se processar um pacote cifrado;
2. envia uma requisição *setup* para o servidor. No cabeçalho de transporte (UDP), especifica-se a porta para o socket de dados RTP criado;
3. aguarda-se a resposta do servidor e analisa-se o cabeçalho de sessão na resposta para obter o ID da sessão. Caso a resposta seja "200 ok", o RTSP muda para o estado *ready*. Inicializa o número de sequência do RTSP com o valor 1.

Play

1. envia uma requisição play;
2. incrementa o número de sequência do RTSP em uma unidade. No cabeçalho desta requisição encontra-se o ID de sessão fornecido na resposta ao setup;
3. se houver uma resposta do tipo "200 ok", vinda do servidor, o RTSP passa para o estado *playing*;
4. inicializa a variável de tempo "*timer*" e seu valor será sempre usado no cabeçalho dos pacotes UDP.

Pause: similar à requisição *play*. Com exceção para a mudança de estado do RTSP para *ready*, caso a resposta seja do tipo "200 ok". Além disso, a variável de tempo "*timer*" é parada, caso esse botão seja requisitado.

Teardown: possui o mesmo comportamento das requisições *play* e *pause*. Após uma resposta válida do servidor, seu estado é alterado para *init*, a variável de tempo "*timer*" é parada e o programa é encerrado.

Nesta classe encontra-se o método "*timerListener*" que manipula os pacotes UDP recebidos. Este método é o único que precisou ser alterado nessa classe e suas modificações são apresentadas no Apêndice A. É nele que os pacotes cifrados serão decifrados.

Neste método a primeira ação é ler, da área de memória temporária, o pacote recebido.

Porém como esta área tem um tamanho de 15.000 bytes, só os *bytes* contendo dados realmente devem ser lidos. Para saber qual é o tamanho usa-se o método *length* da estrutura de dados *array* do Java. Assim cria-se o pacote UDP com o dados que estão nesta área de memória e de tamanho igual a *length*.

O passo seguinte é extrair o conteúdo de dados do UDP. Este campo, contém todo o pacote RTP, que está cifrado. Então este conteúdo é decifrado e só então, o pacote RTP é criado, com seu cabeçalho e dados.

5.3. Servidor

Responde às requisições RTSP, realiza o tratamento destas requisições através de classes Java, que encapsulam os pacotes RTP que em seguida são enviados ao cliente, contendo os dados de um quadro de vídeo. Executa-se esta classe da seguinte maneira:

```
java Server [porta RTSP] [algoritmo criptografia]
```

Onde, tem-se respectivamente a porta RTSP que se estabelece a comunicação com o cliente e o algoritmo de criptografia para esta comunicação.

Implementou-se o empacotamento dos dados de vídeo em pacotes RTP. Criado o pacote necessário, ajustou-se os campos no cabeçalho do pacote e copiou-se a carga útil (o *frame* do vídeo) dentro do pacote.

Quando o servidor recebe a requisição *play* do cliente, aciona-se um temporizador que é ativado a cada 100ms. Esse tempo indica que a cada 100ms um novo frame de vídeo deve ser lido, encapsulado num pacote RTP e enviado para o destinatário. Esse tempo é armazenado no cabeçalho do pacote RTP, no campo *timestamp,* como pode ser visto na Figura 12. Em seguida este pacote RTP é cifrado (incluindo seu cabeçalho) e seu conteúdo é copiado para o campo de dados do pacote UDP.

O servidor chama o primeiro construtor da classe *RTPpacket* para realizar o encapsulamento. Nesta classe o método "*actionPerformed*" é responsável por realizar o envio dos pacotes contendo os *frames* de vídeo. Portanto é nesta classe que ocorre a cifragem dos pacotes. Todo o pacote RTP (incluindo seu cabeçalho) é armazenado no vetor "*packetBits*", esses dados serão cifrados por um dos algoritmos selecionados. O tamanho do dado é calculado, para incluir-se esta informação no cabeçalho do protocolo de transporte UDP e enviar ao cliente.

Da mesma forma como a classe Cliente reserva uma área de memória para receber os pacotes UDP através do *socket*, o servidor também reserva uma área temporária de memória de 15.000 bytes para enviar os pacotes UDP através do *socket* estabelecido com o cliente.

5.4. *RTPpacket*

Realiza o tratamento dos pacotes RTP. Responsável por empacotar e descompactar os pacotes RTP e serve tanto ao cliente como o servidor. Não foi necessário implementar ou modificar nenhum método nesta classe.

Esta classe é idêntica para o cliente e para o servidor, pois é a unidade de comunicação entre ambas.

Na Figura 12, demonstra-se como é o cabeçalho deste pacote. O campo *RTP-version* (*V*) é 2, os campos *padding* (*P*), *extension* (*X*), *number of contributing sources* (*CC*) e *marker* (*M*) são todos 0.

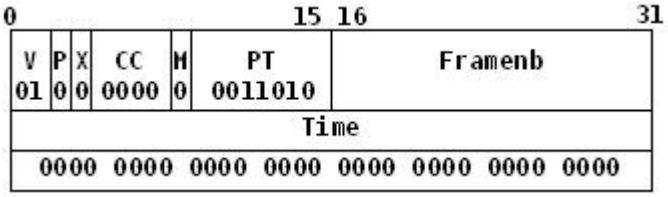

Figura 12 - Esquema do cabeçalho RTP usado na aplicação Servidor

O campo carga útil (*PT*), possui o valor 26, já que o tipo de codificação utilizado é o Mjpeg (*MotionJPEG*). O servidor fornece o número de sequência como argumento *Framenb* para o construtor desta classe. Da mesma forma preenche-se o campo *timestamp* com o argumento *Time.* O identificador da fonte (*SSRC*) identifica o servidor. Optou-se por utilizar o valor 0, como forma de simplificação. Como não há nenhuma outra fonte de contribuição (campo *CC* = 0), o campo *CSRC* não existe. Então, o comprimento do cabeçalho do pacote é de 12 *bytes*.

Estes valores foram utilizados na disposição "*header*" da classe *RTPpacket*. A carga útil (fornecida como argumento "*data"*) é copiada para a variável "*payload*". O comprimento da carga útil é dado no argumento "*data_length*".

Este diagrama encontra-se na ordem de *byte* de rede (também conhecido como *big-endian*). A *Java Virtual Machine* usa a mesma ordem de *byte*, então não é preciso transformar o cabeçalho de pacote na ordem de *byte* de rede.

5.5. *VideoStream*

Esta classe é usada para ler os dados de vídeo do arquivo em disco. Assim como a classe *RTPpacket*, não foi necessário implementar ou modificar nenhum método.

O formato de vídeo usado nesta aplicação é o tipo Mjpeg, que basicamente consiste numa sequência de imagens JPEG, ou seja cada quadro é uma imagem comprimida,

codificada numa sequência de *bytes*. Cada quadro é enviado individualmente em um pacote RTP que pode ser recebido por um cliente que decodifique tal formato. O uso deste formato é comum em câmeras digitais, câmeras IP e é nativo no *browser* Firefox.

A partir da apresentação da aplicação de testes que foi utilizada neste trabalho e com o conhecimento adquirido através dos estudos da pilha de protocolos da Internet e dos algoritmos de criptografia, é possível analisar o desempenho de cada algoritmo aplicado numa transmissão de vídeo através da Internet.

5.6. *Framework* JCE

Na aplicação cliente-servidor foi utilizado o *framework* JCE (*Java Encryption Extension*) para implementar os algoritmos de criptografia. Foi criada uma classe para cada algoritmo e métodos para cifrar e decifrar os blocos de texto claro, usando a classe *Cipher* da JCE. Os métodos "encripta" e "decripta" são análogos e foram construídos da seguinte forma:

1. Usa-se a classe *KeySpec* para criar o objeto que irá armazenar a chave;
2. Usa-se a classe *SecretKeyFactory* para criar o objeto que seleciona o algoritmo. São passados como parâmetro o algoritmo, o modo de cifragem e o *padding*;
3. Converte-se o objeto da chave através do objeto do algoritmo usando o método *generateSecret* da classe *SecretKeyFactory*;
4. Cria-se um objeto da classe *Cipher*, que será responsável pela cifragem ou decifragem. Essa opção é definida no momento de sua inicialização;
5. O método *doFinal* recebe como entrada um *array* de *bytes* do texto claro e retorna um *array* com o texto cifrado.

6. Desempenho dos algoritmos

A fim de obter-se uma comparação entre o desempenho de cada algoritmo apresentado nos capítulos anteriores, apresenta-se agora os resultados encontrados através de envio de pacotes, em uma rede de computadores utilizando aplicação de testes cliente-servidor.

6.1. Ambiente

Os experimentos executados neste trabalho foram realizados no Laboratório de Ciência da Computação, do Instituto de Computação da UFF, utilizando dois computadores, ambos com as seguintes configurações: Processador AMD Sempron 2800+ 1.61 GHz, 512 MB de memória RAM, sistema operacional Windows XP e máquina virtual Java versão 6 atualização 22. A tecnologia da rede utilizada é do tipo *Ethernet*, utilizando uma topologia estrela, ou seja, os computadores estão conectados um a um *switch* com velocidade de 100 Mbps.

Para medir o desempenho de cada algoritmo utilizou-se um arquivo de vídeo, no formato Mjpeg com tamanho de 4,07 MB (4.269.893 *bytes*). Este vídeo contém 500 *frames*, de tamanhos variados, cada um representando uma imagem JPEG. Cada frame é cifrado pelo servidor antes de ser enviado e ao chegar no cliente, ele é decifrado. Durante todo o processo de transferência do arquivo de vídeo a mesma chave é utilizada. Um player no lado cliente reproduz no formato de imagens cada *frame* à medida que seu pacote é recebido. O tempo de uso gasto pela CPU para cifrar cada *frame* foi registrado e calculamos uma média aritmética para chegar aos resultados obtidos nos gráficos apresentados. O mesmo método foi aplicado para registramos a média de *overhead* dos pacotes após o uso de cada um dos algoritmos.

Devido à simplicidade do *player* utilizado nesta aplicação e ao tempo reduzido para realizar este trabalho não foi possível utilizar outros arquivos de vídeo ou até mesmo utilizar outros formatos para os experimentos com os algoritmos de criptografia.

Para analisar-se o cálculo do tempo gasto de CPU, utilizou-se o método

ThreadMXBean do pacote *java.lang.management* do Java 5 [JAVA, 2011]. Este pacote possui licença *free* e seu código-fonte e documentação estão disponíveis na Internet para *download*. Desta forma, possibilitou-se monitorar a execução da JVM (*Java Virtual Machine*) e medir o tempo de uso de CPU por cada *thread* do algoritmo. Apesar dos testes terem sido realizados em um ambiente multitarefa, devido a esta abordagem os tempos não são afetados por outras atividades do sistema operacional. A implementação destes métodos encontra-se no Apêndice B deste trabalho.

A Tabela 12 mostra o tamanho de cada chave utilizado pelos algoritmos deste experimento. O *framework* utilizado possui uma limitação de tamanho de chave de 128 *bits* para o algoritmo AES. Por esta razão não foi possível executar o algoritmo utilizando tamanho de chaves diferentes.

Tabela 12 - Tamanho das chaves utilizadas

	Chaves (*bits*)
DES	56
3DES	56
AES (CBC)	128
AES (ECB)	128
BlowFish	128

6.2. Resultados

Após a execução dos experimentos iniciamos a análise dos resultados, através das médias aritméticas de cada algoritmo executado. Ou seja, a cada processamento dos 500 pacotes, registramos os dados relativos aquele pacote e em seguida obtivemos uma média geral. Desta análise coletamos o tempo de processamento dos pacotes para ser cifrado no servidor e o tempo de processamento para este mesmo pacote ser decifrado no cliente.

A Tabela 13 sumariza os valores obtidos na cifragem dos pacotes nesse experimento. A coluna "*bytes* processados" indica a quantidade média de *bytes* cifrados em cada pacote RTP transferido. A coluna seguinte apresenta o tempo médio gasto para cifrar cada pacote, antes de ser transferido. Por fim, a última coluna indica a capacidade de vazão (*throughput*) de processamento de cada algoritmo baseando-se nas informações das duas colunas anteriores. Por exemplo, para o tipo de conteúdo transferido entre o servidor e o cliente, o

algoritmo *Blowfish* processa aproximadamente 0,0000288 *bytes* a cada segundo. Este dado diz respeito ao tempo de processamento de cada algoritmo em função da quantidade de *bytes* processados.

Esta tabela mostra como os algoritmos AES e *Blowfish*, possuem um desempenho melhor em comparação ao DES e 3DES. É possível perceber também que o algoritmo *Blowfish* possui uma vantagem quando se comparando a sua capacidade de vazão. Por estes resultados permite-se evidenciar que o AES é superior ao DES e 3DES. O 3DES possui um terço do DES de capacidade de vazão, ou seja, ele precisa de três vezes mais tempo para processar a mesma quantidade de dados.

Tabela 13 - Comparação de resultados

	Bytes processados	Tempo gasto (ms)	B/segundos
AES (CBC)	8551	343,750	0,000024875636
AES (ECB)	8551	328,125	0,00002606019
Blowfish	8551	296,875	0,000028803368
DES	8551	312,500	0,0000273632
3DES	8551	815,625	0,000010483985

O algoritmo DES utiliza uma chave de 56 *bits* e blocos de 64 *bits*. Por esta cifra utilizar uma rede de Feistel, o mesmo processamento é gasto para cifrar e decifrar um bloco de dados. Isto comprovou-se na execução deste experimento. O tempo gasto para decifrar foi de 312,400ms. O 3DES utiliza o mesmo conceito, porém executando o mesmo algoritmo do DES três vezes e como esperado seu tempo de decifragem, de 814,900ms, foi aproximadamente igual ao tempo de cifragem.

O algoritmo Rijndael (AES) é uma cifra de bloco iterativo, com um bloco fixo de 128 bits e uma chave que pode ser de 128, 192 ou 256 bits. Ao contrário do seu predecessor DES, o AES é uma rede de permutação-substituição e não uma rede de Feistel. Isso quer dizer que o processo de descriptografia faz uso de códigos parcialmente diferentes. A diferença de código é identificada na operação de *MixColumns*, que usa uma estrutura polinomial diferente correspondendo a uma inversa da operação de criptografia, e assim eleva uma maior complexidade para a descriptografia.

Usando o algoritmo AES no modo CBC gastou-se em média 945,336 ms para se descriptografar os pacotes de vídeo, tempo superior ao gasto pelo algoritmo 3DES para

realizar e mesma descriptografia. No modo ECB este tempo foi de 811,200 ms.

O algoritmo *Blowfish* possui uma chave de tamanho variável e blocos de entrada de 64 *bits*. Ele divide-se em duas fases: a fase de expansão das chaves e a fase de cifragem dos dados. Na fase de cifragem, 16 rodadas da rede de Feistel são aplicadas. A decifragem do *Blowfish* é direta. O bloco a ser decifrado seguirá o fluxo na mesma direção do bloco quando foi cifrado, entretanto, as sub-chaves geradas na fase de expansão estarão em ordem inversa. Por este motivo o tempo de decifragem registrado para o *Blowfish* foi o mesmo para a cifragem.

Devido ao ambiente e a topologia da rede em que os experimentos foram executados não verificou-se nenhuma perda de pacotes durante a transmissão dos vídeos usando algoritmos de criptografia para cifrar e decifrar os pacotes.

7. Conclusão e trabalhos futuros

Neste trabalho implementou-se os métodos de criptografia do *framework* Java JCE, a fim de prover uma segurança na transmissão de um *streaming* de vídeo. Durante todo o desenvolvimento tomou-se o cuidado de não modificar o cabeçalho dos protocolos utilizados, já que há uma padronização a ser seguida evitando-se o risco de tornar a aplicação incompatível com outros clientes ou servidores que utilizem os mesmos protocolos. Outro objetivo deste trabalho foi de analisar o desempenho dos algoritmos de criptografia. Tivemos uma preocupação que esta análise não sofresse a interferência de tarefas em execução paralela. É importante ressaltar que todos os testes foram realizados em um ambiente *multithread* (ou multitarefa), por este motivo recorreu-se às implementações nativas da linguagem Java para se isolar e analisar somente as *threads* que realmente eram interessantes.

Uma dificuldade encontrada nestes experimentos e que havia sido subestimada inicialmente foi o uso de diferentes arquivos de vídeo, a fim de se testar exaustivamente os algoritmos aqui apresentados. Dada à simplicidade do *player* que é implementado na aplicação original não foi possível analisar uma variedade maior de arquivos. Vislumbra-se aqui uma importante possibilidade de trabalhos futuros evoluírem esta aplicação permitindo que outros formatos de vídeo possam ser executados.

Com a implementação desse trabalho, pudemos perceber a dificuldade em se acessar um dado criptografado e a importância de uma comunicação segura entre cliente-servidor. Também pudemos perceber a importância da segurança da informação utilizando-se em algoritmos de criptografia que impedem em um tempo computacionalmente viável que um usuário não autorizado acesse ao conteúdo do dado.

Como se percebe a aplicação utilizada nesses experimentos utiliza chaves simétricas que são armazenadas previamente no cliente e no servidor. Em termos práticos, isso não é muito eficiente, pois seria necessário haver um meio seguro para a troca dessas chaves. Então uma possibilidade para criar esse ambiente seguro seria através do uso de chaves públicas.

Implementando-se os métodos de chaves públicas presentes no framework, seria possível criar uma chave de sessão em tempo real entre cliente e servidor, e a partir do estabelecimento dessa chave utilizar um algoritmo de chave simétrica para a troca de dados. Tem-se aqui mais uma possibilidade para trabalhos futuros.

Apêndice A

Neste apêndice são apresentados trechos do código implementados no servidor e no cliente para realizar a criptografia e decriptografia dos pacotes de dados.

Código 1: Cliente

```
//Constrói um datagrama para receber os dados do socket UDP
rcvdp = new DatagramPacket(buf, buf.length);

//Recebe o pacote UDP do socket
RTPsocket.receive(rcvdp);

//Os dados estão criptografados. A primeira coisa é decriptografá-los
//Não estou interessado nos 15000 bytes. Só naquilo que efetivamente //foi
preenchido
final byte[] encryptedBytes = Arrays.copyOfRange(rcvdp.getData(), 0,
rcvdp.getLength());

//Decriptografa
byte[] decryptedBytes = null;
switch (Client.algorithm) {
      case AES:
            decryptedBytes = decriptaAES(encryptedBytes);
            break;
      case BlowFish:
            decryptedBytes = decriptaBlowfish(encryptedBytes);
            break;
      case DES:
            decryptedBytes = decriptaDES(encryptedBytes);
            break;
      case DESede:
            decryptedBytes = decriptaDES3(encryptedBytes);
            break;
      case XOR:
            decryptedBytes = criptaDecripta(encryptedBytes);
            break;
      default:
            throw new IllegalArgumentException("Algoritmo não previsto");
}

//Constrói o pacote com os dados decriptografados.
final RTPPacket rtpPacket = new RTPPacket(decryptedBytes,
decryptedBytes.length);
```

Código 2: Servidor

```
//Constrói um pacote RTP contendo um frame do vídeo
final RTPPacket rtpPacket = new RTPPacket(Server.MJPEG_TYPE, imagenb,
imagenb * Server.FRAME_PERIOD, buf, imageLength);

//Descobre o tamanho total do pacote a ser enviado
int packetLength = rtpPacket.getLength();

//recupera o pacote de streaming e armazena num vetor de bytes
final byte[] packetBits = new byte[packetLength];
```

```
rtpPacket.getPacket(packetBits);

System.out.println("Tamanho do texto claro: " + packetLength);

//Criptografa
byte[] encryptedBits = null;

switch (Server.algorithm) {
      case AES:
            encryptedBits = encriptaAES(packetBits);
            break;
      case BlowFish:
            encryptedBits = encriptaBlowfish(packetBits);
            break;
      case DES:
            encryptedBits = encriptaDES(packetBits);
            break;
      case DESede:
            encryptedBits = encriptaDES3(packetBits);
            break;
      case XOR:
            encryptedBits = criptaDecripta(packetBits);
            break;
            default:
      throw new IllegalArgumentException("Algoritmo não previsto");
}

      packetLength = encryptedBits.length;
      System.out.println("tamanho    do    texto    criptografado:    "    +
packetLength);

      //Envia o pacote como um datagrama através do protocolo UDP
      senddp     =     new     DatagramPacket(encryptedBits,     packetLength,
clientIPAddr, rtpDestPort);

      rtpSocket.send(senddp);
```

Apêndice B

A forma como foi implementado o método para análise de tempo gasto de CPU para execução de uma *thread* é apresentado a seguir:

```
ThreadTimes tt = new ThreadTimes();
long id = java.lang.Thread.currentThread().getId();

// ...
// execução do trecho de código a ser avaliado
// ...

// tempo gasto durante a execução da thread
long taskCpuTimeNamo = tt.getCpuTime(id);
```

Bibliografia

[3DES, 1996] COPPERSMITH, D; JOHNSON, D. B; MATYAS, S. M. A proposed mode for triple-DES encryption, *IBM Journal of Research and Development*, 40 ed, pp. 253–260. 1996.

[CAMP, 1992] CAMPBELL, K; WIENER, M. Proof that DES Is Not a Group. Em *Proceedings of the Crypto*, Nova Iorque: Springer-Verlag, 1992.

[COPPERSMITH, 1994] COPPERSMITH, D. The Data Encryption Standard (DES) and Its Strength Against Attacks. Em *IBM Journal of Research andDevelopment*, Maio 1994.

[COUTINHO, 2003] COUTINHO, S. C. *Números Inteiros e Criptografia RSA*. Rio de Janeiro: IMPA, 2003.

[DIFFIE, 1976] DIFFIE, W; HELLMAN, M.E. New Directions in Cryptography. Em *IEEE Transactions on Information Theory*, volume IT-22, pp. 644-654, 1976.

[FIPS, 1977] Data encryption standard, Em *Federal Information Processing Standards Publication,* volume 46, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1977.

[FIPS, 1980] DES modes of operation, Em *Federal Information Processing Standards Publication,* volume 81, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1980.

[GOLDWASSER, 2008] GOLDWASSER, Shafi; BELLARE, Mihir. *Lecture Notes on Cryptography*. Massachusetts, 2008. Disponível em: http://cseweb.ucsd.edu/~mihir/papers/gb.pdf. Acessado em 30 janeiro 2011.

[INS, 1979] POSTEL, J. *Internet Name Server*. Disponível em: http://www.networksorcery.com/enp/ien/ien116.txt. Acessado em 01 dezembro 2010.

[JAVA, 2011] Package java.lang.management. Disponível em: http://download.oracle.com/javase/1.5.0/docs/api/java/lang/management/package-summary.html. Acessado em 21 de junho de 2011.

[KALISKI, 1996] KALISKI, B; ROBSHAW, M. Multiple Encryption: Weighing Security and Performance. Em *Dr. Dobb's Journal*, Janeiro 1996.

[KAHN, 1966] KAHN, D. *The Code-breakers: The Comprehensive History Of Secret Communication From Ancient Times To The Internet*. 2. ed. Nova Iorque: Scribner, 1996.

[KUROSE, 2006] KUROSE, J. F; ROSS, K. W. *Redes de Computadores e a Internet*: *uma abordagem top-down*. 3. ed. São Paulo: Pearson Addison Wesley, 2006.

[LENSTRA, 1993] LENSTRA, A.K. et al. The number field sieve. Em *Lecture Notes in Mathematies,* volume 1554, pp. 11-12, Nova Iorque: Springer- Verlag, 1993.

[MENEZES, 1996] MENEZES, A; OORSCHOT, P. Van; VANSTONE, S. *Handbook of Applied Cryptography*. Nova Iorque: CRC Press, 1996

[POMERANCE, 1985] POMERANCE, C. The Quadratic Sieve Factoring Algorithm. Em *Proceeding of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques.* Nova Iorque: Springer-Verlag, pp. 169-182, Setembro 1985.

[RFC, 1350] SOLLINS, Karen R. The TFTP Protocol. Julho 1992. RFC 1350

[RFC, 1889] RTP: A Transport Protocol for Real-Time Applications. Janeiro 1996. RFC 1889

[RFC, 2326] SCHULZRINNE, H. Real Time Streaming Protocol. Abril 1998. RFC 2326.

[RFC, 768] POSTEL, J. User Datagram Protocol. Agosto 1980. RFC 768.

[RIESEL, 1994] RIESEL, H. *Prime Numbers And Computer Methods Of Factorization (Progress in Mathematics).* 2. ed. cap. 6, pp. 174-177, Boston: Birkhäuser Boston, 1994.

[RIVEST, 1978] RIVEST, R. L; SHAMIR, A; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Em *Communications of the ACM*, volume 21, n. 2, pp. 120-126. Nova Iorque: ACM, Fevereiro 1978.

[RSA, 2011] RSA LABORATORIES. *How fast is the RSA algorithm?* Disponível em: http://www.rsa.com/rsalabs/node.asp?id=2215. Acessado em 15 de maio de 2011.

[SHANNON, 1949] SHANNON, C. *Communication Theory of Secrecy Systems*. Bell System Technical Journal, pp 656–715, 1949.

[SCHNEIER, 1994] SCHNEIER, B. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). Em *Fast Software Encryption*, *Cambridge Security Workshop Proceeding*, pp. 191-204. Londres: Springer-Verlag, 1994.

[SHAMIR, 1999] SHAMIR, A. Factoring Large Numbers with the TWINKLE Device. Em *Lecture Notes in Computer Seience*, volume 1717, Nova Iorque: Springer-Verlag, 1999.

[STALLINGS, 2003] STALLINGS, William. *Cryptography and Network Security: Principles and Practice*. 3. ed. Prentice Hall, 2003

[TANENBAUM, 2003] TANENBAUM, A. S. *Redes de Computadores*. 4. ed. São Paulo: Campus, 2003

[TERADA, 2000] TERADA, R. *Segurança de Dados: criptografia em redes de computador*. São Paulo: Edgard Blücher, 2000.

[TUCHMAN, 1979] TUCHMAN, W. Hellman Presents No Shortcut Solutions to DES. Em *IEEE Spectrum*, Julho 1979.