Relatório Trabalho 01 – Teleinformática e Redes 1

Professor: Marcos Caetano

Alunos: Rafael Lopes *11/0056001*

Lucas Vanderlei *11/0015975*

Eliandra Sandes *11/0148053*

Isabella Queiroz *11/0031423*

João Victor Aguiar *10/0107565*

Algoritmos de detecção e correção de erros:

CRC-8:

Esse algoritmo tem por objetivo o controle de redundância cíclica (CRC). É um meio de controle de integridade dos dados potente e fácil de aplicar. Representa o principal método de detecção de erros utilizado nas telecomunicações.

O controle de redundância cíclico consiste em proteger blocos de dados, chamados tramas (frames). Cada trama é associada a um bloco de dados, chamado de controle (às vezes por abuso de linguagem ou FCS para Frame Check Sequência no caso de um código de 32 bits). O código CRC contém elementos redundantes no que diz respeito à trama, permitindo detectar os erros, mas também repará-los.

O princípio do CRC consiste em tratar as sequências binárias como polinómios binários, quer dizer polinómios cujos coeficientes correspondem à sequência binária. Assim, a sequência binária 0110101001 pode ser representada sob a forma polinomial seguinte:

$0*X9 + 1*X8 + 1*X7 + 0*X6 + 1*X5 + 0*X4 + 1*X3 + 0*X2 + 0*X1 + 1*X0$
$X8 + X7 + X5 + X3 + X0 = X8 + X7 + X5 + X3 + 1$

Deste modo, o bit de menor importância (o bit mais a direita) representa o grau 0 do polinômio (x0 = 1), o 4º bit partindo da direita representa o grau 3 do polinômio e assim em diante. Uma sequencia de n bits é constituído por um polinômio de grau no máximo n-1. Todas as expressões polinomiais são manipuladas seguidamente com uma aritmética módulo 2. Neste mecanismo de detecção de erro, um polinômio predefinido (chamado polinômio gerador e notado G (X)) é conhecido do emissor e do receptor. A detecção de erro consiste, para o emissor, em efetuar um algoritmo sobre os bits da trama a fim de gerar um CRC, e transmitir estes dois elementos ao receptor. Basta então ao receptor que efetue o mesmo cálculo a fim de verificar que o CRC é válido.

Hamming:

O código de Hamming é usado para inserir informações de correção de erros em fluxo de dados. Os códigos são concebidos de modo que um erro não só é detectado, mas corrigido. Mesmo ele aumentando a quantidade de dados a serem enviados, essa adição de informação de correção de erros aumenta a confiabilidade das comunicações sobre os meios com altas taxas de erro.

Passos para a montagem do algoritmo:

Exemplo: 11010010

1- Crie a palavra de dados. Qualquer bit com uma posição que for uma potência de dois (primeiro, segundo, quarto, etc.) deve ser reservado para informações de paridade. Use o tamanho que for necessário para que a palavra tenha os dados originais e os bits de paridade. Transformando o exemplo, teríamos: __1_101_0010. Note que os bits originais permanecem na mesma ordem, mas foram espalhados para inserirmos os bits de paridade.
2- Calcule em seguida os bits de paridade. Começando a partir do primeiro bit, lê-se um bit e pula-se outro, ou seja, a leitura se dá de um bit sim e um não até o final da sequencia. Enquanto isso, conta-se o numero de 1s encontrados. Os bits de paridade não contém nesse processo. Se o número de uns for par, defina o primeiro bit como zero. Caso contrário, defina-o como um. Exemplo: Bits 1, 3, 5, 7, 9 e 11 de __1_101_0010, _11101, contêm quatro uns. Este é par, então, o primeiro bit é definido como zero: 0 _ 1 _ 1 0 1 _ 0 0 1 0.

   Calcule os bits de paridade restantes. Começando com o segundo bit, lê-se dois bits e, em seguida, pula-se dois bits e repete-se o procedimento até o final. O quarto bit lê quatro bits, pula outros quatro, começando pelo bit quatro. O mesmo padrão é seguido por todos os bits de paridade, até todos serem computados. Exemplo: Bit 2: 0 _ 1 _ 1 0 1 _ 0 0 1 0 verifica _1, 01, 01, que contêm três uns, então o bit 2 é definido como um. Bit 4: 011_101_0010 verifica _101, 1, que contêm três uns, então o bit 4 é definido como um. Bit 8: 0111101_0010 verifica _0010, que contêm só um, então o bit 8 é definido como um. A palavra é, portanto, codificada como 011110110010. Confirme a palavra. Se

uma palavra estiver corrompida, os bits de paridade não vão coincidir com o que é esperado. Para confirmar que a palavra não esteja corrompida, basta calcular os bits de paridade usando as etapas dois e três. Se os bits não forem iguais, grave suas posições. Corrija o bit errado. Se você encontrar bits de paridade incorretos, simplesmente some as posições dos bits. O valor da soma é a posição do bit incorreto. Troque o valor do bit nesta posição. Por exemplo, se os bits de paridade incorretos forem o um e o quatro, trocar o valor do quinto bit corrigirá o erro.

MD5:

A entrada do MD5 é um fluxo de dados (mensagem) que pode ter um número arbitrário de bits, representado por b, um número inteiro positivo que varia de zero até o infinito. Para obter o digesto da mensagem, seus bits, representados por m0, m1, ..., m{b-1}, onde b = número de bits da mensagem, são submetidos a diversas operações. Este processo é dividido em cinco etapas ou passos.

Passo 1: Preparação do fluxo de dados

Adiciona-se à mensagem os bits necessários para que seu tamanho mais 64 bits seja divisível por 512.

Passo 2: Inclusão do comprimento

Depois da adição de bits, uma representação binária do tamanho original da mensagem e que ocupa 64 bits, é adicionada à mesma. O conjunto obtido é processado em blocos de 512 bits na estrutura iterativa de Damgård/Merkle, sendo que cada bloco é processado em quatro rodadas distintas.

Passo 3: Inicialização do buffer MD

Um buffer de quatro words é usado para calcular o digesto da mensagem. Os registradores de 32 bits A, B, C e D são inicializados com os seguintes valores hexadecimais:

word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10

Passo 4: Processamento da mensagem em blocos de 16 words (512 bits)

Primeiro definse-se quatro funções auxiliares. Cada uma delas usa três words de 32 bits para produzir uma saída de um word de 32 bits.

F(X,Y,Z) = (X and Y) or (not(X) and Z)
G(X,Y,Z) = (X and Z) or (Y and not(Z))
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X or not(Z))

A função F atua como condicional sobre cada um dos bits: se X então Y senão Z. É importante frisar que, se os bits de X, Y e Z são independentes e não induzidos (unbiased) então cada bit de F(X,Y,Z) também será independente e não induzido.

As funções G, H e I são semelhantes à função F quanto à ação "paralela bit a bit" produzindo saídas de bits independentes e não induzidos se os mesmos tiverem estas características. A função H é apenas um "XOR" ou função de "paridade" das suas entradas.

As etapas deste passo usam uma tabela de 64 elementos, T[1] a T[64], construída à partir da função seno. T[i] for o nésimo elemento da tabela e é igual à parte inteira de abs(seno(i)) multiplicada por 4294967296, onde i é expresso em radianos.

Antes de iniciar o processamento, deve-se armazenar os valores de A, B, C e D. Neste texto, as variáveis de trabalho serão expressas em letras minúsculas, portanto armazenamos a = A, b = B, c = C e d = D.

Divide-se cada bloco de 512 bits em 16 sub-blocos de 32 bits, aqui identificados por X[0] a X[15]. Genericamente, os sub-blocos são designados por X[k]. A seguir, aplica-se as funções F, G, H e I em quatro rodadas:

/* Rodada 1
/* Seja [abcd k s i] a operação a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)
/* Faça as seguintes 16 operações.
[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]

[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
/* Rodada 2
/* Seja [abcd k s i] a operação a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)
/* Faça as seguintes 16 operações.
[ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
/* Rodada 3
/* Seja [abcd k s i] a operação a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)
/* Faça as seguintes 16 operações
[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
/* Rodada 4
/* Seja [abcd k s i] a operação a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)
/* Faça as seguintes 16 operações
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
/* Finalmente, faça as adições dos resultados obtidos para a, b, c, d
/* com os valores iniciais de A, B, C e D
A = a + A
B = b + B
C = c + C
D = d + D

Passo 5: A saída

O digesto da mensagem produzido na saída é a concatenação de A, B, C e D. Começa-se com o byte menos significativo de A e termina-se com o byte mais significativo de D.

SHA1:

A família de SHA (Secure Hash Algorithm) está relacionada com as funções criptográficas. A função mais usada nesta família, a SHA-1, é usada numa grande variedade de aplicações e protocolos de segurança, incluindo TLS, SSL, PGP, SSH, S/MIME e IPSec. SHA-1 foi considerado o sucessor do MD5. Ambos têm vulnerabilidades comprovadas1 . Em algumas correntes, é sugerido que o SHA-256 ou superior seja usado para tecnologia crítica.

Os algoritmos SHA foram projetados pela National Security Agency (NSA) e publicados como um padrão do governo Norte-Americano.

O primeiro membro da família, publicado em 1993, foi oficialmente chamado SHA; no entanto, é frequentemente chamado SHA-0 para evitar confusões com os seus sucessores. Dois anos mais tarde, SHA-1, o primeiro sucessor do SHA, foi publicado. Desde então quatro variantes foram lançadas com capacidades de saída aumentadas e um design ligeiramente diferente: SHA-224, SHA-256, SHA-384, e SHA-512 — por vezes chamadas de SHA-2.

Foram feitos ataques a ambos SHA-0 e SHA-12 . Ainda não foram reportados ataques às variantes SHA-2, mas como elas são semelhantes ao SHA-1, pesquisadores estão preocupados, e estão a desenvolver candidatos para um novo e melhor padrão de hashing.

Análises:

Os algoritmos de detecção de erros MD5 e SHA1 não permitem que a mensagem original seja recuperada pela natureza da função hash utilizada na implementação das mesmas. O algoritmo de detecção de erro Hamming não corrige mais de 1 bit em sua execução. O CRC-8 também não é capaz de corrigir nenhum erro.

Comparações:

CRC-8:

-Não flipar nenhum bit: Não detectou erro / Não deveria detectar erro (CERTO)

-Flipagem de bits impares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits pares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits aleatórios: Detectou erro / Deveria detectar erro (CERTO)

Hamming:

-Não flipar nenhum bit: Não detectou erro / Não deveria detectar erro (CERTO)

-Flipagem de bits impares: Não detectou erro/ Deveria detectar erro (ERRADO)

-Flipagem de bits pares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits aleatórios: Detectou erro / Deveria detectar erro (CERTO)

MD5:

-Não flipar nenhum bit: Não detectou erro / Não deveria detectar erro (CERTO)

-Flipagem de bits impares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits pares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits aleatórios: Detectou erro / Deveria detectar erro (CERTO)

SHA1:

-Não flipar nenhum bit: Não detectou erro / Não deveria detectar erro (CERTO)

-Flipagem de bits impares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits pares: Detectou erro / Deveria detectar erro (CERTO)

-Flipagem de bits aleatórios: Detectou erro / Deveria detectar erro (CERTO)

Conclusão:

A partir das pesquisas realizadas sobre os algoritmos acima somados aos testes, chegamos a conclusão que o CRC-8, é muito bom para detectar um erro na mensagem, mas não é capaz de detectar aonde ou quantos bits vieram errados, consequentemente, não possui condições para recuperar a mensagem enviada.

Já o MD5 e o SHA-1 são muito parecidos. Ambos também não possuem capacidade de detectar qual, ou quais bits possuem erro. Logo, devido a sua própria natureza, não é possível realizar a recuperação da mensagem original. Segundo nossas pesquisas, constatamos que são os mais usados no mercado hoje em dia para algoritmos maiores. O SHA-1 já possuem novas versões mais novas, mas também é muito usado. O MD5, mesmo sendo mais antigo, ainda hoje possui muitos sistemas que o usam. Sendo assim, concluímos que tanto o MD5 quanto o SHA-1 são melhores para dados maiores.

O Hamming possui a vantagem de quando há um erro de apenas um bit, além de detectar que houve um erro, ele retorna a posição exata em que o bit com erro foi encontrado, entretanto quando há mais de um erro, ele só consegue reportar o erro, já a posição onde ocorreu o erro não é retornado corretamente. Detectamos, também, que ele possui uma falha em que quando flipamos os bits impares, ele não detectou o erro e isso é um grave problema para o algoritmo que trabalha com confiabilidade.

O CRC-8 é bom para algoritmos menores. O hamming possui a vantagem de quando detecta um erro e é apenas de um bit, ele não só retorna a posição do erro encontrado como também é capaz de corrigir o erro, mas a desvantagem de não detectar o caso de erro em bits impares é algo preocupante.