UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIAS E GEOCIÊNCIAS
DEPARTAMENTO DE ELETRÔNICA E SISTEMAS

Pós-Graduação em Engenharia Elétrica

Análise de Códigos Detectores de Erros Utilizados na Camada de Transporte

por

Ricardo da Silva Barboza

Dissertação de Mestrado

Recife – Agosto de 2003

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIAS E GEOCIÊNCIAS
DEPARTAMENTO DE ELETRÔNICA E SISTEMAS

RICARDO DA SILVA BARBOZA

Análise de Códigos Detectores de Erros Utilizados na Camada de Transporte

Este trabalho foi apresentado à Pós-Graduação em Engenharia Elétrica do Centro de Tecnologias e Geociências da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

ORIENTADOR: Prof. Dr. Rafael Dueire Lins

Recife – Agosto de 2003

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DE MESTRADO DE

RICARDO DA SILVA BARBOZA

TÍTULO

"ANÁLISE DE CÓDIGOS DETECTORES DE ERROS UTILIZADOS NA CAMADA DE TRANSPORTE"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, DES/UFPE, RICARDO MENEZES CAMPELLO DE SOUZA, DES/UFPE e RICARDO MASSA FERREIRA LIMA, DI/UFPE, sob a presidência do primeiro, consideram o candidato **RICARDO DA SILVA BARBOZA APROVADO**.

Recife, 06 de agosto de 2003.

RAFAEL DUEIRE LINS

RICARDO MENEZES CAMPELLO DE SOUZA

RICARDO MASSA FERREIRA LIMA

À minha esposa Andréia e aos meus filhos Brenda, Rebeca e Rafael.

AGRADECIMENTOS

Agradeço em primeiro lugar a Deus, pois sem Ele não seria possível a realização desse trabalho.

Aos meus pais, Selma da Silva Barboza e João Raimundo de Freitas Barboza, pelo apoio dado em toda minha vida e seus conselhos.

Aos professores do curso que muito contribuíram solucionando minhas dúvidas.

Ao professor Rafael Dueire Lins pelo seu esforço em viabilizar nosso curso, orientação e pelo apoio dispensado em minha estada em Recife.

Aos amigos, Jucimar Maia da Silva Jr. e Raimundo Corrêa de Oliveira pelo apoio dado em todo o curso de mestrado e de suas contribuições para este trabalho.

RESUMO

Esta dissertação analisa as taxas de falhas na detecção de erros em alguns dos mais utilizados códigos detectores de erros na camada de transporte. Um simulador que gera erros em dados uniformemente e não uniformemente distribuídos foi desenvolvido. Nossos resultados mostraram altas taxas de falhas nos chamados *checksums* quando utilizamos parâmetros que geram padrões de erros que alteram poucos bits nas palavras código. Contra o que tem sido relado em artigos recentes, o *Internet Checksum* exibiu melhor desempenho quando dados não uniformemente distribuídos foram utilizados.

ABSTRACT

This thesis analyses the failure rates in the detection of errors in some of the most frequently used error detecting codes in the transport layer. An simulator that generates errors in uniformly and non-uniformly distributed data was developed. The results obtained show high failure rates in checksum codes when the errors modify few bits in the code words. Contrarywise to what has been reported in recent papers, the Internet Checksum exhibited better performance when non-uniformly distributed data was used.

ÍNDICE

LISTA DE FIGURAS

1. INTRODUÇÃO

Este capítulo descreve a motivação e os objetivos desta Dissertação. Apresenta-se, também, a idéia principal dos demais capítulos que constituem este trabalho.

1.1. MOTIVAÇÃO

Desde a padronização do *Internet Protocol* (IP) [RFC791], dois protocolos de transporte estão disponíveis para aplicativos da camada de aplicação: o *Transmission Control Protocol* (TCP) [RFC793] e o *User Datagram Protocol* (UDP) [RFC768]. Devido a necessidades de novos serviços requeridos por aplicações recentes [RFC2960], atualmente estão sendo desenvolvidos outros protocolos para a camada de transporte pelo *Internet Engineering Task* Force (IETF), como o *Stream Control Transport Protocol* (SCTP) [RFC2960], o *Datagram Congestion Control Protocol* (DCCP) [KHL02] e o *UDP Lite Protocol* [LRZ02].

Sendo um dos objetivos da camada de transporte o fornecimento de comunicação fim-a-fim de forma confiável, geralmente são empregados códigos de

detecção de erros para verificar se a informação não foi alterada pelo canal de comunicação.

No caso do SCTP, o código detector de erro escolhido foi o Adler32 [RFC2960], e devido a um trabalho de J. Stone [STN02], foi verificado que para pacotes de dados de comprimento pequeno o Adler32 apresenta uma detecção de erros inferior ao esperado.

Segundo Stone, o problema encontra-se em um acumulador de 16 bits, chamado *S1*, que é a soma dos bytes do bloco de dados de entrada módulo 65521 (o maior primo menor que 2^{16}). Como *S1* acumula valores utilizando oito bits por vez, o bloco de entrada deve ter no mínimo 257 bytes para que todos os bits deste acumulador sejam afetados. Ou seja, para blocos de entrada com menos de 257 bytes, este acumulador terá, pelo Teorema do Limite Central, uma distribuição normal que resultará em uma distribuição desigual de valores de *S1*. Este quadro se agrava com o fato de que os dados que geralmente são transportados não possuem uma distribuição uniforme, resultando em uma probabilidade maior de um erro deixar um pacote danificado com um valor válido do que se todos os valores fossem uniformemente iguais. Stone [STN98] advoga que este problema atinge os códigos que geram seus bits de paridade através de alguma forma de soma dos dados de entrada (estes códigos são geralmente chamados de *checksums*).

Devido a esta fraqueza inerente, exarcebada pelo fato que o SCTP foi projetado para inicialmente transportar mensagens de sinalização (SS7) da rede pública de telefonia comutada (PSTN), que possuem menos que 128 bytes, o grupo que desenvolve este protocolo está alterando o método utilizado para detecção de erros. O que sem dúvida, trará prejuízos a quem já o está utilizando e principalmente onde o mesmo foi implementado em hardware.

Os grupos do IETF que estão desenvolvendo os protocolos DCCP e UDP-Lite, também, estão escolhendo os códigos detectores de erros que irão utilizar, e um trabalho de análise comparativa de vários códigos seria de grande valia para evitar problemas vindouros.

1.2. OBJETIVOS

A finalidade dos códigos detectores de erros é diminuir a taxa de erro das mensagens entregues a um receptor.

Como nem todos os erros podem ser detectados, sempre existe uma taxa de erro residual [HLZ91]. Desta forma, este trabalho tem como objetivo, analisar vários códigos detectores de erros levando em conta suas característica de detecção de erros em canais pouco ruidosos e que possuem erros em surtos, e encontrar suas taxas de falha.

Os resultados desta análise podem ser de grande ajuda para a escolha de um entre os seguintes códigos detectores de erros empregados atualmente em protocolos da camada de transporte:

- Adler – Criado por Mark Adler para a biblioteca de compressão ZLIB [RFC1950] e utilizado na versão inicial do Protocolo SCTP;
- Fletcher – Proposto por John Fletcher [FLT82] e utilizado na Camada de Transporte do Modelo OSI (*Open Systems Interconnection*) [SKL89], no trabalho serão analisados quatro versões, duas destas geram 16 bits de paridade e as outras duas 32 bits de paridade;
- *Internet Checksum* – Utilizado pelo conjunto de protocolos TCP/IP, em sua versão de 16 bits. Será analisada também uma versão pouco estudada que gera 32 bits de paridade;
- *Cyclic Redundancy Checks* (CRC) – Os códigos de redundância cíclica estão entre os mais utilizados em comunicação digital [WCK95]. Analisaremos os que possuem polinômio gerador do CCITT (16 bits), o utilizado em redes locais Ethernet (CRC32B) e o encontrado por Castagnoli [CST93] (CRC32C), este último substituirá o Adler32 no SCTP.

1.3. METODOLOGIA

Para a análise de desempenho, será utilizado um software que irá gerar pacotes de tamanhos definidos, a partir de dados reais e randômicos. Estes pacotes serão distorcidos pela adição de erros em surto e alteração de bits. Através do número de vezes que os códigos falharem em detectar os erros iremos calcular a Probabilidade P_{ue} de não detecção de erros (***Probability of undetected error***) dos códigos em diversas situações.

1.4. ORGANIZAÇÃO

Esta Dissertação é composta, além deste capítulo introdutório, de quatro outros capítulos, referências bibliográficas e apêndices conforme detalhamento feito a seguir:

No capítulo 2 – Apresentação da teoria de códigos com uma descrição dos códigos detectores de erros utilizados na camada de transporte atualmente.

No capítulo 3 – Descrição dos métodos utilizados para a escolha dos parâmetros utilizados na simulação para avaliação dos códigos detectores erros e validação dos resultados iniciais.

No capítulo 4 – Apresentação dos resultados obtidos a partir dos dados gerados pelas simulações realizadas e discussão dos mesmos.

Finalizando, no capítulo 5 – Apresentação de conclusões fundamentadas nos resultados e propostas de novas pesquisas.

Apêndice A – Descrição das estruturas algébricas necessárias para a construção dos códigos detectores de erros analisados nesta dissertação.

Apêndice B – Descrição do Software utilizado na simulação.

Apêndice C – Email do Dr. Dafna Sheinwald do IBM Haifa Research Lab para o grupo que desenvolve o protocolo iSCSI por Julian Satran da IBM, onde é citado algumas propriedades dos códigos estudados e o modelo de erro em surto implementado neste trabalho.

Apêndice D – Email ao grupo que desenvolve o SCTP informando a descoberta da fraqueza do Adler32 em mensagens de comprimento pequeno.

Apêndice E – Comparação do tempo de codificação das palavras código dos códigos estudados.

2. CÓDIGOS DETECTORES DE ERROS

Neste capítulo será apresentada uma visão geral da teoria de códigos de controle de erros em conjunto com uma descrição dos códigos detectores de erros em estudo.

2.1. CÓDIGOS DE BLOCO

Códigos de controle de erro introduzem quantidades de redundância controlada em um fluxo de dados transmitido, fornecendo ao receptor um meio para detectar e possivelmente corrigir erros causados por ruído no canal de comunicação.

Um código de bloco para controle de erro **C** consiste de um conjunto de palavras código $\{c_0, c_1, c_2, ..., c_{M-1}\}$. Cada palavra código é da forma $c = (c_0, c_1, ..., c_{n-1})$; se as coordenadas individuais recebem seus valores de um Corpo Finito GF(q) (Apêndice A), então o código é dito q-ário. O processo de codificação consiste em segmentar o fluxo de dados em blocos, e mapear estes blocos em palavras código de **C** (Figura 1). Este mapeamento é geralmente de um para um, assegurando que o processo de codificação possa ser revertido no receptor e os dados originais possam ser recuperados.

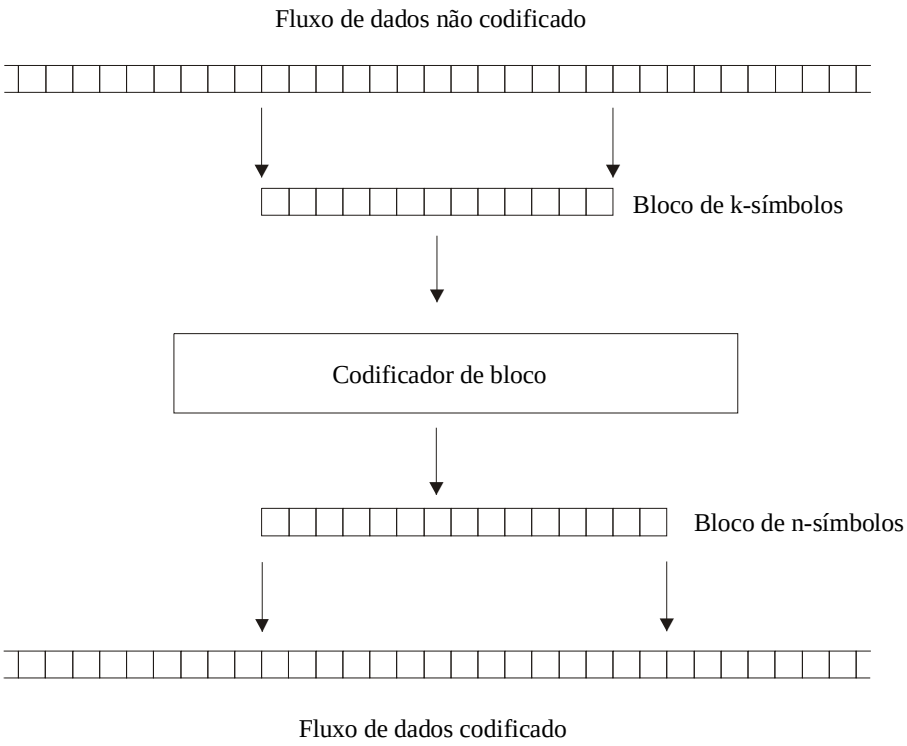


Figura 1 – Codificação em bloco [WCK95].

Se os símbolos no fluxo de dados podem assumir qualquer valor em GF(q), então a coleção de todas as k-tuplas $\mathbf{m} = (m_0, m_1, ..., m_{k-1})$ forma um espaço vetorial sobre GF(q). Existem q^k possíveis vetores de dados com k-símbolos. Se $M = q^k$, o codificador segmenta o fluxo de dados em blocos de k-símbolos, senão o codificador deve trabalhar com mensagens de tamanho variável.

A coleção de todas as possíveis n-tuplas sobre GF(q) forma um espaço vetorial sobre GF(q) contendo q^n vetores. Existem então $(q^n - M)$ padrões de n-símbolos que não estão associados com blocos de dados e, portanto não são palavras código válidas. O código **C** contém redundância, se o número de palavras código é menor que o número de possíveis blocos de n-símbolos. A redundância r é expressa na forma logarítmica.

$$r = n - \log_q M$$

Se $M = q^k$, então r é simplesmente a diferença $r = (n - k)$ entre o comprimento das palavras código e dos blocos de dados. A redundância é também freqüentemente expressa em termos da taxa do código.

A taxa do código é definida da seguinte forma. Seja M o número de palavras código, cada qual de comprimento n, em um código **C**. A taxa de **C** é

$$R = \frac{\log_q M}{n}$$

A razão de código é simplesmente $R = k/n$ para os casos em que $M = q^k$.

A alteração de uma palavra código por ruído de canal é freqüentemente modelada como um processo aditivo, como mostrado na Figura 2. Este modelo é particularmente útil com canais binários, onde a adição do padrão de erro é realizada módulo 2. Neste trabalho o erro é introduzido nas palavras código desta forma.

A determinação se erros estão presentes em uma palavra recebida é chamada detecção de erros.

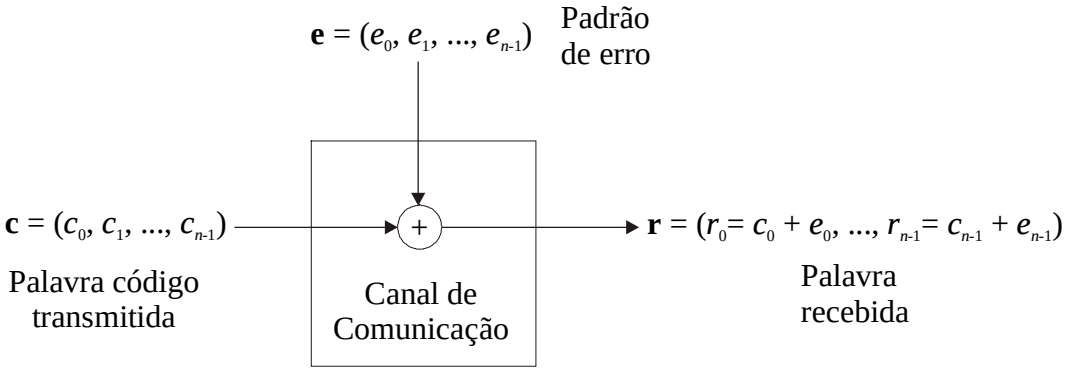


Figura 2 – Um modelo de canal ruidoso aditivo [WCK95].

Um padrão de erro não é detectável se e somente se ele faz com que a palavra recebida seja uma outra palavra código diferente da transmitida. Dado uma

palavra código transmitida **c**, existem (M – 1) palavras código diferentes de **c** que podem chegar no receptor, por isso há (M – 1) padrões de erros indetectáveis.

O decodificador pode reagir a um erro detectado com uma das seguintes respostas:

- Requisitar a retransmissão da palavra.
- Marcar a palavra como incorreta e enviar adiante para a próxima camada.
- Tentar corrigir os erros na palavra recebida.

Requisições de retransmissões são usadas em um conjunto de estratégias de controle de erro chamados protocolos de requisição com repetição automática (ARQ). Alguns exemplos são os protocolos da camada de transporte do TCP/IP como o TCP e o SCTP.

A segunda opção é típica de aplicações em que um menor atraso é o fator de maior importância.

A opção final é chamada de FEC (*forward error correction*). Em sistemas FEC a estrutura algébrica do código é utilizada para determinar qual das palavras código é a mais provável de ter sido transmitida, dada uma palavra recebida com erro.

No estudo de códigos de blocos algumas definições são importantes:

- O **peso de Hamming de uma palavra** é o número de coordenadas diferentes de zero na palavra código. O peso de uma palavra código **c** é geralmente escrito como $w(\mathbf{c})$.
- A **distância de Hamming** entre duas palavras **v** e **w** é o número de coordenadas em que as mesmas diferem, ou seja,

$$d_{Ham\min g}(v,w) = d(v,w) = |\{i \mid v_i \neq w_i, i = 0,1,...,n-1\}|$$

- A **distância mínima** de um código **C** é a menor distância de Hamming entre todos os pares distintos de palavras código em **C**.

Os únicos padrões de erros indetectáveis são aqueles que fazem com que uma palavra código se torne outra palavra código. Seja d_{min} a distância mínima de um código em uso, garante-se então que uma palavra código transmitida seja diferente ao menos em d_{min} coordenadas de qualquer outra palavra código. Para que um padrão de erro seja indetectável, ele deve alterar os valores da palavra código no mínimo em d_{min} coordenadas. Então um código com distância mínima d_{min} pode detectar todos os padrões de erros de peso menor ou igual a $(d_{min} - 1)$.

2.2. CÓDIGOS DE BLOCO LINEARES

Códigos de bloco lineares são os mais facilmente implementáveis e, portanto, os tipos mais usados de códigos de bloco. Por definição eles formam um subespaço vetorial sobre um corpo finito [WCK95].

Lin [LIN83] cita que uma propriedade desejável de um código de bloco linear é que suas palavras código possam ser representadas de forma sistemática. Desta forma a palavra é divida em duas partes, uma com a mensagem consistindo de k dígitos inalterados de informação e outra com $n - k$ dígitos de paridade dependentes da informação. Todos os códigos estudados neste trabalho são sistemáticos. Como o processo de codificação da mensagem é uma operação computacionalmente cara [FEL95], a grande maioria dos códigos detectores de erros utilizados em redes de computadores (principalmente na camada de transporte) são sistemáticos.

Geralmente é utilizada a seguinte notação quando se trabalha com códigos lineares: Um código linear de comprimento n e dimensão k é chamado de código (n, k). Um código (n, k) com símbolos em GF(q) tem um total de q^k palavras código de comprimento n.

Podemos citar como propriedades de códigos lineares [WCK95]:

- A combinação linear de qualquer conjunto de palavras código é uma palavra código. Uma conseqüência disto é que códigos lineares sempre possuem o vetor com todas suas coordenadas iguais a zero.

- A distância mínima de um código linear é igual ao peso da palavra código de menor peso diferente de zero.
- Os padrões de erros não detectáveis para um código linear são independentes da palavra código transmitida e sempre consistem do conjunto de todas as palavras código diferentes de zero.

2.3. CÓDIGOS CÍCLICOS

Os códigos de detecção de erro CRC (*cyclic redundancy check*) estão entre os conjuntos de códigos mais usados em comunicações e armazenamento de dados, devido a sua codificação simples e rápida e suas propriedades de detecção de erro [COS98].

Um código de bloco linear (*n*, *k*) **C** é dito cíclico se para qualquer palavra código $\mathbf{c} = (c_0, c_1, ..., c_{n-2}, c_{n-1}) \in \mathbf{C}$, existe também uma palavra código $\mathbf{c}' = (c_{n-1}, c_0, c_1, ..., c_{n-2}) \in \mathbf{C}$. A palavra código **c**' é uma rotação à direita da palavra código **c**. Como **c** foi escolhida arbitrariamente entre as palavras código de **C**, então para todas as *n* distintas rotações de **c** também devem existir palavras código em **C**.

Os CRCs são códigos cíclicos sistemáticos encurtados [WCK95]. Tais códigos são construídos da seguinte maneira. Seja **C** um código cíclico sistemático (*n*, *k*). Seja **S** o subconjunto de palavras código em **C** tais que as coordenadas de informação de mais alta ordem *j* (as coordenadas *j* mais à direita das palavras código) possuem valor igual a zero. Seja **C**' o conjunto de palavras obtidas pela remoção dessas coordenadas *j* de todas as palavras em **S**. **C**' é um código cíclico sistemático encurtado (*n* - *j*, *k* - *j*).

Como o encurtamento diminui a taxa do código, códigos encurtados possuem capacidades de detecção e correção de erro, no mínimo tão boas quanto os dos códigos de que foram derivados.

Códigos obtidos pelo encurtamento de códigos cíclicos são quase sempre não cíclicos [WCK95], sendo freqüentemente chamados de códigos polinomiais devido ao polinômio gerador que gera tanto o código cíclico original quanto o código encurtado

derivado. Apesar disso, ainda é possível utilizar os mesmos codificadores e decodificadores (circuitos ou algoritmos) utilizados com o código cíclico original.

O processo de codificação ocorre da seguinte forma: Seja $g(x)$ um polinômio gerador de grau r.

$$g(x) = x^r + g_{r-1}x^{r-1} + \mathbf{K} + g_1 x + g_0 .$$

Dado um bloco de mensagem m,

$$m = (m_0, m_1, \mathbf{L}, m_{k-1})$$

e o polinômio associado da mensagem:

$$m(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \mathbf{L} + m_1 x + m_0$$

o codificador calcula o resto $d(x)$ obtido da divisão de $x^r m(x)$ por $g(x)$. Este resto é anexado à mensagem completando a codificação sistemática. A palavra código resultante é da forma $c(x) = x^r m(x) + d(x)$.

O codificador pode também ser usado como decodificador. Por construção, $c(x) = x^r m(x) + d(x)$ deve ser divisível por $g(x)$. Se os dados codificados não forem alterados durante a transmissão, a decodificação da versão recebida de $c(x)$ irá resultar em um resto igual a zero. A detecção de resto diferente de zero após o processamento dos dados recebidos indicará a presença de erros durante a transmissão.

Como exemplo prático poderíamos tomar $g(x) = x^5 + x^2 + 1$ (polinômio gerador primitivo em GF(2^5)) e a mensagem a ser transmitida igual a "11011100", desta forma o polinômio associado à mensagem é $m(x) = x^7 + x^6 + x^4 + x^3 + x^2$. O primeiro passo seria multiplicar o polinômio associado à mensagem por um fator igual ao mais alto grau do polinômio gerador, neste caso x^5, para dar espaço aos bits de redundância. Isto simplesmente significa deslocar os bits na palavra código 5 vezes a esquerda. Então, divide-se o polinômio associado à mensagem pelo polinômio gerador e o resto é anexado à mensagem.

1) $(x^7 + x^6 + x^4 + x^3 + x^2) * x^5 = x^{12} + x^{11} + x^9 + x^8 + x^7$

2) $x^{12} + x^{11} + x^9 + x^8 + x^7$ | $x^5 + x^2 + 1$

x^{12} x^9 x^7 | $x^7 + x^6 + x$

x^{11} x^8

x^{11} x^8 x^6

x^6

x^6 x^3 x

x^3 x ⟶ Resto

$d(x)$ é igual a $x^3 + x$ e a palavra código transmitida é “1101110001010”.

É interessante notar que se as primeiras j coordenadas de símbolos da mensagem forem iguais a zero, sua entrada no codificador não altera o estado do CRC. A remoção dos j símbolos de mais alta ordem do bloco de mensagem então não causa nenhum impacto no processo de codificação exceto pela remoção dos j zeros nas correspondentes posições da mensagem na palavra código. Como a inserção de bits iguais a zero no início de uma mensagem não seria detectada por um CRC desta maneira, geralmente o codificador inicializa o CRC com um valor diferente de zero.

Além disto os códigos CRC são geralmente alterados na prática para evitar a possibilidade de que os bits de redundância sejam confundidos com campos de sinalização de protocolos (delimitatores de quadros, por exemplo) ou para manter uma densidade de transição mínima (uma importante consideração na gravação em meio magnético). A modificação do código tipicamente envolve a complementação de alguns ou todos os bits de redundância. Apesar do código não ser mais linear desta forma, o desempenho de detecção de erros não é afetado [COS98]. Esta é a forma que geralmente o CRC é codificado na camada de enlace em redes de computadores.

Quando utilizamos códigos detectores na camada de transporte em redes de computadores é prática não utilizar hardware para a codificação e decodificação [SLT84]. A camada de transporte é composta de um cabeçalho e nele é reservado um campo para armazenar os bits de redundância. Na codificação, este campo é preenchido com zeros, a mensagem passa pela codificação e os bits de paridade são inseridos neste

campo [TAN02] [RFC2960] [KHL02] [LRZ02]. Mais uma vez o código perde sua linearidade desde modo, mas mantém sua capacidade de detecção.

Para verificarmos as propriedades de detecção dos CRCs iremos começar com as seguintes considerações. Como o CRC é um código linear, todo padrão de erro $e(x)$ deve ser igual a alguma palavra código para que o erro não seja detectado. Se $m(x)$ é o polinômio associado à mensagem e $g(x)$ o polinômio gerador de grau r, o resto $d(x)$ possui grau $r - 1$ e é definido como sendo o resto de:

$$\frac{m(x).x^r}{g(x)}$$

A palavra código a ser transmitida é:

$$c(x) = m(x).x^r - d(x)$$

Um erro de transmissão adiciona um polinômio erro $e(x)$ à palavra código transmitida. Quando o receptor divide a palavra recebida pelo polinômio gerador o termo de erro é encontrado.

$$\frac{c(x)+e(x)}{g(x)} = \frac{c(x)}{g(x)} + \frac{e(x)}{g(x)} = \frac{e(x)}{g(x)}$$

Um erro só não é detectado se o resto da divisão do padrão de erro $e(x)$ pelo polinômio gerador $g(x)$ for zero. Se $e(x)$ não for divisível por $g(x)$ ou ter um grau menor que $g(x)$, a divisão sempre terá resto diferente de zero [HLZ91]. Isto significa que todos os surtos de comprimento r ou menores serão detectados.

Um surto de comprimento maior que r não será detectado ser ele for divisível por $g(x)$. Então a falha na detecção de um erro depende somente do padrão de erro $e(x)$. Assumindo que todos os padrões de erros são equiprováveis temos que com palavras código de comprimento $n + r$, existe um total de 2^{n+r} possíveis padrões de

erros. O número de múltiplos de um polinômio gerador de grau r em uma palavra código de comprimento $n + r$ é igual a 2^n. Isto significa que uma fração

$$\frac{2^n}{2^{n+r}} = \frac{1}{2^r}$$

de todos os erros não são detectados. Ou seja, $P_{ue} = 1/2^r$.

A literatura sobre códigos CRCs descreve várias regras para a escolha de polinômios geradores.

Wicker [WCK95] cita que é prática comum selecionar $g(x) = p(x)(x + 1)$ onde $p(x)$ é primitivo. O fator $(x + 1)$ assegura que todos os erros com peso ímpar são detectáveis. Códigos CRCs gerados por polinômios deste tipo possuem uma distância mínima igual a 4. Então qualquer erro que afete 3 bits é detectado.

Wolf [WLF94] investigou o código CRC16 CCITT que gera 16 bits de paridade. Este código é gerado por um polinômio da forma $g(x) = p(x)\ (x + 1)$ sendo $p(x)$ um polinômio primitivo de grau $r - 1$ (grau 15 neste caso). Utilizaremos os resultados de seu trabalho para validarmos nossa simulação.

Wolf comentou que todos os polinômios primitivos geram códigos com o mesmo desempenho, quando os códigos não estão encurtados. Mas para as versões encurtadas, a escolha de determinado polinômio primitivo influi no código resultante. Através de um hardware especial [WLF94B], Wolf encontrou um polinômio primitivo que (quando multiplicado por $(x + 1)$) produz um polinômio gerador que é superior ao do CRC16 CCITT por uma ordem de magnitude. Para CRCs que geram 32 bits de paridade, ele citou dois polinômios que diferem em suas probabilidades de não detecção de erro com surtos de 33 bits, em 4 ordens de magnitude.

Baicheva *et al.* [BAI00] fez uma comparação de diferentes polinômios geradores de grau 16 da forma $g(x) = p(x)\ (x + 1)$, e de outras formas. Ele calculou suas distâncias mínimas para comprimentos de palavras código de até 1024 bits.

Fujiwara *et al.* [FUJ89] determinou os pesos de todas as palavras do CRC utilizado no padrão de rede IEEE 802.3 (CRC32B). Este código é gerado por um polinômio primitivo de grau 32, sendo assim sua distância mínima é igual a 3. Esta distância permanece a mesma para palavras código de até $2^{32} - 1$ bits. Entretanto, o

formato de quadro para o MAC (*Media Access Control*) da camada de enlace no IEEE 802.3 proíbe comprimentos de quadros maiores que 12.144 bits. Então, Fujiwara somente investigou palavras código até esse comprimento. A distância mínima igual a 4 foi encontrada para palavras código entre 4.096 e 12.144 bits; para palavras código de 512 à 2.048, d_{min} foi igual a 5; o valor de d_{min} chegou até 15 quando as palavras código possuíam de 33 até 42 bits.

Castagnoli *el al.* [CST93] melhorou a técnica de Fujiwara e explorou um número maior de códigos CRCs com 24 e 32 bits de redundância. Foram estudados vários códigos construídos pela multiplicação de polinômios irredutíveis de baixo grau. Na classe comum (polinômio irredutível de grau 31) * (x + 1) foram estudados 47.000 polinômios (nem todos os possíveis). Os melhores em termos de distância mínima tinham d = 6 até 5.275 bits e d = 4 até $2^{31} - 1$ bits. E entre estes *melhores* está o que Castagnoli denominou CRC32/4 ($p(x)$ = $11EDC6F31_H$), e neste trabalho chamaremos de CRC32C, este código irá substituir o código Adler32 no protocolo SCTP.

Para comparação o CRC32B (IEEE 802.3) possui d = 4 até no mínimo 64.000 bits (Fujiwara parou em 12.144 bits) e d = 3 até $2^{32} - 1$ bits.

O trabalho de Castagnoli foi realizado com o auxílio de um processador de propósito especial.

Tabela 1 – Polinômios Geradores utilizados neste trabalho

CRC	Polinômio Gerador
CRC16 CCITT [WCK95]	$g(x) = x^{16} + x^{12} + x^5 + 1 = (x^{15} + x^{14} + x^{13} + x^{12} + x^4 + x^3 + x^2 + x + 1)(x + 1)$
CRC32B (IEEE 802.3) [WCK95]	$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
CRC32C (CRC32/4) [CST93]	$g(x) = x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1$

2.4. INTERNET CHECKSUM

A tecnologia chave que dá suporte a Internet (e várias internets, intranets e variantes) é o conjunto de protocolos que recebem o nome de TCP/IP. Este nome é devido a dois protocolos constituintes do conjunto, o Transmission Control Protocol (TCP) e o Internet Protocol (IP), embora este conjunto contenha realmente vários outros protocolos. O TCP/IP permite a interligação de redes de computadores que são baseadas em tecnologias diferentes. Por exemplo, TCP/IP permite conexão entre redes ATM e sistemas baseados em Ethernet (IEEE 802.3).

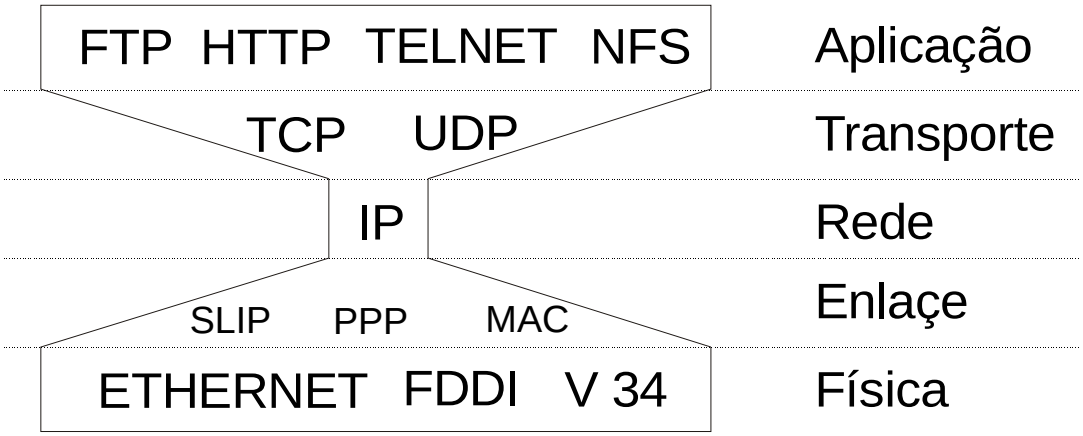


Figura 3 – Estrutura em camadas de algumas tecnologias de rede e protocolos TCP/IP.

A arquitetura TCP/IP pode ser dividida em quatro camadas básicas (Figura 3):

- Camada de Aplicação que consiste de processos do usuário (TELNET, FTP, SMTP, etc.).
- Camada de Transporte que provê transferência de dados fim-a-fim.
- Camada de Inter-rede que fornece uma imagem de uma única rede virtual para as camadas superiores pelo roteamento de datagramas entre as redes interconectadas.
- Camada de interface da rede (também chamada de camada de enlace) que possui uma interface real com o hardware das redes (como X.25, ATM e FDDI).

O Internet Protocol (IP) é o principal componente da Camada Inter-rede. Ele é um protocolo não orientado à conexão que não supõe ou provê qualquer medida para uma conexão confiável. O único controle de erro incorporado ao datagrama IP é um ***checksum*** que gera 16 bits de paridade e se aplica somente ao cabeçalho. O ***checksum*** é obtido pelo cálculo do complemento da soma em complemento de um de todas as palavras de 16 bits do cabeçalho (o campo que contém o ***checksum*** é mantido zerado durante este cálculo) [RFC1071]. Se um datagrama IP possui um ***checksum*** incorreto, ele é descartado. Não há provisão para uma requisição de retransmissão, em parte por que não se saberia de quem pedir a requisição de transmissão. Deve-se notar que o IP não faz qualquer verificação do campo de dados dentro do datagrama. Este serviço deve ser providenciado pela camada de transporte.

Existem dois protocolos básicos na camada de transporte: o Transmission Control Protocol (TCP – orientado à conexão) e o User Datagram Protocol (UDP – não orientado à conexão), ambos protocolos são encapsulados por datagramas IP. O UDP possui um ***checksum*** opcional de 16 bits de paridade que, quando usado, é calculado de uma maneira similar à do ***checksum*** do datagrama IP. O ***checksum*** UDP é o complemento da soma em complemento de um de todas as palavras de 16 bits de um pseudo do cabeçalho do cabeçalho IP, cabeçalho UDP, e o dados UDP. Como no IP, o UDP não fornece o reconhecimento dos pacotes. O UDP simplesmente repassa os pacotes recebidos para a aplicação. A vídeo conferência é uma aplicação típica que usa UDP para enfatizar a latência sobre a confiabilidade.

O TCP, por outro lado, faz uso completo de seu ***checksum*** de 16 bits. Ele designa um número de seqüência para cada pacote transmitido, e espera um reconhecimento positivo da camada de transporte da entidade receptora. Se um reconhecimento não é recebido em um certo tempo, o pacote é retransmitido.

Segundo Peterson [PET00], o *Internet Checksum* foi utilizado nos protocolos TCP/IP devido à experiência na ARPANET, que sugeria que um código desta forma era adequado.

Um exemplo do cálculo do *Internet Checksum* (ou *TCP Checksum*) seria a codificação da seguinte mensagem (em hexadecimal):

$$48656c6c6f20776f726c642e_H$$

O primeiro passo é separar a mensagem em palavras de 16 bits e somá-las:

```
 4865
 6c6c
 6f20
 776f
 726c
+642e
 ----
271fa
```

Como a soma é em complemento de um, o *overflow* é somado com os dígitos menos significantes.

```
71fa
+  2
----
71fc = 0111000111111100B
```

O resultado é então complementado e anexado à mensagem.

$$\sim 71fc = 0000111000000011_B$$

Quando a palavra código é recebida a soma de todas as palavras de 16 bits, incluindo os bits de paridade, deve ser igual a 1111111111111111_B. Se outro valor for encontrado, o pacote é descartado.

As propriedades do *Internet Checksum* foram originalmente discutidas por Bill Plummer [IEN45]. Plummer comentou que a operação utilizada no cálculo, ou seja, a adição em complemento de um (+) possui as seguintes propriedades:

- + é comutativa. Então, a ordem em que as palavras são somadas não é importante.
- + possui um elemento identidade. Sendo que dois elementos (0000_H –0 e $ffff_H$ +0) podem ser somados a um valor sem alterá-lo. Isto permite que a codificação da mensagem possa ser feita considerando o campo com os bits de paridade iguais a zero.
- + possui inverso. Que é utilizado pelo receptor para a verificação do pacote.
- + é associativa, permitindo que os bits de paridade estejam em qualquer lugar do pacote e mesmo assim se utilize uma leitura seqüencial para decodificá-lo.

Outras propriedades do *Internet Checksum*, e desejáveis em qualquer código utilizado em redes de computadores, citadas por Plummer são:

- É rápido de ser calculado pelos computadores atuais, e em nossa simulação ele é o código mais rápido de ser codificado e decodificado dos códigos estudados (Apêndice E).
- Não altera o conteúdo da informação, ele é sistemático.
- Pode ser modificado incrementalmente, ou seja, se um byte é alterado não é necessário recalcular o *checksum* completamente.

O *Internet checksum* possui as seguintes propriedades de detecção de erros [IEN45]:

- Todos os erros de 1 bit são detectados (possui distância mínima igual a 2).
- Todos os surtos de 15 bits ou de menor comprimento são detectados.
- Todos os surtos de 16 bits são detectados, exceto o que substitui a palavra 0000_H pela $ffff_H$ e vice versa.
- Considerando que os restantes dos surtos são equiprováveis, o código apresenta uma taxa de falha de 1 em 2^{16} surtos ($P_{ue} = 1/2^{16}$).

Stone [STN98] cita que esta taxa só possui este valor quando as mensagens são compostas de dados uniformemente distribuídos, em seu trabalho com dados não uniformemente distribuídos a P_{ue} do *Internet Checksum* foi da ordem de $1/2^{10}$.

Chegamos a resultados iguais a estes somente sob certas condições dos modelos de erros aplicados, e não pela utilização de dados não uniformemente distribuídos.

Neste trabalho será avaliado o *Internet Checksum*, que será chamado TCP16, e uma versão deste código que gera 32 bits de paridade, que chamaremos de TCP32.

2.5. FLETCHER

No final dos anos 70, uma técnica de detecção de erro foi desenvolvida com o objetivo de ser implementável eficientemente em computadores e com propriedades de detecção mais próximas às do CRC (em relação às do TCP16). Este método, chamando de *checksum* Fletcher, foi desenvolvido por John G. Fletcher [KOD92]. O algoritmo e uma análise de suas características foram publicados na revista *IEEE Transactions on Communications* em janeiro de 1982 [FLT82]. Uma versão do *checksum* Fletcher (Fletcher16 255, que utiliza soma em complemento de um) deste então foi adotada para uso da camada de transporte do protocolo de rede padronizado pela ISO (*International Standards Organization*) [SKL89]. Nesse artigo, Fletcher analisou várias medidas comuns da habilidade de detecção de erro de seu algoritmo e comparou os resultados a um CRC, que obteve melhor desempenho.

Há uma RFC [RFC1145] que propôs incluir o código de Fletcher como alternativa ao TCP16 no Protocolo TCP, o que não aconteceu na prática.

Para a codificação do Fletcher são mantidos dois somatórios [FLT82]. Um somatório *A*, é a soma da mensagem tomada *K* bits por vez. O outro somatório, *B*, é a soma dos *K* bits multiplicados por sua posição do final da mensagem. Esta multiplicação incorpora informação posicional nos bits de redundância e protege a mensagem contra movimento ou transposição de dados. Os dois somatórios são concatenados para gerar

$2K$ bits de redundância. O algoritmo não necessita multiplicar valores se os somatórios forem calculados conforme a Listagem 1.

Cada seqüência de K bits é tratada como um inteiro e as operações são realizadas nestes inteiros *módulo M*. Os valores de M considerados por Fletcher foram $M = 2^K$ (aritmética de complemento de dois) e $M = 2^K - 1$ (aritmética de complemento de um). No primeiro caso, se uma adição ultrapassar os K bits, o *overflow* é descartado. No segundo caso o *overflow* é somado de volta ao resultado como demonstrado no *Internet Checksum*. Estes dois valores de M são implementáveis eficientemente na maioria dos computadores.

Quando Sklower [SKL89] estava otimizando implementações do código Fletcher notou que as versões desse código que geravam 32 bits de redundância eram mais rápidas do que as que geravam 16 bits. Verificamos que isto ocorre não só com o Fletcher, mas com a maioria dos códigos estudados (Apêndice E), desde que os *loops* utilizados nas codificações afetem um número maior de bits de cada vez (o valor ótimo geralmente é o comprimento da palavra do computador utilizado para a codificação).

Pode-se citar como propriedades de detecção de erro destes códigos [FLT82]:

- Distância mínima igual a 2, devido ao somatório A.
- Utilizando $M = 2^K - 1$, o código possui distância mínima igual a 3, quando a palavra código possui até $K(2^K - 1)$ bits, e todos os surtos de comprimento até $K - 1$ bits são detectáveis.
- Utilizando $M = 2^K$, todos os surtos de comprimento até $2K - 1$ bits são detectáveis.
- Se todos os outros erros forem equiprováveis, a P_{ue} é a mesma dos códigos já citados, ou seja, $1/2^r$, onde r é o número de bits redundantes.

Foram implementadas em nossa simulação quatro versões deste código:

- Fletcher16 255 – Onde $K = 8$ e $M = 2^K - 1$
- Fletcher16 256 – Onde $K = 8$ e $M = 2^K$

- Fletcher32 65535 – Onde $K = 16$ e $M = 2^K - 1$
- Fletcher32 65536 – Onde $K = 16$ e $M = 2^K$

Listagem 1 – Pseudo-código para a codificação Fletcher16 255

```
inteiro i, A, B;

A = 0;
B = 0;
para i de 1 até comprimento_da_mensagem faça
    A = ( A + message[i] ) modulo 255;
    B = ( A + B          ) modulo 255;
fim para;
retorne (B << 8) + A) // concatena os valores
```

2.6. ADLER

O código Adler32 foi desenvolvido por Mark Adler para a biblioteca de compressão *zlib* [ZLIB02]. A motivação para o desenvolvimento do Adler32 foi o mesmo de Fletcher, um código que fosse de rápida codificação em software e tivesse propriedades de detecção próximas às do CRC.

Os bits de redundância gerados pela codificação do Adler32 são compostos por duas somas *S1* e *S2* [CAV01]. *S1* possui 16 bits e é a soma da mensagem de 8 em 8 bits módulo 65521 (que é o maior primo menor que 2^{16}). *S2* também possui 16 bits e é a soma da mensagem de 8 em 8 bits multiplicada por sua posição do final da mensagem módulo 65521. *S1* possui valor inicial igual a 1 para fazer com que o comprimento da mensagem seja parte de *S2* e desta forma a inserção ou remoção de bytes na mensagem seja detectado. A utilização do primo 65521, segundo Adler é para fazer com que todos os surtos de 16 bits sejam detectados e não aconteça o mesmo problema encontrado em algumas versões do Fletcher32.

Na prática para a codificação do Adler não é necessário utilizar o operador de multiplicação. O efeito multiplicativo resulta da forma em que a soma é acumulada (Listagem 2). O Adler32 é essencialmente igual ao Fletcher, exceto que o *S1* é inicializado em 1 ao invés de zero, e a soma é reduzida módulo 65521. Outra diferença que possui implicações na capacidade de detecção de surtos do Adler32 é que a soma é tomada de byte em byte ao invés de dois bytes por vez como no Fletcher32.

Os códigos apresentados até agora possuem uma probabilidade de não detecção de erros P_{ue} da ordem de:

$$P_{ue} = \frac{1}{2^r}$$

Onde r é o número de bits de paridade. Para códigos que geram 32 bits de paridade isto seria igual a 2,3283 x 10^{-10}, como os bits de paridade do Adler32 são limitado a 65521^2 valores, devido ao fato que as suas duas somas são reduzidas módulo 65521, sua P_{ue} possui um valor um pouco maior [ZLIB02].

$$P_{ue(Adler32)} = \frac{1}{65521^2} = 2{,}3294 \times 10^{-10}$$

Sheinwald [SHW01] (apêndice C) mostra um padrão de erro que o Adler32 não detecta. Este padrão é um surto de 24 bits que altera três bytes consecutivos de x, y, z para x', y', z', quando os bytes estão relacionados na seguinte forma:

$$z'-z = -(x'-x)-(y'-y) \text{ e } 2x'+y' = 2x+y$$

Por exemplo, se uma seqüência consistindo dos valores "4, 2, 1" é alterada para "5, 0, 2" os bits de verificação permanecem inalterados. Este surto passa desapercebido pelo código, pois ele manipula um byte por vez.

Sheinwald também cita a falta de embasamento teórico no desenvolvimento do Adler32.

Devido ao seu objetivo de desenvolvimento, o de ser um algoritmo rápido quando codificado por software, o grupo que está desenvolvendo o Stream Control Transport Protocol (SCTP) [RFC2960] o escolheu como código detector de erros.

Stone [STN02] detectou uma fraqueza neste código quando ele é utilizado em conjunto com mensagens de comprimento pequeno, com o qual o SCTP deve manipular, por isso, ele teve que ser substituído (o email informando este problema e a possibilidade de substituição deste código está no apêndice D).

O problema com mensagens de pequeno comprimento se encontra no acumulador *S1*. Como ele guarda valores até 16 bits e a soma é de 8 em 8 bits, deve haver várias iterações até que todos os seus bits sejam afetados, o que resulta em uma distribuição de valores de *S1* próximo a uma curva normal.

Como o Adler32 foi projetado para assegurar a integridade de arquivos comprimidos, que geralmente possuem vários megabytes, e não para pacotes de dados transitando por redes de computadores, que possuem um comprimento pequeno em comparação a estes arquivos, este problema não é de grande preocupação, segundo seus criadores [ZLIB02].

Listagem 2 – Pseudo-código para a codificação Adler32

```
inteiro i, s1, s2;

s1 = 1;
s2 = 0;
para i de 1 até comprimento_da_mensagem faça
    s1 = ( s1 + message[i] ) modulo 65521;
    s2 = ( s2 + s1         ) modulo 65521;
fim para;
retorne (s2 << 16) + s1) // concatena os valores
```

2.7. CONCLUSÃO

Neste capítulo citamos algumas definições importantes na teoria de códigos de controle de erros e discutimos as propriedades dos códigos em estudo. A Tabela 2 traz um resumo da capacidade de detecção de erros destes códigos.

Tabela 2 – Capacidade de detecção de erros dos códigos analisados

Código	d_{min}	Detecção de Surto (bits)	P_{ue}
CRC16 CCITT	4 até 32.752 bits*	16	$1{,}53 \times 10^{-5}$
CRC32B	3 até 2.147.483.616 bits*	32	$2{,}33 \times 10^{-10}$
CRC32C	4 até 1.073.741.792 bits*	32	$2{,}33 \times 10^{-10}$
TCP16	2	15	$1{,}53 \times 10^{-5}$
TCP32	2	31	$2{,}33 \times 10^{-10}$
Fletcher16 255	3 até 2.040 bits*	7	$1{,}53 \times 10^{-5}$
Fletcher16 256	2	16	$1{,}53 \times 10^{-5}$
Fletcher32 65535	3 até 524.280 bits*	15	$2{,}33 \times 10^{-10}$
Fletcher32 65536	2	32	$2{,}33 \times 10^{-10}$
Adler32	3	23	$2{,}33 \times 10^{-10}$

* comprimento da mensagem a ser codificada.

3. METODOLOGIA

Neste capítulo iremos descrever os métodos utilizados para o desenvolvimento da simulação utilizada na avaliação dos códigos detectores de erros. Assim como os modelos de erro, tipos de arquivos e parâmetros empregados.

A simulação desenvolvida visa, a partir de modelos de erros, introduzir erros em palavras código dos códigos discutidos no capítulo anterior.

A idéia básica é gerar padrões de erros e verificar se o código consegue detectar as alterações. Através do número de vezes em que o código falhar, calcularemos a sua P_{ue} (Probabilidade de não detecção de erros) e analisaremos seu comportamento.

3.1. MODELOS DE ERRO

Lin [LIN83] classifica os erros em dois tipos principais que afetam canais com memória e canais sem memória.

Nos canais sem memória, o ruído afeta cada símbolo transmitido independentemente. Por exemplo, no canal BSC (*binary symmetric channel*) da Figura 4, cada bit transmitido possui uma probabilidade p de ser recebido incorretamente e uma

probabilidade 1 – *p* de ser recebido corretamente, independente de outros bits transmitidos. Em sistemas computacionais, certos erros de memória e a gradual degradação de hardware também causam este tipo de erro que afeta bits de forma independente.

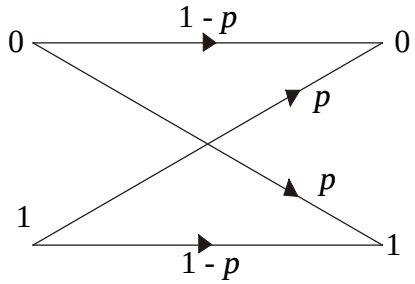


Figura 4 – *Binary symmetric channel.*

Em canais com memória, o ruído não é independente entre os símbolos transmitidos. Um modelo simplificado com memória é mostrado na Figura 5. Este modelo contém dois estados, um *estado bom*, no qual erros de transmissão ocorrem com pouca freqüência, $p_1 \approx 0$, e um *estado mal*, em que erros de transmissão são altamente prováveis, $p_2 \approx 0{,}5$. O canal está no estado bom na maioria do tempo, mas quando ele muda para o estado mal devido a uma alteração nas características de transmissão do canal, os erros de transmissão ocorrem em surto devido à alta probabilidade de transição neste estado. Falhas transientes ou intermitentes em hardware também podem introduzir erros sobre vários símbolos consecutivos. Isto cria, do ponto de vista de decodificador, erros em surto nas palavras código recebidas. Erros em surto são predominantes também em sistemas de comunicação móvel, onde o canal de comunicação é afetado por *multipath fading*.

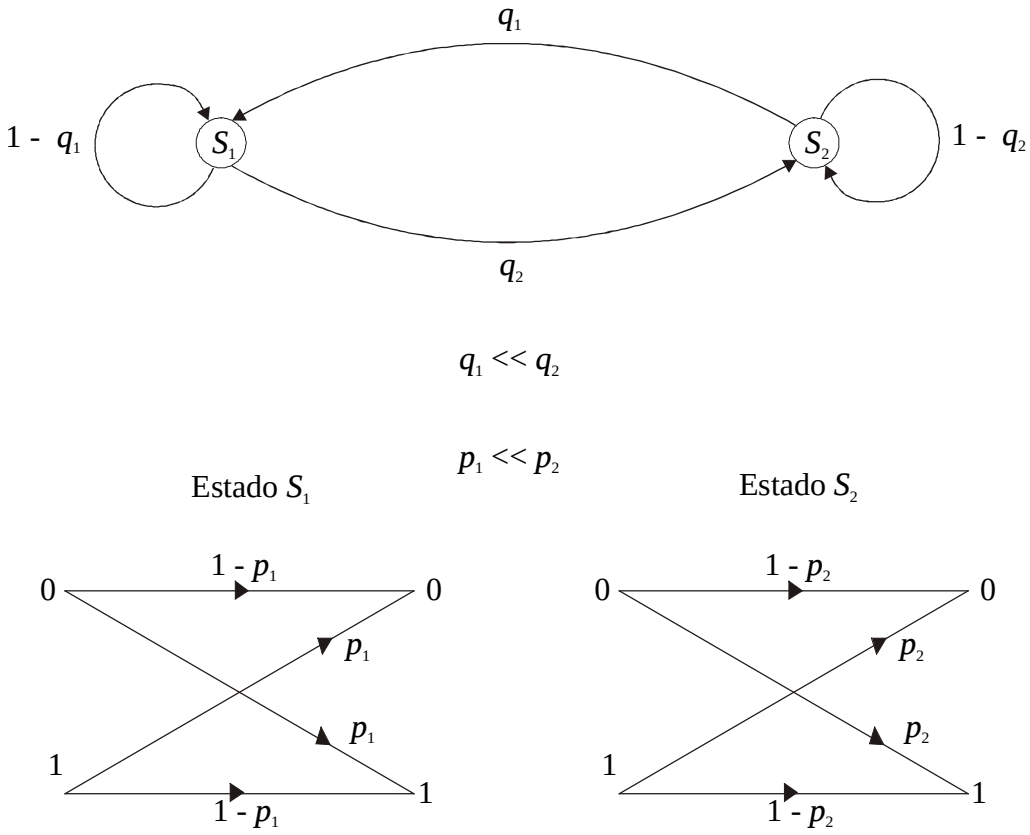


Figura 5 – Modelo simplificado de um canal com memória [LIN83].

Vários autores definem o erro em surto como sendo um padrão de erro que afeta *b* bits, onde o padrão de erro começa e termina com um valor diferente de zero [WCK95] [TAN02] [LIN83]. Por exemplo, um padrão de erro de um surto que afetasse 7 bits em um canal binário, teria a seguinte aparência:

...000001XXXXX10000...

Neste trabalho iremos adotar a definição proposta por Wolf [WLF94]. Para erros em surto, Wolf sugeriu um modelo de surto (*b*:*p*), onde os erros ocorrem randomicamente dentro de um intervalos de *b* bits, com uma probabilidade de erro de cada bit igual à *p* ($0 \leq p \leq 1$).

Os códigos estudados no artigo de Wolf foram os CRCs, e para eles o valor de $p = 0,5$, era considerado o pior caso. Os valores das P_{ue} para os códigos CRCs estudados por Wolf são conhecidos ($1/2^r$) e são independentes da escolha do polinômio primitivo. Com esta nova definição de surto, onde p pode ser diferente de 0,5, para um determinado b podem existir valores de p, diferentes de 0,5 que maximizam a P_{ue} do surto (b:p).

Desta forma, implementamos dois modelos de erros:

- Erros em surto afetando b bits consecutivos na palavra código, conforme definição de Wolf;
- Poucos erros independentes afetando alguns bits na palavra.

3.2. PARÂMETROS DE SIMULAÇÃO

Ao trabalharmos com o modelo de erros independentes, teremos apenas um parâmetro específico deste modelo, o número n de bits afetados na palavra código. Este parâmetro irá variar de 2 à 9 bits que serão escolhidos randomicamente na palavra código. Estes valores refletem a baixa quantidade de bits que são afetados por este modelo [LIN83] [HLZ91].

Com o modelo de surto, temos dois parâmetros específicos, b e p. O comprimento do surto b assumirá os valores de 8, 16, 24, 32, 40, 48, 56 e 64 bits, devido ao fato que os computadores atuais trabalham com palavras múltiplas de 8 bits [STL99] [FLT82]. Erros gerados por software e barramentos de dados resultam em surtos com estes comprimentos. Já p irá variar de 0,01 (1%) à 1 (100%) em incrementos de 0,1 para que possamos encontrar a máxima P_{ue} dos códigos.

Se tivermos um experimento que teremos que repetir b vezes uma prova, e em cada prova particular existir uma probabilidade associada a um determinado evento, chamada de probabilidade de sucesso p, temos que a probabilidade do evento ocorrer x vezes em b provas é dada pela função:

$$p(x) = \left(\frac{b!}{x!(b-x)!} \right) \left(p^x (1-p)^{(b-x)} \right)$$

A prova neste caso chama-se prova de Bernoulli, e *x* diz-se distribuída segundo Bernoulli, ou distribuída binomialmente.

A distribuição da quantidade de bits alterados no modelo de surto (*b*:*p*) de Wolf, segue esta distribuição, pois dentro da amplitude *b* do surto (número de provas), cada bit é alterado independentemente do outro por uma probabilidade *p*.

Deste modo, dada a equação da distribuição binomial anterior, podemos traçar alguns gráficos preliminares para verificar o comportamento deste modelo.

Na Figura 6, fixamos $p = 0,5$ e traçamos várias curvas[1] com os diferentes valores de *b* que utilizaremos na simulação. Para os menores valores de *b* verificamos que o valor central do surto (número de bits com erros) possui uma grande probabilidade de ocorrência (mais de 25% do surto alterar 4 bits quando $b = 8$), sendo que este valor decai a medida em que aumentamos o comprimento do surto.

Já na Figura 7, fixamos $b = 40$ e investigamos o comportamento da distribuição a medida em que alteramos o valor *p*. Para $p < 0,5$, padrões que alteram menos da metade dos bits no surto são mais prováveis ($p = 0,1$); para $p > 0,5$ padrões com mais da metade dos bits em erro são mais prováveis ($p = 0,9$); para $p = 0,5$ temos a situação que no capítulo anterior menciona como padrões equiprováveis, ou seja, todos os padrões possuem a mesma probabilidade de ocorrer ($p = 0,5$); por fim, se $p << 0,5$ a distribuição estará centrada em poucos erros ($p = 0,01$).

[1] Utilizamos o programa DERIVE para traçar os gráficos da Figura 6 e 7.

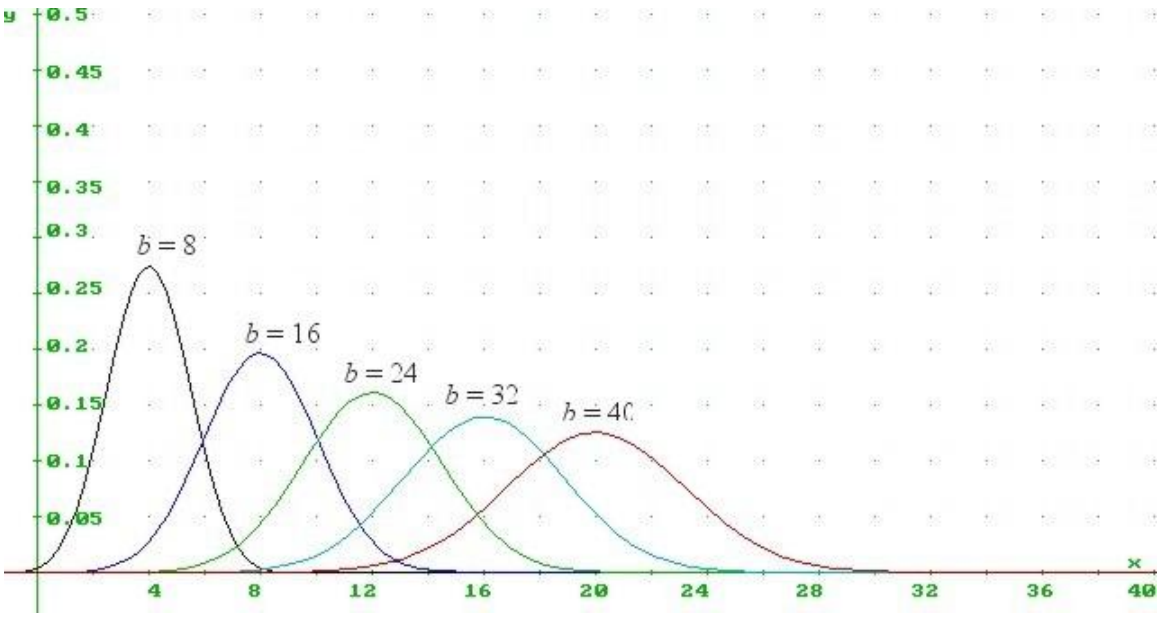


Figura 6 – Distribuição binomial, utilizando como parâmetros p = 0,5 e b = [8; 16; 24; 32; 40].

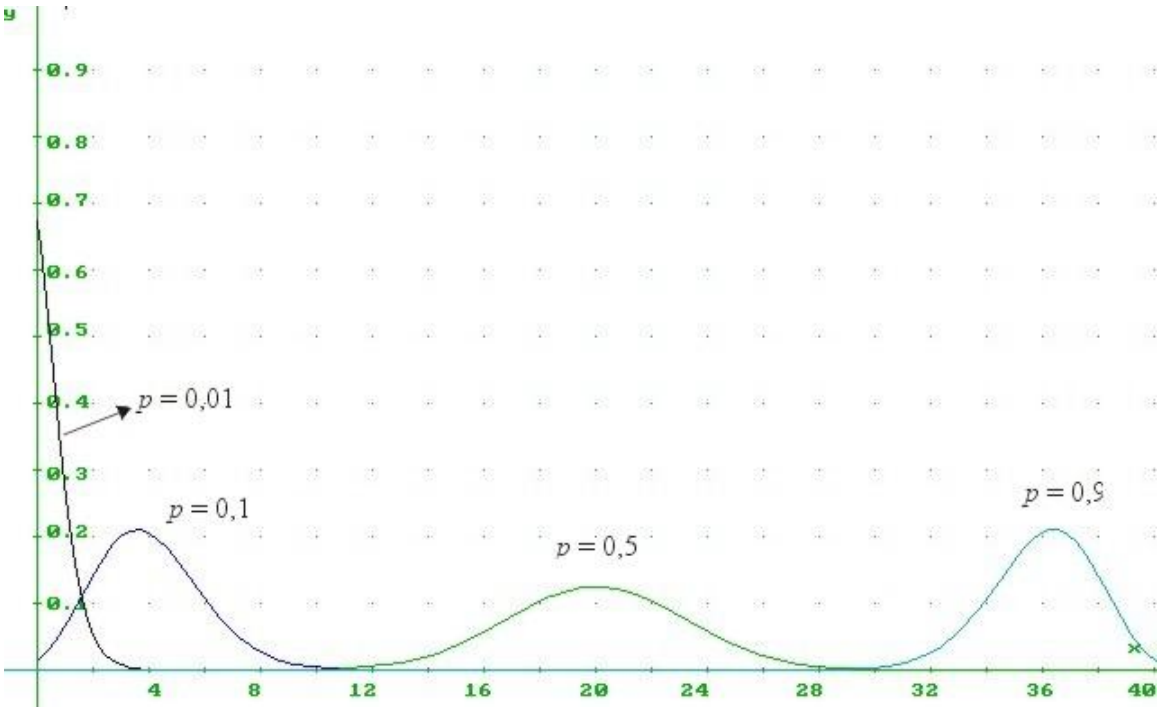


Figura 7 – Distribuição binomial, utilizando como parâmetros b = 40 e p = [0,01; 0,10; 0,5; 0,9].

Para tornarmos o resultado da simulação o mais próximo do que aconteceria em uma rede real, e para verificarmos se existe qualquer fraqueza dos códigos quando as mensagens possuem dados não uniformemente distribuídos (que indicaria a não linearidade dos códigos), além dos dados randômicos, que são geralmente empregados em simulações como esta [HLZ91], utilizaremos mais 3 tipos de dados nos testes. O conjunto de tipos de dados empregado no teste consiste de:

- Dados uniformemente distribuídos – para simular este tipo de informação utilizamos a função random() (C++ Builder 5 SP1) que utiliza um gerador congruente multiplicativo de números randômicos com período igual a 2^{32} para retornar sucessivos números pseudo randômicos.
- Mensagens SS7 – São mensagens de sinalização entre centrais telefônicas, o arquivo em questão possui vários minutos de troca de informação real, como requisições de estabelecimento de chamadas. O SCTP foi desenvolvido inicialmente para transportar este tipo de dados.
- Português – Possui várias obras clássicas, em texto puro (ASCII), da língua portuguesa que incluem entre outros autores Machado de Assis, José de Alencar e Aluísio de Azevedo.
- Inglês – Para este tipo de dados montamos um arquivo com todas as obras de Shakespeare codificadas em HTML.

Outro parâmetro que utilizamos em nossa simulação foi o comprimento da mensagem que seria codificada

Em redes Ethernet o limite de um pacote é de 12.144 bits (1518 bytes), e devido ao algoritmo de descoberta de MTU (***maximum transmission unit***) utilizado pelo TCP/IP [RFC1191], que verifica o comprimento máximo dos pacotes que podem trafegar entre dois pontos antes de iniciar a transmissão de dados, desta forma limitando o comprimento máximo da maioria dos pacotes na Internet, geramos pacotes com 1.500 bytes para simular o comprimento máximo de quadro que pode transitar em redes Ethernet.

Segundo Stone [STN02] as mensagens de sinalização (SS7) são geralmente menores que 128 bytes, por isso para verificarmos palavras código de comprimento pequeno iremos gerar também pacotes de 100 bytes.

3.3. DETERMINAÇÃO DO NÚMERO DE ITERAÇÕES

Um fator importante para alcançar resultados confiáveis é a determinação do número de iterações, ou seja, o número de vezes em que iremos gerar palavras código e alterá-las para verificar se o código tem sucesso na detecção das alterações.

O ponto inicial foi escolher no modelo de erro parâmetros para os quais já soubéssemos o valor esperado da taxa de falhas. Como Wolf [WLF94] notou em seu trabalho que a utilização de um surto (b:0,5) gera resultados iguais aos esperados teoricamente, escolhemos $b = 40$, pois os códigos estudados não possuem garantia de detectar todos os surtos deste comprimento e variamos o número de iterações[2] até que o valor do P_{ue} encontrado estivesse bem próximo a $1/2^r$, onde r é o número de bits redundantes.

Para os códigos onde $r = 16$ a P_{ue} é igual a $1/2^{16} \cong 1{,}53\text{x}10^{-5}$, já para os códigos que geram 32 bits de paridade, $r = 32$ e a P_{ue} é igual a $1/2^{32} \cong 2{,}33\text{x}10^{-10}$.

Verificamos, após alguns testes iniciais, que para termos resultados confiáveis para os códigos que possuem $r = 16$, poderíamos utilizar entre 10 milhões e 100 milhões de iterações, e traçamos os gráficos para cada um dos tipos de dados estudados (Figuras 8, 9, 10 e 11).

Para os códigos com 32 bits de paridade, encontramos mais estabilidade nos resultados utilizando o modelo de surto (40:0,5) em torno de 11 e 20 bilhões de iterações (Figura 12), um valor inviável para os nossos recursos computacionais.

[2] Gostaríamos de agradecer ao Prof. Dr. Valdemar C. da Rocha Jr. pela sugestão.

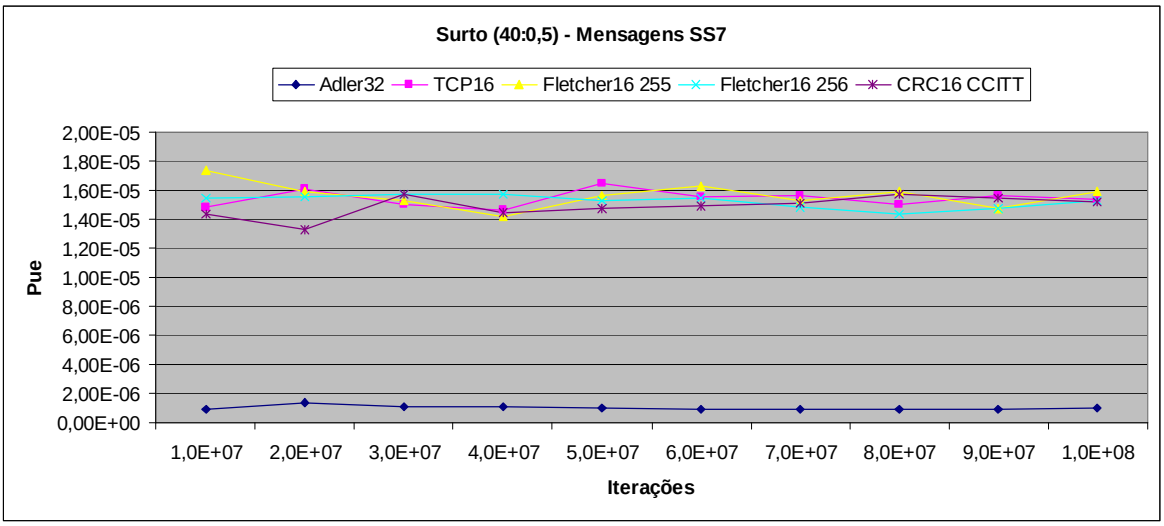


Figura 8 – Resultado da aplicação do surto (40:0,5) nos códigos estudados utilizando dados não uniformemente distribuídos pela variação do número de iterações (Dados: Mensagens SS7; Comprimento da mensagem: 100 bytes).

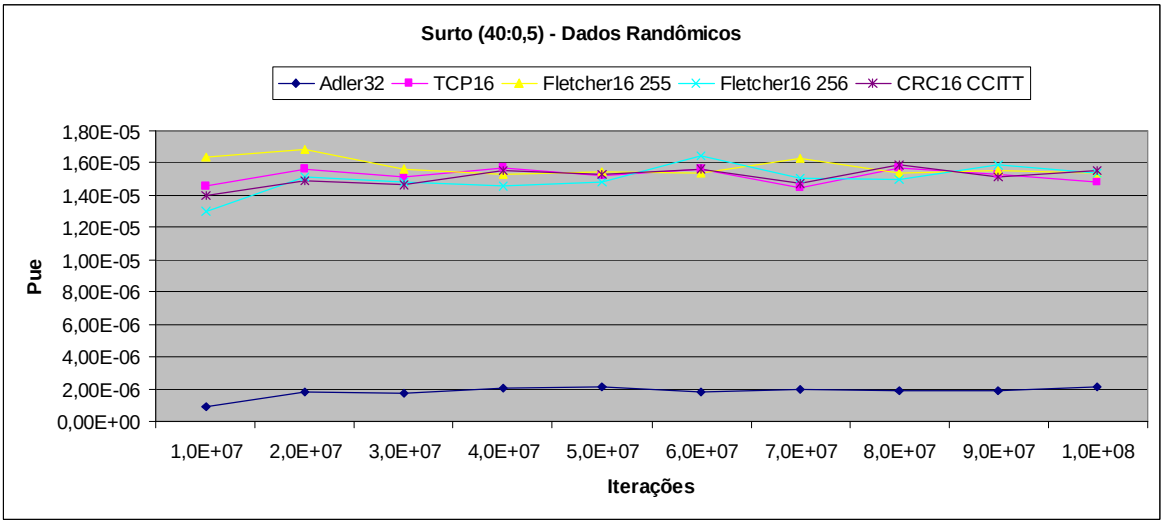


Figura 9 – Resultado da aplicação do surto (40:0,5) nos códigos estudados utilizando dados uniformemente distribuídos pela variação do número de iterações (Dados: Randômicos; Comprimento da mensagem: 100 bytes).

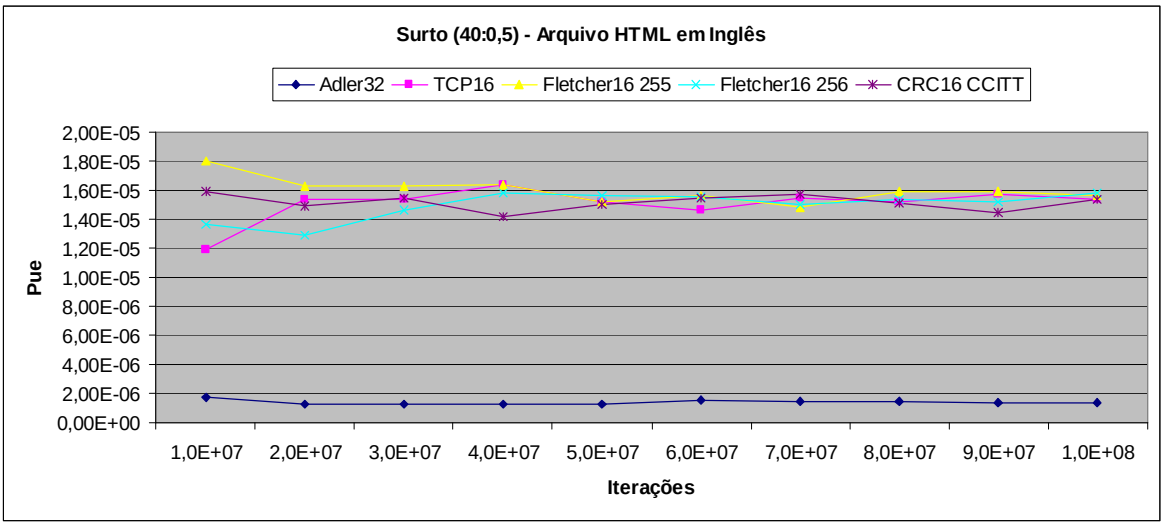


Figura 10 – Resultado da aplicação do surto (40:0,5) nos códigos estudados utilizando dados não uniformemente distribuídos pela variação do número de iterações (Dados: Arquivo HTML em Inglês; Comprimento da mensagem: 100 bytes).

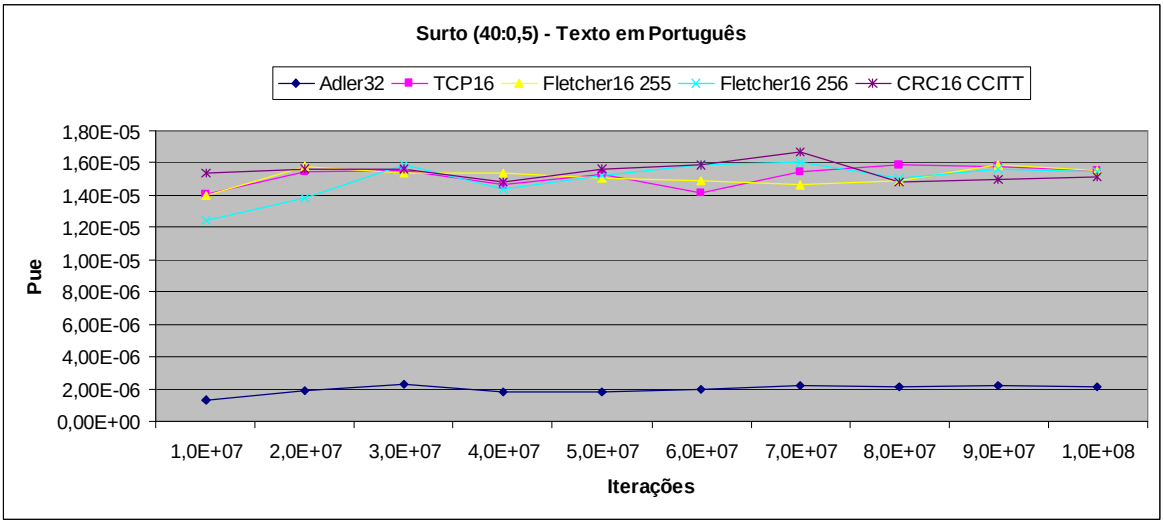


Figura 11 – Resultado da aplicação do surto (40:0,5) nos códigos estudados utilizando dados não uniformemente distribuídos pela variação do número de iterações (Dados: Texto em Português; Comprimento da mensagem: 100 bytes).

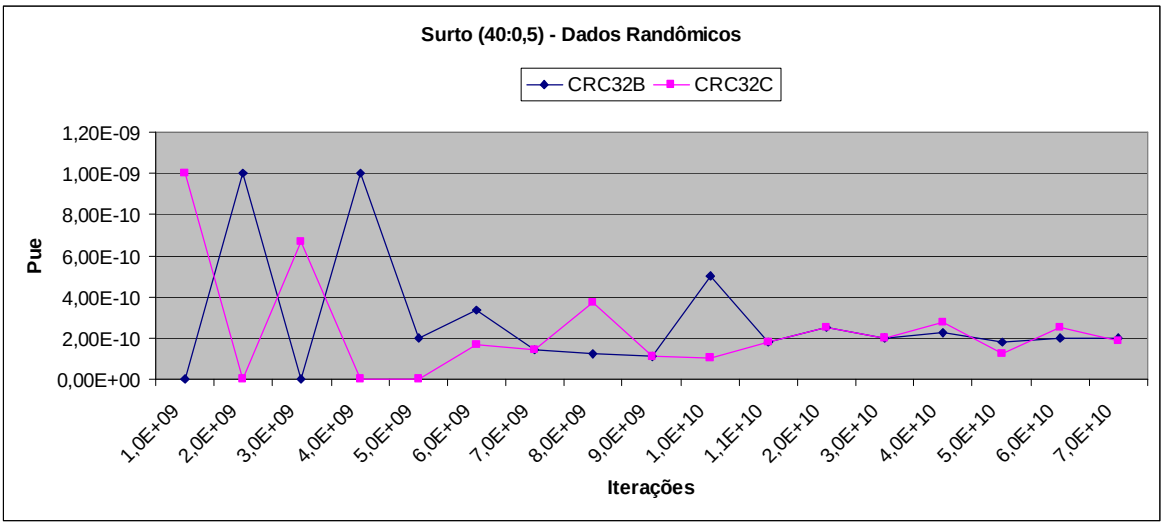


Figura 12 – Resultado da aplicação do surto (40:0,5) nos códigos CRC32B e CRC32C utilizando dados uniformemente distribuídos pela variação do número de iterações (Dados: Randômicos; Comprimento da mensagem: 5 bytes).

Como verificado nos gráficos anteriores, a P_{ue} dos códigos, realmente tendeu ao valor teórico esperado. E em 50 milhões de iterações obtivemos uma diferença em relação ao esperado teoricamente ($1/2^r$) da ordem +/- 5% (Tabela 3). Devido a este resultado e ao fato que se utilizássemos uma quantidade de iterações maior do que esta não poderíamos completar o trabalho em tempo hábil, a quantidade de iterações utilizada para a obtenção dos resultados de agora em diante, se não houver nota em contrário, será de 50 milhões. Esta quantidade de iterações, no entanto, é insuficiente para mensurar falhas em detectar erros, quando a P_{ue} for menor que $1/(5x10^7)$.

Tabela 3 – Variação dos resultados em relação ao esperado teoricamente.

	(%) de variação			
Tipo de dados	TCP16	Fletcher16 255	Fletcher16 256	CRC16 CCITT
Mensagens SS7	5,78	2,37	0,14	-3,27
Dados randômicos	-0,52	1,19	-2,74	0,53
Arquivo HTML em Inglês	-0,39	-0,52	2,24	-1,96
Texto em Português	0,27	-1,30	-0,39	2,24

De acordo com gráficos obtidos dos dados de nossa simulação, a variação dos códigos (de 16 bits de paridade) ou do tipo de dados não influi nos resultados de forma significativa, no modelo de surto (40:0,5).

O Adler32 foi o único código com 32 bits de paridade que já nestes testes iniciais apresentou uma taxa alta de falhas. Conforme calculado no capítulo anterior, ele devia apresentar uma $P_{ue} \cong 2{,}33\text{x}10^{-10}$, mas encontramos uma valor próximo a $2{,}00\text{x}10^{-6}$.

Para o modelo de poucos erros independentes, utilizamos um número de alterações igual a 4, que é maior ou igual a distância mínima de todos os códigos estudados quando são utilizadas palavras código de 100 bytes. Os resultados também apresentaram uma variação pequena em torno de 50 milhões de iterações (Figura 13). Mas neste teste todos os códigos, exceto o CRC32B e CRC32C, apresentaram um valor superior à $1/2^r$ (Tabela 4). A variação do tipo de dados utilizado, mais uma vez não resultou em variações significativas.

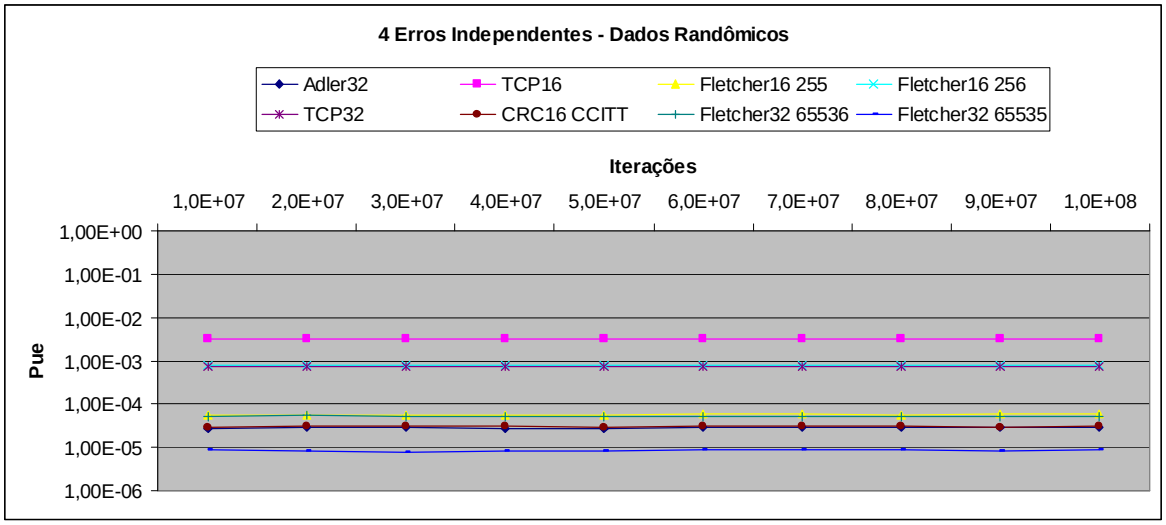


Figura 13 – Resultados da alteração de 4 bits nas palavras código dos códigos estudados. (Dados: Randômicos; Comprimento da mensagem: 100 bytes).

Tabela 4 – Resultado da alteração de 4 bits, nas palavras código, dos códigos estudados, em $5{,}00 \times 10^{7}$ iterações.

Código	P_{ue} (4 bits)
Adler32	$2{,}70 \times 10^{-5}$
TCP16	$3{,}10 \times 10^{-3}$
TCP32	$7{,}17 \times 10^{-4}$
Fletcher16 255	$5{,}54 \times 10^{-5}$
Fletcher16 256	$7{,}97 \times 10^{-4}$
Fletcher32 65535	$8{,}22 \times 10^{-6}$
Fletcher32 65536	$5{,}07 \times 10^{-5}$
CRC16 CCITT	$2{,}99 \times 10^{-5}$

3.4. VALIDAÇÃO DOS RESULTADOS

Apesar dos resultados anteriores com surtos (*b*:0,5) serem próximos aos calculados teoricamente, para validar o modelo de surto de Wolf, iremos comparar os resultados obtidos neste trabalho com os que foram publicados em seu artigo [WLF94].

Inicialmente, Wolf procurou a probabilidade p^* que maximizava a P_{ue} dos códigos CRC16, CRC16 CCITT e CRC16 Q utilizando um surto de comprimento $b = 20$ ($P_{ue}(20{:}p^*)$). Através da variação do valor de p de 0,1 a 1,0. Wolf encontrou o gráfico da Figura 15, onde a p^* do CRC16 CCITT foi igual a 0,2. Em nosso trabalho, utilizando os mesmos parâmetros traçamos o gráfico da Figura 14, e encontramos o mesmo valor de p^*, ou seja, 0,2. O gráfico também se comportou de forma idêntica ao de Wolf, apresentando duas máximas locais, uma para p menor que 0,5 e uma para p acima de 0,5.

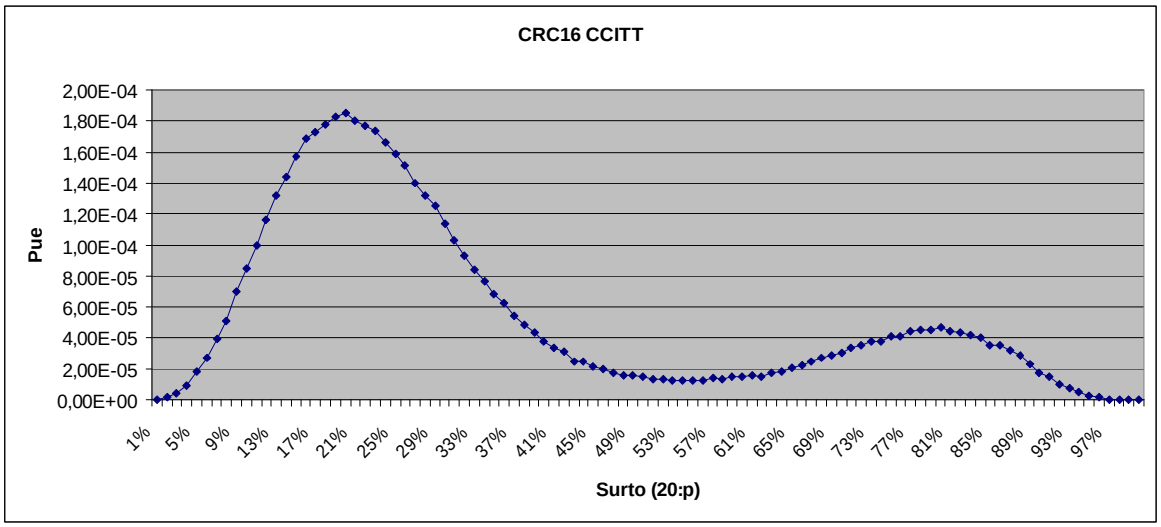


Figura 14 – Resultados da aplicação de surto (20:*p*) no código CRC16 CCITT. (Dados: Randômicos; Comprimento da mensagem: 100 bytes). Escala linear.

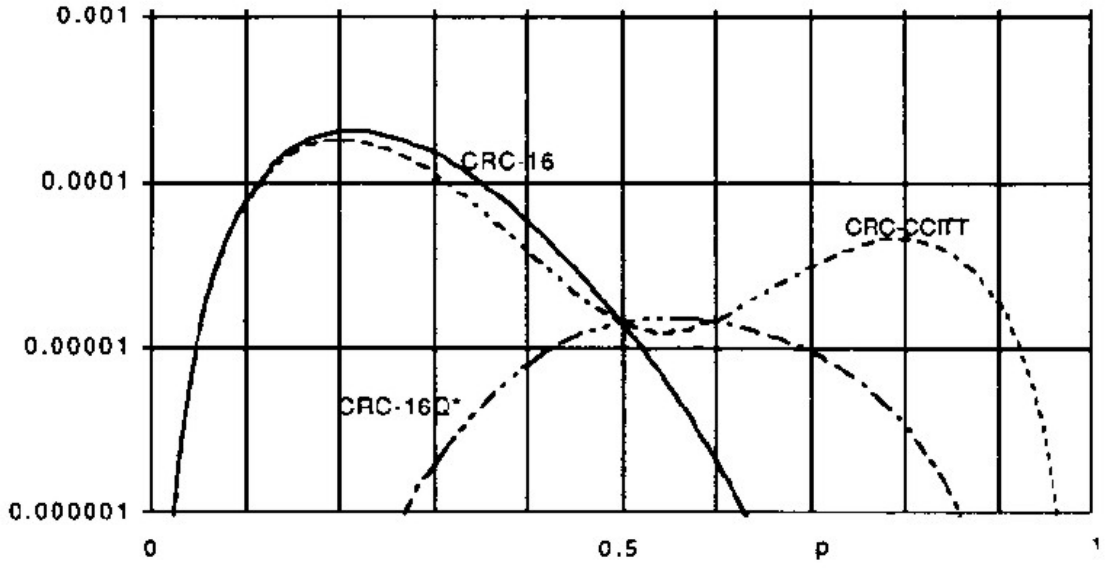


Figura 15 – Resultados da aplicação de surto (20:*p*) no código CRC16, CRC CCITT (CRC16 CCITT) e CRC16 Q [WLF94].

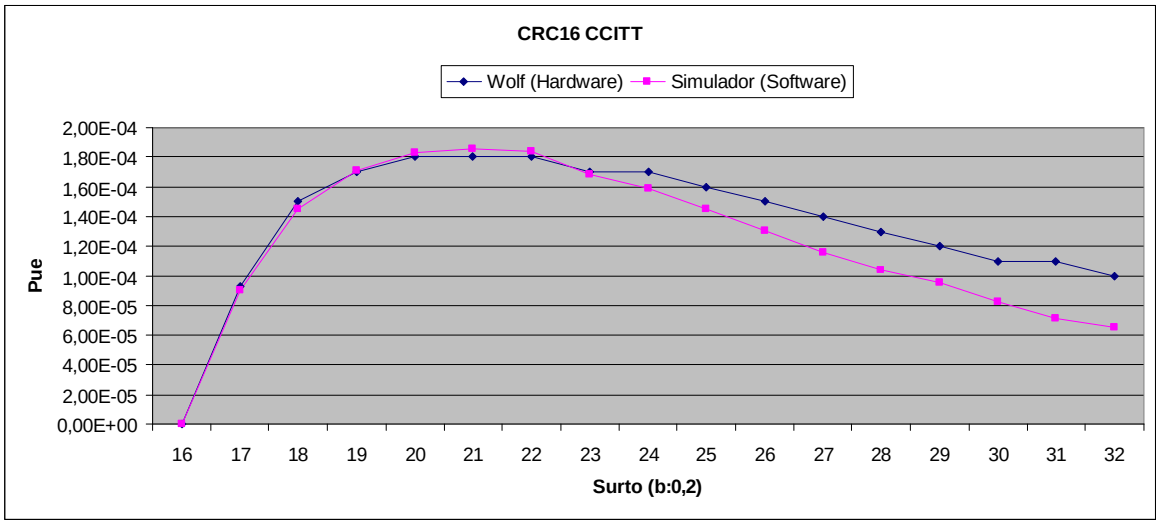


Figura 16 – Comparação entre os resultados obtidos por Wolf [WLF94] e os deste trabalho.

Após encontrar o p^* do CRC16 CCITT, para surto de comprimento $b = 20$, Wolf variou o parâmetro b de comprimento do surto entre 16 e 32 bits, registrando estes valores em uma tabela, utilizando estes mesmos parâmetros traçamos o gráfico da Figura 16 e comparamos com os encontrados por Wolf. Os valores são praticamente idênticos até 23 bits.

Utilizamos dois algoritmos diferentes, um que gerava as palavras código e depois adicionava o ruído e outro que somente gerava o ruído para verificar se era uma palavra código do CRC16 CCITT válida, os mesmos resultados foram encontrados nos dois casos.

3.5. CONCLUSÃO

Neste capítulo listamos os modelos de erros e os parâmetros que foram utilizados em nosso trabalho e comparamos alguns resultados com valores esperados teoricamente. Utilizando os parâmetros que Wolf [WLF94] usou em seu trabalho, refizemos os testes e comparamos os resultados.

Também notamos que os resultados iniciais de nossa simulação já possuem algumas informações interessantes. O código Adler foi o único com 32 bits de paridade que apresentou uma alta taxa de falhas, mesmo quando o surto utilizado possuía probabilidade ***p*** igual a 0,5. No modelo de erros aleatórios com 4 bits alterados, todos os códigos (exceto o CRC32B e o CRC32C) apresentaram uma taxa alta de falhas.

4. DISCUSSÃO DOS RESULTADOS

Neste capítulo iremos expor os resultados de nosso trabalho, assim como uma análise destes resultados para cada código.

Inicialmente verificaremos se os códigos sofrem alguma influência do tipo de dados ou do comprimento da mensagem a ser codificada. Para tanto codificaremos duas versões de mensagens (uma com 100 bytes, e outra 1.500 bytes) com os seguintes tipos de dados:

- Texto em português
- HTML em inglês
- Mensagens SS7
- Dados randômicos

Aplicaremos a estas palavras código, o modelo de surto (b:0,5), b = [8, 16, 24, 32, 40, 48, 56, 64], e o modelo de erros independentes com n = [2, 3, 4, 5, 6, 7, 8, 9] 50 milhões de vezes para traçarmos os gráficos de P_{ue} de cada modelo de erro.

Para verificarmos se a p* (a probabilidade que maximiza um surto(b:p)), é afetada pelos tipos de dados, aplicaremos um surto (40:p) nos códigos (nenhum deles

detecta todos os erros com estes parâmetros, Tabela 2), para cada tipo de dados e condensamos os resultados em um gráfico por código.

Por fim, aplicaremos um surto(b:p), b = [8, 16, 24, 32, 40, 48, 56, 64], em mensagens randômicas para encontrarmos o comportamento da P_{ue} a medida em que variamos o comprimento do surto.

4.1. CÓDIGOS CÍCLICOS

Os CRCs obtiveram os melhores resultados de detecção de erros, seu comportamento como era de se esperar pela teoria apresentada, não é influenciado pelos tipos de dados utilizados na simulação nem pelo comprimento da palavra código, somente pelo modelo de erro empregado (Figura 17, 18 e 19).

Não conseguimos falhas suficientes para traçar os gráficos dos códigos CRC32B e CRC32C. Como teoricamente eles devem apresentar uma falha em 2^{32} ($4{,}29 \times 10^{9}$) e utilizamos 5×10^{9} iterações (5×10^{7} iterações x 100 passos) geralmente só um erro passava desapercebido por estes dois códigos em cada um dos testes realizados.

No surto (b:0,5) o CRC16 CCITT apresentou um comportamento que se repetiu na maioria dos códigos (Figura 17), quando teoricamente ele deveria detectar todos os erros (b = 8, 16) a P_{ue} foi igual a zero, nos outros casos (b = 24, 32, 40, 48, 56, 64) a P_{ue} manteve-se em 2^{-16}.

No modelo de erros independentes, o CRC16 CCITT se comportou de uma forma única entre os códigos estudados (Figura 18). Devido ao seu polinômio gerador ser da forma $p(x)(x - 1)$, todos os erros com peso ímpar foram detectados, mas com erros de peso par, a P_{ue} foi o dobro do teórico, ou seja, 2×2^{-16}. Encontramos somente uma citação para este comportamento, Fletcher [FLT82] cita que apesar de alguns CRCs detectarem todos os erros de peso ímpar, eles teriam algum tipo de fraqueza em algum outro tipo de erro.

Como utilizamos mensagens de 100 e 1.500 bytes, e em ambos o CRC16 CCITT possui a mesma distância mínima, também não houve influência do comprimento das mensagens nos resultados.

Verificamos também que a medida em que aumentávamos o comprimento do surto, havia uma diminuição das duas máximas locais encontradas por Wolf (Figura 19 e 20).

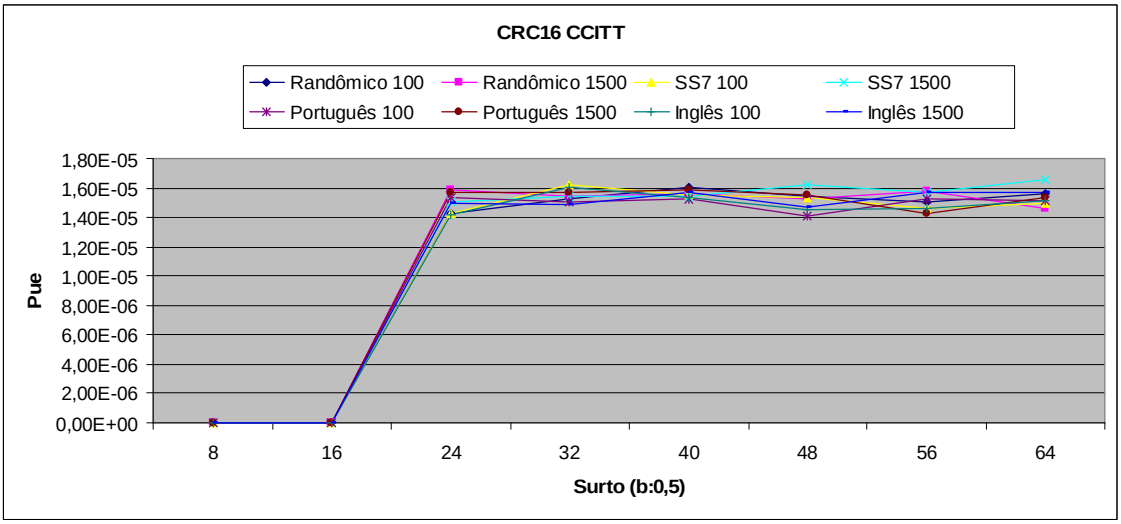


Figura 17 – CRC16 CCITT; Surto (*b*:0,5), variando o tipo de dados e o comprimento da mensagem.

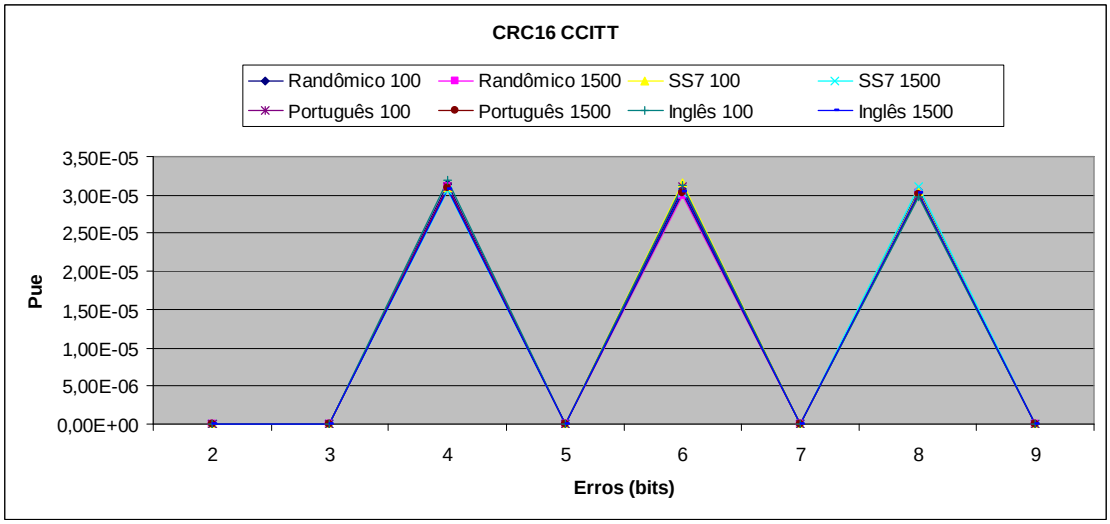


Figura 18 – CRC16 CCITT; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem.

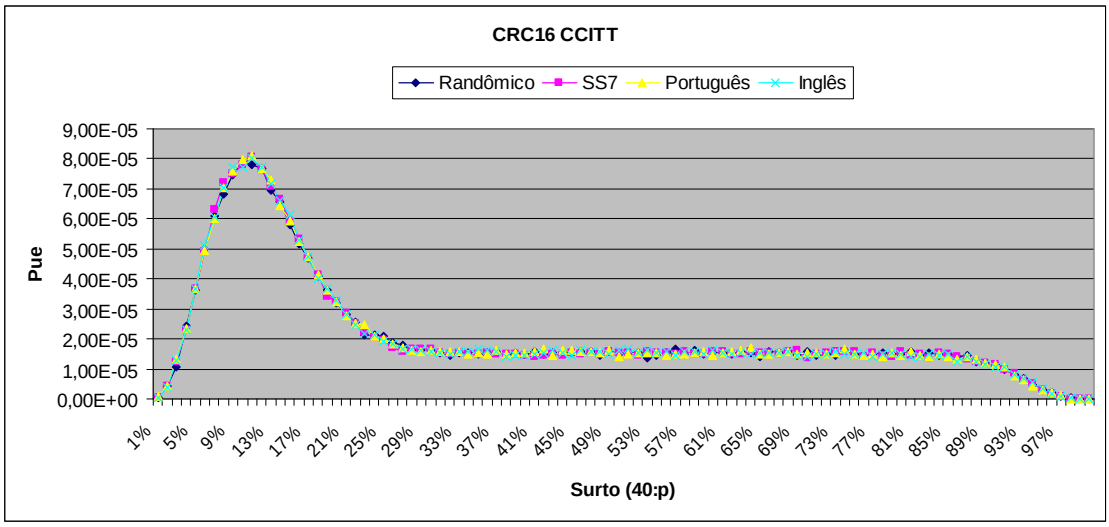
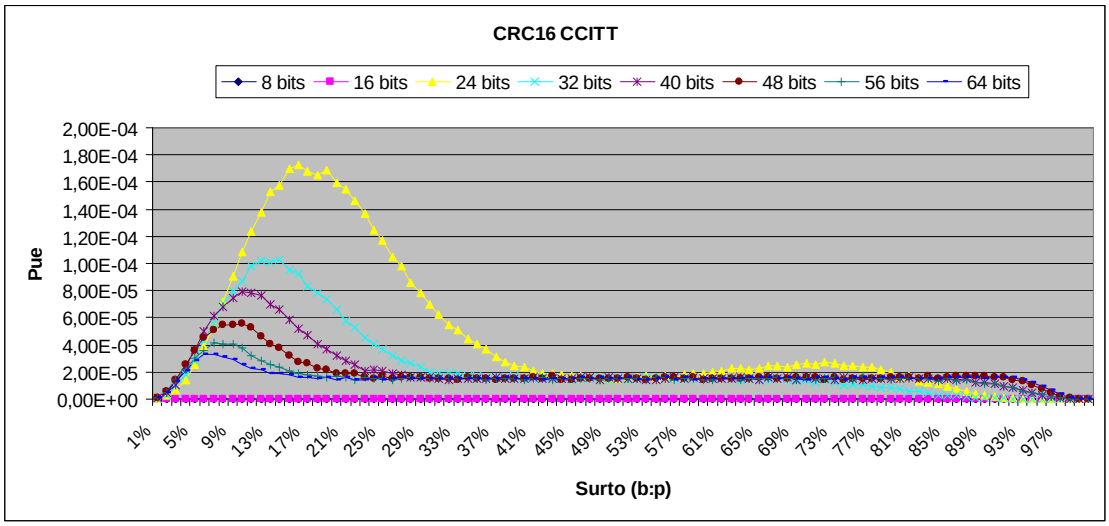
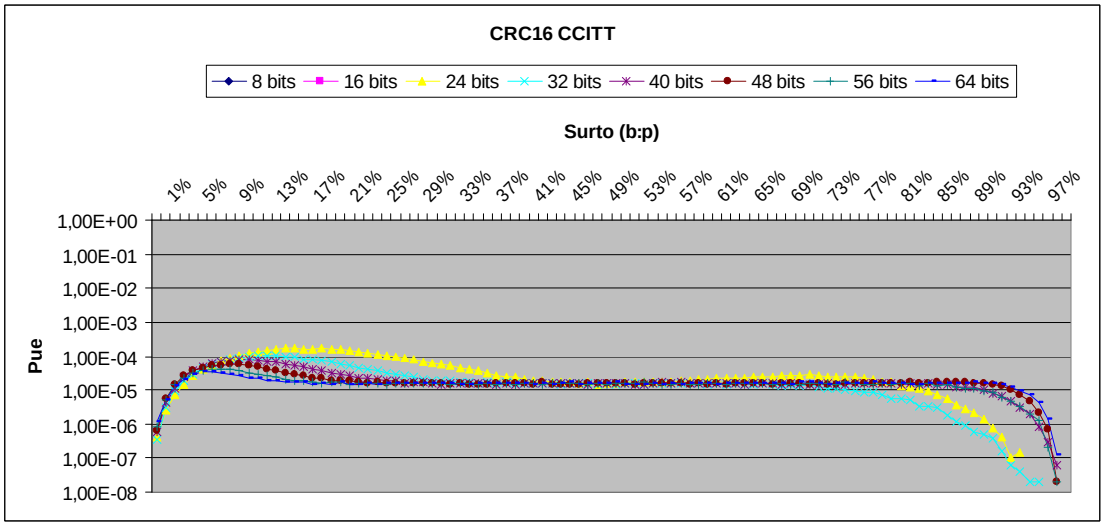


Figura 19 – CRC16 CCITT; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). Escala linear.



(a)



(b)

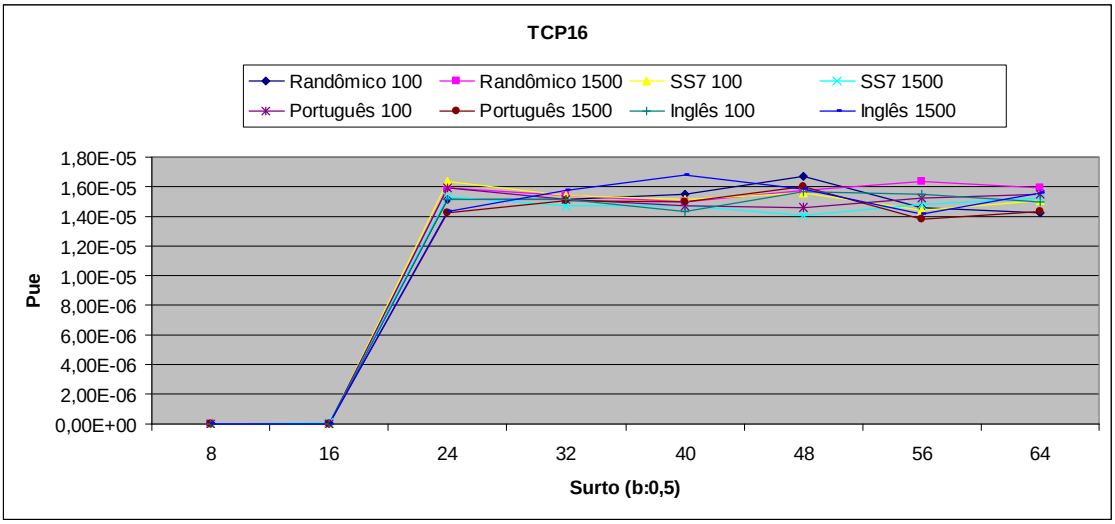
Figura 20 – CRC16 CCITT; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

4.2. INTERNET CHECKSUM

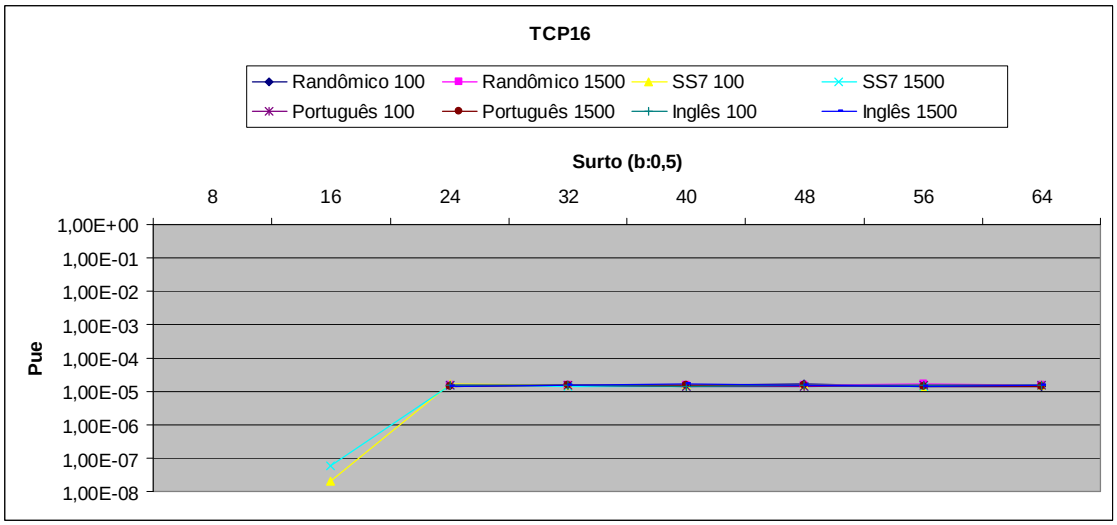
O TCP16 e o TCP32 apresentaram comportamentos semelhantes, ambos possuem uma altíssima taxa de falhas quando os padrões de erros alteram poucos bits (chegando a 1 falha para 100 detecções corretas). Neste caso eles possuem as piores taxas de detecção que encontramos (Figura 22, 23, 24, 25, 26, 27). Só foi possível traçar parte dos gráficos do TCP32 devido a esta alta taxa de falhas (traçamos as regiões dos gráficos que possuíam P_{ue} maior do que 10^{-8}).

Não encontramos, porém resultados que dão apoio a idéia que dados não uniformemente distribuídos afetem de forma negativa e significativa a capacidade de detecção de códigos como este, pois dos tipos de dados incluídos na simulação o que apresentou o pior resultado foi o que possuía dados uniformemente distribuídos (dados randômicos) (Figura 22, 23, 25, 26). Mas ao contrário do CRC16 CCITT, o *Internet Checksum* é afetado pela distribuição dos dados, como podemos verificar nos gráficos (Figura 22, 23, 25, 26), há uma diferença de 60% entre os valores obtidos entre os dados randômicos e o texto em português.

A única situação em que os dados não uniformemente distribuídos apresentaram uma taxa de falhas maior em relação aos dados uniformemente distribuídos foi quando havia uma condição inicial para ocorrência da falha na detecção do erro. No caso do TCP16 isto ocorre quando um surto de 16 bits substitui a palavra $\texttt{0000}_{\text{H}}$ pela $\texttt{ffff}_{\text{H}}$ e vice versa. Então se os dados forem uniformemente distribuídos a probabilidade de ocorrência destas palavras no fluxo de dados é de $2x2^{-16}$, e com um surto no qual $p = 0,5$, a probabilidade de falha seria de $2x2^{-16}x2^{-16} = 2x2^{-32} = 4,66x10^{-10}$. Como entre os tipos de dados, o que era composto de mensagens SS7 apresentava o maior número destas palavras, esta probabilidade foi maior, em torno de $5x10^{-8}$ (Figura 21b). Quando p é igual a 1, utilizando dados uniformemente distribuídos, a probabilidade teórica de falha é de $2x2^{-16} = 3,05x10^{-5}$, encontramos em nossa simulação o valor de $3,12x10^{-5}$ (Figura 24b).

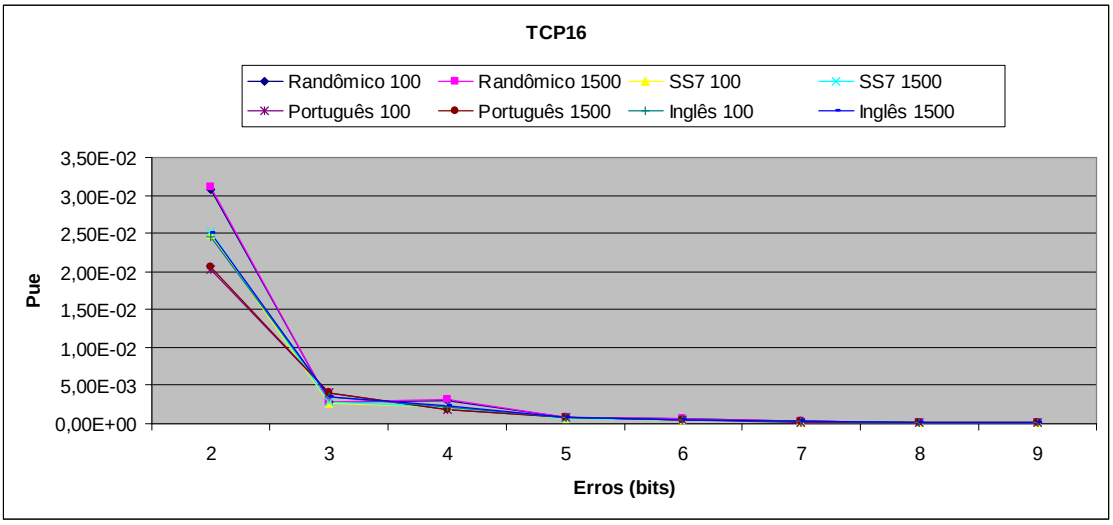


(a)

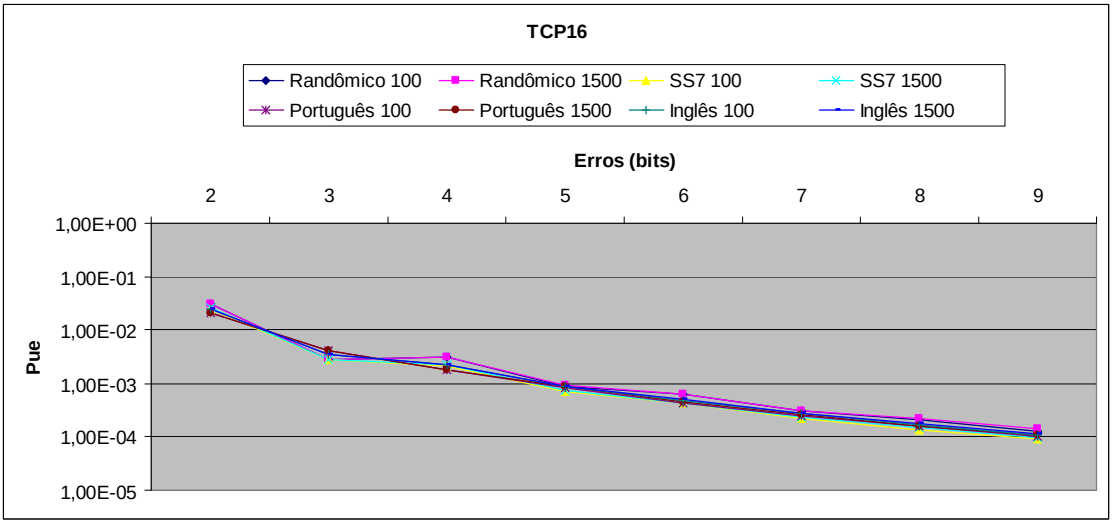


(b)

Figura 21 – TCP16; Surto (*b*:0,5), variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

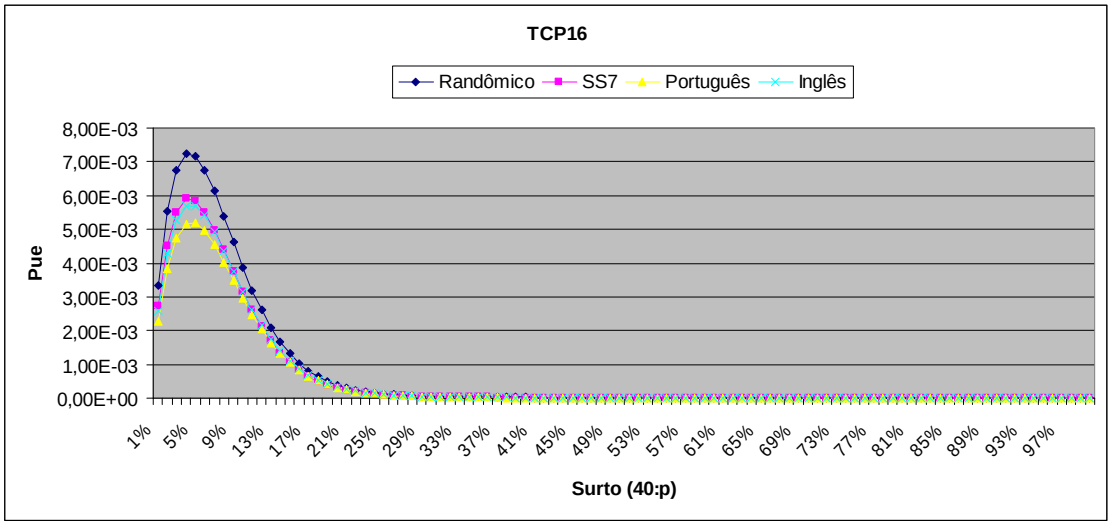


(a)

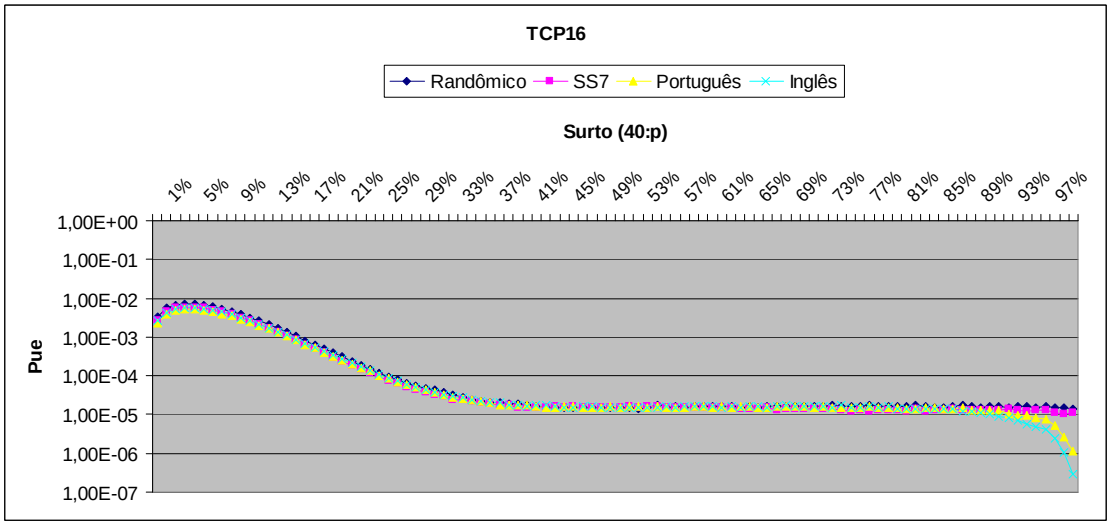


(b)

Figura 22 – TCP16; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

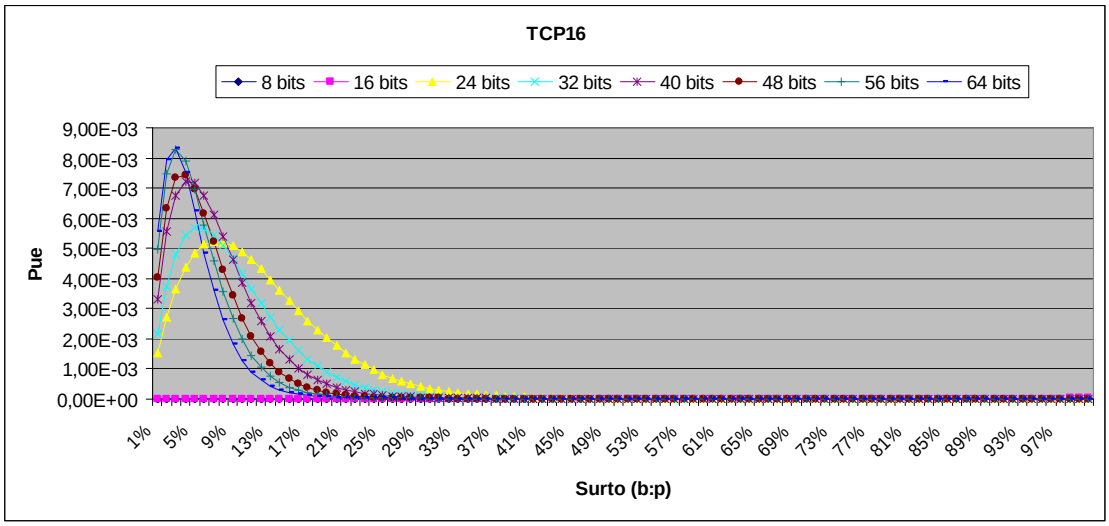


(a)

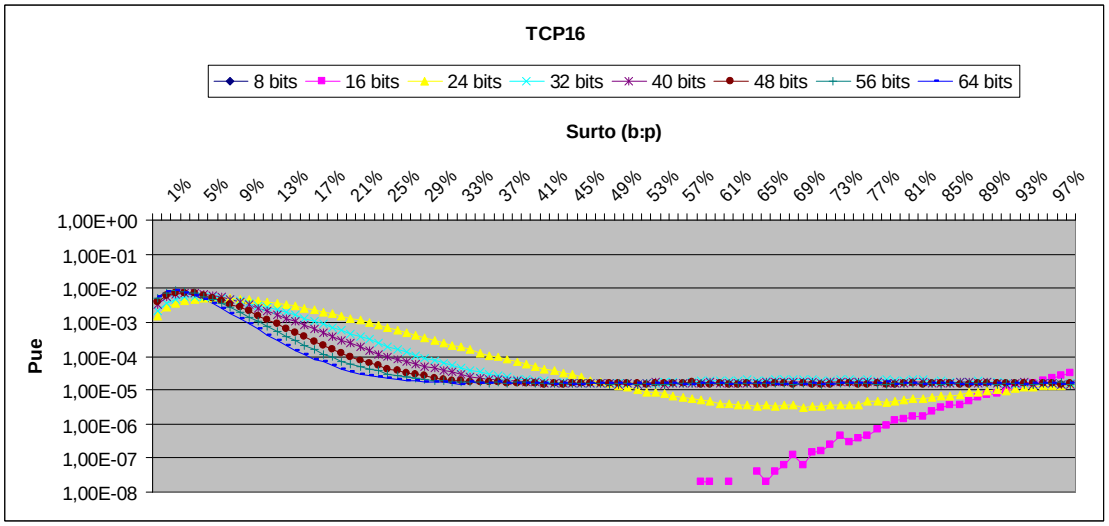


(b)

Figura 23 – TCP16; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

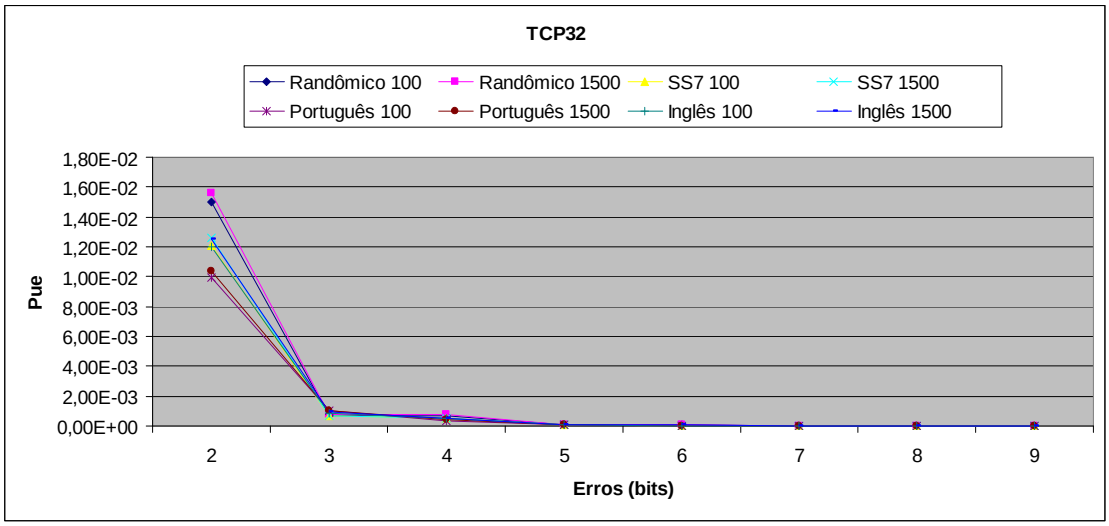


(a)

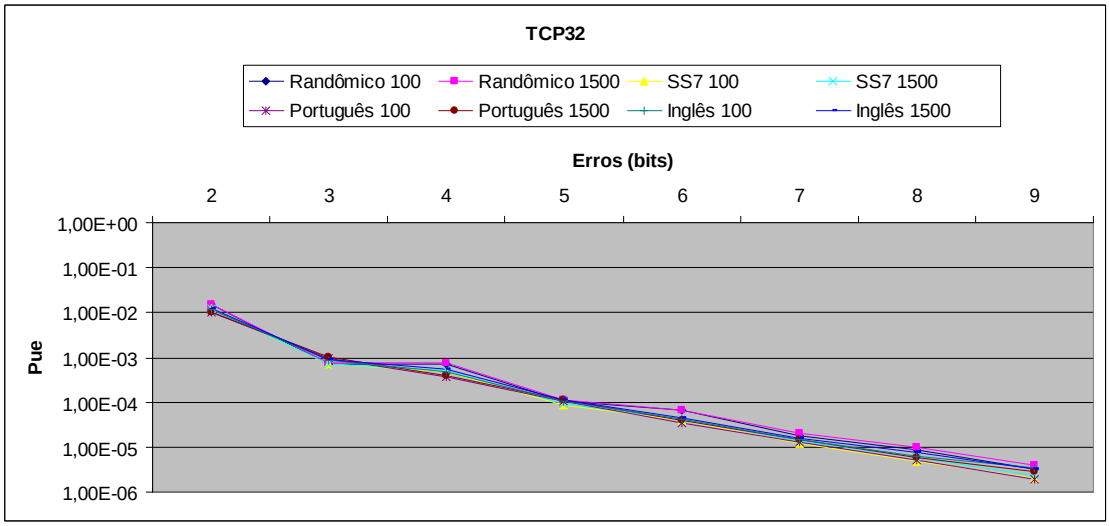


(b)

Figura 24 – TCP16; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

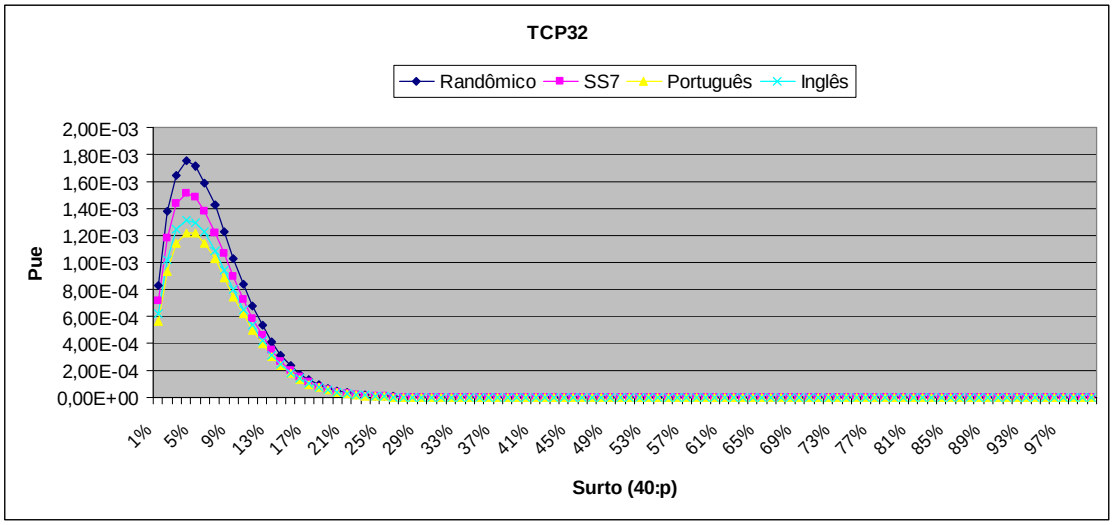


(a)

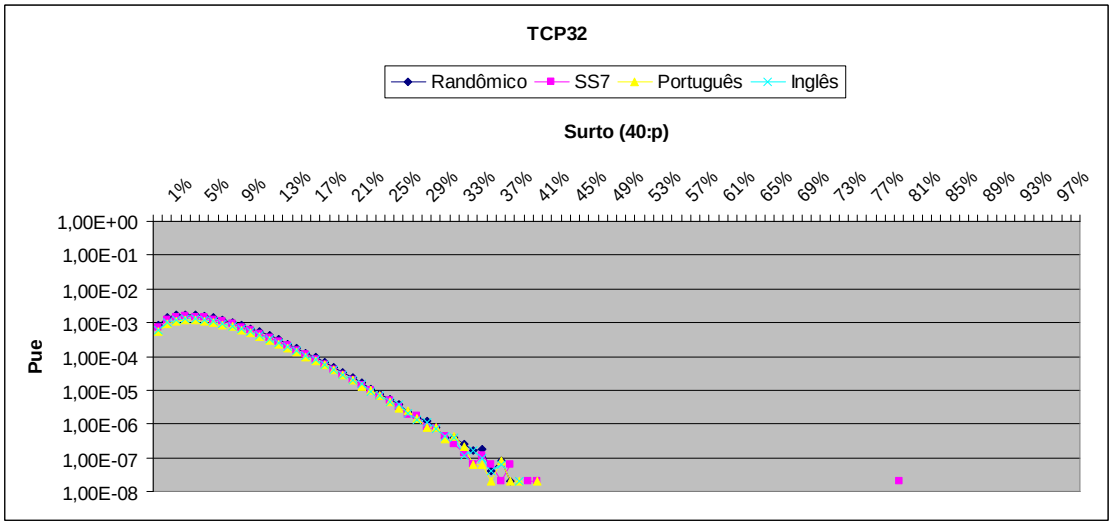


(b)

Figura 25 – TCP32; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

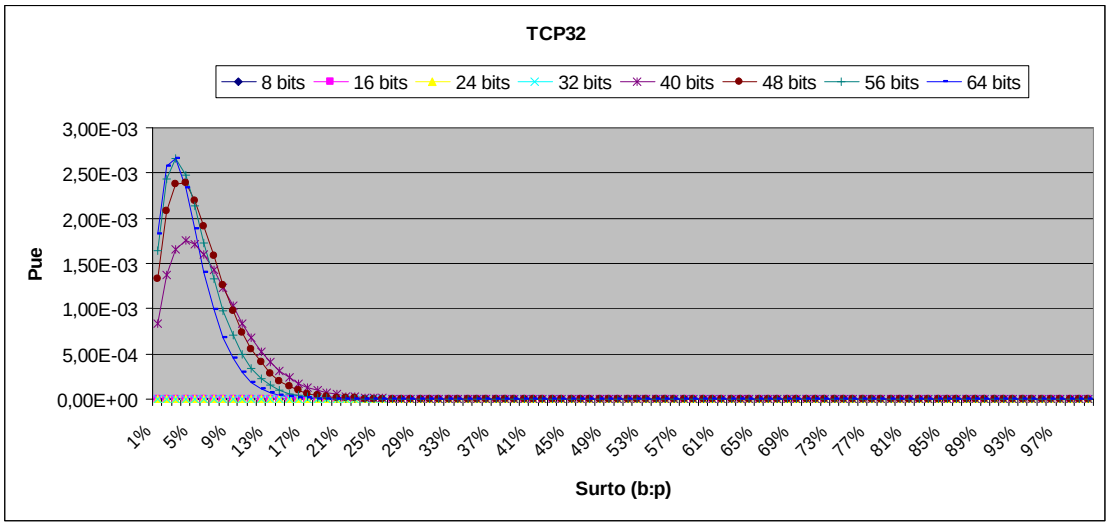


(a)

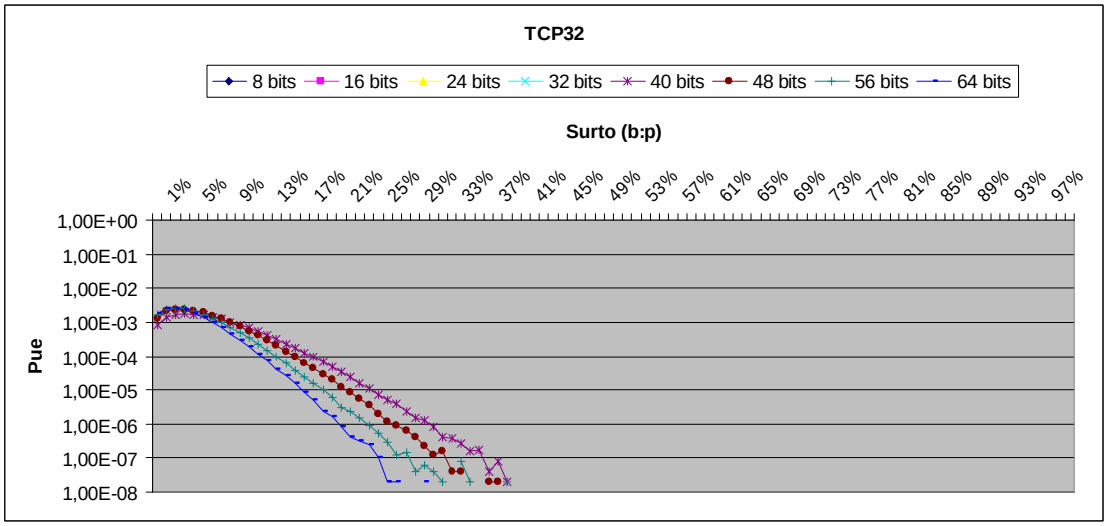


(b)

Figura 26 – TCP32; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.



(a)



(b)

Figura 27 – TCP32; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

4.3. FLETCHER

Em geral os códigos Fletcher possuem altas taxas de falhas em padrões de erros que alteram poucos bits, mas não chegam aos níveis do TCP16 e TCP32.

Um dos fatores determinantes das características do ***checksum*** de Fletcher foi a escolha de ***M***, ou seja, a aplicação da aritmética de complemento de um ou complemento de dois.

Desta forma podemos separar nossa análise em dois grupos, um do complemento de um, do qual implementamos os códigos:

- Fletcher16 255
- Fletcher32 65535

E o grupo formado pelo conjunto de códigos que utilizam aritmética em complemento de dois:

- Fletcher16 256
- Fletcher32 65536

O grupo formado pelo complemento de um possui um desempenho 10 vezes melhor, em relação ao complemento de dois, na máxima P_{ue} com o modelo de surto (Figura 30, 34), mas não há garantia de detecção de surtos relativamente pequenos. Por exemplo, o Fletcher16 255 só detecta todos os surtos até o comprimento de 7 bits (Figura 28), o pior resultado desta característica de detecção entre os códigos estudados. E como observado no *Internet Checksum*, este grupo de códigos é influenciado pelo tipo de dados sendo a diferença de 50% entre o pior e melhor caso.

Outro comportamento que também encontramos neste grupo de códigos foi os dos casos em que certas falhas em detectar erros necessitavam de uma condição inicial, o resultado é que os tipos de dados que apresentavam um maior número destas condições como as palavras $\texttt{ff}_{\text{H}}$ e $\texttt{00}_{\text{H}}$, para o Fletcher16 255 e $\texttt{ffff}_{\text{H}}$ e $\texttt{0000}_{\text{H}}$ para o Fletcher32 65535, também possuíam as maiores taxas de falhas. Por exemplo, as taxas

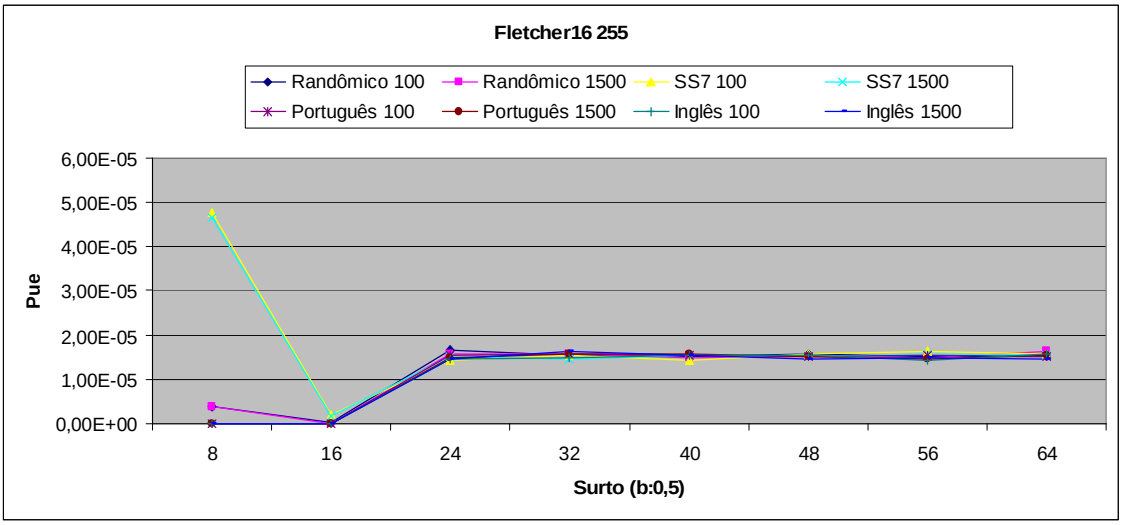
de falhas para o surto (8:0,5) e surto (16:0,5) da Figura 28, são maiores quando utilizamos os dados que possuíam mensagens SS7.

O surto (8:1,0) aplicado no Fletcher16 255 (Figura 31) apresentou um alta taxa de falhas, neste ponto calculamos a P_{ue} teórica. Para que esse erro ocorra é necessário que no fluxo de dados esteja presente a seqüência $\texttt{ff}_H$ ou $\texttt{00}_H$ alinhadas em byte e após o erro ocorrer as palavras sejam alteradas da mesma forma que o *Internet Checksum*, então a P_{ue} do surto (8:1,0) = $(2*(1/2^8))/8 = 9{,}77\text{x}10^{-4}$, encontramos em nossa simulação o valor de $9{,}74\text{x}10^{-4}$, uma diferença de 0,3%.

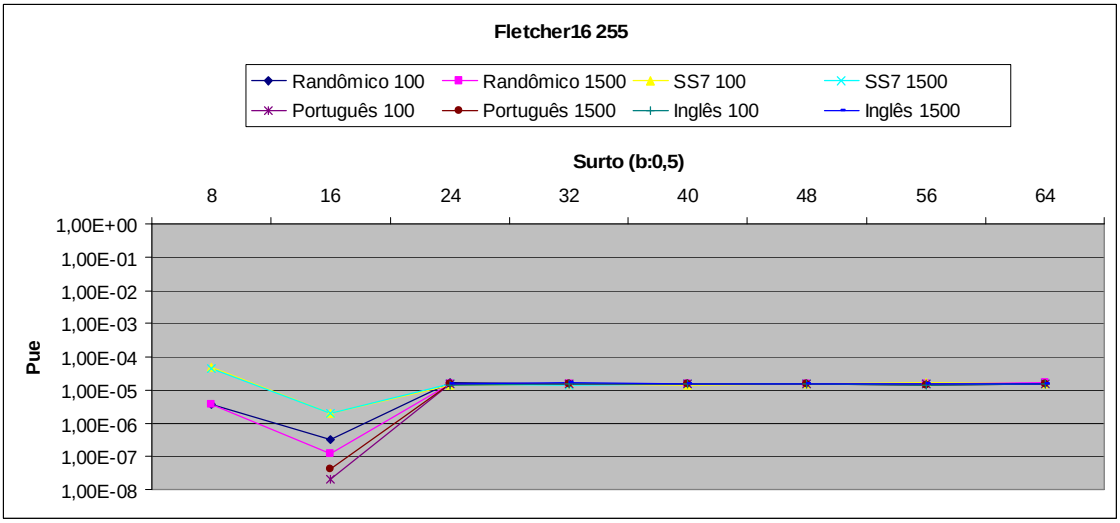
No Fletcher16 255 também notamos a alteração da distância pela alteração do tamanho das mensagens. Como a distância mínima deste código é 3 até 2.040 bits, quando codificamos mensagens de 100 bytes todos os erros de 2 bits foram detectados, mas nas mensagens de 1.500 bytes houve uma alta taxa de falhas pela alteração de 2 bits (Figura 29a). No Fletcher 32 65535 não verificamos esta situação, pois a alteração da distância só ocorre quando são utilizadas palavras código maiores que 524.280 bits, um valor superior aos geralmente encontrados em redes de computadores.

No Fletcher32 65535 o tamanho das mensagens influenciou na capacidade de detecção (Figura 36). Houve uma taxa 10 vezes pior quando utilizamos as mensagens de 100 bytes em relação às de 1.500 bytes.

Nos códigos que utilizam complemento de dois a alteração do comprimento da palavra código foi bem baixa (em torno de 5% da taxa de falhas), já a aplicação dos diferentes tipos de dados não resultou em uma alteração significativa nos resultados.

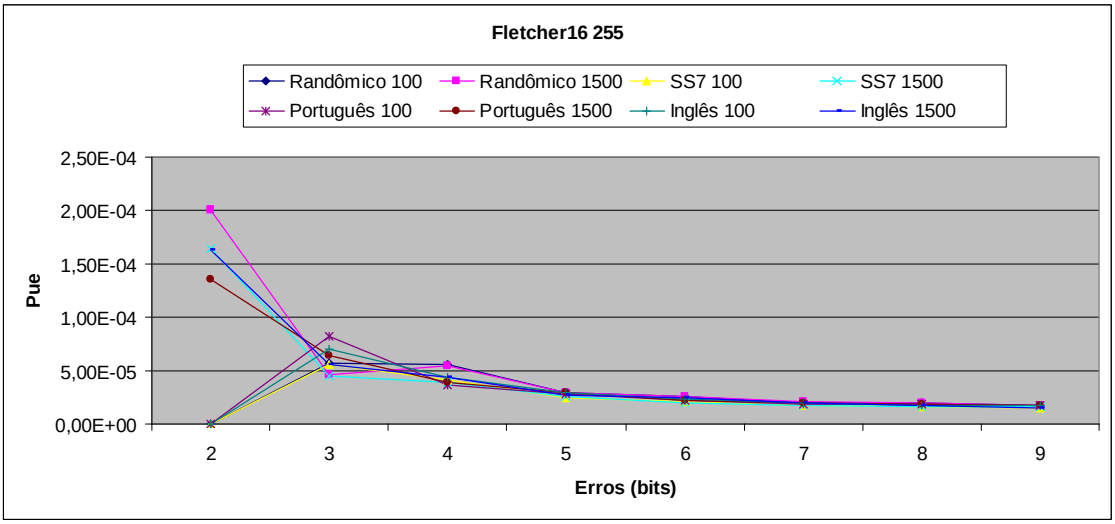


(a)

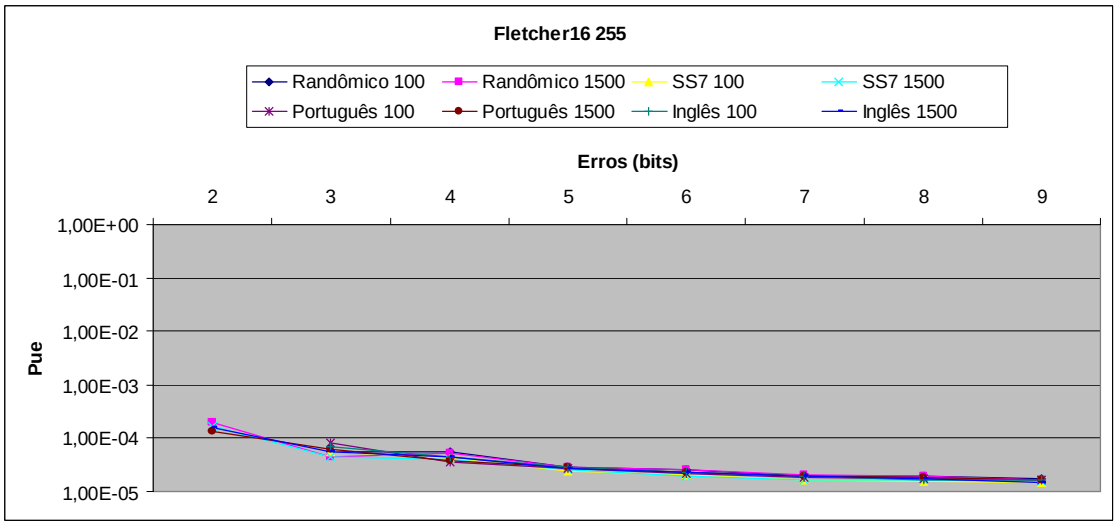


(b)

Figura 28 – Fletcher16 255; Surto (*b*:0,5), variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

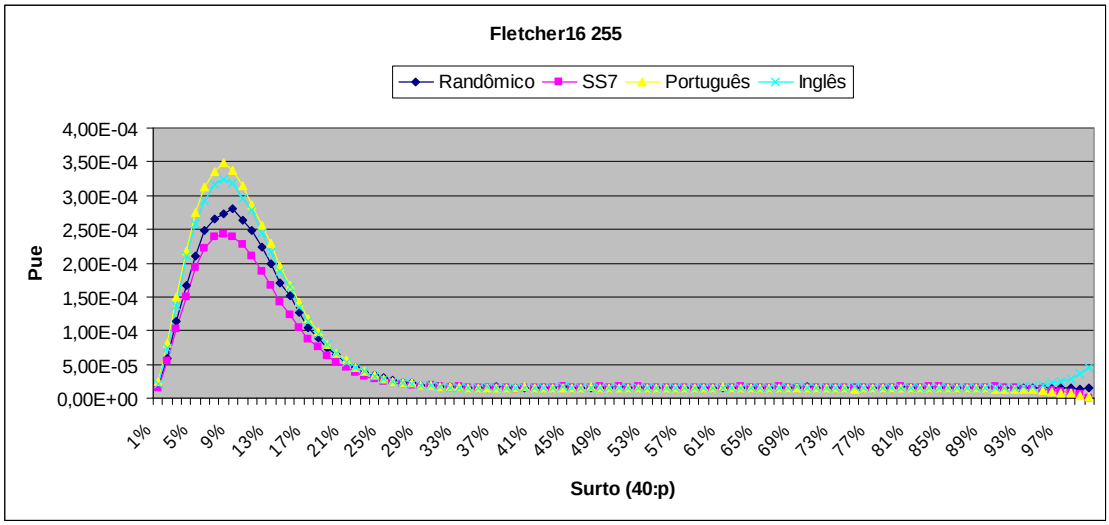


(a)

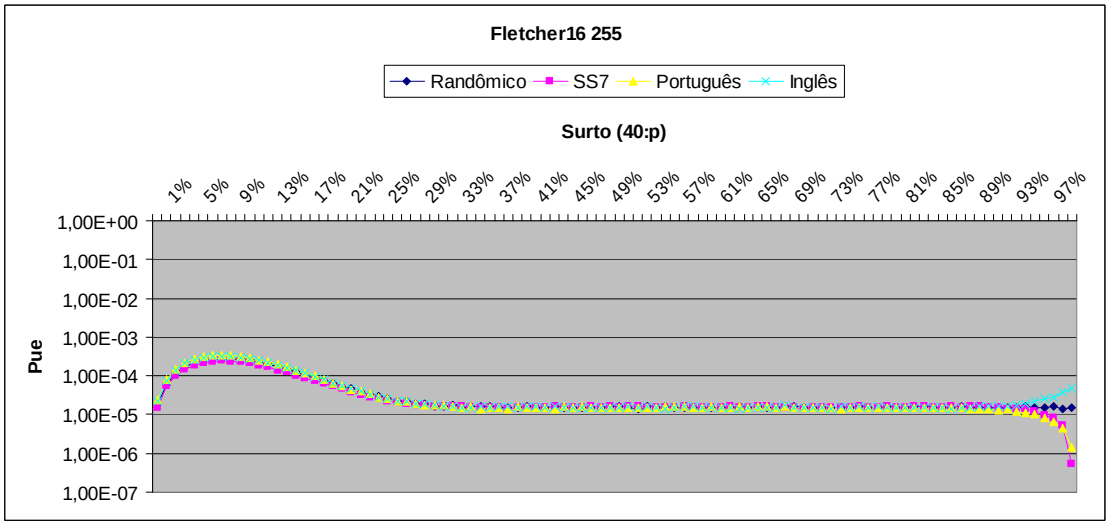


(b)

Figura 29 – Fletcher16 255; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

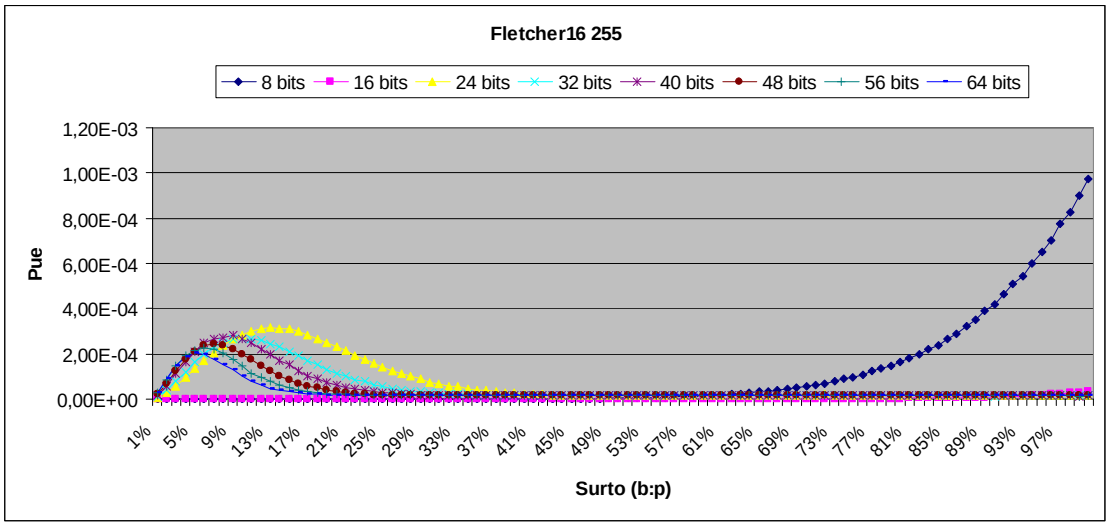


(a)

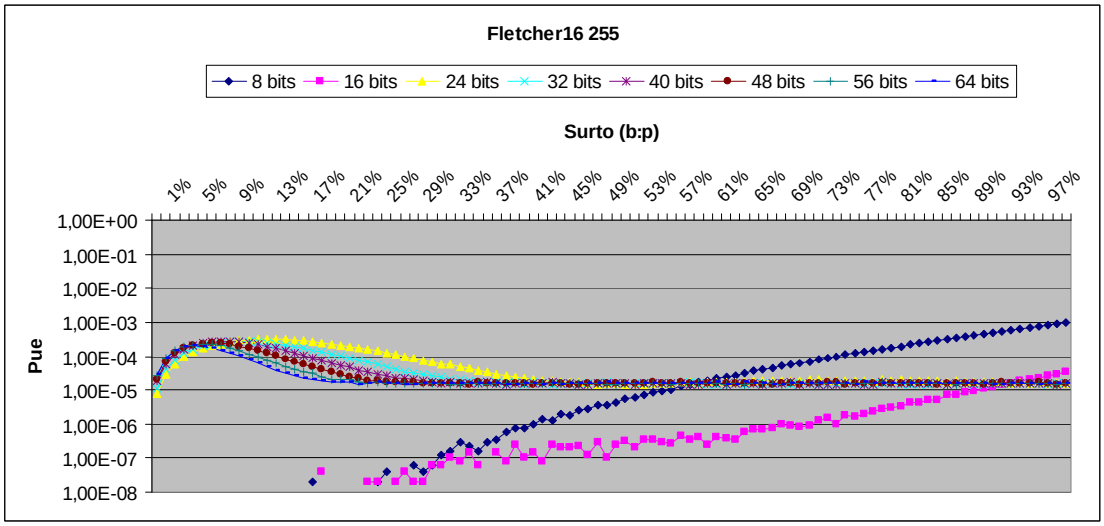


(b)

Figura 30 – Fletcher16 255; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.



(a)



(b)

Figura 31 – Fletcher16 255; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

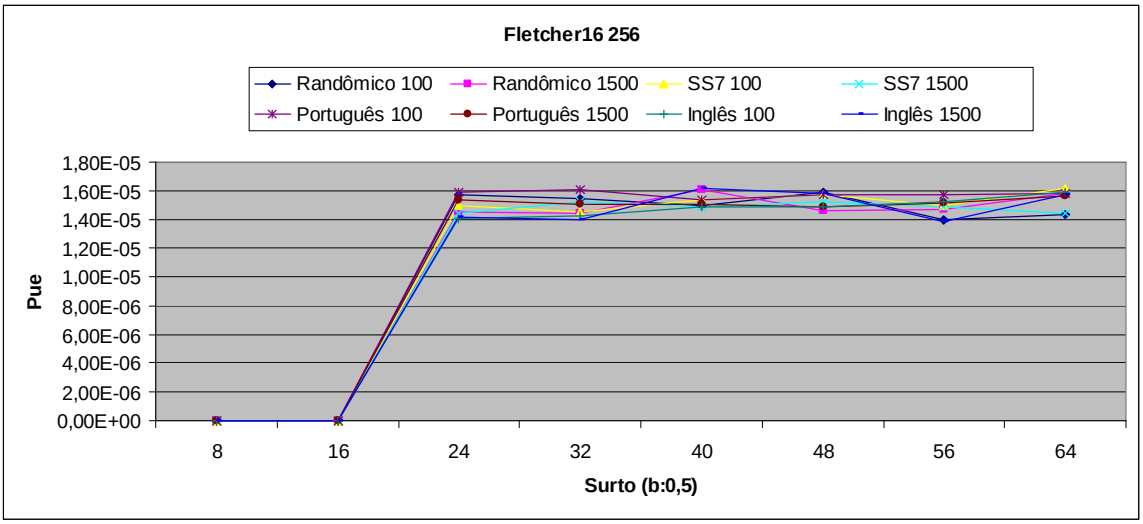
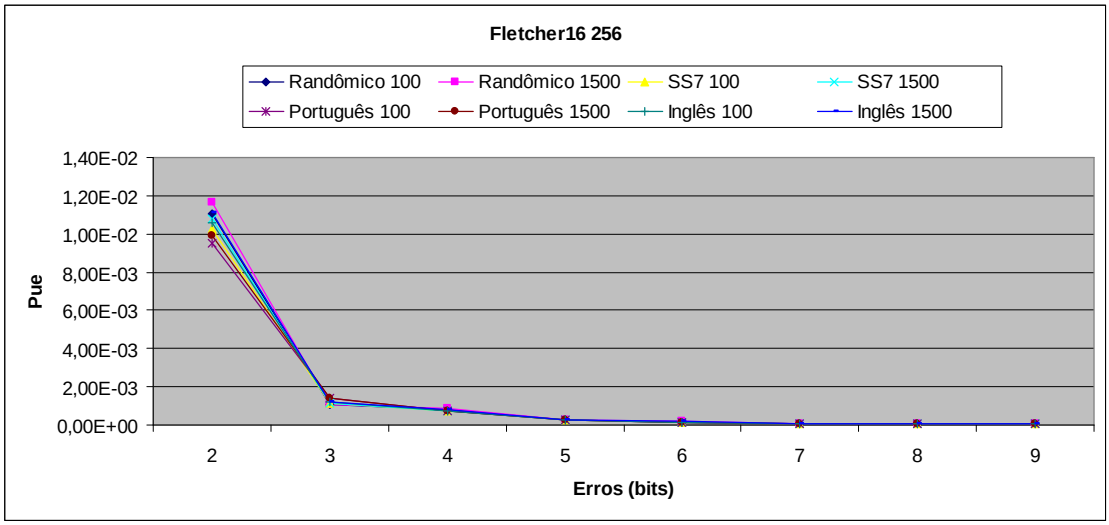
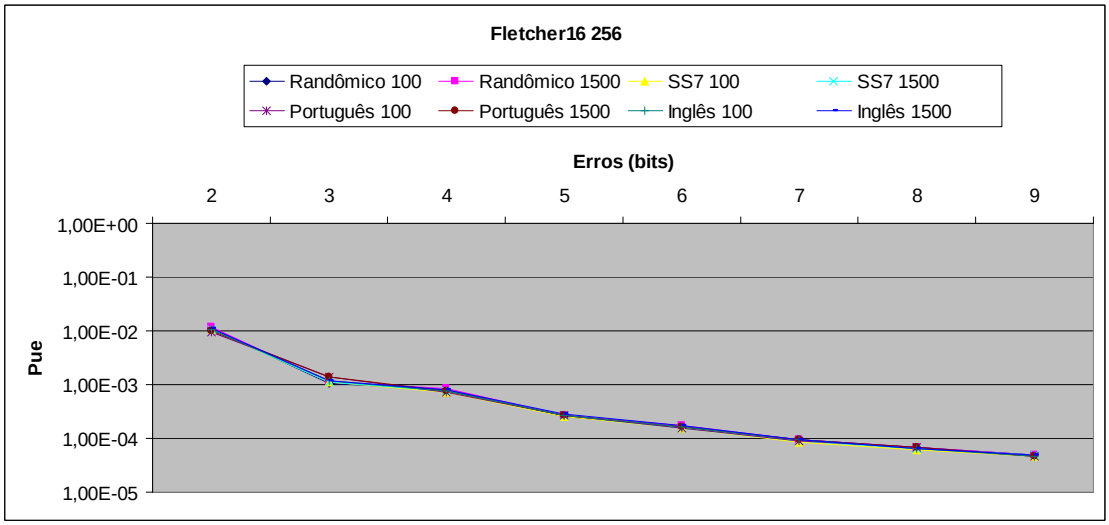


Figura 32 – Fletcher16 256; Surto (*b*:0,5), variando o tipo de dados e o comprimento da mensagem.

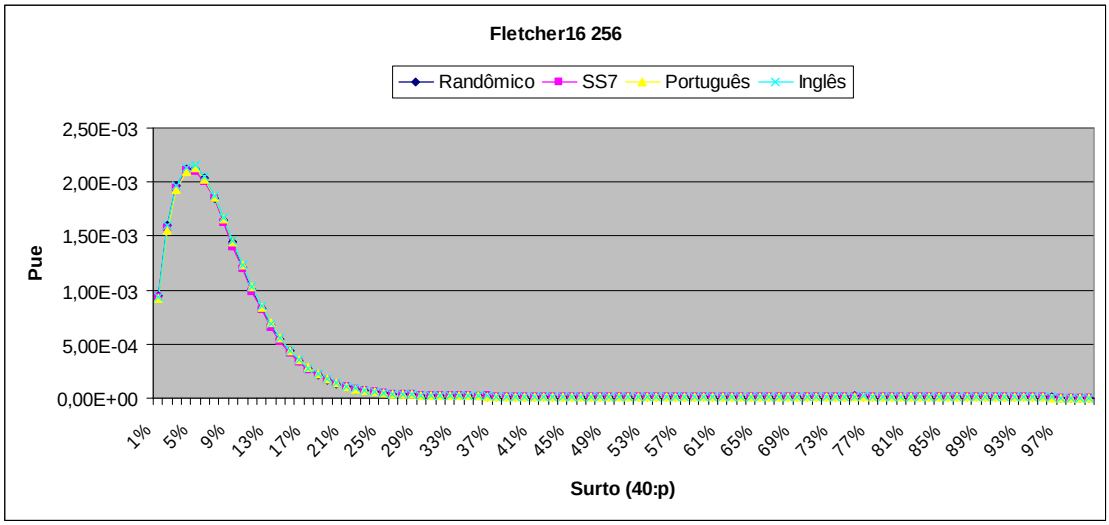


(a)

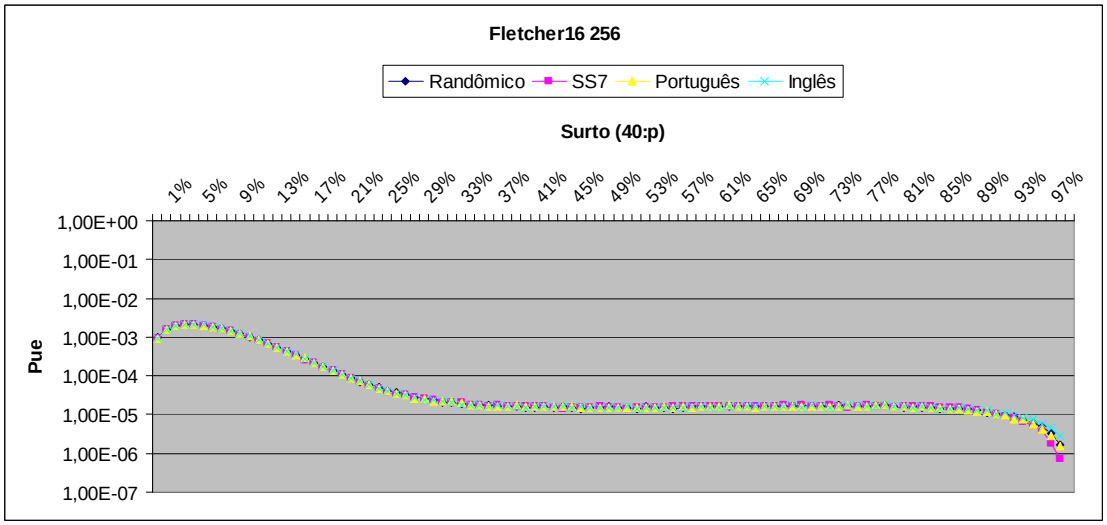


(b)

Figura 33 – Fletcher16 256; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

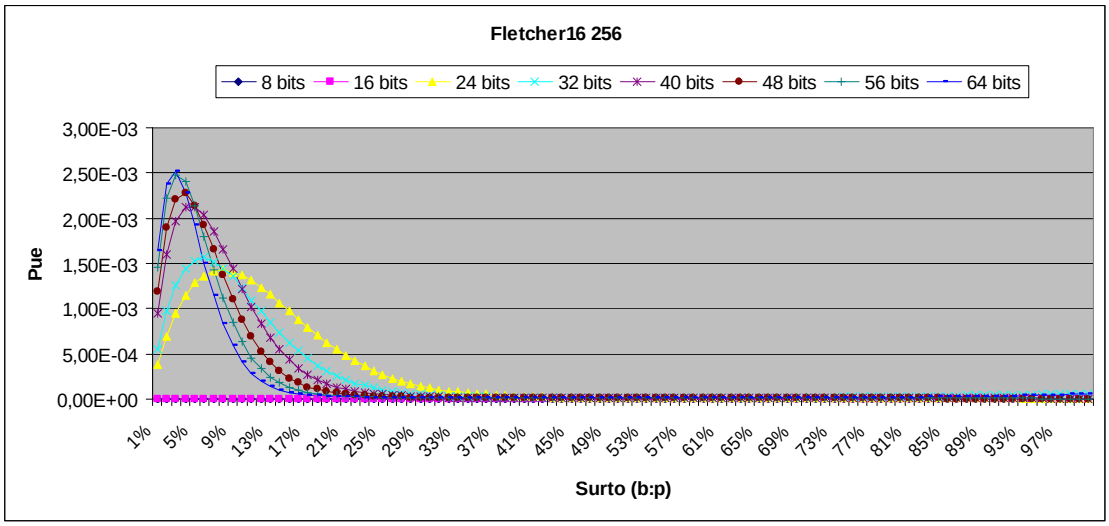


(a)

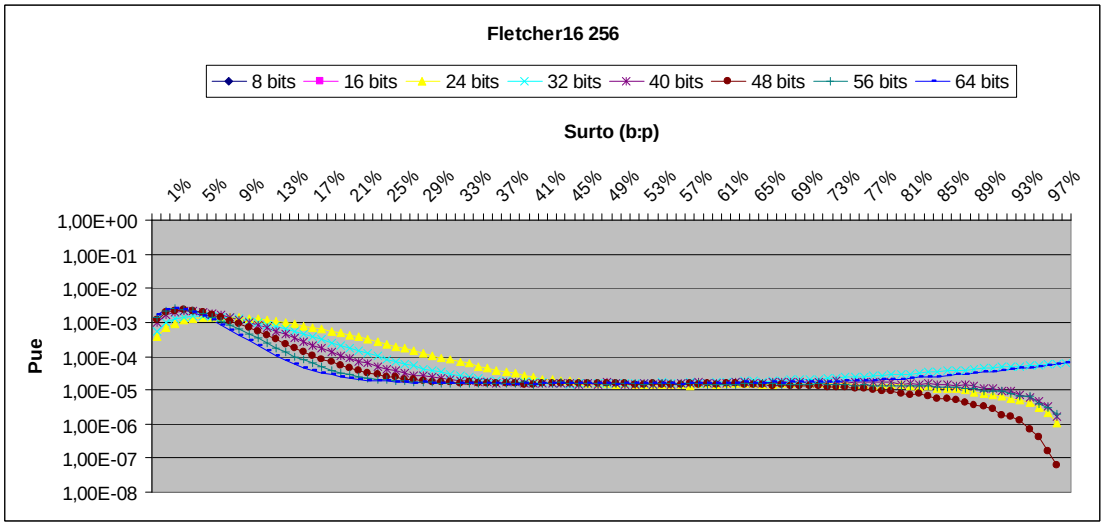


(b)

Figura 34 – Fletcher16 256; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

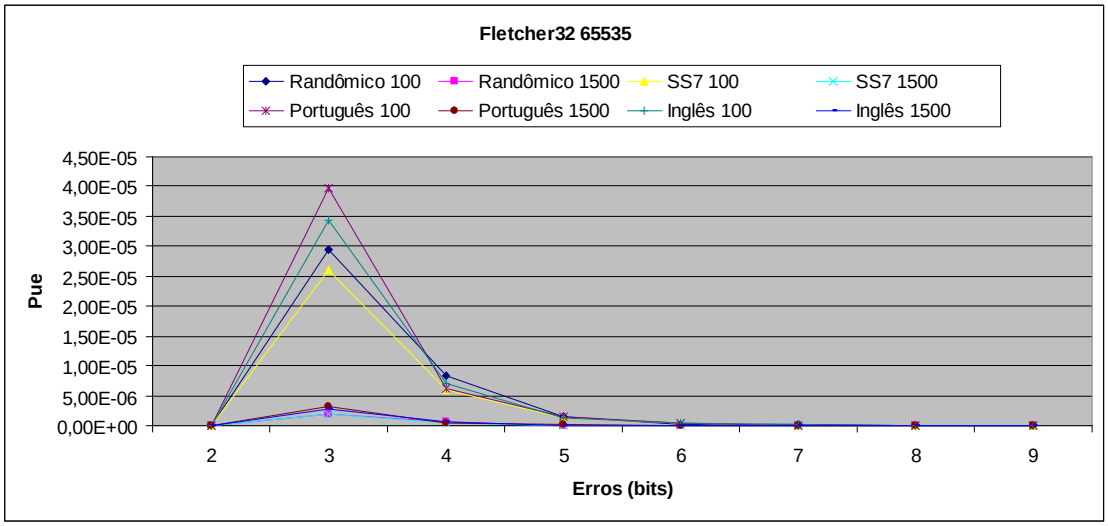


(a)

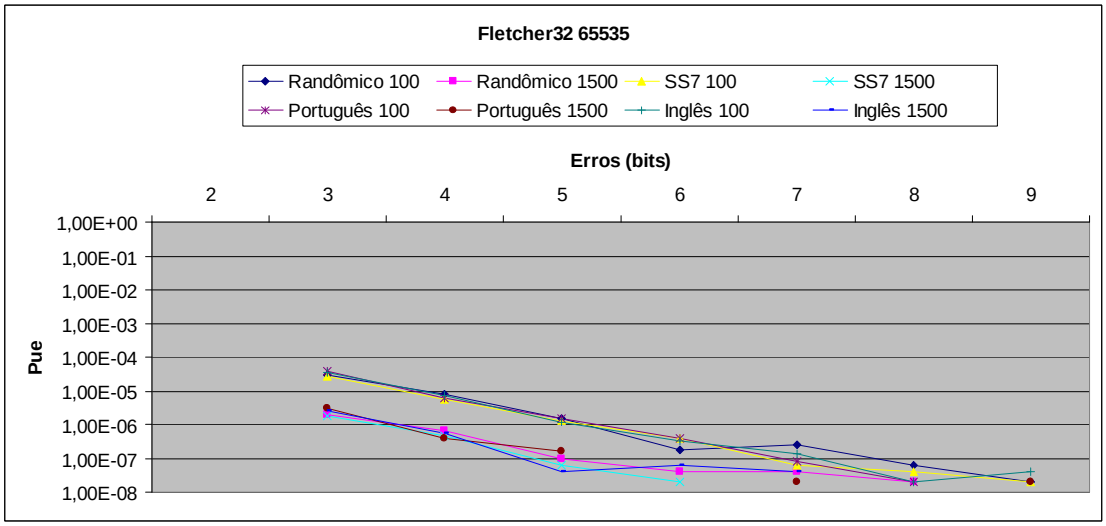


(b)

Figura 35 – Fletcher16 256; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

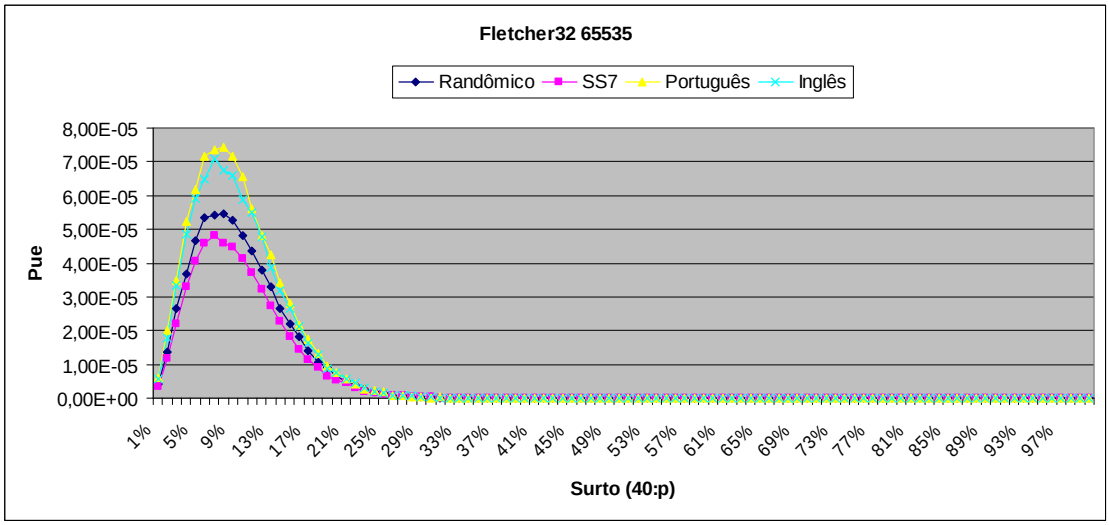


(a)

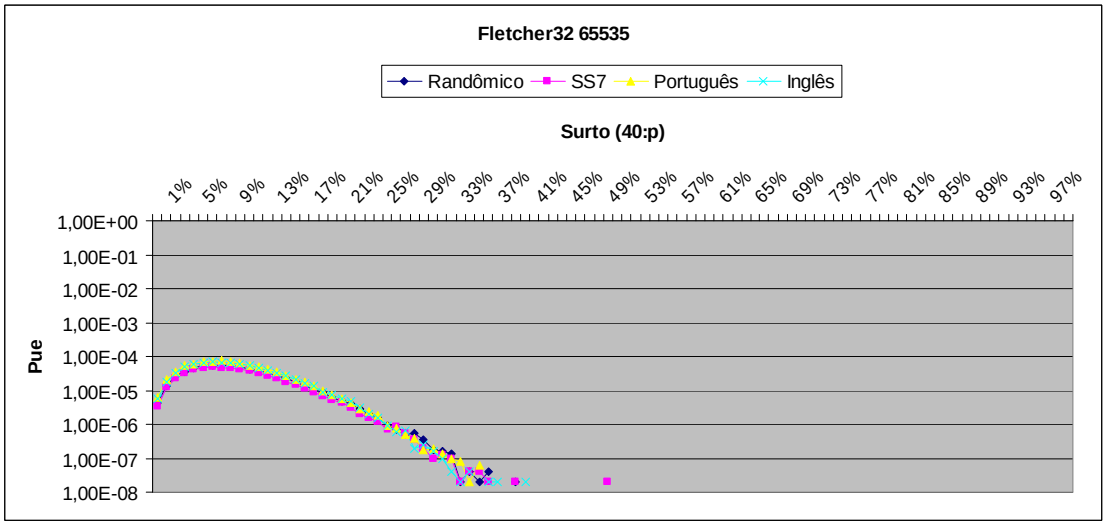


(b)

Figura 36 – Fletcher32 65535; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

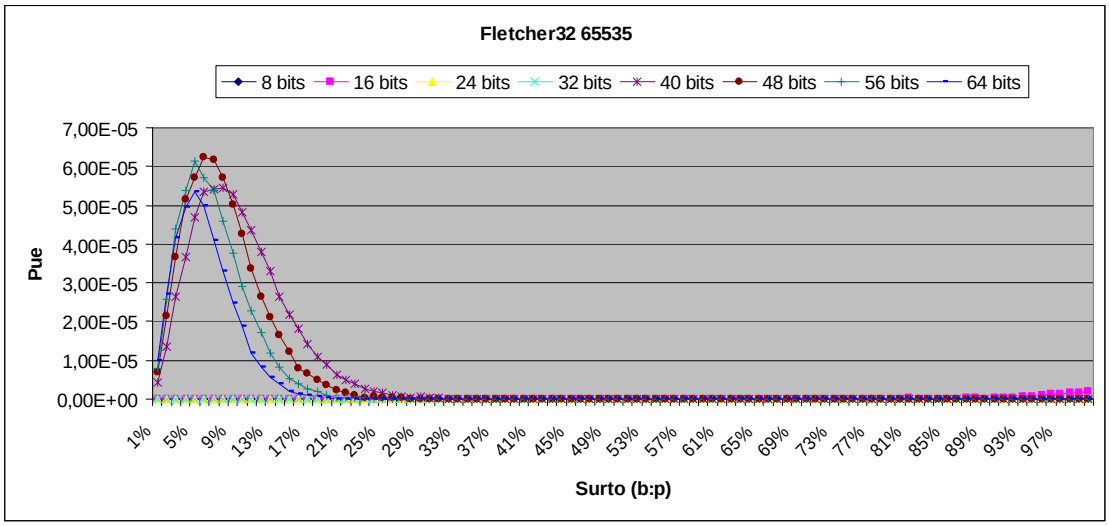


(a)

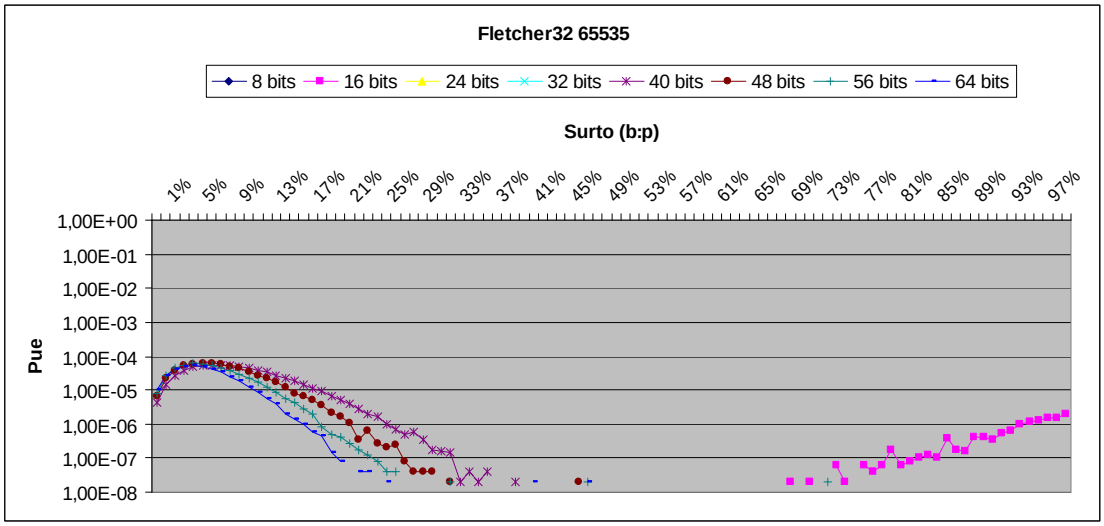


(b)

Figura 37 – Fletcher32 65535; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

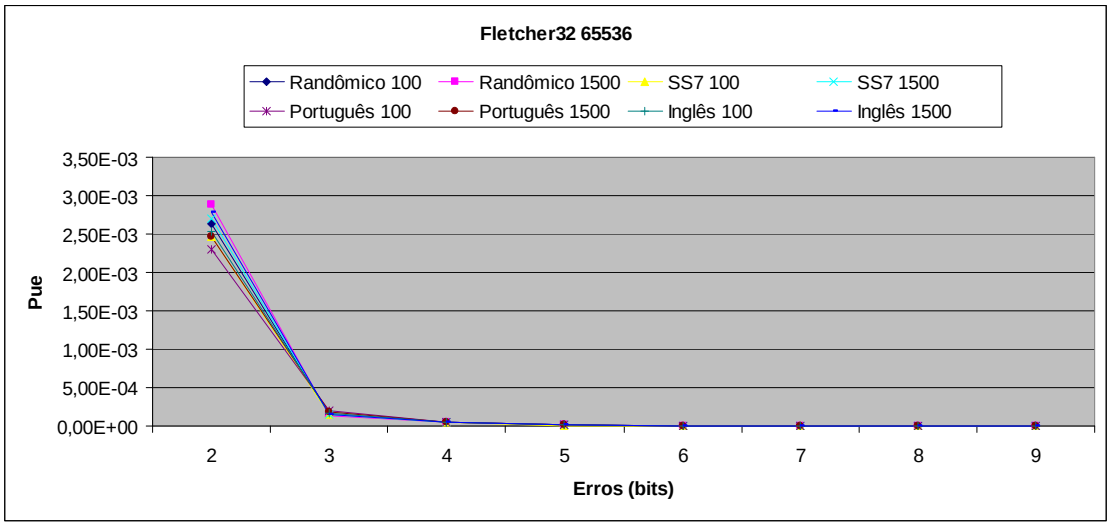


(a)

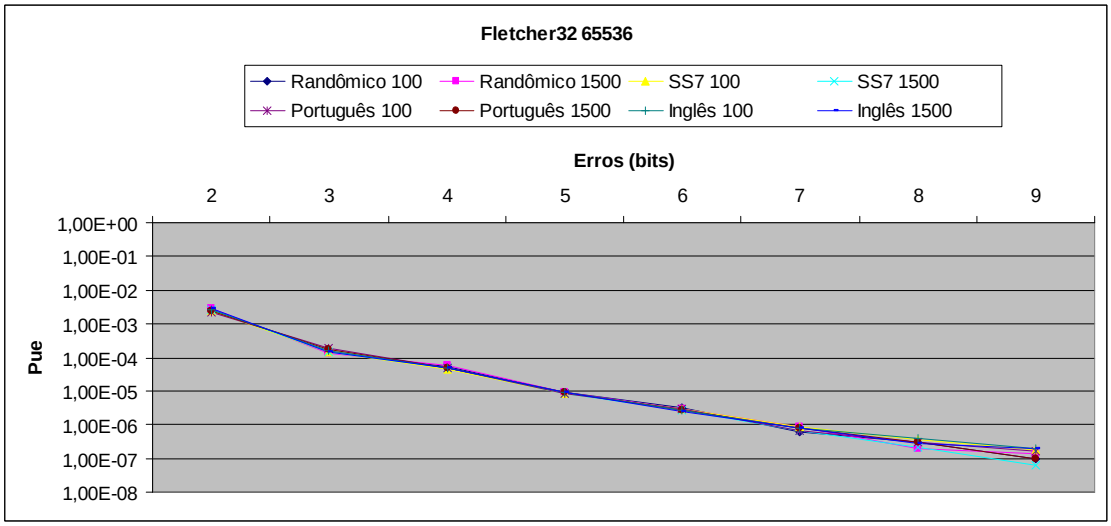


(b)

Figura 38 – Fletcher32 65535; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

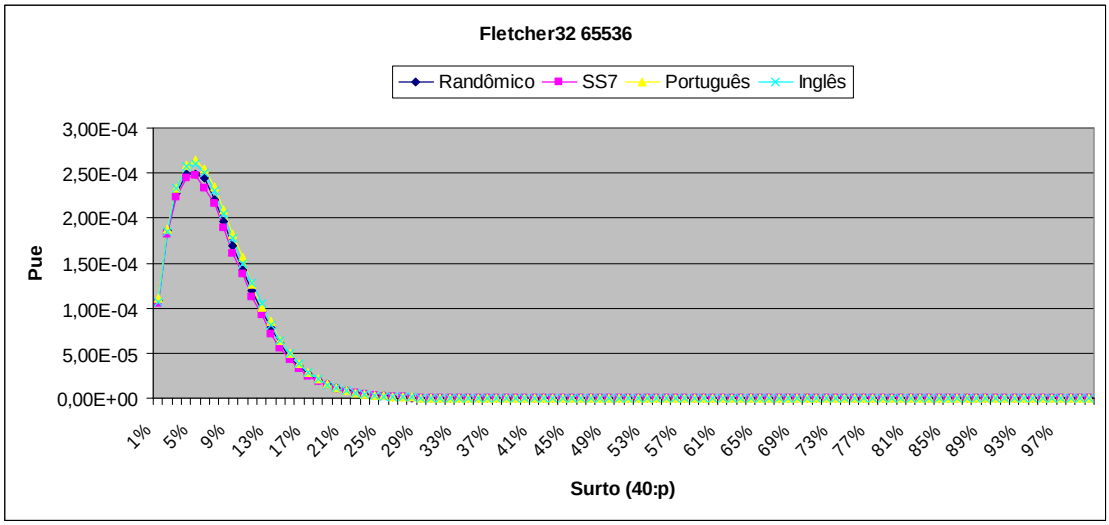


(a)

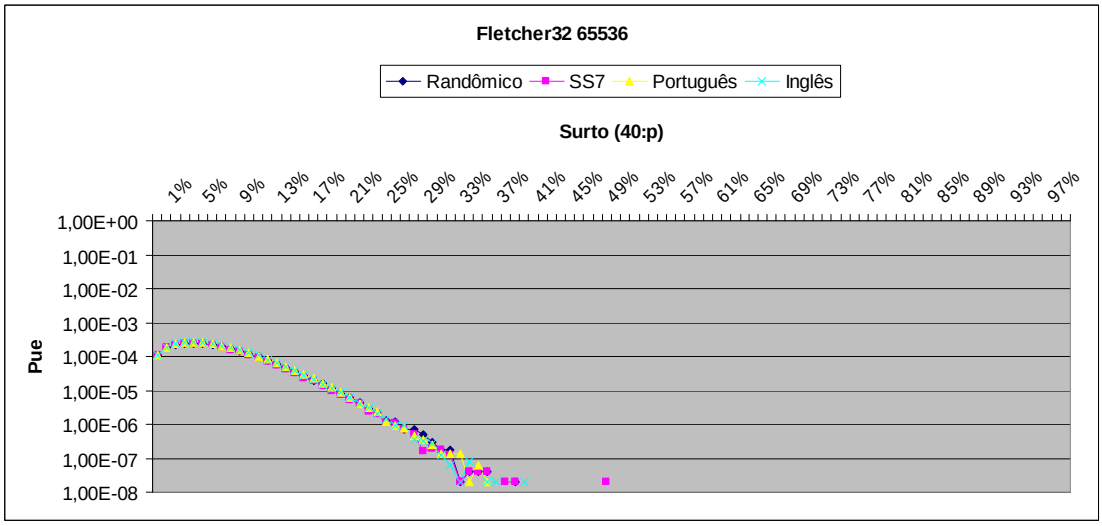


(b)

Figura 39 – Fletcher32 65536; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

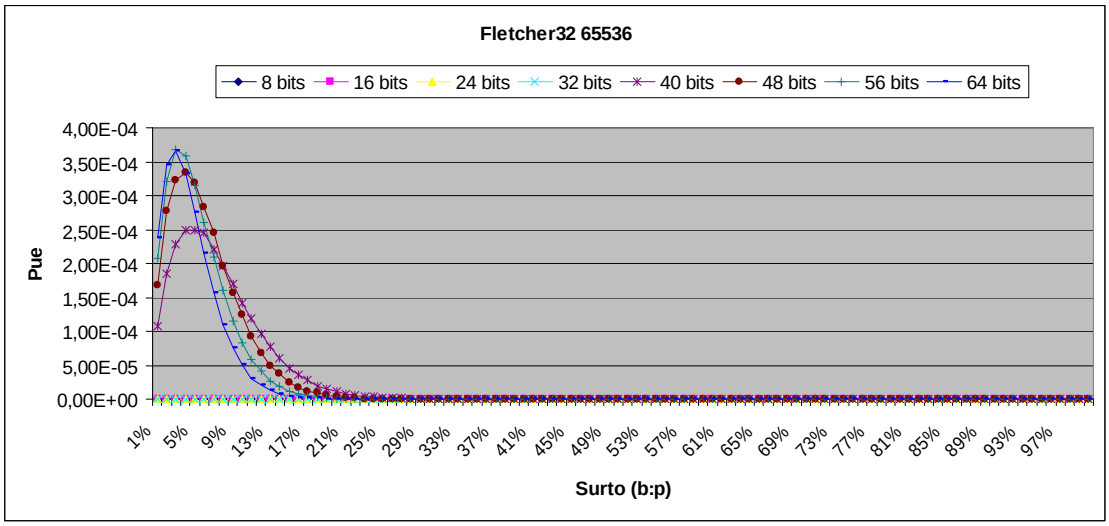


(a)

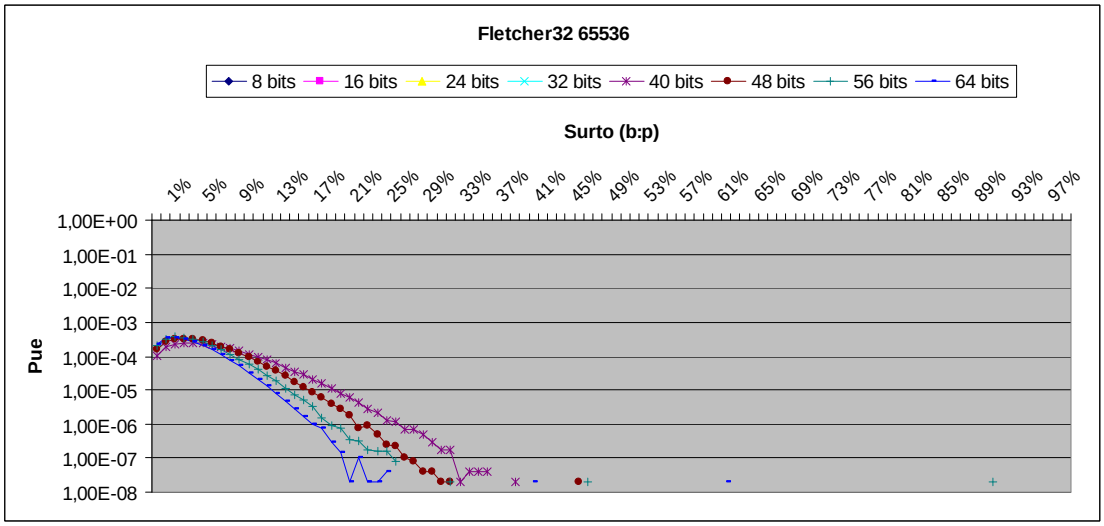


(b)

Figura 40 – Fletcher32 65536; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.



(a)



(b)

Figura 41 – Fletcher32 65536; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

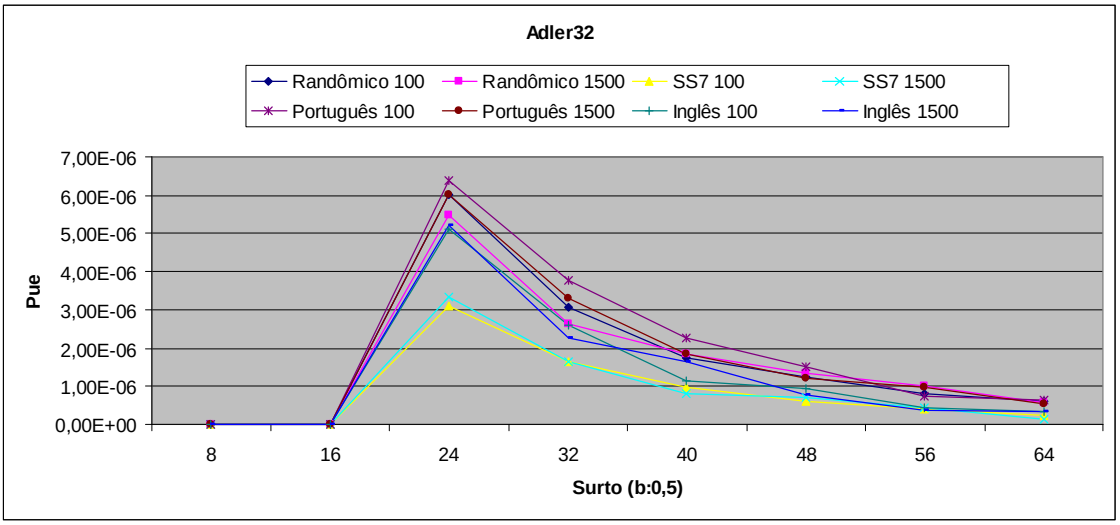
4.4. ADLER

Os resultados do Adler32 foram parecidos com os do Fletcher32 65535, ambos são susceptíveis ao comprimento das mensagens e ao tipo de dados empregados (40% de diferença entre o melhor e o pior caso).

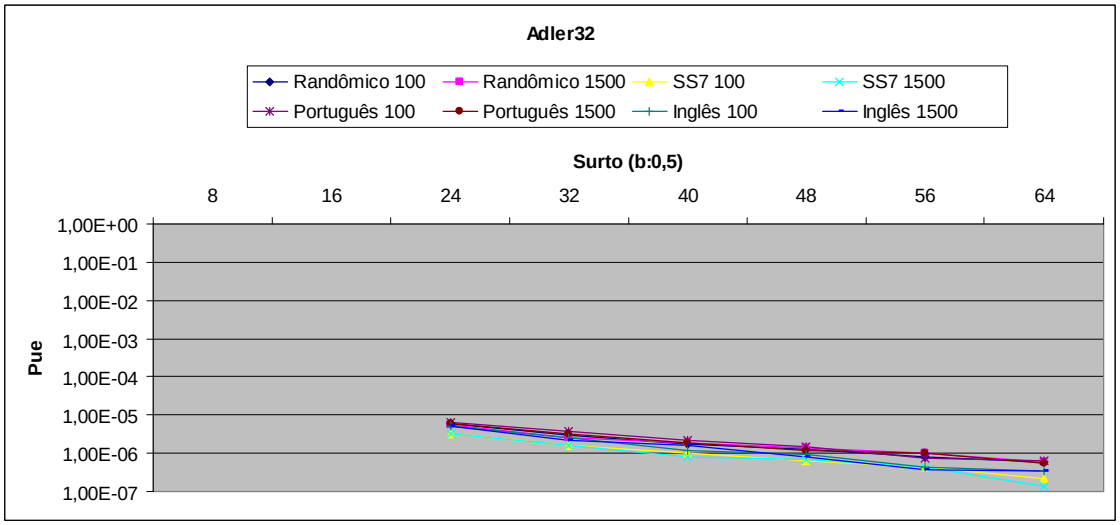
Em relação ao comprimento da mensagem o emprego de mensagens de 1.500 bytes no modelo de erros aleatórios resultou em taxas de falhas de 10 a 20 vezes melhores que o de mensagens de 100 bytes (Figura 43). No modelo de surto este valor foi de 20% a 80% melhor (Figura 42).

O padrão de erro que afeta 24 bits, citado por Sheinwald [SHW01] resultou nas maiores taxas de falhas do Adler32 (Figura 42, 45), mesmo assim não chegou aos piores níveis do TCP32.

Na faixa em que os códigos anteriores chegaram ao nível teórico de $1/2^r$, a P_{ue} do Adler32 ficou em torno de 10^{-5} a 10^{-6}, como se estivéssemos utilizando um código que gera de 14 a 19 bits redundantes (Figura 44, 45). Desta forma encontramos as taxas de falhas de detecção deste código em unidades de transporte típicas, como propusemos no capítulo 1.

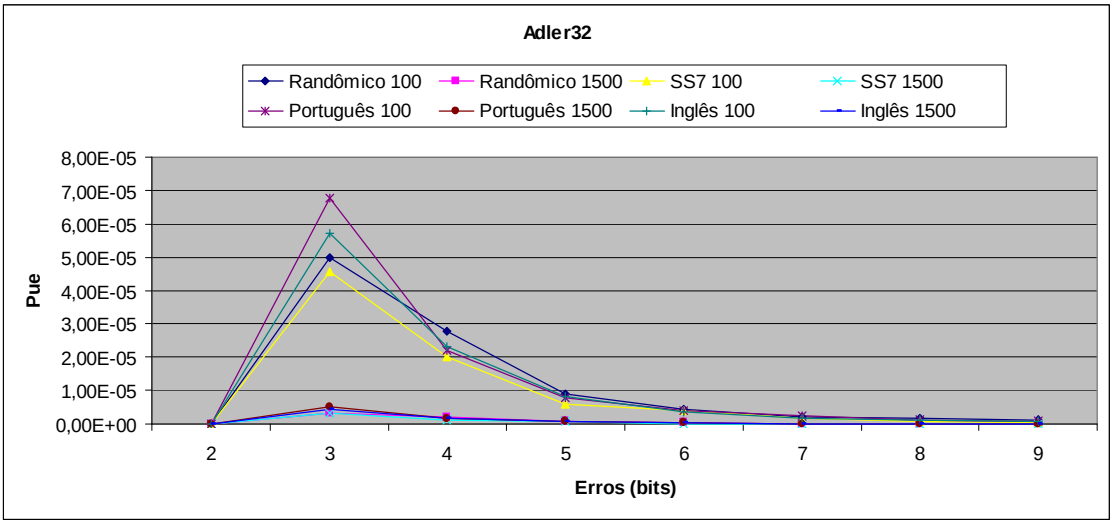


(a)

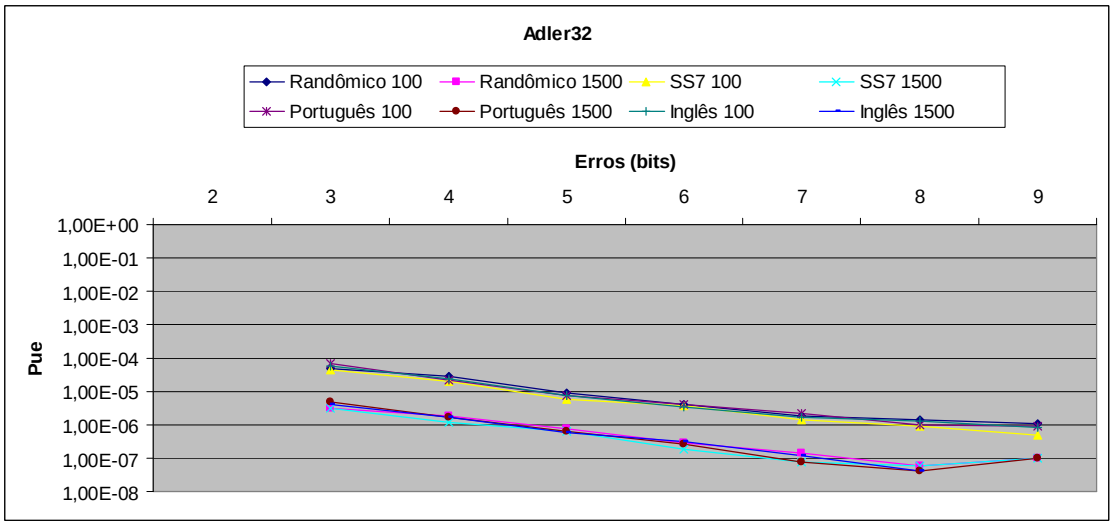


(b)

Figura 42 – Adler32; Surto (*b*:0,5), variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

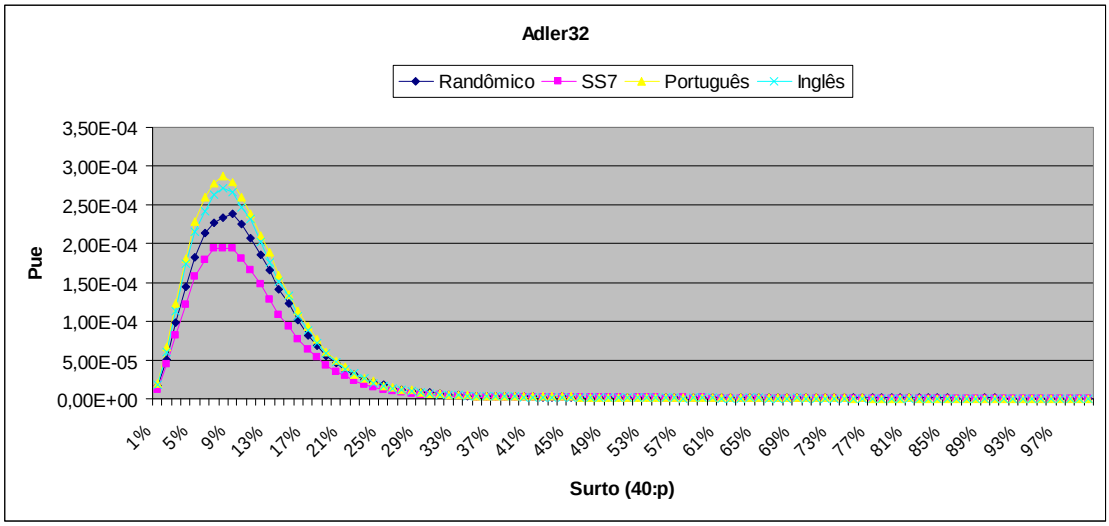


(a)

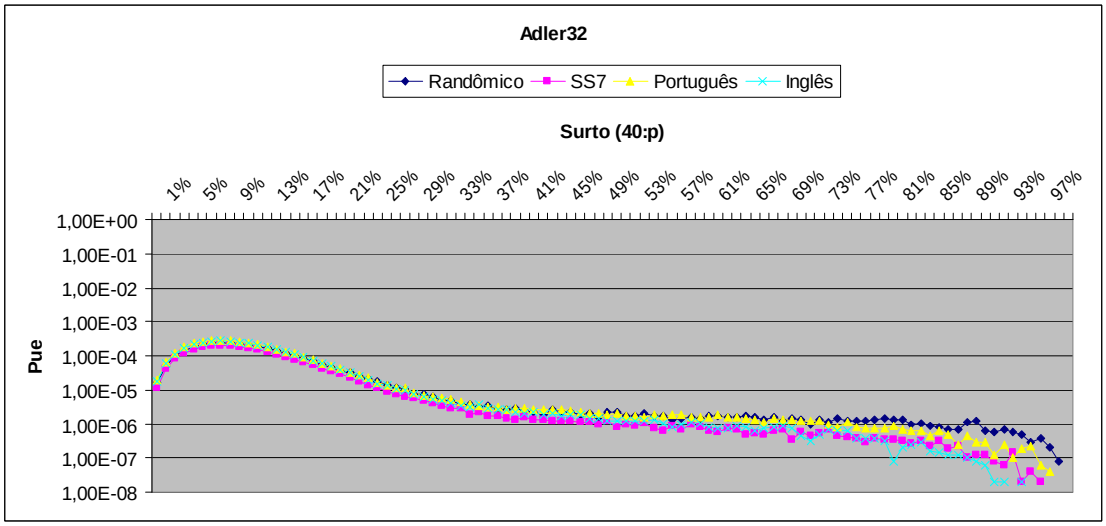


(b)

Figura 43 – Adler32; Erros aleatórios, variando o tipo de dados e o comprimento da mensagem. (a) Escala linear; (b) Escala logarítmica.

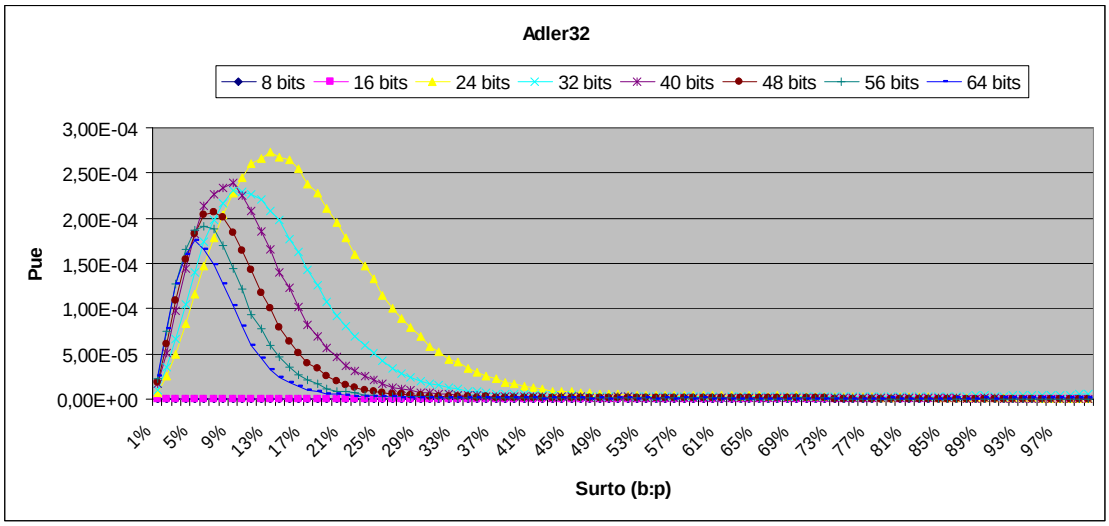


(a)

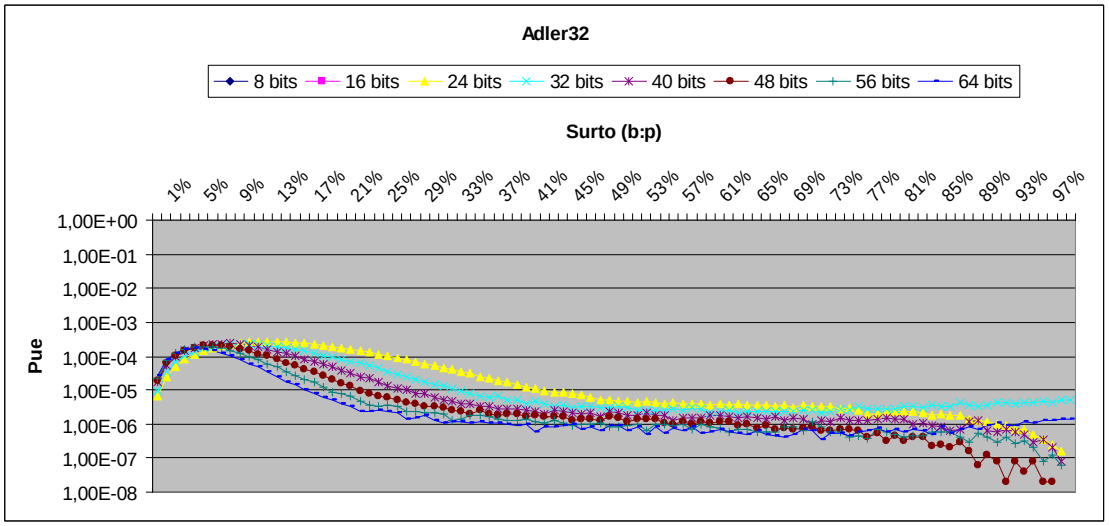


(b)

Figura 44 – Adler32; Surto (40:*p*), variando o tipo de dados e a probabilidade do surto (Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.



(a)



(b)

Figura 45 – Adler32; Surto (*b*:*p*), variando o comprimento e a probabilidade do surto (Dados: Randômicos; Comprimento da mensagem: 100 bytes). (a) Escala linear; (b) Escala logarítmica.

4.5. CONCLUSÃO

Neste capítulo mostramos os resultados de nossa simulação, e traçamos os gráficos dos testes realizados nos códigos analisados. Para termos uma visão da capacidade de detecção de surto dos códigos, reunimos na Figura 46 o gráfico do modelo de surto (40:*p*). Nela podemos notar que todos os códigos tendem ao valor teórico esperado de $1/2^r$, quando *p* se aproxima de 0,5, menos o Adler32. E na região que $p < 0,2$ onde a maioria dos códigos apresentam uma alta taxa de falhas, mesmo utilizando 32 bits de redundância o TCP32 só possui um desempenho melhor do que o TCP16 e o Fletcher16 256.

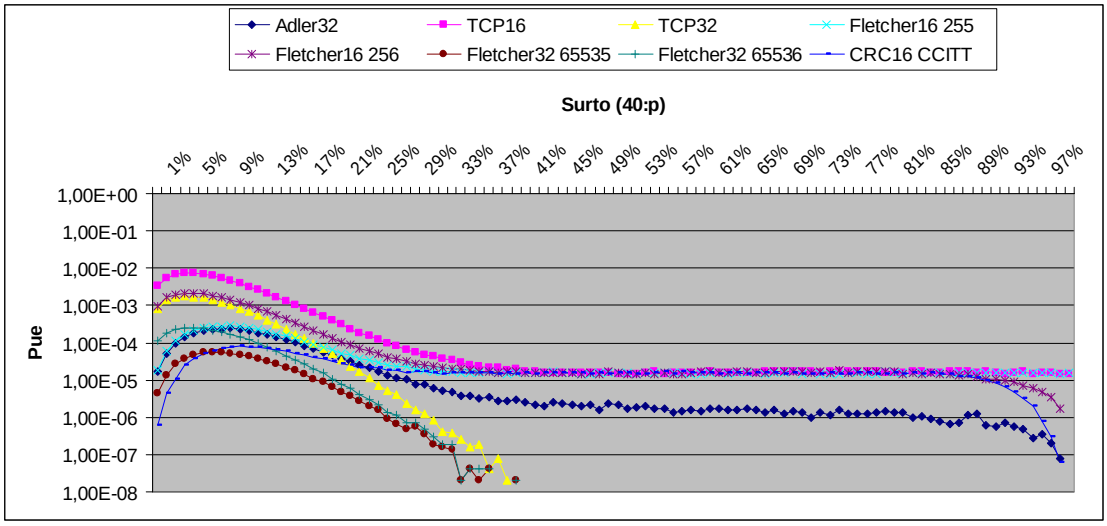


Figura 46 – Comparação das capacidade de detecção dos códigos estudados, pelo emprego de surto(40:*p*). (Dados: Randômicos; Comprimento da mensagem: 100 bytes)

Apesar de termos encontrado altas taxas de falhas para o TCP16, verificamos que isso foi devido ao modelo de erro e não da utilização de dados não uniformemente distribuídos como citado por Stone [STN98].

Sendo assim, analisamos o modelo de erro utilizado por Stone e verificamos que ele pode ser simplificado em alguns casos a um surto que atinge 48 bytes, desta forma simulamos um surto (384:*p*) (Figura 47).

No caso do TCP16, somente a utilização de valores baixos de *p* (menores que 0,07), resultavam em uma taxa alta de falhas, em todo o restante do gráfico a P_{ue} se manteve em seu valor teórico.

Estes valores de *p* poderiam ter ocorrido na prática, pois o modelo de Stone substituía um conjunto de 48 bytes por outro, e se estes dois blocos fossem dados contínuos de um arquivo não uniformemente distribuído, a probabilidade de haver vários bytes iguais nas mesmas posições seriam altas.

Neste caso, dados não uniformemente distribuídos poderiam causar modelos de erros com uma baixa probabilidade *p* de transição dos bits, o que na maioria dos códigos aqui estudados resultaria em altas taxas de falhas.

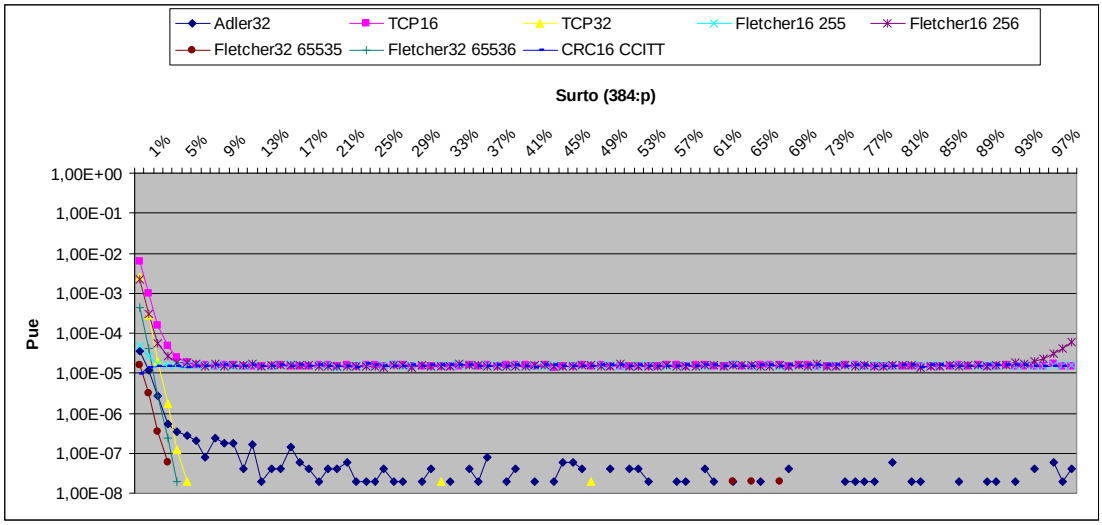


Figura 47 – Verificação do comportamento dos códigos estudados em surtos de grande comprimento; surto (384:*p*). (Dados: Randômicos; Comprimento da mensagem: 100 bytes)

5. CONCLUSÕES E TRABALHOS FUTUROS

Durante o desenvolvimento de nosso trabalho procuramos implementar um simulador que pudesse ser efetivo no teste de códigos detectores de erro em situações que fossem próximas às encontradas em uma rede de computadores. Através do desenvolvimento do simulador e dos resultados das simulações podemos citar algumas conclusões baseadas nos capítulos anteriores.

O principal fator que influenciou os resultados foi o modelo de erro empregado. É interessante notar que os protocolos de transporte citados no Capítulo 1 são utilizados em conjunto com o protocolo de rede IP, e como um pacote de dados trafegando, neste caso pela Internet, pode passar por enlaces sem fio, fibras óticas e cabos de par trançado, além de roteadores e *middle-boxes* (NATs e *Proxies*) [STN00], a modelagem de um padrão de erro para esta situação se torna difícil. Então o código que minimize a taxa de erro residual nesse caso poderia ser o que apresentasse bons resultados na maioria dos testes, como opção.

Em nosso trabalho os CRCs obtiveram os melhores resultados de taxa de erro residual, portanto são ótimos candidatos a serem empregados na camada de transporte. Um dos problemas enfrentados pelos CRCs no entanto, é que em software sua codificação ainda é lenta comparada aos *checksums* (Apêndice E).

Outra característica analisada dos CRCs foi sua capacidade de detecção de erros de peso ímpares. No CRC16 CCITT observamos que apesar de serem detectados todos os erros de peso ímpares, a taxa de falha de detecção dos erros de peso par era o dobro do esperado teoricamente. Se os pesos dos erros forem distribuídos igualmente no canal, essa característica será vantajosa somente para aumento da distância mínima do código que passa de 3 para 4.

O tipo de dado empregado em nossos testes somente influenciou os resultados dos códigos não lineares, mas de forma menos significativa do que os modelos de erro utilizado neste trabalho. O tipo de dado que levou a maior taxa de falhas no *Internet Checksum* foi o uniformemente distribuído. Vale ressaltar que várias fontes citam que este tipo de dado levaria a uma taxa de falhas bem menor em relação a dados não uniformemente distribuídos.

Confirmamos o problema existente no Adler32 para pacotes em torno de 100 bytes e verificamos que ele também possui uma taxa de erros elevada mesmo quando são utilizados pacotes de 1.500 bytes. Esses resultados mostram que este código não é aconselhável para a verificação da existência de erros em pacotes de redes de computadores.

5.1. RECOMENDAÇÕES PARA TRABALHOS FUTUROS

Em um primeiro momento sugerimos a continuidade das pesquisas dos códigos CRC32B e CRC32C. Como não traçamos os gráficos desses códigos devido aos nossos recursos limitados, a obtenção destes resultados indicaria o melhor polinômio a ser utilizado em futuros protocolos, sob o ponto de vista dos modelos de erros implementados no simulador.

Sugere-se também a realização dos testes deste trabalho na família de códigos chamada *Weighted Sum Codes* [MCA94]. Suas propriedades de detecção de erros são comparáveis às dos CRCs e suas implementações são eficientes tanto por software quanto por hardware, apesar disso estes códigos não estão sendo utilizados em

protocolos da camada de transporte ou até mesmo nos protocolos em desenvolvimento citados na introdução de nosso trabalho.

Outro trabalho interessante seria a alteração do método de geração de números randômicos no simulador. Como verificado na comparação dos resultados deste trabalho com os de Wolf (Figura 16), houve um distanciamento dos valores a partir do ponto que correspondia ao surto (23:0,2), uma possível causa pode ser a diferença dos métodos de geração de números randômicos entre os dois trabalhos. Esse estudo poderia apontar tendências nos números gerados por esses algoritmos.

6. BIBLIOGRAFIA

[BAI00] T. Baicheva, S. Dodunekov, P. Kazakov, *Undetected error probability performance of cyclic redundancy-check codes of 16-bit redundancy*, IEEE Proceedings on Communications, 147:253-256, October 2000.

[CAV01] V. Cavanna, *et al.*, *ISCSI Digests "CRC or Checksum?"*, Internet draft, draft-cavanna-iscsi-crc-vs-chsum-01.txt. Work in progress, March 2001.

[COS98] D. Costello Jr., *et al.*, *Applications of Error-Control Coding*, IEEE Transactions on Information Theory, vol. 44, n° 6, pp. 2531-2560, October 1998.

[CST93] G. Castagnoli, S. Braeuer, M. Herrman, *Optimization of Cyclic Redundancy-Check Codes with 24 and 32 Parity Bits*, IEEE Transactions on Communications, Vol. 41, No. 6, pp. 883-892, June 1993.

[DAE02] J. Daemen, V. Rijmen, *The Design of Rijndael*, Berlin: Springer, 2002.

[FEL95] D. Feldmeier, *Fast Software Implementation of Error Detection Codes*, IEEE/ACM Transactions on Networking, Vol. 3, No. 6, December 1995.

[FLT82] J. Fletcher, *An Arithmetic Checksum for Serial Transmissions*, IEEE Transactions on Communications, Vol. COM-30, n° 1, pp. 247-252, January 1982.

[FUJ89] T. Fujiwara, T. Kasami, S. Lin, *Error Detecting Capabilities of the Shortened Hamming Codes Adopted for Error Detection in IEEE standard 802.3*, IEEE Transactions on communications, September 1989.

[HLZ91] G. Holzmann, *Design and Validation of Computers Protocols*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[HYK88] S. Haykin, *Digital Communications*, New York: John Wiley & Sons, 1988.

[IEN45] W. Plummer, *TCP Checksum Function Design*, Internet Engineering Note 45, BBN, 1978. Reimpresso na RFC 1071.

[KAY96] J. Kay, J. Pasquale, *Profiling and Reducing Processing Overheads in TCP/IP*, IEEE/ACM Transactions on Networking, Vol. 4, n^{o} 6, pp. 817-828, December 1996.

[KHL02] E. Kohler, *et al.*, *Datagram Congestion Control Protocol (DCCP)*, Internet draft, draft-kohler-dcp-03.txt. Work in progress, May 2002.

[KOD92] J. Kodis, *Fletcher's Checksum – Error detection at a fraction of the cost*, Dr. Dobb's Journal, May 1992.

[LIN83] S. Lin, D. Costello, *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice Hall, 1983.

[LRZ02] L. Larzon, *et al.*, *The UDP Lite Protocol*, Internet draft, draft-ietf-tsvwg-udp-lite-00.txt. Work in progress, January 2002.

[MCA94] A. McAuley, *Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes*, IEEE/ACM Transactions on Networking, Vol. 2, No. 1, pp. 16-22, February 1994.

[NKS88] T. Nakassis, *Fletcher's Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls*, ACM Computer Communication Review, Vol. 18, No. 5, pp. 86-94, October 1988.

[PET00] L. Peterson, *Computer Networks*, Morgan Kaufmann, 2a Ed. San Francisco, California, 2000.

[RFC1071] R. Braden, *et al.*, *Computing the Internet Checksum*, Internet RFC 1071, September 1988.

[RFC1145] J. Zweig, *et al.*, *TCP Alternate Checksum Options*, Internet RFC 1145, February 1990.

[RFC1191] J. Mogul, *et al.*, *Path MTU Discovery*, Internet RFC 1191, November 1990.

[RFC1950] P. Deutsch, *et al.*, *ZLIB Compressed Data Format Specification version 3.3*, Internet RFC 1950, May 1996.

[RFC2960] R. Stewart, *et al.*, *Stream Control Transmission Protocol*, Internet RFC 2960, October 2000.

[RFC768] J. Postel, *et al.*, *User Datagram Protocol*, Internet RFC 768, August 1980.

[RFC791] J. Postel, *Internet Protocol*, Internet RFC 791, September 1981.

[RFC793] J. Postel, *Transmission Control Protocol*, Internet RFC 793, September 1981.

[SHW01] *On Fletcher and Adler codes, and classic CRCs*, Email from Dr. Dafna Sheinwald from the IBM Haifa Research Lab to iSCSI group via Julian Satran of IBM circa, January 2001.

[SKL89] K. Sklower, *Improving the Efficiency of the OSI Checksum Calculation*, ACM Computer Communication Review, Vol. 19, No. 5, pp. 32-43, October 1989.

[SLT84] J. Saltzer, D. Reed, D. Clark, *End-to-end arguments in system design*, ACM Transactions in Computer Systems, 2, pp. 277-288, November 1984.

[STL99] W. Stallings, *Computer Organization and Architecture*, 5ª Ed. Prentice Hall, New Jersey, 1999.

[STN02] J. Stone, *et al.*, *SCTP Checksum Change*, Internet draft draft-ietf-tsvwg-sctpcsum-07.txt. Work in progress, May 2002.

[STN98] J. Stone, M. Greenwald,C. Partridge, J. Hughes, *Performance of checksums and CRCs over real data*, IEEE/ACM Transactions on Networking, Vol. 6, N° 5 , pp. 529-543. October, 1998.

[STW02] R. Stewart, Q. Xie, *Stream Control Transmission Protocol (SCTP): A Reference Guide*, Indianapolis, IN: Addison-Wesley, 2002.

[TAN02] A. Tanenbaum, *Computer Networks*, 4ª ed. Addison Wesley, 2002.

[WCK95] S. Wicker, *Error Control Systems for Digital Communication and Storage*, Englewood Cliffs: Prentice Hall, 1995.

[WIL93] R. Williams, *A Painless Guide to CRC Error Detection Algorithms*, Internet: ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt. August, 1993.

[WLF94] J. Wolf, D. Chun, *The Single Burst Error Detection Performance of Binary Cyclic Codes*, IEEE Transactions on Communications, Vol. 42, n° 1, pp. 11-13, January 1994.

[WLF94B] D. Chun, J. Wolf, *Special Hardware for computing the probability of undetected error for certain binary CRC codes and test results*, IEEE Transactions on Communications, COM-42:2769-2772, October 1994.

[ZLIB02] J. Gailly, M. Adler, *zlib Technical Details*, Internet: http://www.gzip.org/zlib/zlib_tech.html. April, 2002.

APÊNDICE A – ESTRUTURAS ALGÉBRICAS

O objetivo deste apêndice é descrever algumas estruturas algébricas que são utilizadas em códigos detectores de erros. Entre estas estruturas estão grupos, anéis e corpos finitos.

A.1. GRUPOS

Definição A.1 – Um grupo $\langle G,* \rangle$ consiste de um conjunto G e uma operação definida em seus elementos:

$$*: G \times G \to G : (a,b) \ \mathbf{a} \ a*b. \qquad \text{(A.1)}$$

A operação * deve obedecer às seguintes condições:

Fechamento: $\forall a,b \in G : a*b \in G$ (A.2)

Associatividade: $\forall a,b,c \in G : (a*b)*c = a*(b*c)$ (A.3)

Elemento Identidade: $\exists e \in G, \forall a \in G : a*e = e*a = a$ (A.4)

Elementos Inversos: $\forall a \in G, \exists b \in G : a*b = e = b*a$ (A.5)

O grupo é chamado de Abeliano (ou comutativo), se além de obedecer estas quatro condições, ser válida a comutatividade:

Comutatividade: $\forall a,b \in G : a*b = b*a$ (A.6)

Definição A.2 – Um grupo $\langle G,* \rangle$ é finito se $|G|$ é finito, onde $|G|$ é a ordem ou cardinalidade do grupo, ou seja, o número de elementos de *G*.

Um exemplo de um grupo Abeliano é $\langle Z,+ \rangle$: o conjunto dos números inteiros, em conjunto com a operação de adição. A estrutura $\langle Z_n,+ \rangle$ é um segundo exemplo. O conjunto contém os números inteiros de 0 até $n - 1$ e a operação é adição módulo *n*. A operação de multiplicação também pode ser utilizada para a formação de grupos, um exemplo seria o grupo $\langle Z_p^*,. \rangle$: um conjunto de números inteiros de 1 até $p - 1$, com a operação de multiplicação módulo *p*, onde *p* é primo.

Definição A.3 – A ordem de um elemento *g* em um grupo é a menor quantidade de vezes que *g* é operado consigo mesmo, resultando na identidade do grupo. Por exemplo, no caso de um grupo multiplicativo, é o menor expoente que resulta no elemento 1 (se a operação fosse de adição o elemento identidade seria o 0), ou seja, o menor inteiro positivo *t* tal que $g^t = 1$. Se tal inteiro t não existir, então a ordem de *g* é definida como ∞.

Definição A.4 – Um grupo com *n* elementos e que possui um elemento *g* de ordem *n* é dito ser um grupo cíclico, sendo *g* chamado de elemento gerador do grupo.

A.2. ANÉIS

Definição A.5 – Um anel $\langle R,+,.\rangle$ consiste de um conjunto R com duas operações definidas em seus elementos, aqui identificadas por '+' e '.'. Para ser classificado como um anel, as operações devem seguir as seguintes condições:

1. A estrutura $\langle R,+\rangle$ deve ser um grupo Abeliano.
2. A operação '.' deve ser fechada, e associativa sobre R.
3. As operações '+' e '.' devem ser relacionadas pela lei da distributividade:

$$\forall\, a,b,c \in R : (a+b).c = (a.c)+(b.c). \qquad \text{(A.7)}$$

Um anel $\langle R,+,.\rangle$ é chamado de anel comutativo se a operação '.' for comutativa. É dito ser um anel com identidade se a operação '.' Tem um elemento identidade, denotado por 1.

Um exemplo de um anel é $\langle Z,+,.\rangle$: o conjunto de inteiros, com as operações adição multiplicação. Este anel é um anel comutativo, com identidade.

A.3. CORPOS

Definição A.6 – A estrutura $\langle F,+,.\rangle$ é um corpo se as duas condições seguintes forem satisfeitas:

1. $\langle F,+,.\rangle$ ser um anel comutativo, com identidade e sem divisores de zero.
2. Para todo elemento de F, existe o elemento inverso em relação à operação '.', exceto para o elemento 0, o elemento identidade de $\langle F,+\rangle$

A estrutura $\langle F,+,. \rangle$ é um corpo se, e somente se ambos $\langle F,+ \rangle$ e $\langle F \setminus \{0\},. \rangle$ forem grupo Abelianos e obedecerem a lei da distributividade.

Definição A.7 – A característica m de um corpo é 0 se 1 + 1 + ... + 1 (m vezes) nunca for igual a 0 para qualquer $m \geq 1$. Caso contrário, a característica de um corpo é o menor inteiro positivo m tal que $\sum_{i=1}^{m} 1$ é igual a 0.

Um exemplo bem conhecido de um corpo é o conjunto de números reais, com as operações de adição e multiplicação. Outros exemplos são o conjunto de números complexos e o conjunto de números racionais, com as mesmas operações. Para estes exemplos o número de elementos é infinito.

A.4. CORPOS FINITOS

Um corpo finito é um corpo com um número finito de elementos. Um corpo com ordem q existe, se e somente se q for potência de um número primo p, ou seja, $q = p^n$, onde p é a característica do corpo finito.

Corpos de mesma ordem são isomórficos: eles possuem a mesma estrutura algébrica diferindo somente na representação dos elementos. Em outras palavras, para cada potência de um número primo existe somente um corpo finito, representado por GF(p^n). (GF significa *Galois Field* ou Corpo Finito)

Como exemplos de corpos finitos, pode-se citar corpos de ordem prima p. Os elementos de um corpo finito GF(p) podem ser representados pelos inteiros 0, 1, ..., $p - 1$. As duas operações do corpo são a adição módulo p e a multiplicação módulo p.

Para corpos finitos com ordem não prima, as operações de adição e multiplicação não podem ser representadas pela adição e multiplicação de inteiros com a operação de módulo.

A.5. POLINÔMIOS SOBRE CORPOS

Um polinômio sobre um corpo F é uma expressão da forma:

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \mathbf{L} + b_2x^2 + b_1x + b_0, \qquad \text{(A.8)}$$

O grau de um polinômio é igual a l se $b_j = 0$, $\forall\, j > l$, e l for o menor número com esta propriedade. O conjunto de polinômios sobre um corpo F é denominado $F[x]$. Quando o conjunto possui grau menor que l, é representado por $F[x]|l$.

Em computadores, os polinômios $F[x]|l$ podem ser armazenados eficientemente representando os l coeficientes como uma palavra [DAE02].

Se o corpo F for GF(2), e l = 8. Os polinômios podem ser armazenados como valores de 8 bits, ou bytes:

$$b(x) \;\mathbf{a}\; b_7b_6b_5b_4b_3b_2b_1b_0 \qquad \text{(A.9)}$$

Sendo assim o polinômio $x^6 + x^4 + x^2 + x + 1$ corresponderia a palavra binária 01010111, ou (57) em notação hexadecimal.

As seguintes operações podem ser definidas sobre polinômios:

Adição – A soma de polinômios consiste da soma dos coeficientes de potências iguais de x, o resultado da soma dos coeficientes mantém-se no corpo F.

$$c(x) = a(x) + b(x) \Leftrightarrow c_i = a_i + b_i, 0 \le i \le n \qquad \text{(A.10)}$$

O elemento neutro para a adição é o polinômio com todos os coeficientes iguais a 0. O grau de $c(x)$ é o máximo dos graus de $a(x)$ e $b(x)$. A estrutura $\langle F[x]\,|\,l, + \rangle$ é um grupo Abeliano.

Exemplificando, considerando F o corpo GF(2). A soma dos polinômios:

$a(x) = x^6 + x^4 + x^2 + x + 1$
$b(x) = x^7 + x + 1$

resulta em $c(x) = (x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1)$
$= x^7 + x^6 + x^4 + x^2 + (1 + 1)x + (1 + 1)$
$= x^7 + x^6 + x^4 + x^2.$

Em notação binária tem-se 01010111 + 10000011 = 11010100. Como se pode notar, a adição pode ser implementada com a operação de OU-EXCLUSIVO.

Multiplicação – A multiplicação de polinômios é associativa (A.3), comutativa (A.6) e distributiva (A.7) em relação à adição de polinômios. Existe um elemento identidade: o polinômio de grau 0 e com coeficiente de x^0 igual a 1. Para que a multiplicação seja fechada (A.2) sobre $F[x]|l$, deve-se selecionar um polinômio $m(x)$ de grau l, chamado polinômio redutor (*reduction polynominal*).

A multiplicação de dois polinômios $a(x)$ e $b(x)$ é então definida como o produto dos polinômios módulo o polinômio $m(x)$:

$$c(x) = (a(x) . b(x)) \pmod{m(x)} \qquad \text{(A.11)}$$

Desta forma, a estrutura $\langle F[x] | l, +, . \rangle$ é um anel comutativo.

Exemplo – Sendo o polinômio $m(x) = x^8 + x^4 + x^3 + x + 1$ o polinômio redutor da multiplicação entre $a(x) = x^6 + x^4 + x^2 + x + 1$ e $b(x) = x^7 + x + 1$:

$a(x) . b(x) = (x^6 + x^4 + x^2 + x + 1) . (x^7 + x + 1)$
$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$

como:

$$
\begin{array}{l|l}
x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 & x^8 + x^4 + x^3 + x + 1 \\
\cline{2-2}
x^{13} \quad + x^9 + x^8 + x^6 + x^5 & x^5 + x^3 \\
\cline{1-1}
x^{11} \quad + x^4 + x^3 + 1 & \\
x^{11} \quad + x^7 + x^6 + x^4 + x^3 & \\
\cline{1-1}
\text{(resto)} \quad x^7 + x^6 \quad + 1 &
\end{array}
$$

temos:

$$a(x) \,.\, b(x) \equiv x^7 + x^6 + 1 (\text{mod } x^8 + x^4 + x^3 + x + 1)$$

Definição A.8 – Um polinômio $d(x)$ é irredutível sobre o corpo GF(p) se e somente se não existir dois polinômios $a(x)$ e $b(x)$ com coeficientes em GF(p) de forma que $d(x) = a(x) \,.\, b(x)$, sendo os graus $a(x)$ e $b(x)$ maiores que 0.

Definição A.9 – Seja $a(x)$ um polinômio irredutível de grau m sobre GF(p). Diz-se que $a(x)$ pertence ao expoente e se $a(x)$ divide $(x^e - 1)$, e não divide $(x^n - 1)$ para $n < e$. Além disso, se $e = p^m - 1$, $a(x)$ é chamado polinômio primitivo.

O elemento inverso para a multiplicação pode ser encontrado em termos do algoritmo de Euclides. Seja $a(x)$ um polinômio que se deseje encontrar o inverso. O algoritmo de Euclides pode ser utilizado para encontrar dois polinômios $b(x)$ e $c(x)$ de tal forma que:

$$a(x) \,.\, b(x) + m(x) \,.\, c(x) = mdc(a(x), m(x)) \qquad \text{(A.12)}$$

Aqui $mdc(a(x),m(x))$ significa o maior divisor comum dos polinômios $a(x)$ e $m(x)$, que é sempre igual a um polinômio de grau 0 se e somente se $m(x)$ for irredutível. Aplicando a redução modular a (A.12), tem-se:

$$a(x) \,.\, b(x) = 1 (\mathrm{mod}\, m(x)), \qquad \text{(A.13)}$$

que significa que $b(x)$ é o elemento inverso de $a(x)$ para a definição de multiplicação '.' dada pela definição A.5.

Se F for o corpo GF(p), escolhendo-se um polinômio irredutível como polinômio redutor, a estrutura $\langle F[x] | n, +, . \rangle$ se torna um corpo com p^n elementos, geralmente representado por GF(p^n).

APÊNDICE B – SOFTWARE DE SIMULAÇÃO

Desenvolvemos nosso simulador em C++, nele implementamos os algoritmos dos códigos descritos no capítulo 2, e os modelos de erros citados no capítulo 3 (Listagem B1).

Antes de iniciar as simulações, informávamos o arquivo que seria utilizado na simulação, a quantidade de iterações e o tamanho das mensagens a serem codificadas (Figura B1). Também era necessário indicar qual o modelo de erro a ser utilizado, e suas opções (Figura B2).

A maior dificuldade do desenvolvimento deste software é que os computadores atuais manipulam as informações em bytes, e a alteração de bits específicos necessários para a implementação dos modelos de erros necessitou de um vetor de byte com 8 elementos, cada elemento possuía um bit diferente dos outros elementos com valor igual 1, desta forma se desejássemos que um bit específico da mensagem fosse alterado era executado uma operação de OU-EXCLUSIVO de um elemento deste vetor com o byte da mensagem (Listagem B1).

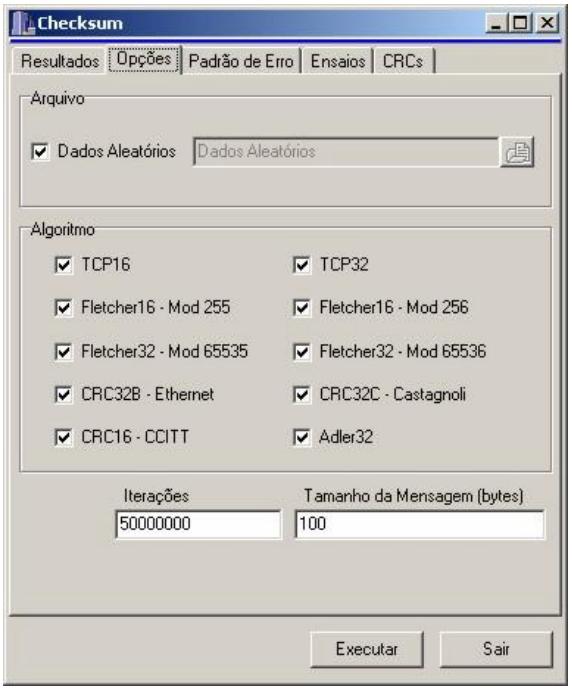


Figura B1 – Opções do Simulador.

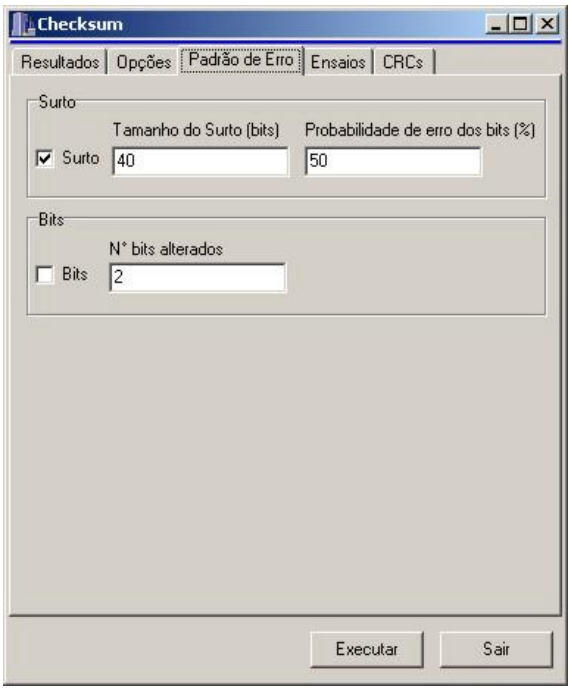


Figura B2 – Opções dos Modelos de erros.

Listagem B1 - Unidade onde foram implementados os modelos de erro (erro.c)

```
//-------------------------------------------------------------------------

unsigned char tab_erro[] = { 0x80U ,  // 10000000
                             0x40U ,  // 01000000
                             0x20U ,  // 00100000
                             0x10U ,  // 00010000
                             0x08U ,  // 00001000
                             0x04U ,  // 00000100
                             0x02U ,  // 00000010
                             0x01U }; // 00000001

//-------------------------------------------------------------------------

void bit_erro(unsigned char *buf, // buffer de dados
              int len,            // tamanho do buffer
              int num_bit)        // numero de bits alterados
{
 int i , j , teste , bit, temp;
 int tab_falha[100];

 i = 0;

 while (i < num_bit) {
   bit = random(len * 8);
   teste = 1;
   for (j = 0;j < i; j++) {
     if (tab_falha[j] == bit) {
       teste = 0;
     }
   }
   if (teste) {
     temp = bit / 8;
     buf[temp] = buf[temp] ^ tab_erro[bit % 8];
     tab_falha[i] = bit;
     i++;
   }
 }
}

//-------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------

int surto_erro(unsigned char *buf, // buffer de dados
               int len,            //  tamanho do buffer em bytes
               short int surto,    // tamanho do surto em bits
               short int prob)     // probabilidade de erro dos bits no surto
{
 int i , inicio_surto, fim_surto, retorno , temp;

 retorno = 1;

 inicio_surto = random((len * 8) - 1 - surto);
 fim_surto = inicio_surto + surto;
 for (i = inicio_surto;i < fim_surto;i++) {
   if (random(100) < prob) {
     temp = i / 8 ;
     buf[temp] = buf[temp] ^ tab_erro[i % 8];
     retorno = 0;
   }
 }
 return(retorno);
}

//---------------------------------------------------------------------------
```

APÊNDICE C – ON FLETCHER AND ADLER CODES, AND CLASSIC CRC-S

Email do Dr. Dafna Sheinwald do IBM Haifa Research Lab para o grupo que desenvolve o protocolo iSCSI, disponível em:

http://www.pdl.cmu.edu/mailinglists/ips/mail/doc00002.doc.

To: ips@ece.cmu.edu
Subject: A memo on some checksums
From: julian_satran@il.ibm.com
Date: Mon, 15 Jan 2001 22:18:08 +0200
cc: Dafna_Sheinwald@il.ibm.com

Fletcher Code [3] and Adler Code [4] detect much less than CRC-32 codes [5]

Fletcher and Adler code seem to lack a theoretical ground, and their error detection capability is inferior to that attainable by CRC of the same number of bits.

CRC-32, Adler-32, and Fletcher-32 (2*16), all append 32 bits to the information data.

With this number of bits, CRC-32 can detect **any** single burst of up to 32 bits, on a stream of data of any length [1], [2],[5].

Here are examples for relatively short bursts that Adler and Fletcher can not detect.

A 24-bit burst error which Adler-32 code can not detect:

Recall that Adler-32 runs two sums: s1 and s2 [4]. Suppose that at some point on the data stream s1=a and s2=b. Suppose that both a and b are way smaller than 65521, the value called "BASE" at [4]. Also suppose that at that point the original data stream continues with bytes of values **4,2,1**. At the end of these three bytes, the values of s1 grows to a+4+2+1 = a+7, whereas s2 grows to b+a+4+a+6+a+7 = b+3a+17. Now, suppose that a 24 burst occurred on these three bytes, which modified them to **5,0,2**. Now, when doing the detection, s1=a and s2=b just before these three bytes, and on completion of their processing, s1=a+5+0+2 = a+7, and s2=b+a+5+a+5+a+7=b+3a+17. Detecting from that point and on to the end of the data stream, s1 and s2 will trace the very same values they had on encoding, and thus the burst is not detected.

This occurs for every three consecutive bytes x,y,z which are modified to x',y', and z', such that 2x+y = 2x'+y', and z and z' are any two numbers such that z'-z = x+y-x'-y'. Too likely.

A 16-bit burst error which Fletcher-32 code can not detect:

Because Fletcher does 1's complement calculations, the addition of 0x0000 to any number other than 0 yields the same result as adding of 0xFFFF. Thus, Fletcher code can not detect two consecutive bytes which turned both from 0x00 to 0xFF .

Fast Software implementation

The celebrated property of both Fletcher and Adler is the speed of their software implementations.

Nevertheless, there are known techniques for programing CRC-32 [6], such that the coding (as well as the decoding) of each byte calls for one table look up, in a table of 256 32-bit words, one AND operation

(which can be avoided in machine code -- taking AL from the whole of EAX, say), one XOR, and one 8-bit shift.

Adler [4] demands the expensive **modulo** operation with a very lare prime, for the processing of each byte, or some test to indicate the necessity in that operation. Fletcher [7] only demands an addition, and can work on 16-bit at a time.

CRC-32 too can be worked 16-bit at a time, but this would mean using a table of 2^16=64K entries of 32-bit each, which might hurt caching.

Good CRC codes

Every CRC code detects **every** one burst error that is not longer than the number of bits of the CRC.

With probability 2^-r (where r is the number of bits or the CRC) it fails to detect a burst longer than r.

Wolf [2] presents an interesting definition of a burst error, where the burst is parameterized by both - its width and the probability of inverting a bit within that width. With this definition, the probability of detecting a burst error that is wider than the number of CRC bits depends on the specific generator polynomial picked, and not only on its degree (=number of CRC bits). Wolf found that good generating polynomials are of the form (1+x)p(x) where p(x) is a primitive polynomial.

Looking at good CRC-64 codes, we would thus want to investigate primitive polynomials of degree 63 [8].

Bibliography

[1] Error Correcting Codes by Peterson and Weldon, the MIT Press, 1961

[2] "The single burst error detection performance of binary cyclic codes" by Jack Wolf and Dexter Chun, IEEE Trans. on Communications, Vol 42, pp 11-13, January 1994.

[3] RFC1146: TCP Alternate Checksum Options, at http://rfc.net/rfc1146.html

[4] RFC1150**:** ZLIB Compressed Data Format Specification version 3.3, at http://rfc.net/rfc1150.html

[5] "Cyclic Redundancy Checking for Ethernet" at http://lev.yudalevich.tripod.com/ECC/crc.html

[6] "Fast CRC32 in Software" by Richard Black, 1994, at www.cl.cam.ac.uk/Research/SRG/bluebook/21/crc/crc.html

[7] "Nasa FITS documents" at http://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/general/checksum/node26.html

[8] "Information on Primitive and Irreducible Polynomials" at http://www.theory.csc.uvic.ca/~cos/inf/neck/PolyInfo.html

APÊNDICE D – CARTA SOBRE A ALTERAÇÃO CÓDIGO UTILIZADO NO SCTP

Date: Thu, 12 Apr 2001 08:42:41 -0400
To: sigtran@standards.nortelnetworks.com
From: Chip Sharp <chsharp@cisco.com>
Subject: SCTP checksum problems
Cc: rrs@cisco.com
In-Reply-To: <LYRIS-1442-559-2001.04.11-18.57.25--chsharp#CISCO.COM@lyri
 s.nortelnetworks.com>
Mime-Version: 1.0
Content-Type: text/plain; charset="us-ascii"; format=flowed
X-Mozilla-Status2: 00000000

Lyndon,
It seems that some researchers (Craig Partridge, Jonathan Stone (Stanford), Jim Wendt (HP)) have finally gotten around to looking at the SCTP checksum... and barfed. :-\
Unfortunately, they didn't do this before the RFC was issued.

The problem with the Adler-32 checksum is that it is noticeably weaker than the alternatives for short packets. Of course, since the primary application of SCTP is Signaling Transport and call signaling typically uses packets less than 128 bytes, this is a major problem.

This problem was basically ferreted out by the iSCSI people.

Randally has already sent an email to the A-Ds. He inadvertently left you off the CC: list.

We currently have a mail thread going with the iSCSI people and with the ADs to figure out the best way to approach this. We have asked the researchers to provide us with a better checksum. Then we have to figure out a way to get it into SCTP. The researchers so far have

proposed either a modified Adler-32 using 16 bit accumulator instead of 8 bit, Fletcher's checksum or CRC-32.

The methods that we have thought of to migrate to a new checksum:

1) Obsolete the current SCTP RFC and come out with a new one with the modified checksum. This will fix the problem, but introduces instant non-backward compatibility.

2) Multiple checksum recalculation on INIT. The transmitter transmits the INIT and the receiver tries different checksums until it matches one. This has the obvious drawback of increasing the work on the receiver and possibly enhancing a DOS.

3) Checksum negotiation: We add a new parameter to the INIT to allow the endpoints to negotiate the checksum used. The disadvantage of this is that it adds complexity to the initialization routine (i.e., deciding when to switch over). The advantage is that it allows for backward compatibility and fallback to original SCTP operation and it allows the addition of new checksums down the road.

These are the alternatives so far. We are separating the choice of checksum algorithm from the method of negotiating the algorithm since they are pretty much orthogonal problems. Once we get these nailed down we can put together a notice to the list, perhaps in the form of an I-D describing each method.

This is an unfortunate occurence, but I believe everyone would like to fix the checksum as early as possible. Randall thinks that moving to a 16 bit accumulator in the Adler's checksum would be a very easy fix in the code.

Chip Sharp Consulting Engineering
Cisco Systems

APÊNDICE E – CUSTO COMPUTACIONAL DOS CÓDIGOS ESTUDADOS

Segundo Kay [KAY96] além da taxa residual, um outro fator a ser considerado na escolha de um código para a detecção de erros na camada de transporte é a velocidade de processamento das palavras código. Em seu trabalho ele verificou que do tempo total gasto na preparação de um pacote TCP, 40% era utilizado no cálculo do *Internet Checksum*, em pacotes de 1.500 bytes.

Para verificarmos a velocidade de codificação das palavras código, executamos os algoritmos 100 mil vezes em mensagens de 1.500 bytes (Figura E1). Estes valores relativos ao TCP16 variam de acordo com processador e memória utilizados (neste caso utilizamos um processador Pentium IV 1,7 GHz e memória PC133).

Verificamos que o número de bits processados a cada *loop* do algoritmo do código é um dos principais fatores que influenciam a velocidade de processamento. As versões com 32 bits de paridade foram mais rápidas do que as que geravam 16 bits de paridade. Sklower [SKL89] fez esta constatação quando estava otimizando o código Fletcher. A única exceção foi o TCP32, que foi penalizado pela inclusão de alguns testes dentro de seu *loop*.

Os algoritmos citados no gráfico E1 estão na listagem E1.

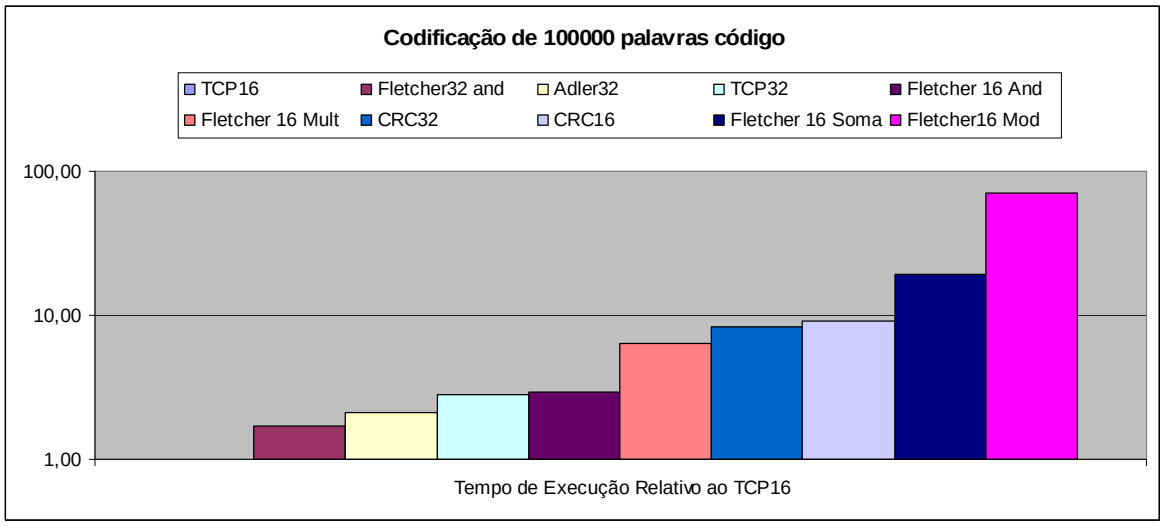


Figura E1 – Tempo de codificação de 100 mil palavras código de 1.500 bytes em relação ao TCP16. (Escala logarítmica)

Listagem E1- Código fonte dos algoritmos utilizados no gráfico E1

```
//  adler32 --------------------------------------------------------------

unsigned int adler32(const unsigned char *input, unsigned int length)
// written and placed in the public domain by Wei Dai
{
        unsigned int adler = 1L;

        unsigned long s1 = adler & 0xffff;
        unsigned long s2 = (adler >> 16) & 0xffff;

      while (length % 8 != 0)
      {
            s1 += *input++;
            s2 += s1;
            length--;
      }

      while (length > 0)
```

```
      {
            s1 += input[0]; s2 += s1;
            s1 += input[1]; s2 += s1;
            s1 += input[2]; s2 += s1;
            s1 += input[3]; s2 += s1;
            s1 += input[4]; s2 += s1;
            s1 += input[5]; s2 += s1;
            s1 += input[6]; s2 += s1;
            s1 += input[7]; s2 += s1;

            length -= 8;
            input += 8;

            if (s1 >= BASE)
                  s1 -= BASE;
            if (length % 0x8000 == 0)
                  s2 %= BASE;
      }

        return ((unsigned int)(s2 << 16) + s1);
}

//  TCP16 ------------------------------------------------------------------------

/* ----------------------- INTERNET Checksum --------------------------- */

/* Calculate the Internet Protocol family checksum algorithm.
   This code is taken from Steven's "Unix Network Programming" pp454 -455.
   The algorithm is simple, using a 32-bit accumulator (sum),
   we add sequential 16-bit words to it, and at the end, fold back
   all the carry bits from the top 16 bits into the lower 16 bits.
 */

unsigned int tcp16(unsigned short *addr, int nbytes)
{
    unsigned long                sum;
    unsigned short      oddbyte, answer;

    sum = 0L;
    while(nbytes > 1) {
        sum += *addr++;
        nbytes -= 2;
    }
```

```
    if(nbytes == 1) {   /* mop up an odd byte if necessary */
        oddbyte = 0;    /* make sure that the top  byte is zero */
        *((unsigned char *)&oddbyte) = *(unsigned char *)addr; /* 1 byte only */
        sum += oddbyte;
    }
    /* Now add back carry outs from top 16 bits to lower 16 bits */
    sum = (sum >> 16) + (sum & 0xffff); /* add hi-16 to lo-16 */
    sum += (sum >> 16);                 /* add carry bits */
    answer  = ~sum;     /* one's complement, then truncate to 16 bits */
    return(answer);
}

//  TCP32 ------------------------------------------------------------------------

unsigned int tcp32(const unsigned char *buf, int len)

// sctp reference implementation

{
  unsigned long s1 = 0;
  unsigned long s2;
  unsigned long wrap = 0;
  unsigned long *bufp;
  int n,nlen;
  int odd;
  nlen = len / 4;
  bufp = (unsigned long *)buf;
  for (n = 0; n < nlen; n++,bufp++) {
    s2 = (s1 + *bufp);
    if((s2 < s1) && (s2 < *bufp)){
      wrap++;
    }
    s1 = s2;
  }
  odd = len-(nlen*4);
  if(odd){
    unsigned long final;
    unsigned char a,b,c,d;
    if(odd == 1){
      a = buf[(len-1)];
      b = c = d = 0;
    }else if(odd == 2){
      a = buf[(len-2)];
```

```
      b = buf[(len-1)];
      c = d = 0;
    }else{
      a = buf[(len-3)];
      b = buf[(len-2)];
      c = buf[(len-1)];
      d = 0;
    }
    final = (a << 24) | (b << 16) | (c << 8) | d;
    s2 = (s1 + final);
    if((s2 < s1) && (s2 < final)){
      wrap++;
    }
    s1 = s2;
  }
  /* Now add the roll over */
  s2 = s1 + wrap;
  if((s2 < s1) && (s2 < wrap)){
    /* roll it again */
    s2 += 1;
  }
  s1 = ~s2;
  return(s1);
}

// Fletcher32 and ------------------------------------------------------------

unsigned int fletcher32mod65536and(const unsigned char *buf, int len)

// sctp reference implementation

{
  unsigned long s1 = 0;
  unsigned long s2 = 0;
  unsigned short *bufp;
  int n,nlen;
  nlen = len / 2;

  bufp = (unsigned short *)buf;
  for (n = 0; n < nlen; n++,bufp++) {
    s1 = ((s1 + *bufp) & 0x0000ffff);
    s2 = ((s2 + s1) & 0x0000ffff);
  }
```

```
  if(len > (nlen *2)){
    /* do the odd byte, we don't
     * expect this code to execute.
     */
    unsigned short x;
    x = buf[(len-1)] << 8 | 0;
    s1 = ((s1 + x) & 0x0000ffff);
    s2 = ((s2 + s1) & 0x0000ffff);
  }
  return ((unsigned int)(s2 << 16) + s1);
}

//  Fletcher 16 and------------------------------------------------------------

unsigned short int fletcher16mod256and(const unsigned char *buf, int len)
{
  unsigned short int s1 = 0;
  unsigned short int s2 = 0;
  int n;

  for (n = 0; n < len; n++,buf++) {
    s1 = ((s1 + *buf) & 0x00ff);
    s2 = ((s2 + s1) & 0x00ff);
  }

  return ((s2 << 8) + s1);
}

//  Fletcher16 mult ------------------------------------------------------------

unsigned short int fletcher16mod256mult(const unsigned char *buf, int len)
{
  unsigned short int s1;
  unsigned short int s2;
  int n;
  unsigned int is1 = 0;
  unsigned int is2 = 0;

  for (n = 0; n < len; n++,buf++) {
    is1 = is1 + *buf;
    is2 = is2 + ( (len - n) * *buf);
  }
```

```
  s1 = (is1 % 256);

  s2 = (is2 % 256);


  return ((s2 << 8) + s1);
}

//  CRC32 ---------------------------------------------------------------------

unsigned long crc32b_table[256];
/* Initialized first time "crc32()" is called. If you prefer, you can
 * statically initialize it at compile time.
 */

/*
 * Build auxiliary table for parallel byte-at-a-time CRC-32.
 */

#define CRC32B_POLY 0x04c11db7     /* AUTODIN II, Ethernet, & FDDI */

void init_crc32b()
{
        int i, j;
        unsigned long c;

        for (i = 0; i < 256; ++i) {
                for (c = i << 24, j = 8; j > 0;  --j)
                        c = c & 0x80000000 ? (c << 1) ^ CRC32B_POLY : (c << 1);
                crc32b_table[i] = c;
        }
}

unsigned long crc32b(unsigned char *buf, int len)
{
        unsigned char *p;
        unsigned long  crc;

//        if (!crc32b_table[1])    /* if not already done, */
//                init_crc32b();   /* build table */
//        crc = 0xffffffff;       /* preload shift register, per CRC-32 spec */
        crc = 0x00000000; // alterei
        for (p = buf; len > 0; ++p,  --len)
```

```
                crc = (crc << 8) ^ crc32b_table[(crc >> 24) ^ *p];
//        return(~crc);            /* transmit complement, per CRC -32 spec */
        return(crc); // alterei
}

//  CRC16 ------------------------------------------------------------------

/*
 * CRC 010041 // bell labs
 */
static unsigned short crc_table[256] = {
 0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
 0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
 0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0 x62D6,
 0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
 0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
 0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
 0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x569 5, 0x46B4,
 0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
 0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
 0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
 0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0 x3A33, 0x2A12,
 0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
 0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
 0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
 0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E3 2, 0x1E51, 0x0E70,
 0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
 0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
 0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
 0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0 xD31C, 0xE37F, 0xF35E,
 0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
 0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
 0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
 0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75 F, 0xF77E, 0xC71D, 0xD73C,
 0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
 0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
 0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
 0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0 x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
 0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
 0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
 0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
 0xEF1F, 0xFF3E, 0xCF5D, 0xDF7 C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
 0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0 };
```

```
unsigned short crc_ccitt(unsigned char *s, int n)

{

register unsigned short crc=0;

while (n-- > 0)

crc = crc_table[(crc>>8 ^ *s++) & 0xff] ^ (crc<<8) ;

return crc;

}

//  Fletcher16 soma ------------------------------------------------------

unsigned short int fletcher16mod255soma(const unsigned char *buf, int len)

// dr. dobb's journal

{
 unsigned short int s1 = 0;
 unsigned short int s2 = 0;

 int i;

 for (i=0; i<len; i++) {
   s1 += *buf++;
   if (s1 >= 255) s1 -= 255;
   s2 += s1;
   if (s2 >= 255) s2 -= 255;
 }
 return ((s2 << 8) + s1);
}

//  Fletcher16 mod -------------------------------------------------------------

unsigned short int fletcher16mod255(const unsigned char *buf, int len)

// dr. dobb's journal

{
```

```
  unsigned short int s1 = 0;
  unsigned short int s2 = 0;
  int n;

  for (n = 0; n < len; n++,buf++) {
    s1 = ((s1 + *buf) % 255);
    s2 = ((s2 + s1) % 255);
  }

  return ((s2 << 8) + s1);
}
```