

Universidad del Valle de Guatemala
Facultad de ingeniería

The logo consists of the letters 'UVG' in a large, bold, white sans-serif font, centered on a solid green rectangular background.

UNIVERSIDAD
DEL VALLE
DE GUATEMALA

Redes
Laboratorio 2.1 - Informe
Esquemas de detección y corrección de errores

Andrés de la Roca
Jun Woo Lee

Guatemala, 2023

Descripción de la práctica

El objetivo principal de este laboratorio es practicar y entender el uso de los algoritmos de corrección y detección de errores, ya que estos son muy importantes y utilizados en el ámbito de redes, por lo cual vale la pena entender a un nivel más práctico como es que funcionan estos por lo que se ideó esta actividad para identificar las ventajas y desventajas de los algoritmos que elijamos luego de hacer nuestra propia implementación de estos en versión de emisor y receptor. Se harán pruebas que pondrán a la luz las ventajas y desventajas de los algoritmos de detección y corrección de errores elegidos.

Algoritmo de detección de errores

Para el algoritmo de detección de errores se optó por utilizar *Fletcher checksum* cuyo cálculo se realiza a través de la suma de comprobación de los datos del mensajes y consiste en dos sumas separadas, las cuales son comprueban la longitud correcta del mensaje y la comprobación de datos correctos. Se eligió este para poder trabajar con uno diferente al CRC-32 y observar sus diferencias, especialmente porque se dice que el Fletcher checksum no es tan robusto como el CRC, pero es rápido y de una complejidad baja, por lo que será interesante observar más adelante en bajo que situaciones es donde más flaquea este algoritmo.

Algoritmo de corrección de errores

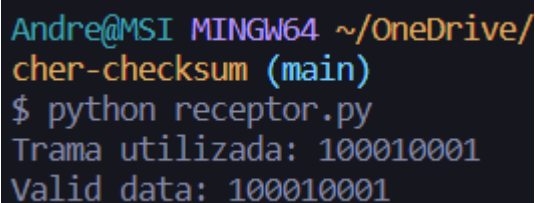
Para el algoritmo de corrección de errores, se optó por utilizar el código de Hamming. Este algoritmo permite detectar y corregir errores en los datos transmitidos mediante la adición de bits de paridad en posiciones específicas dentro del mensaje. Se eligió este algoritmo debido a su capacidad para corregir errores de un solo bit y detectar errores de dos bits, lo que lo hace adecuado para aplicaciones donde la tasa de error es baja pero la corrección de errores es importante. Además, el código de Hamming es relativamente simple y fácil de implementar, lo que lo hace una buena opción para aplicaciones donde la eficiencia y la simplicidad son importantes.

Resultados

Algoritmo de detección de errores

Pruebas sin manipulación

- 100010001



```
Andre@MSI MINGW64 ~/OneDrive/
cher-checksum (main)
$ python receptor.py
Trama utilizada: 100010001
Valid data: 100010001
```

- 110101101

```
Andre@MSI MINGW64 ~/OneDrive/
cher-checksum (main)
$ python receptor.py
Trama utilizada: 110101101
Valid data: 110101101
```

- 1010110001

```
Andre@MSI MINGW64 ~/OneDrive/
cher-checksum (main)
$ python receptor.py
Trama utilizada: 1010110001
Valid data: 1010110001
```

Pruebas con manipulacion de 1 bit

- 1011011 -> 1011010

```
Andre@MSI MINGW64 ~/OneDrive/
cher-checksum (main)
$ python receptor.py
Trama utilizada: 1011011
Modificacion trama: 1011010
Invalid data
```

- 11011001 -> 11011011

```
Andre@MSI MINGW64 ~/OneDrive/
cher-checksum (main)
$ python receptor.py
Trama utilizada: 11011001
Modificacion trama: 11011000
Invalid data
```

- 101011011 -> 100011011

```
cher-checksum (main)
$ python receptor.py
Trama utilizada: 101011011
Modificacion trama: 100011011
Invalid data
```

Pruebas con manipulacion de 2 bits

- 101101111 -> 101001101

```
Andre@MSI MINGW64 ~/OneDrive/Un
cher-checksum (main)
$ python receptor.py
Trama utilizada: 101101111
Modificacion trama: 101101001
Invalid data
```

- 100011010 -> 111011010

```
Andre@MSI MINGW64 ~/OneDrive/Un
cher-checksum (main)
$ python receptor.py
Trama utilizada: 100011010
Modificacion trama: 111011010
Invalid data
```

- 1010100110 -> 0010100100

```
Andre@MSI MINGW64 ~/OneDrive/Unive
cher-checksum (main)
$ python receptor.py
Trama utilizada: 1010100110
Modificacion trama: 1010000010
Invalid data
```

Pruebas con manipulación para evitar detección

- 10010011101 -> 01101100010 (Inversión de bits)

```
$ python receptor.py
Trama utilizada: 10010011101
Modificacion trama: 01101100010
Valid data: 01101100010
```

- 1101001010 -> 1010110100

```
er-checksum (main)
$ python receptor.py
Trama utilizada: 1101001010
Modificacion trama: 1010110100
Valid data: 1010110100
```

- 11110000111 -> (Inversión de bits)

```
er-checksum (main)
$ python receptor.py
Trama utilizada: 11110000111
Modificacion trama: 00001111000
Valid data: 00001111000
```

Aquí se puede observar dos vulnerabilidades en el algoritmo causados por inversión de bits y otro con ciertos bits modificados con el objetivo de explotar la vulnerabilidad principal del algoritmo de Fletcher, el cual es la forma en la que realiza la suma para hacer el cálculo de la suma lo que lo hace no tan robusto ante este tipo de errores en los que partes clave del cálculo son cambiados, lo que termina causando que la suma termine siendo la misma tanto para la trama original como para la trama modificada, lo que produce que la trama modificada haga un *bypass* de la detección de errores y el algoritmo la tome como correcta cuando compara los cálculos (la suma).

Para las pruebas con errores de inversión de bytes el resultado del cálculo que se realiza es prácticamente el mismo para las dos tramas, lo que, al no tener ningún chequeo adicional, se toma como correcta la trama modificada.

En la prueba donde se cambiaron ciertos bits lo que sucede es que se tiene una serie de cálculos bastante similar y como se mencionó anteriormente, se toma como una trama correcta al tener la misma suma.

Esta falta de chequeos dentro de la estructura de este algoritmo es lo que lo hace ser tan eficiente en sus cálculos, lo que lo hace ideal para ciertas aplicaciones, sin embargo, esto es algo que se discutirá más adelante.

Algoritmo de corrección de errores

Pruebas sin manipulación

- 100010001

```
Encoded data: 1111000010001  
Decoded data: 100010001  
No errors detected
```

- 110101101

```
Encoded data: 1011101101101  
Decoded data: 110101101  
No errors detected
```

- 1010110001

```
Encoded data: 00100101110001  
Decoded data: 1010110001  
No errors detected
```

Pruebas con manipulación de 1 bit

- 1011011 -> 1011010

```
Encoded data: 1011000  
Decoded data: 1010  
Corrected error at position 6
```

- 11011001 -> 11011011

```
Encoded data: 11011011
Decoded data: 1111
Corrected error at position 5
```

- 101011011 -> 100011011

```
Encoded data: 100011011
Decoded data: 11101
Corrected error at position 3
```

Pruebas con manipulacion de 2 bits

- 101101111 -> 101001101

```
Encoded data: 101001101
Decoded data: 1100
Corrected error at position 2
```

- 100011010 -> 111011010

```
launcher 50271 -- C:\Dev\K
Encoded data: 111011010
Decoded data: 1101
Corrected error at position 3
```

- 1010100110 -> 0010100100

```
launcher 50281 -- C:\Dev\K
Encoded data: 0010100100
Decoded data: 0001
Corrected error at position 6
```

Pruebas con manipulación para evitar detección

- 10010011101 -> 01101100010 (Inversión de bits)

```
launcher 50489 -- C:\Dev\K
Encoded data: 101000100011101
Decoded data: 1000
No errors detected
```

- 1101001010 -> 1010110100

```
launcher 50498 -- C:\Dev\K
Encoded data: 11111010001010
Decoded data: 1110
No errors detected
```

- 1111000011 -> (Inversión de bits)

```
launcher 50506 -- C:\Dev
Encoded data: 00111110000011
Decoded data: 0010
No errors detected
```

Aquí se pueden observar dos vulnerabilidades en el algoritmo del código Hamming, una causada por la inversión de bits y otra por la modificación de ciertos bits, con el objetivo de explotar la vulnerabilidad principal del código. Dicha vulnerabilidad radica en la forma en que realiza el cálculo de paridad para detectar y corregir errores, lo que lo hace menos robusto ante ciertos tipos de errores en los que partes clave del cálculo son cambiadas. Esto provoca que, tanto para la trama original como para la trama modificada, la paridad resultante sea la misma, lo que produce que la trama modificada evite la detección de errores y el algoritmo la considere como correcta al comparar las paridades.

En las pruebas con errores de inversión de bytes, el resultado del cálculo de paridad es prácticamente el mismo para ambas tramas. Debido a la falta de chequeos adicionales, la trama modificada es tomada como correcta, lo que representa una vulnerabilidad significativa.

En la prueba donde se cambiaron ciertos bits, ocurre una serie de cálculos de paridad bastante similar, y como se mencionó anteriormente, la trama se toma como correcta debido a la igualdad en la paridad.

Discusión

Según Maxino (2006) el algoritmo de Fletcher Checksum es significativamente más eficiente y rápido en la realización de sus cálculos a comparación de otros como CRC y Adler, sin embargo, el precio de este alto rendimiento es el hecho que no hace varios de los chequeos a la trama que los demás algoritmos realizan, lo que puede llevar a que para ciertas tramas el algoritmo falla en la detección de errores.

Si hablamos de las pruebas normales realizadas, se puede observar que se detectaron errores correctamente cuando la trama era modificada, esta detección sucede a través de los cálculos de suma realizados durante la operación característicos de este algoritmo.

Tomando un enfoque en las pruebas manipuladas para evitar detección se puede decir que, el algoritmo de Fletcher puede fallar especialmente en tramas cuyos bits sean invertidos, adicionalmente, se puede comentar que debido a cómo el algoritmo calcula la suma esta puede llegar a fallar incluso cuando no se encuentran invertidos los bits pero si cuando se hacen ciertas sumas en el proceso que coinciden entre la trama modificada y la trama original, el proceso mediante el cual este sucede es cuando se realizan las suma de los bits en posiciones complementarias se cancelan entre sí, por lo cual el algoritmo no es capaz de detectar este de una trama a otra.

El código Hamming, al igual que el algoritmo de Fletcher, está diseñado para la detección y corrección de errores en la transmisión de datos. Si bien ambos tienen funcionalidades similares, existen algunas diferencias importantes en su implementación y rendimiento.

El código Hamming utiliza la técnica de redundancia para agregar bits de paridad a la trama original, lo que le permite detectar y corregir errores de un solo bit. La capacidad de corrección de errores del código Hamming puede mejorarse al agregar más bits de paridad, pero esto aumenta la sobrecarga y complejidad del algoritmo.

A diferencia del algoritmo de Fletcher, el código Hamming tiene una mayor capacidad para detectar y corregir errores de un solo bit, lo que lo hace más confiable en ciertos escenarios. Sin embargo, al igual que el algoritmo de Fletcher, el código Hamming también puede tener vulnerabilidades, especialmente frente a errores en ráfaga y múltiples errores dentro del mismo bloque de código.

Comentarios

Las fallas encontradas en los algoritmos nos dejan ver cómo cada uno de ellos tienen sus propias fortalezas y debilidades, por lo que al momento de que se utilicen en un ambiente real se debe utilizar el algoritmo correcto, no hay una sola opción “óptima” para todo tipo de problemas, sino que, hay que saber elegir dependiendo de las necesidades de la solución a la problemática con la que se encuentre. Por ejemplo si se necesita consumir pocos recursos y tener una respuesta lo mas rápido posible se puede utilizar el Fletcher Checksum, si se quisiera más seguridad dentro de la detección de errores se podría utilizar alguna de las variantes de CRC, todo depende siempre del contexto en el que se utilice y elegir el que mejor se pueda desempeñar de acuerdo con los objetivos que se tienen.

En el caso del código Hamming, al igual que con el algoritmo de Fletcher, es esencial considerar sus ventajas y limitaciones al seleccionar un mecanismo de detección y corrección de errores. El código Hamming es especialmente eficiente en la detección y corrección de errores de un solo bit, lo que lo hace adecuado para aplicaciones donde se prioriza la precisión en la detección de errores puntuales.

Conclusiones

- El algoritmo Fletcher Checksum es de los algoritmos de detección de errores más veloces y eficientes y que además provee con una relativamente alta efectividad sin consumir muchos recursos.
- El algoritmo Fletcher Checksum es bastante propenso a fallar cuando se encuentra con bits inversos entre la trama original y la trama recibida, ya que debido a sus cálculos no le es posible diferenciar en la mayoría de los casos.
- Es necesario conocer las necesidades del proyecto/solución a una problemática antes de implementar un algoritmo en concreto.
- Todos los algoritmos tienen sus fallas o puntos débiles, lo cual hace interesante y útil conocer una gran variedad de algoritmos para implementar.
- El código Hamming es eficaz en la detección y corrección de errores de un solo bit, pero puede ser limitado en la detección de errores en ráfaga o múltiples errores dentro de la misma trama.

Bibliografia

Maxino, T. (2006) Revisiting Fletcher and Adler Checksums. Carnegie Mellon University

Maxino, T.C., Koopman, P.C. (2009) The effectiveness of checksum for embedded control networks. IEEE transactions on dependable and secure computing.

NASA (2007) Fletcher Checksum. FITS Documents