

Análisis y Detección de Esteganografía en Audio

Luciano Bello
111863-8
luciano@linux.org.ar

Procesamiento de Señales
Facultad Regional Buenos Aires
Universidad Tecnológica Nacional
Profesor: Dr. Ing. Marcelo Risk
Curso: K-5051

Índice

1. Introducción	3
2. Métodos	3
2.1. Características de la funciones y lenguajes utilizados	3
2.2. Manipulación de bits	5
3. Resultados	7
3.1. Efecto en la amplitud	8
3.2. Efecto en la frecuencia	9
4. Discusión y Conclusiones	11
5. Anexos de códigos fuente	12
5.1. steganography.R	12
5.2. steganography.c	13

1. Introducción

La esteganografía es la rama de la criptología que trata sobre la ocultación de mensajes para evitar que se perciba la existencia del mismo. En contraste con la criptografía tradicional, en donde la existencia del mensaje es clara, pero el contenido del mensaje está oculto, la esteganografía intenta ocultar la existencia misma del secreto[8]. Se busca que un mensaje de este tipo se disimule en otro mensaje que lo contiene, con sentido, cuyo conocimiento pueda ser público y no levante sospechas sobre el secreto que camufla.

Las expresiones de esteganografía históricamente usadas hacen referencia a tinta invisible, marcas de agua o pequeñas variaciones en la letra manuscrita, por citar solo alguno de los ejemplos. Más recientemente, se busca ocultar mensajes en imágenes[5]. Una técnica conocida consiste modificar los bits menos significativos de cada pixel, tratando de que estas diferencias pasen por alto para el ojo humano. Así se explota la falta de precisión de la visión humana a la que la imagen le resulta sin alteración, ya que la mayoría de los estándares permiten utilizar escalas de colores con niveles tan parecidos que el ojo humano no puede distinguir.

Generalmente, los mensajes a ocultar son cifrados de forma tradicional (típicamente, utilizando criptografía simétrica) antes de camuflarse en el mensaje contenedor. Los mensajes, una vez cifrados, tienen una apariencia caótica que les permite disimularse dentro del mensaje inocente. Por otro lado, si se descubriera la existencia del secreto, el adversario todavía tendría que lidiar con el criptoanálisis necesario para recuperar la clave utilizada.

Existen variantes de la esteganografía cuyo oponente no es un humano, sino un sistema computacional. Las funciones *mimic*, creadas por Peter Wayner, ofuscan un mensaje para que tenga propiedades estadísticas de otro, como ser una obra de Shakespeare o un artículo periodístico¹. Este tipo de esteganografía no puede engañar a una persona, pero sí a alguna gran computadora que busca en Internet mensajes con ciertas características [7, 5].

Ocultarse de un humano tiene problemáticas relacionadas con la naturaleza analógica e imprecisa de los sentidos, como en el caso de las imágenes. Ocultarse de un sistema implica modificar patrones y características estadístico-matemáticas, como lo demuestra las funciones *mimic*. Pero ocultarse de un humano armado con herramientas de análisis digital es otro tema.

Este trabajo busca analizar las características que puede tener una señal de audio esteganografiado, dependiendo de la distribución y la forma de las perturbaciones, así como también la cantidad de información que puede contenerse sin delatar su presencia.

2. Métodos

2.1. Características de las funciones y lenguajes utilizados

En este trabajo se utilizó audio en formato WAV², ya que no tiene compresión y es estándar. Para el procesamiento y análisis de los resultados se utilizó el lenguaje de programación R. La manipulación de bits se realizó en C y se desarrolló el conector entre ambos lenguajes.

¹Para más información, consulte en <http://www.wayner.org/texts/mimic/>.

²WAVEform audio format

Todas las pruebas se realizaron desde el interprete de comandos *R*. Las funciones desarrolladas para este trabajo se encuentran en el archivo `steganography.R`, cuyo código fuente se adjunta tanto en formato digital como en el 5.1.

Para ejecutar el contenido de este archivo se debe correr el siguiente comando:

```
> source("steganography.R")
```

Esta acción declara las funciones que se explicarán más adelante, aunque lo que primero se ejecuta es `library(sound)`, cargando la biblioteca `sound`[3]. Éste paquete debe estar preinstalado y se requiere para la manipulación de archivos de audio. La función nativa que carga el archivo se llama `loadSample(filename, filecheck=TRUE)` y devuelve un objeto de tipo *Sample*, donde las muestras son de tipo *double* (real, en coma flotante) entre -1 y 1. Basado en esta función se escribió una nueva rutina llamada `loadSampleAsBits(filename, filecheck=TRUE)` que devuelve el mismo tipo de objeto, pero con muestras representadas por *integers* (enteros), donde el *string de bits* puede manipularse fácilmente.

```
> # Carga del archivo de audio en ambos formatos.
> sonido1<-loadSample("archivo.wav")
> sonido2<-loadSampleAsBits("archivo.wav")
> # Ambas funciones devuelven un objeto de tipo Sample
> is.Sample(sonido2)$test && is.Sample(sonido1)$test
[1] TRUE
> # Las muestras se almacenan dentro del atributo
> # "sound" en forma de matriz. Comparación entre los tipos
> # de muestra de cada Sample (primer valor del canal 1).
> typeof(sonido1$sound[[1,1]])
[1] "double"
> typeof(sonido2$sound[[1,1]])
[1] "integer"
```

Nótese que, si bien la salida de ambas funciones es del mismo tipo (*Sample*), las muestras de la función modificada no tienen valores entre -1 y 1. Esto hace que las gráficas se vayan de escala, porque la función `plot(x, y, ...)` espera valores cuyo valor absoluto sea menor a uno. Una situación semejante ocurre con la función `play(s, stay=FALSE, command=WavPlayer())`, encargada de reproducir el sonido.

Para compatibilizar ambos objetos y que las gráficas y reproducciones no degeneren se desarrolló una función que transforma las muestras devolviéndolas a su estado original de números reales de coma flotante: `stegaSampleAsDouble(Sample)`.

```
> # Carga del archivo con funciones esteganográficas.
> source("steganography.R")
> # Tipo de la muestra del Sample antes y después.
> typeof(sonido2$sound[[1,1]])
[1] "integer"
> typeof(stegaSampleAsDouble(sonido2)$sound[[1,1]])
[1] "double"
```

Así, ambos tipos se compatibilizan, de manera tal que se pueden manipular en un formato para después convertirlo al otro y analizarlo.

Por último, se desarrolló la función `stegaBits(Sample, bits=0, porcentaje=100, accion=0)`, que es la interfaz de conexión con la función `void bits(int *sound, int *nsound, unsigned short *bits, int *porcentaje, short *accion)`, escrita en *C* e incluida en `steganography.c` cuyo código fuente también se adjunta (ver sección 5.2). Una vez compilada es `steganography.so` y es cargado por la segunda línea de `steganography.R` a través del comando `dyn.load("steganography.so")`.

Esta interfaz de conexión permite abstraerse de la lógica manipuladora de bits de bajo nivel que se explicará más adelante.

El objetivo de la función `stegaBits()` es modificar los bits menos significativos de las muestras de un objeto *Sample*. Recibe cuatro parámetros, aunque sólo el primero es obligatorio y lo demás pueden tomar valores por defecto:

Sample: Campo obligatorio. Es el objeto *Sample* a modificar. Sus muestras deben estar en formato *integer*, es decir, haber sido cargado con la función `loadSampleAsBits()`.

bits: Valor por defecto: 0 (sin cambios). Es la cantidad de bits menos significativos a modificar en cada muestra. Sus valores prácticos posibles son enteros entre 1 y 15, ya que cada muestra está codificada en 16 bits (más tarde se explicará la falta de sentido de modificarlos todos). El valor 0 equivale a no realizar modificaciones.

porcentaje: Valor por defecto: 100 (todas las muestras). Es el porcentaje de muestras que van a modificarse. Sus valores prácticos posibles son enteros entre 1 y 100. Las muestras a modificar tratarán de estar distribuidas uniformemente en la matriz.

accion: Valor por defecto: 0 (modificar con ceros). Es el tipo de bits con el que se modificará la muestra. Los valores posibles son: 0 para convertir los bits menos significativos en ceros, 1 para hacerlo con unos, y cualquier otro número para convertirlo en bits al azar. Todos los valores distintos de 1 y 0 serán tratados de la misma manera. Cuando una muestra es modificada se conserva su signo, incluso cuando se alteren los 16 bits de la misma.

La modificación aleatoria de bits tiene particular relevancia en este trabajo ya que, como se dijo, la información que es almacenada en esteganografía está cifrada previamente. Así, la información a ocultar en el audio, tendrá una apariencia estocástica y, a fines de este trabajo, reemplazable con azar.

2.2. Manipulación de bits

La manipulación de bits se realiza con la función `stegaBits()`. A continuación, se detallan algunos ejemplos sobre qué implica dicha manipulación en sentido numérico.

En primer lugar, se debe tener en cuenta la forma en la que se codifican las muestras. En las señales analizadas, la calidad del audio es de 16 bits. Esto significa que cada muestra puede codificarse, en términos enteros, con límites ± 32767 . La capacidad del *integer*, tanto en *R* como en *C*, supera sobradamente este límite, con lo que es un formato útil para extraerlo del archivo. En *C*, se almacenaron las muestras en variables de tipo *short* que tiene exactamente 16 bits y permite realizar operaciones lógicas más fácilmente.

Por ejemplo, supóngase que se tiene una muestra cuyo valor es 1737 y si quiere utilizar esteganográficamente los últimos 5 bits. Inicialmente, se puede observar los límites de la deformación que sufrirá la muestra transformando los bits contenedores en ceros y en unos.

original	1737	00000110 110	01001
ceros	1728	00000110 110	00000
unos	1759	00000110 110	11111

Cuando se almacene información en esta muestra, el valor m que se obtenga entrará entre las 32 (2^5) posibilidades que presenta el rango $1728 \leq m \leq 1759$.

En el entorno de programación R ya presentado se obtiene este efecto de la siguiente forma:

```
> # Muestra original (número 34210 del canal 1).
> sonidito$sound[1,34210]
[1] 1737
> # Los últimos 5 bits se transforman en ceros.
> stegaBits(sonidito,5,accion=0)$sound[1,34210]
[1] 1728
> # Los últimos 5 bits se transforman en unos.
> stegaBits(sonidito,5,accion=1)$sound[1,34210]
[1] 1759
> # Los últimos 5 bits se transforman aleatoriamente.
> stegaBits(sonidito,5,accion=1)$sound[1,34210]
[1] 1742
```

Las alteraciones de este tipo pueden aplicarse a todas o algunas de las muestras. Por ejemplo, la deformación al azar de los 10 bits menos significativos de toda una señal puede verse gráficamente en la figura 1, comparada con la señal original. Nótese la fuerte diferencia con la figura 2, cuando a la misma entrada se la alteran en 11 bits, pero sólo a la mitad de las muestras. El bit adicional aumenta el desvío exponencialmente ($2^{10} = 1024$ y $2^{11} = 2048$). Ambos gráficos muestran solo un detalle para que las diferencias puedan evidenciarse. Esta característica de modificar la densidad de las alteraciones es podría ser importante, porque el oído humano tiene a rellenar espacios con relativa facilidad.

Para afectar solo algunas muestras a los cambios, hay que modificar la variable **porcentaje** de la función `stegaBits()`, que es su tercer parámetro. En los ejemplos:

```
> modificado<-stegaBits(sonido, bits=10, porcentaje=100)
Resumen:
 91646 muestras cambiadas (100%)
114557 bytes de random
> modificado1<-stegaBits(sonido, bits=11, porcentaje=50)
Resumen:
 45824 muestras cambiadas (50%)
 63008 bytes de random
> # Para graficarlos o reproducirlos, las muestras
> # deben estar en formato double.
```

```
> play(stegaSampleAsDouble(modificado1))  
> plot(stegaSampleAsDouble(modificado2))
```

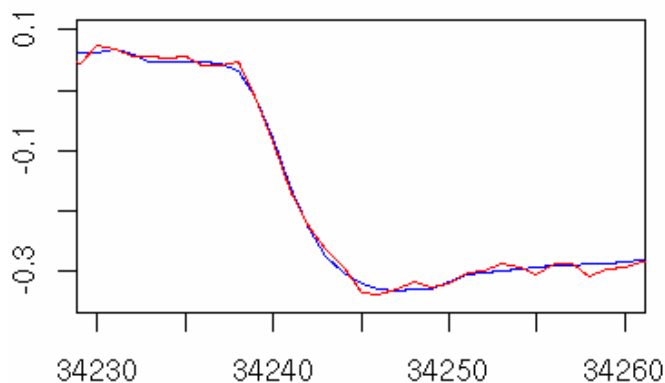


Figura 1: Señal original (azul) y alterada (roja) con bits al azar. Deformación de 10 bits en el 100 % de las muestras. (detalle)

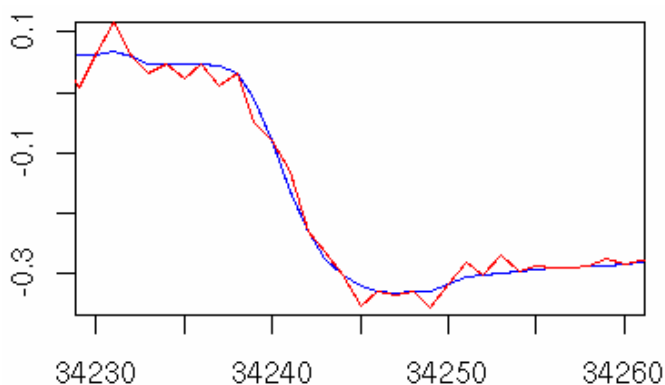


Figura 2: Señal original (azul) y alterada (roja) con bits al azar. Deformación de 11 bits en el 50 % de las muestras. (detalle)

Como puede observarse, las variaciones de los parámetros modifican la capacidad de almacenamiento del audio que contendrá el mensaje a esteganografiar.

3. Resultados

La esteganografía que se estudia en este trabajo reside en aplicar el concepto explicado en el apartado anterior, es decir, manipular los bits menos significativos de todos o algunas de las muestras. En estas pruebas se estudiará las alteraciones auditivas y analíticas que afectan a tres tipos de señales: un audio real con amplitud decreciente, tono de baja frecuencia y otro de alta frecuencia. En todos los casos el muestreo es de 22050 por segundo y con una duración aproximada de 3,46 segundos.

3.1. Efecto en la amplitud

Para esta prueba se utilizó el archivo `sonido.wav`, que se adjunta. Este audio tiene 76287 muestras, a 22050 por segundo, 16 bits de calidad y en estero.

```
> sonido<-loadSampleAsBits("sonido.wav")
> sonido
type      : stereo
rate      : 22050 samples / second
quality   : 16 bits / sample
length    : 76287 samples
R memory  : 610296 bytes
HD memory : 305192 bytes
duration  : 3.46 seconds
> # Modificar todos los bits del 100% de las muestras.
> muerto<-stegaBits(sonido,16,100)
Resumen:
 152574 muestras cambiadas (100%)
 305148 bytes de ceros
```

Son 152.574 palabras de 16 bits, por lo que puede contener una capacidad máxima teórica de casi 300 Kbytes (2 bytes en cada muestra). Es obvio que poco sentido tendría utilizar su capacidad máxima, porque la señal se vería estocásticamente deformada en su totalidad y el mensaje secreto perdería su camuflaje.

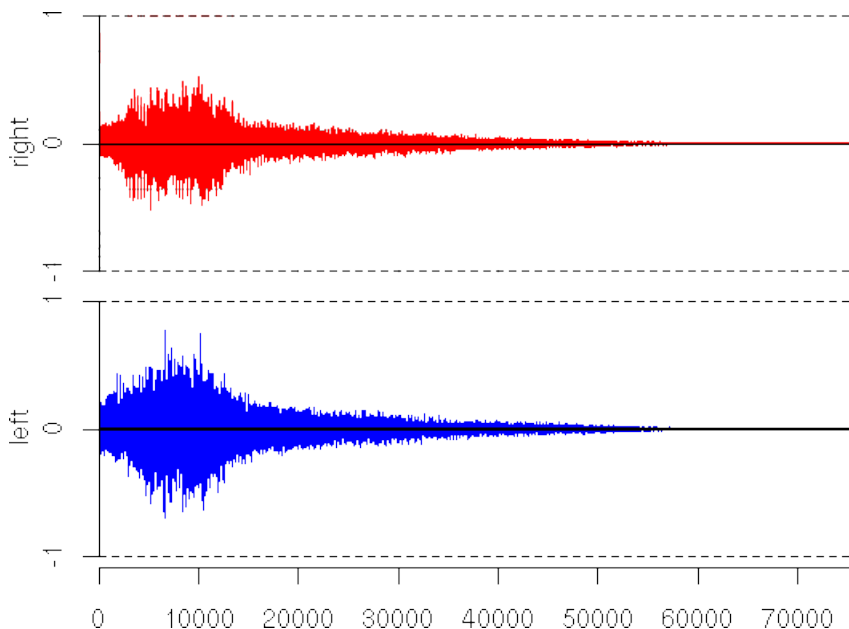


Figura 3: Sonido `sonido.wav`.

Tal como puede verse en la figura 3, tiene una gran cantidad de bajas amplitudes. Éstas son muy sensibles a las alteraciones, porque contienen mucha información en los bits poco significativos. Hacia la muestra 50000 (2 segundos), hay muy pocas muestras con información por arriba del décimo bit.

La resta entre la señal original y la modificada permite evidenciar las alteraciones sufridas. La salida de los siguientes comandos es la figura 4 donde se alteran 10 bits con azar:

```
> original<-stegaSampleAsDouble(sonido)
> # 10 bits de random en el 100% de las muestras
> modificado<-stegaSampleAsDouble(stegaBits(sonido,10,100,-1))
Resumen:
 152574 muestras cambiadas (100%)
 190717 bytes de random
> # Para graficar espectros, se utiliza la biblioteca seewave
> library(seewave)
> spectro(original-modificado,sonido$rate,flim=c(0,2.5))
```

La mancha hacia el final de la señal refleja altos índices de amplitud, ya que estas muestras son muy distintas a las del audio original. Auditivamente se evidencia un ruido muy similar al blanco y muy notorio hacia el final de la señal.

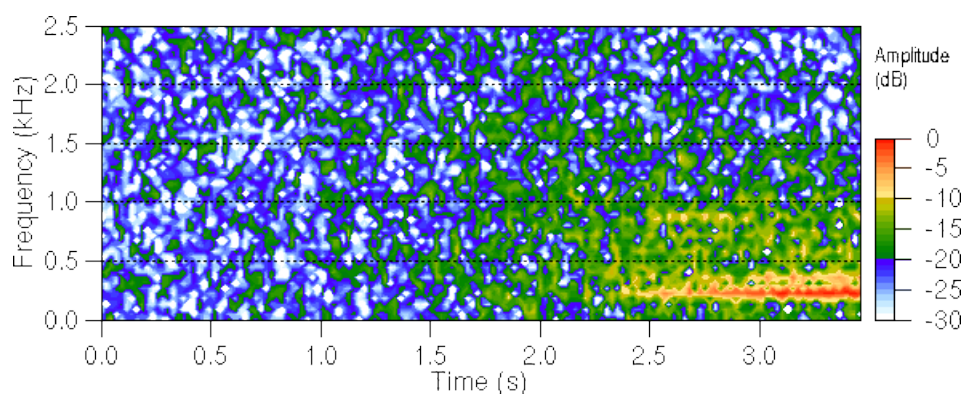


Figura 4: Análisis espectral de la diferencia de audio `sonido.wav` cuando se alteran los últimos 10 bits con azar. (detalle)

3.2. Efecto en la frecuencia

En la gráfica anterior (figura 4) se puede ver que las diferencias más acentuadas están en el espectro de las bajas frecuencias. Esto ocurre porque, aunque el azar afecta a todas las frecuencias, las altas se cancelaron con el audio original.

Este efecto invita a reflexionar sobre cómo afecta a la frecuencia la esteganografía.

El archivo `grave.wav` que se adjunta es un audio de baja frecuencia (150 Hz) en estereo, creado con la función `Sine()` de la biblioteca `sound`³. Al graficar la misma alteración de 10 bits al azar en todas las muestras se obtiene la comparación de la figura 5.

```
> grave<-loadSampleAsBits("grave.wav")
> grave
type      : stereo
```

³generado con el comando: `saveSample(Sine(150,3.46,rate=22050,channels=2),"grave.wav")`

```

rate      : 22050 samples / second
quality   : 16 bits / sample
length    : 76293 samples
R memory  : 610344 bytes
HD memory : 305216 bytes
duration  : 3.46 seconds
> original<-stegaSampleAsDouble(grave)
> # 10 bits de random en el 100% de las muestras.
> modificado<-stegaSampleAsDouble(stegaBits(grave,10,100,-1))
Resumen:
152586 muestras cambiadas (100%)
190732 bytes de random
> # Gráfica del espectro (muy ampliado en amplitud).
> spec(original,22050,type='l',flim=c(0.0,0.6),alim=c(0,0.003))
> spec(modificado,22050,type='l',flim=c(0.0,0.6),alim=c(0,0.003))

```

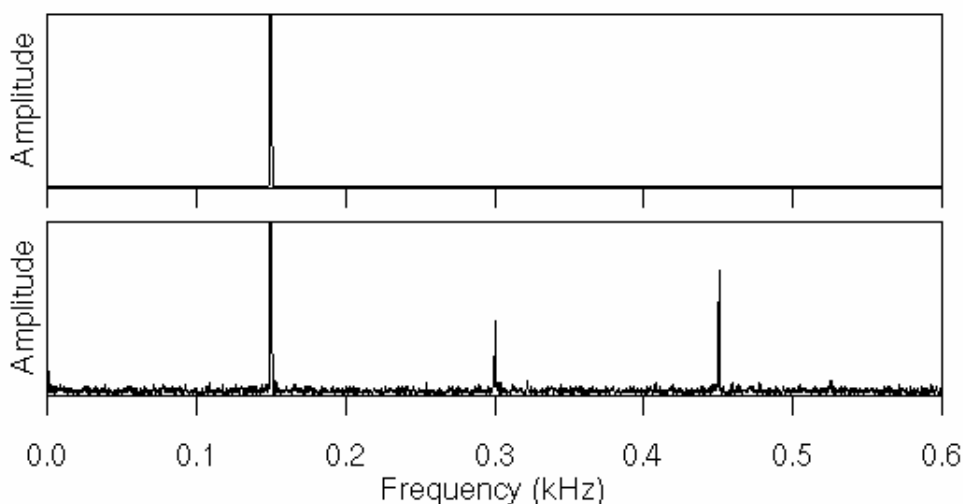


Figura 5: Comparación de espectros. Arriba, un tono de baja frecuencia (150 Hz) puro. Abajo, el mismo alterando los últimos 10 bits con azar. (detalle)

La deformación adicionada parece obvia, pero hay que tener en cuenta que el detalle aumenta el efecto. De todas formas, auditivamente se evidencia con fuerza.

Cuando se realiza la misma operación en un audio alta frecuencia (1000 Hz) la diferencia auditiva es inapreciable con respecto al no manipulado. El archivo en cuestión es `agudo.wav`⁴ y puede apreciarse que no hay cambios desde el punto de vista gráfico en la figura 6:

```

> agudo<-loadSampleAsBits("agudo.wav")
> grave
type      : stereo

```

⁴generado con el comando: `saveSample(Sine(1000,3.46,rate=22050,channels=2),"agudo.wav")`

```

rate      : 22050 samples / second
quality   : 16 bits / sample
length    : 76293 samples
R memory  : 610344 bytes
HD memory : 305216 bytes
duration  : 3.46 seconds
> original<-stegaSampleAsDouble(agudo)
> # 10 bits de random en el 100% de las muestras
> modificado<-stegaSampleAsDouble(stegaBits(agudo,10,100,-1))
Resumen:
152586 muestras cambiadas (100%)
190732 bytes de random
> # Gráfica del espectro (muy ampliado en amplitud).
> spec(original,22050,type='l',flim=c(0.7,1.3),alim=c(0,0.003))
> spec(modificado,22050,type='l',flim=c(0.7,1.3),alim=c(0,0.003))

```

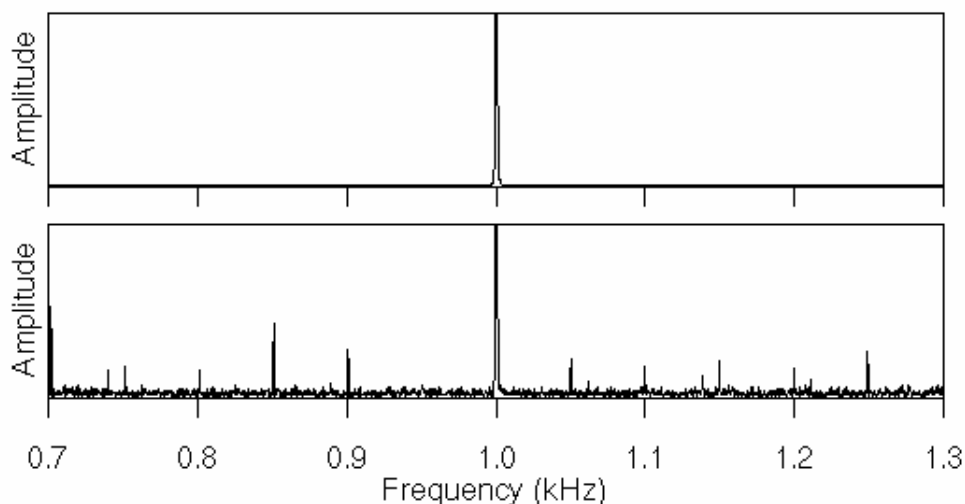


Figura 6: Comparación de espectros. Arriba, un tono de alta frecuencia (1000 Hz) puro. Abajo, el mismo alterando los últimos 10 bits con azar. (detalle)

Ambos resultados se adjuntan bajo los nombres `stegaGrave.wav` y `stegaAgudo.wav`.

4. Discusión y Conclusiones

Las bajas amplitudes parecen ser muy sensibles a la manipulación esteganográfica. Cuando el sonido contenedor disminuye su volumen, el ruido de las alteraciones surge y se evidencia. Una posible buena modificación a esta prueba de concepto puede ser la implementación de una opción para no alterar las muestras cuya amplitud este por debajo de cierto umbral, relacionado éste con la cantidad de bits a alterar.

Por otra parte, las altas frecuencias parecen *tapar* la distorsión. Esto puede deberse a que el tímpano tiene un método de protección ante las altas frecuencias. El mismo

consiste en aumentar la rigidez del tímpano para bajar el volumen (amplitud) del sonido y así evitar lesiones[2]. Esta baja en la amplitud puede estar ocultando las alteraciones efectuadas ⁵.

Se realizaron pruebas complementarias modificando solo un porcentaje de las muestras. Auditivamente las alteraciones son notables. Se escuchan golpes rítmicos de fondo. Esta manifiesta distorsión, sumada a la fuerte baja en la capacidad contenedora del archivo, hacen creer que afectar muestras salteadas carezca de sentido.

Puede concluirse que las altas amplitudes son preferibles para la esteganografía, porque las alteraciones se absorben mejor. Así como las altas frecuencias tienden a disimular al oído humano el ruido producido por las perturbaciones esteganográficas.

5. Anexos de códigos fuente

5.1. steganography.R

```
library(sound)
dyn.load("steganography.so")
stegaBits <- function(sonido,bits=0,porcentaje=100,accion=0) {
  return(as.Sample(
    .C("bits",
      sonido[["sound"]],
      length(sonido[["sound"]]),
      as.integer(bits),
      as.integer(porcentaje),
      as.integer(accion)[[1]],
      sonido$rate,
      sonido$bits
    )
  )
}

loadSampleAsBits <- function(filename,filecheck=TRUE) {
#Copiado, con pequeñas modificaciones, de "sound,loadSample()"
  if (!is.null(class(filename)) && class(filename)!="Sample")
    return(filename)
  if (mode(filename)!="character")
    stop("Argument 'filename' must be a character string.")
  if (substr(filename,nchar(filename)-3,nchar(filename))!=".wav")
    stop("Filename must have the extension .wav.")
  if (filecheck){
    if (file.access(filename)==-1)
      stop("File not found.")
    if (file.access(filename,4)==-1)
      stop("No read permission for this file.")
  }
}
```

⁵esta conclusión fue realizada gracias al invaluable aporte de Licenciada en Terapia Ocupacional Patricia Bello y sus fuertes conocimiento de fisiología humana

```

}

fileR <- file(filename,"rb")

if(readChar(fileR, nchars=4) != 'RIFF')
  stop("File is not RIFF format.")
readBin(fileR, "integer", n = 4, size = 1)
if(readChar(fileR, nchars=4) != 'WAVE')
  stop("File is not WAVE format.")

      readBin(fileR,"integer",n=10,size=1)
channels <- readBin(fileR,"integer",      size=2, endian='little')
rate      <- readBin(fileR,"integer",      size=4, endian='little')
      readBin(fileR,"integer",n= 6,size=1)
bits      <- readBin(fileR,"integer",      size=2, endian='little')
      readBin(fileR,"integer",n= 4,size=1)
Length    <- readBin(fileR,"integer",      size=4, endian='little')
if (bits==8)
  data <- readBin(fileR,"integer",n=Length ,
                  size=1,signed=FALSE, endian='little')
else
  data <- readBin(fileR,"integer",n=Length/2,
                  size=2,signed=TRUE , endian='little')
close(fileR)

if (channels==2)
  dim(data) <- c(channels,length(data)/channels)

return(as.Sample(data,rate,bits))
}

stegaSampleAsDouble <- function (sonido) {
  if (sonido$bits==8)
    sonido$sound <- sonido$sound/128-1
  else
    sonido$sound <- sonido$sound/32768
  return(sonido)
}

```

5.2. steganography.c

```

#include <R.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
void bits(int *sound, int *nsound,

```

```

        unsigned short *bits, int *porcentaje, short *accion)
{
    short buildMask(short unsigned);
    unsigned short operar(unsigned short, short, short);
    int i, cambiar,cambios;
    float k,salto;
    short mask,umask;
    cambiar=(int)((*nsound)*(*porcentaje)/100);
    mask=buildMask(*bits);
    umask=(-1)^mask;
    srand ( time(NULL) );
    if (cambiar!=0){
        salto=(float)(*nsound-1)/(cambiar-1);
        k=0;
        for(i=0; i < *nsound/2; i++)
            if (i==(int)k ) {
                if (sound[2*i] >= 0)
                    sound[2*i]=(int)(operar((short)sound[2*i],mask,*accion));
                else
                    sound[2*i]=(int)(operar(abs((short)sound[2*i]),
                                                mask,*accion))*-1;
                if (sound[2*i+1] >= 0)
                    sound[2*i+1]=(int)(operar((short)sound[2*i+1],mask,*accion));
                else
                    sound[2*i+1]=(int)(operar(abs((short)sound[2*i+1]),
                                                mask,*accion))*-1;
                cambios+=2;
                k+=salto;
            }
    }
    Rprintf("Resumen:\n %i muestras cambiadas (%hd\%)\n %i bytes de"
           ,cambios,(short)(((float)cambios/*nsound))*100),
           (cambios*(bits))/8);
    switch (*accion){
    case 0: Rprintf("ceros\n");
            break;
    case 1: Rprintf("unos\n");
            break;
    default: Rprintf("random\n");
            break;
    }
}
unsigned short operar(unsigned short value, short mask, short accion){
    short umask;
    short azar;

```

```
    umask=(-1)^mask;
    switch (accion){
    case 0: return (value&mask);
        break;
    case 1: return (abs(value|umask));
        break;
    default:
        azar=(short)(rand()%32767);
        return((azar&umask)|(value&mask));
        break;
    }
}
short buildMask(unsigned short value){
    unsigned short c;
    unsigned short ret=0;
    ret--;
    for (c = 1; c <= value; c++)
        ret=ret << 1;
    return(ret);
}
```

Referencias

- [1] Caroline Simonis-Sueur Author Jérôme Sueur, Thierry Aubin. The seewave package. <http://cran.r-project.org/doc/packages/seewave.pdf>.
- [2] Arthur C. Guyton. *Anatomía y fisiología del sistema nervioso*. Editorial Médica Panamericana.
- [3] Matthias Heymann. The sound package. <http://cran.r-project.org/doc/packages/sound.pdf>.
- [4] Kurt Hornik. The R FAQ, 2007. <http://cran.r-project.org/doc/FAQ/R-FAQ.html>.
- [5] Bruce Schneier. *Applied Cryptography*, chapter 1. John Wiley & Sons, Inc., 1996.
- [6] Misha Schwarts. *Transmisión de la información*.
- [7] Peter Wayner. Mimic functions and tractability. Draft. <http://www.funet.fi/pub/crypt/mirrors/dsi/docs/mimic-two.ps.gz>.
- [8] Wikipedia. Esteganografía. Website. <http://es.wikipedia.org/wiki/Esteganograf%C3%ADa>.