

Esteganografia VS Criptografia

Eduardo Ruiz Duarte
beck@math.co.ro

Definiciones

Criptografía:

Rama de la teoría de la información que estudia datos como un objeto matemático con el propósito de encontrar métodos de transmisión seguros de los mismos.

Esteganografía:

Ciencia que estudia las estructuras de mensajes como objeto matemático con el propósito de ocultar estos dentro de la naturaleza de otro mensaje

Un poco mas informal

Criptografía: Cifra datos , si estos son robados es porque se sabe que existen estos datos, aunque no puedan ser visualizados mas que por el receptor verdadero.

Esteganografía: Oculta datos, estos dificilmente pueden ser robados debido a que el mensaje esta oculto en la naturaleza de otro mensaje.

Características de un conjunto de datos cifrado

Sea F un automorfismo (Ya que vamos a suponer que vamos a cifrar de un alfabeto al mismo alfabeto), se dice que F cumple con ser una función criptografía si pasa lo que sigue:

- 1) F es automorfismo por lo tanto tiene inversa
- 2) Confusion (La relación entre la llave y el texto original debe ser muy compleja)
- 3) Difusion (X es estadísticamente independiente de $F(X)$) o sea que las probabilidades son independientes $\text{Prob}(X \text{ y } F(X)) = \text{Prob}(X) \text{ Prob}(F(X))$ o sea que $F(x_1, x_2, \dots, x_n)$ y x_1, x_2, \dots, x_n son eventos totalmente distintos en términos de probabilidad

Pregunta?

A que nos lleva esto ?

Al parecer los datos tienen una estructura facil de reconocer en terminos probabilisticos, que podemos decir de los datos.

Vamos a verlo mas a detalle...

Tests

Veamos que estructura tienen.

primero usaremos una medida muy basica para ver el “promedio de un archivo”:

```
while (read (archivo, &byte, sizeof (char)) > 0)  
{  
    s += byte;  
    c++;  
}  
fprintf (stdout, "Promedio = %.2f\n", s / c);
```

Tests

Archivos ejecutables:

beck@dirichlet:~\$./a.out /bin/mount

Promedio = 85.15

beck@dirichlet:~\$./a.out /bin/sh

Promedio = 91.36

beck@dirichlet:~\$./a.out /bin/cp

Promedio = 93.70

beck@dirichlet:~\$./a.out /bin/lis

Promedio = 97.94

beck@dirichlet:~\$./a.out /bin/uname

Promedio = 82.40

beck@dirichlet:~\$./a.out /bin/ln

Promedio = 90.65

Vemos que el promedio de bytes en ejecutables esta entre 80 y 100 , por lo menos en este espacio que me agarre

Tests

Ahora veamos con un archivo cifrado con AES

```
beck@dirichlet:~$ openssl enc -aes256 -in /etc/services -out  
services.aes
```

```
enter aes-256-cbc encryption password:josejose
```

```
beck@dirichlet:~$ ./a.out services.aes
```

```
Promedio = 127.93
```

```
beck@dirichlet:~$ openssl enc -aes256 -in /etc/services -out  
services.aes
```

```
enter aes-256-cbc encryption password:josejosue
```

```
beck@dirichlet:~$ ./a.out services.aes
```

```
Promedio = 128.11
```

```
beck@dirichlet:~$ openssl enc -aes256 -in /etc/passwd -out passwd.aes
```

```
enter aes-256-cbc encryption password:mipassword
```

```
beck@dirichlet:~$ ./a.out passwd.aes
```

```
Promedio = 128.13
```

Tests

AES cumple con los requerimientos del NIST (National Institute of Standards and Technology)

los cuales son similares a los que mencione (confusion , difusion , prevencion de correlacion)

podemos ver que el promedio siempre esta por ahi de $256/2$, y tenemos un alfabeto de 256 caracteres, o sea todos tienen la misma probabilidad

Esto nos da una caracterizacion importante, los archivos cifrados tienen a ser random
(con las características ya dichas)

Veamos la misma prueba pero ahora sacando datos pseudo aleatorios de un device.

```
beck@dirichlet:~$ dd if=/dev/urandom of=rand_file bs=1024 count=1024
1024+0 records in
1024+0 records out
beck@dirichlet:~$ ./a.out rand_file
Promedio = 127.43
beck@dirichlet:~$
```

Una medida mas seria

Entonces , ya sabemos que son asi , pero bueno , existen metodos para medir la entropia de datos como lo es la siguiente serie, donde $p(x_i)$ aqui es la probabilidad en X vamos a implementarla en C y veamos que hace.

$$H(X) = \sum_{i=1}^n p(x_i) \log_2 \left(\frac{1}{p(x_i)} \right) = - \sum_{i=1}^n p(x_i) \log_2 p(x_i)$$

Mas tests

Cifraremos con mi algoritmo RUIX que (teoricamente) cumple con la difusion , confusion etc... y veamos su comportamiento en funcion con datos pseudo aleatorios

Cifrando con RUIX

Mas adelante veremos el codigo.. ahora veamos su funcionamiento

```
dirichlet@beck ~/steg/ruix $ gcc cook_key.c ruix.c -o ruix
```

```
dirichlet@beck ~/steg/ruix $ ./ruix.exe
```

```
arg1 = infile
```

```
arg2 = outfile
```

```
arg3 = { -d || -e }
```

RUIX

```
dirichlet@beck ~/steg/ruix $ ./ruix.exe /etc/passwd pass.cr -e
```

```
(passphrase) :
```

```
Passphrase is:
```

```
:8a:ef:41:42:e2:2b:71:44
```

```
Expansion is:
```

```
Short Rounds = 44675
```

```
Real Rounds = 13
```

```
s=6069e76b
```

```
t=668fb82d
```

```
u=509051db
```

```
v=548945df
```

```
w=b003d998
```

```
x=ce7033ad
```

```
y=a0f99a74
```

```
z=adba4264
```

```
Encoding/Decoding..
```

```
s=c126685c
```

```
t=63f83669
```

```
u=0a790026
```

```
v=1ec91c54
```

```
w=1cc3bfb9
```

```
x=1bdc7ce6
```

```
y=585e9d99
```

```
z=89f27bf4
```

RUIX

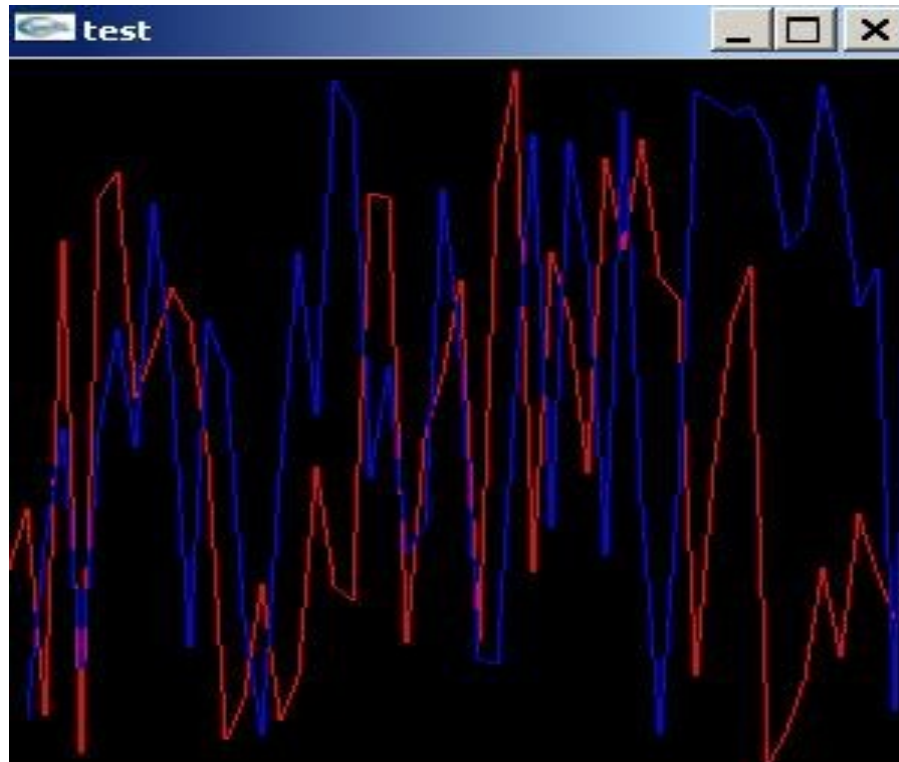
Vamos a ver el comportamiento de un archivo cifrado con RUIX contra datos pseudo aleatorios

```
dirichlet@beck ~/steg/ruix $ ls -al pass.cr  
--wxrw--wt 1 dirichlet None 896 Oct 18 13:45 pass.cr
```

```
dirichlet@beck ~/steg/ruix $ dd if=/dev/urandom of=rand_file  
bs=896 count=1  
1+0 records in  
1+0 records out  
896 bytes (896 B) copied, 0.015 seconds, 59.7 kB/s
```

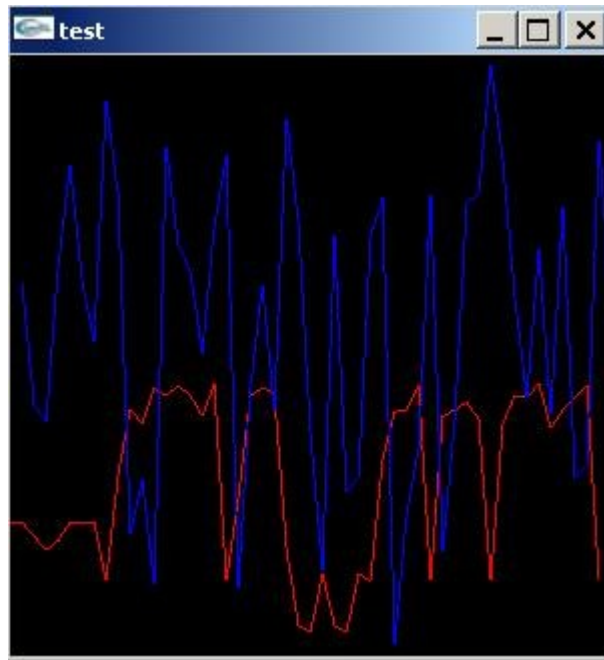
Dibujitos

Grafica de Archivo cifrado con RUIX VS bytes pseudo aleatorios de /dev/urandom



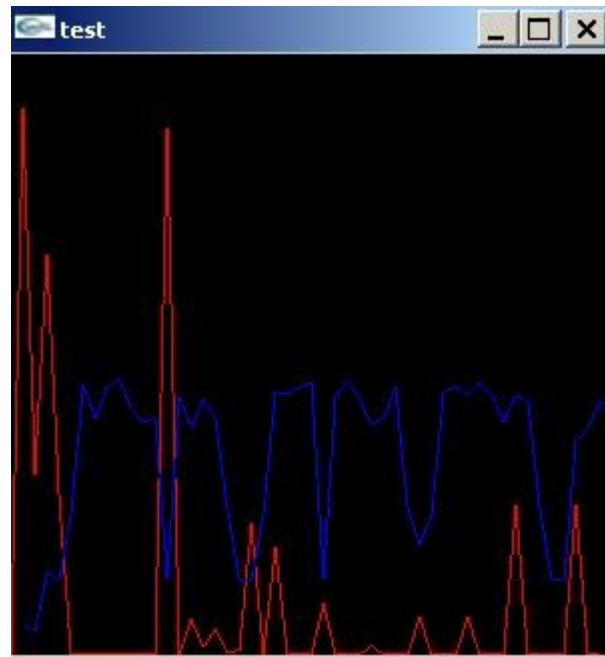
Dibujitos

Grafica de archivo Random VS Text
(/etc/services)



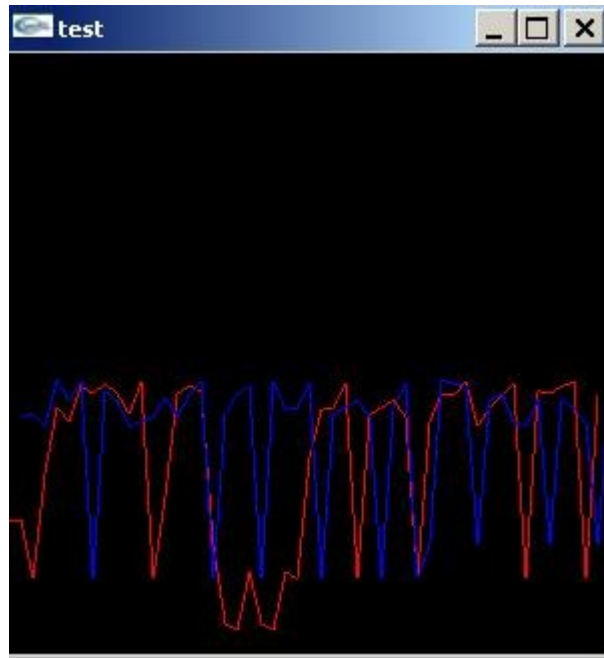
Dibujitos

Grafica de archivo de texto VS binario (/bin/lS)



Dibujitos

Grafica de archivo de texto VS otro archivo de texto



Entropia

Regresando , la funcion de entropia mencionada anteriormente.

Calcula el "desorden", en terminos mas formales, calcula el promedio de bits por byte en un mensaje (bytes en este caso) que se necesitan para poder codificarlo , comprimirlo. Esto nos da una idea muy acertada de la redundancia en el archivo. Asi podemos detectar en internet archivos cifrados de forma segura

Pseudofilosofia

La naturaleza de los mensajes jamas sera aleatoria ya que existe una logica una semantica , una gramatica que hace que exista una secuencia. Asi como en los archivos de sonido , archivos de imagenes etc...Entonces ahora vemos que una desventaja de los archivos cifrados es que es sumamente detectable que algo "secreto viaja por ahi"

Debilidad

Esto facilita la intercepcion de mensajes importantes y la probabilidad de saberlo obviamente es mas grande que si no lo tuviera en sus manos el interceptor existen ataques a DES a RSA (en la implementacion de openssl), entre otras cosas.

Veamos rapidamente la entropia de algunos archivos y de dumps de internet que recopile de un router concurrido

Codigo de entropia

Codigo de implementacion de funcion de entropia
(Frecuencia)

```
int
freq (char *f, int x[256])
{
    int fd, bindex;
    unsigned char byte;
    if ((fd = open (f, O_RDONLY)) < 0)
    {
        printf ("Error opening file\n");
        exit (EXIT_FAILURE);
    }
    stat (f, &arch_info);
    while (read (fd, &byte, sizeof (unsigned char)) > 0)
    {
        bindex = (int) byte;
        x[bindex] = x[bindex] + 1;
    }
    return 0;
}
```

Codigo entropia

Codigo de implementacion de funcion de entropia
(entropia)

```
float
entropia (int f[256])
{
    float x[256];
    float ret_entropia = 0.0;
    int i;
    memset (&x, 0x0, sizeof (x));
    for (i = 0; i < 256; i++)
    {
        x[i] = (float) f[i] / arch_info.st_size;
        if (x[i] > 0.0)
            ret_entropia -= x[i] * log ((double) x[i]) / log ((double) 2.0);
    }
    return ret_entropia;
}
```

Codigo entropia

La funcion que integra esto de manera simple dado el nombre del archivo el cual se quiere calcular la entropia:

```
float  
calc_entropia (char *f)  
{  
    int freq[256];  
    memset (&freq, 0, sizeof (freq));  
    freq (f, freq);  
    return entropia (freq);  
}
```

Pruebas

```
dirichlet@beck ~/steg $ ./entropy.exe /etc/passwd  
5.30
```

```
dirichlet@beck ~/steg $ perl -e "print 'X' x 10000;" > constante  
dirichlet@beck ~/steg $ ./entropy.exe constante  
0.00
```

```
dirichlet@beck ~/steg $ ./entropy.exe random  
8.00
```

```
dirichlet@beck ~/steg $ ./entropy.exe ruix/pass.cr  
7.77
```

Entropia en internet

Ahora dumps de internet de distintos routers y firewalls UNIX a los cuales tengo acceso, recopilando 100 paquetes tcp/udp/icmp en cada uno.
en uno de ellos lo ejecute asi:

```
% gcc captura_paquetes.c -Wall -pedantic -L /opt/csw/lib/ -I  
/opt/csw/include -lpcap -lnsl -lsocket -o captura_paquetes  
% ./captura_paquetes dmfe0 100 > dump3.raw
```

Calculo de entropia de paquetes de red

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump1.raw
```

3.40

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump2.raw
```

3.66

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump3.raw
```

5.41

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump4.raw
```

5.05

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump5.raw
```

4.52

```
dirichlet@beck ~/steg $ ./entropy.exe dump_inet/dump6.raw
```

3.96

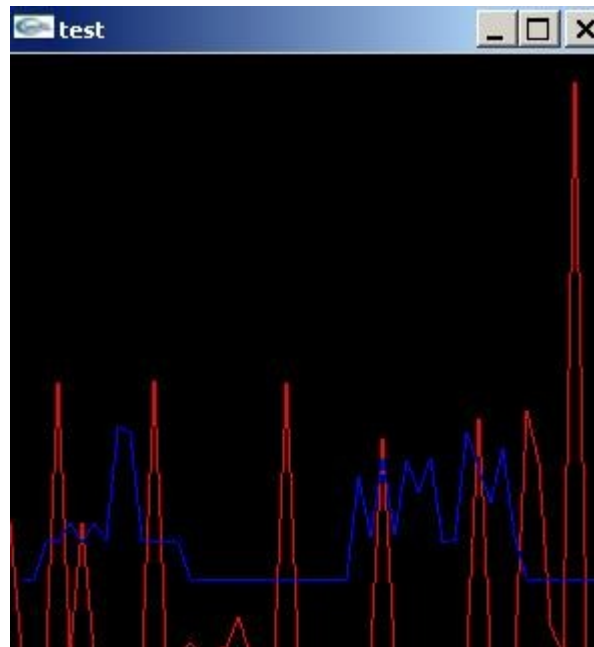
Comparacion

Aqui podemos ver que entre mas tendencia al desorden (los random) se necesitan mas bits por byte para codificar (o sea que aqui podemos ver que un archivo random es dificilmente comprimible) un archivo de texto , se necesitan 5.3 bits por byte (en el caso particular de mi /etc/passwd) , mi RUIX no salio ser tan bueno... no fue tan aleatorio el output pero por ahi va, los dumps de red pues salieron buenos para ser codificables.. en especial el ultimo con 2.43 bits por byte donde cada uno mide:

```
dirichlet@beck ~/steg/dump_inet $ ls -al *.raw
-rw-r--r-- 1 dirichlet None 45494 Oct 18 17:16 dump1.raw
-rw-r--r-- 1 dirichlet None 60924 Oct 18 17:16 dump2.raw
-rw-r--r-- 1 dirichlet None 51943 Oct 18 17:16 dump3.raw
-rw-r--r-- 1 dirichlet None 44380 Oct 18 17:16 dump4.raw
-rw-r--r-- 1 dirichlet None 53500 Oct 18 17:16 dump5.raw
-rw-r--r-- 1 dirichlet None 26831 Oct 18 17:16 dump6.raw
```

Dibujitos

La grafica del dump2 y dump6 se ve asi



Por fin...

Podemos ver entonces que un archivo cifrado , entre mas se acerque a la perfeccion , sera aleatorio el mensaje. y vimos que esto es facilmente detectable.

Ahora veamos que pasa con la esteganografia

BMP's

La esteganografía pretende lo que la criptografía carece.
que es evitar la detección del mensaje.

Veamos una técnica muy sencilla

usando BMP's

Como son los bmp's

los BMPS creo yo que son la estructura mas sencilla de imagenes.

estas pueden ser de 8 bits , 16 bits y 24 bits por pixel.

esto significa que se distribuyen los 3 colores RGB en esos bits

la que me interesa ahorita es la de 24 bits ya que son 3 bytes por pixel.

LSB

El header de BMP me indica el offset en donde comienza la imagen.
la tecnica es muy sencilla , y solo es sustituir el bit menos significativo.
El bit menos significativo es aquel que afecta de menos manera al color.
suponiendo que la maquina es big endian
los pixeles serian de 24 bits

Por ejemplo

010101011101010000101110

El ultimo cero es el bit menos significativo (depende del byte order pero intuitivamente es ese)

si son 24 bits por pixel quiere decir que existen 2^{24} colores diferentes.
o sea 16777216 colores , y si este cambia a una unidad o no va a ser muy notorio que digamos.

Funcionamiento

Ya que son 16 colores basicos eso quiere decir que existen 2^{20} colores por cada color basico

o sea 1048576 tonos de rojos , azules , violetas , verdes , etc...

entonces esta pequenia tecnica mas tecnicamente consiste en lo siguiente:

Rompemos cada byte de un mensaje por bits en un arreglo de 8 chars:

Veamos mi implementacion:

```
typedef struct xbyte_t {  
char byte[8];  
}xbyte;
```

```
xbyte break_byte (unsigned char byte)  
{  
int i;  
xbyte r;  
for (i = 0; i < 8; i++)  
r.bit[7 - i] = ((byte >> i) & 1);  
return r;  
}
```

Funcionamiento

Esto quiere decir que por cada pixel me tomare un bit (Aqui encontramos el primer contra de la esteganografia)
podria tomarlo por cada tono de RGB pero por las prisas lo hice a 24 bits

ahora mapeamos la imagen en un puntero con mmap() y nos ponemos en el offset adecuado para comenzar a escribir en los pixeles
usando el header

Mapeamos y nos ponemos en el offset donde comienzan los pixeles segun el header BMP (offset):

```
BMP *
map_bmp (char *bmp_file)
{
    int r;
    BMP *Img;
    unsigned char *handler;
    bmp_header hdr;
    if ((r = get_bmp_header (bmp_file, &hdr)) < 0)
        return NULL;
    if (stat (bmp_file, &imginfo) < 0)
        return NULL;
    if ((image = open (bmp_file, O_RDWR)) < 0)
        return NULL;
    if ((Img =
        mmap (NULL, imginfo.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, image, 0) == NULL)
        return NULL;

    if (close (image) < 0)
        return NULL;

    Image_bak = Img;
    printf("map_bmp: %c%c\n",Img->r,Img->g);
    handler = (unsigned char *)Img;
    handler = handler + hdr.offset;
    Img = (BMP *)handler;
    return Img;
}
```

Funcionamiento

Verificamos que lo que vas a meter quepa en la imagen

```
if ((fileinfo.st_size*8) > (hdr.pixwidth * hdr.pixheight * 3))  
{  
    fprintf (stderr, "File to insert too large for the BMP\n");  
    close (fd);  
    unmap_bmp ();  
    exit(EXIT_FAILURE);  
}  
fprintf(stderr, "Inserting file..\n");
```

Funcionamiento

Y ya estamos listos para meter cada bit dentro del bit menos significativo de cada PIXEL de 24 bits.

con la siguiente funcion modificamos el bit menos significativo de alguno de los componentes de cada pixel (R , G o B)

unsigned char

modify_lsb (unsigned char byte, unsigned char x)

```
{  
  return (unsigned char)((byte & 0xfe) | (x & 0x01));  
}
```

donde "byte" es el byte del componente del pixel y x es el bit que queremos meterle (0 o 1)

lo unico que hace es lo siguiente

C=(BYTE & 0xFE) , obliga a que C tenga un cero en el octavo bit y conserve intactos los 7 anteriores

P=(x&0x01) , acorta el byte x a 1 bit por si le pusieron alguna otra cosa que no sea 1 o 0

C|P se complementan, para agregar el bit 0 o bit 1 dependiendo del valor de X

Si no entiendes esto preguntale al de alado..asi funcionan las computadoras :p

Observacion

Aqui hay una observacion:

la probabilidad de coincida el bit menos significativo y el bit del mensaje es de $1/2$ a pesar de ser estadisticamente independientes asi que aparte de que no se nota... a veces puede que el pixel se quede intacto. y estadisticamente hablando , por lo menos la mitad de los pixeles estaran intactos.

Ejemplos

Veamos algunos ejemplos con este programa sencillo

Tests

Ahora veamos la entropía entre una imagen con datos embebidos y una que no.

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ./bsteg.exe -i  
chava.bmp /etc/services
```

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ../../entropy.exe  
chava.bmp ; ../../entropy.exe chava-original.bmp
```

7.66

7.66

Conserva "casi" su complejidad .

Funcionamiento

Insertamos los bytes descompuestos en bits distribuido en toda la imagen (vease la variable K como incrementa)

```
while (read (fd, &byte, sizeof (byte)) > 0)  
{  
    nbyte = break_byte (byte);  
    for (i = 0; i < 8; i++) {  
        imghandler[k] = (unsigned char)modify_lsb  
(imghandler[k], nbyte.bit[i]);  
        k+=((imgnfo.st_size)/(filenfo.st_size*8));  
    }  
}
```

Inverso

Para recuperar el tamaño del archivo , lo meti dentro del header de BMP en la parte `hdr.imp_colors` , que generalmente no se usa.... pero si ustedes quisieran ser mas discretos , podrian usar los primeros 32 pixeles para alojar 32 bits que serian el tamaño del archivo

Y para Regresar el archivo , veamos ... es el mismo procedimiento , leemos el header para saber cuanto vamos a recuperar, y comenzamos a movernos por los pixeles , de manera distribuida como en el proceso inicial, pero vamos recopilando bits , y cada 8 pixeles construimos un byte y lo escribimos al output.

Construccion de datos

```
unsigned char  
build_byte (xbyte byte)  
{  
    int i;  
    unsigned char r = 0;  
    for (i = 0; i < 8; i++)  
        r |= ((byte.bit[i] & 1) << (7 - i));  
    return r;  
}
```

Copiemos el bmp original para que no haya chanchuyo

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ cp carshow-orig.bmp carshow.bmp
```

Insertemos el archivo /etc/passwd

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ./bsteg.exe -i carshow.bmp /etc/passwd
```

```
Inserting /etc/passwd into carshow.bmp
```

```
Signature: 424d
```

```
Size of bmp: 507f2
```

```
Reserved : 0
```

```
Offset: 54
```

```
BMP Info: 40
```

```
Pix width in pixels: 0x19e
```

```
Pix height in pixels: 0x109
```

```
Planes: 1
```

```
Pixel size in bits: 24
```

```
Compression: 0
```

```
Size of images with padding: 329660
```

```
Horizontal resolution in pixels per meter: 0
```

```
Vertical resolution in pixels per meter: 0
```

```
Number of colors in image (zero max) : 0
```

```
Number of important colors in image (zero max): 868
```

```
sizeof header 54
```

```
map_bmp: òM
```

```
Inserting file..
```

```
Done!
```

Verificacion

Extraemos el archivo y checamos MD5

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ./bsteg.exe -g  
carshow.bmp nuevo.txt  
Output to nuevo.txt  
map_bmp: òM  
Getting 868 bytes  
Got all bytes (868)!
```

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ md5sum.exe  
/etc/passwd nuevo.txt  
d75b974505c8615e7000bc9ecbd31929 */etc/passwd  
d75b974505c8615e7000bc9ecbd31929 *nuevo.txt
```

Checamos entropia

Entropia de la imagen original y la que tiene los datos

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ../../entropy.exe  
carshow-orig.bmp
```

7.88

```
dirichlet@beck ~/steg/bsteg-0.1/bsteg $ ../../entropy.exe  
carshow.bmp
```

7.88

Es "casi la misma"

JPEG

Al final veremos como funcionan estos programas.. ya voy a acabar , jpeg sera solo una embarrada

JPEG

Veremos esta imagen.



Separacion de componentes

Luminocidad



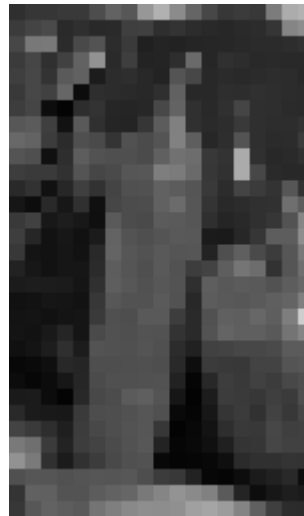
Separacion de componentes

El componente Cb pierde pixeles



Separacion de componentes

Al igual que el Cr



Lossy compression (con perdida de datos)

EL proceso de compresion en JPEG pierde datos.

ya que se le aplica una transformada (DCT) y despues de quantiza (se divide entre otros coeficientes redondeando para ser enteros) ahi es donde se pierden datos.... todo esto se hace con matrices de 8x8.

Ya cuando tienes los coeficientes de la DCT quantizados , puedes ya meter datos en el LSB de cada coeficiente que puedas caracterizar. (ceros por ejemplo) los de alta frecuencia son los mejores , es donde no se nota.

Fin de JPEG

Y así es como se hace en JPEG....

pronto pondré un pdf en mi sitio con más detalles... ya que el motivo de la plática

es llegar a la última slide que son las conclusiones.

Observaciones en Criptografia

- * La criptografia simetrica asegura que nadie pueda ver la informacion si este es un buen algoritmo
- * Pero es tan bueno que es detectable
- * La criptografia es recomendable que sea software libre (algoritmos libres?) para que se fortalezcan los pasos , y mas cuando se usa teoria de anillos y/o grupos y/o campos finitos para perfeccionar las operaciones de cada estructura algebraica .
- * El algoritmo con el que fue cifrado un archivo puede ser publico sin repercutir en la seguridad del archivo cifrado ya que este depende de una contraseña

Observaciones en esteganografía

- La esteganografía asegura que nadie pueda detectar información oculta si es un buen algoritmo
- Requiere más espacio poder meter un archivo en otro
- Se podría cifrar y meter un archivo dentro de otro y el cifrado no repercute con la entropía
- Es peligroso que el algoritmo con el que fue incrustado un dato dentro de otro sea público
ya que este no depende de una llave, por eso
- por lo anterior es sumamente conveniente cifrar los datos antes de incrustarlos.
- Existen ataques de probabilidad

Ejemplos

Veamos algunos ejemplos de esto con cygwin y discutir un poco informalmente sobre fractales y este tema.

FIN

Gracias

Eduardo Ruiz Duarte

Slides , programas etc.. en:

<http://badc0ded.org.ar/steg/>