Criptografía y seguridad informática

Práctica 1

Grupo 4

Mario Valdemaro García Roque
Alberto Cabello Álvarez

Consideraciones sobre la entrega y el código

La entrega se encuentra estructurada en subdirectorios con el contenido de la práctica. En **src** se encuentran los fuentes principales para la creación de cada programa y los ficheros para crear el filtro.

Además, se encuentran los directorios **includes** donde están todos los ficheros .h, que son los de la librería ya mencionada y los necesarios para crear el filtro, y los directorios **obj** donde se compilan todos los programas (una vez usados se borran automáticamente) y el directorio **doc** donde se encuentra este documento.

Para compilar, utilizar make sobre el directorio raíz. En ese directorio se generarán todos los ejecutables necesarios. Estos son los ejecutables propios de cada programa más un filtro programado en flex para transformar los ficheros de entrada al formato deseado (todo letras minúsculas sin caracteres extraños o espacios).

Cabe mencionar que este filtro no es necesario ejecutarlo a mano. Cada programa que recibe texto de fichero tiene una llamada al filtro con system para limpiar el texto de entrada. Esto creará un nuevo fichero del tipo "Fx" donde x es el nombre del fichero original. Por ejemplo, en nuestras pruebas, utilizando el man de gcc (man gcc > gcc.txt) se crea un "Fgcc.txt" al llamar a un programa con ese fichero como parámetro.

El método de ejecución de cada programa corresponde al método indicado en el guión de la práctica, utilizando getopt para el control de parámetros. Se puede consultar este método ejecutando sin parámetros o con parámetros incorrectos, en cuyo caso se detendrá y mostrará la ejecución correcta. Esto es cierto para todos salvo para el programa de criptoanálisis, que tiene un flag, -nl (no limit), que al emplearlo no parará la ejecución para tamaños de clave superiores a 32, que con el método estadístico usado resulta lento para tamaños de clave superiores (Consultar apartado correspondiente para más información).

En el makefile se proporcionan targets como ejemplos de entrada para cada programa, para testeo rápido de los mismos, del tipo:

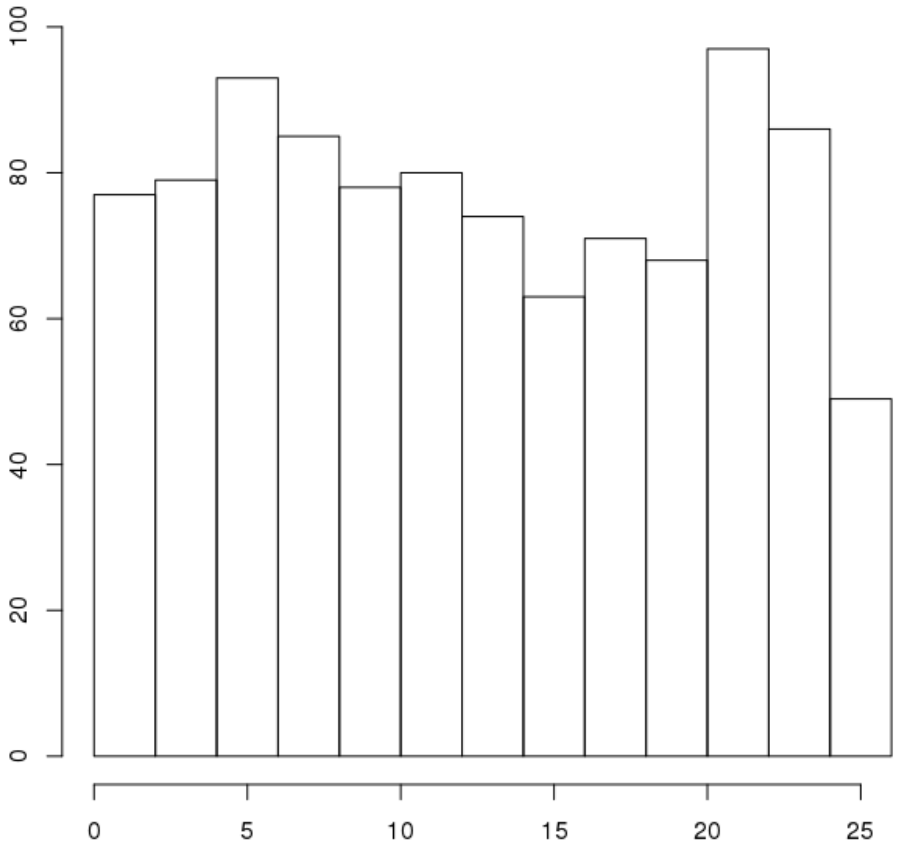
```
testafin:
	./afin -C -m 26 -a 17 -b 24 -i gcc.txt -o afinc.txt
	./afin -D -m 26 -a 17 -b 24 -i afinc.txt -o afind.txt
```

1) Seguridad Perfecta
    - Comprobación empírica de la seguridad perfecta del método afín.
2) Implementación del DES
    - Programación del DES.
3) Principios de diseño del DES.
    - El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC) del DES
    - Estudio de la no linealidad de las S-boxes del DES
4) Principios de diseño del AES
    - El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC) del AES
    - Estudio de la no linealidad de las S-boxes del AES

Comprobación empírica de la seguridad perfecta del método afín:

Para probar este concepto nos creamos 2 funciones de cifrado que usan el concepto de afín pero con la particularidad de que la "a" y la "b" de cada letra que se cifra es aleatoria uniformemente y no uniformemente dependiendo de la función que se use.

Para generar la función uniforme sencillamente hemos usado la función rand de C ya que esta nos devuelve números aleatorios distribuidos uniformemente, como podemos observar en la siguiente gráfica.



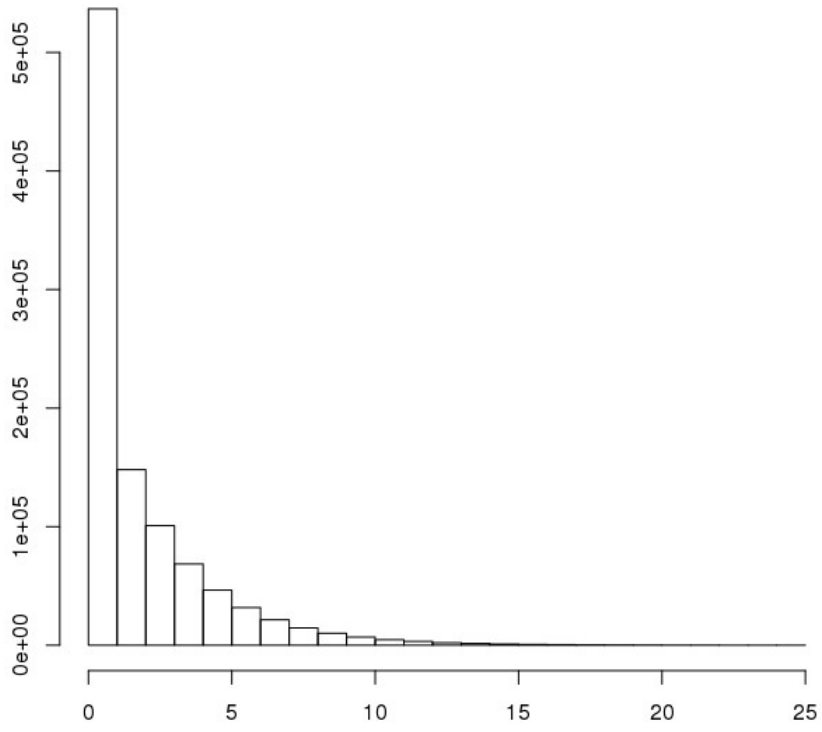
Para generar una distribución no uniforme nos hemos creado al función "método" que tiene el siguiente código:

```
int metodo(int width){

     double rate=width*1.0;
```

```
        int num=width;
        rate/=10.0;
        while(num>=width){
                num=rand();
                num=-rate*log(num*1.0/INT_MAX);
        }
        return num;
}
```

Lo que hace esta función es que siguiendo la función inversa de la función de distribución de la exponencial nos genera una valor aleatorio según esta distribución, como muestra la siguiente gráfica.



Y por ultimo se puede comprobar los resultados del programa ejecutándolo, como muestra ponemos los resultados de uniforme y no uniforme las primeras letras de la "a" donde la a tiene una probabilidad de:

Pp(a)=0.072319

No uniforme

Pp(a|a) = 0.587013	Pp(a|b) = 0.500478	Pp(a|c) = 0.286869	Pp(a|d) = 0.200053
Pp(a|e) = 0.105360	Pp(a|f) = 0.075573	Pp(a|g) = 0.058976	Pp(a|h) = 0.041992
Pp(a|i) = 0.023573	Pp(a|j) = 0.020279	Pp(a|k) = 0.016220	

Uniforme

Pp(a|a) = 0.072084	Pp(a|b) = 0.072562	Pp(a|c) = 0.072835	Pp(a|d) = 0.072712
Pp(a|e) = 0.074932	Pp(a|f) = 0.072412	Pp(a|g) = 0.071338	Pp(a|h)=0.072842
Pp(a|i) = 0.071055	Pp(a|j) = 0.075407	Pp(a|k) = 0.072734	

Programación del DES.

En este apartado se ha realizado la programación del algoritmo de cifrado DES, para el modo de operación ECB.

El tipo de dato utilizado como base para este desarrollo ha sido enteros de 64 bits, (uint64_t en c). Hemos considerado esta opción preferible frente a usar arrays de chars porque nos parecía más limpio a la hora de realizar desplazamientos y otras operaciones a nivel de bit sobre una sola variable.

Como punto de partida para programar este algoritmo, hemos comenzado por cada pieza independiente que se usará como base para el DES. Estas funciones son la aplicación de una permutación a modo de tabla de shorts sobre un número de 64 bits, el paso de un valor por cada una de las sboxes y el algoritmo de generación de las 16 subclaves. Además, como auxiliar, se ha usado una función de rotación de bits para generar claves. Estos métodos tienen las siguientes cabeceras.

```
uint64_t permutacion(uint64_t entrada, const unsigned short *tabla, int sizeEntrada, int sizeTabla)

uint64_t sbox(uint64_t entrada)

uint64_t rotate(uint64_t x, int rotation, int size, uint64_t mask)

uint64_t* generarClaves(uint64_t clave)
```

Una vez que se disponen de estas piezas, se combinan de manera adecuada para programar el cifrado Feistel en sí, la base del DES. Como sabemos, este cifrado es idéntico al descifrar, con la salvedad del orden en el que se aplican las claves, de tal manera que usamos solo una función, con un flag que indique si se cifra o descifra, para invertir el orden de las claves.

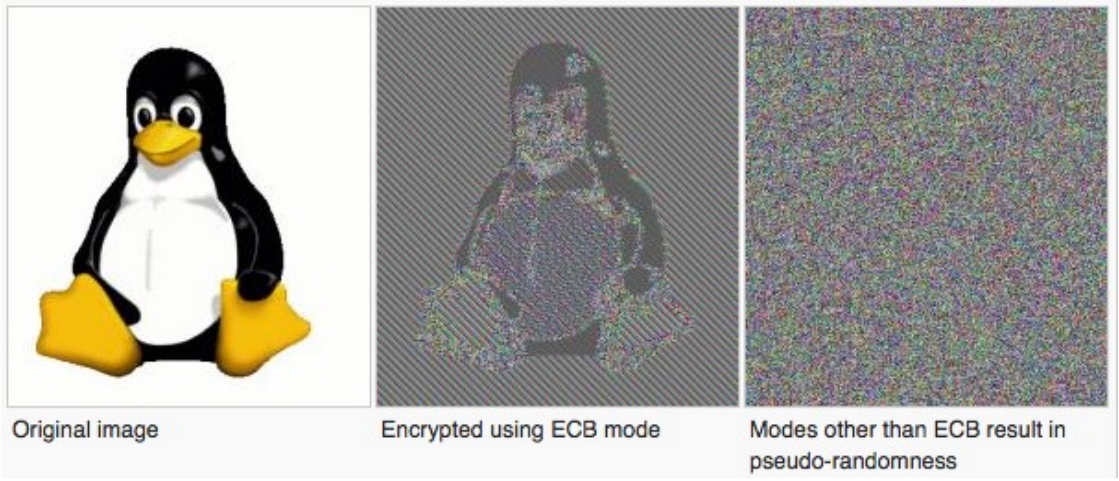
El programa que ejecuta este algoritmo se encuentra en maindes.c, y se puede ejecutar con los argumentos usuales desde el makefile. Las claves se introducen al programa como parámetro en hexadecimal.

El método de operación es el ECB, sencillamente, se leen en bloques de 64 bits, se cifran y se escriben en el descriptor de fichero de salida.

Como se leen los ficheros y se escriben en binario, este método puede cifrar sin mayor dificultad todo tipo de ficheros, como pdf's o imágenes.

Mencionar que este modo de operación resulta inseguro para ficheros grandes, especialmente si tienen muchas repeticiones, como el caso de imágenes, porque para una entrada al cifrado igual, la salida es idéntica.

Este problema resulta aparente al cifrar imágenes. Hemos probado a mostrar este comportamiento, pero por cuestiones del formato de las cabeceras de las imágenes, no podemos abrir imágenes cifradas en sí, aunque, por supuesto, al descifrarlas, sí que es posible, y coinciden con la imagen original. En cualquier caso, los resultados serían del siguiente tipo:



Como detalle, el manejo del padding para el final del cifrado es muy simple, como en fread se leen en bloques de 64 bits y se guardan en la variable correspondiente. Para el momento en el que no se leen exactamente 64 bits, se sobreescribe la variable, y se mantienen algunos valores de la lectura anterior. Hemos hecho comprobaciones con pdf's, archivos comprimidos y ejecutables, y no hemos encontrado que esto fuese un problema en ningún caso.

El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC) del DES

SAC y BIC son dos de los tres principios de diseño del DES.

El criterio SAC postula que un bit de la salida debe cambiar si complementamos un bit de la entrada con una probabilidad de 1/2. Es decir, la probabilidad de que un bit en concreto de la salida tenga valor 1 es la misma que la probabilidad de que ese bit tenga valor 0.

Para probar este concepto hemos realizado un programa que hace lo siguiente:

Genera unas entradas aleatorias, complementa uno de los bits (este proceso se repite con los 6 bits) y las inserta en la función sbox, que hace el proceso de sustitución . Una vez obtenida la salida comprobamos por cada bit el numero de 1's que hemos obtenido y por ultimo mostramos la probabilidad de que haya un 1 en un bit en concreto, lo hacemos para los 32 bits, siendo la salida del programa la siguiente.

```
bit 0 300223 cambios (50.037 %)

bit 1 300686 cambios (50.114 %)

bit 2 299965 cambios (49.994 %)

...

bit 29 298956 cambios (49.826 %)

bit 30 300423 cambios (50.070 %)

bit 31 299732 cambios (49.955 %)
```

BIC nos dice que cualquier par de bits debe cambiar independientemente cuando un bit de la entrada se complementa. Para probar este concepto tenemos que calcular la probabilidad que tienen 2 bits concretos de ser 1, por ejemplo los bit 1 y 2, y después comprobar la probabilidad de que esos 2 bits sean 1 a la vez, si esta ultima probabilidad es la misma que el producto de las probabilidades de ambas de ser 1 entonces consideraremos que el criterio BIC se cumple.

Para comprobar este concepto hemos hecho un programa que hace lo siguiente: genera todos los posibles valores de entrada de las sbox y complementa uno de los bits (en total 64 entradas), después genera las salidas, y con las salidas comprobamos para 2 bits en concreto la probabilidad de que el bit 1 sea 1, la probabilidad de que el bit 2 sea 1 y

la probabilidad de que los bits 1 y 2 sean 1 y 1 a la vez. Por ultimo comprobamos que la probabilidad de que los bits 1 y 2 sean 1 es igual a la probabilidad de que el bit 1 sea uno por la probabilidad de que el bit 2 sea 1. Como ejemplo ponemos la salida obtenida de probar este concepto para la ultima sbox permutando el primer bit, probando para todas las posibles combinaciones de bits:

bits 01, p(01)=0.250000 == p(0)*p(1):0.250000=-> p(0):0.500000 p(1):0.500000

bits 02, p(02)=0.250000 == p(0)*p(2):0.250000=-> p(0):0.500000 p(2):0.500000

bits 03, p(03)=0.250000 == p(0)*p(3):0.250000=-> p(0):0.500000 p(3):0.500000

bits 10, p(10)=0.250000 == p(1)*p(0):0.250000=-> p(1):0.500000 p(0):0.500000

bits 12, p(12)=0.250000 == p(1)*p(2):0.250000=-> p(1):0.500000 p(2):0.500000

bits 13, p(13)=0.250000 == p(1)*p(3):0.250000=-> p(1):0.500000 p(3):0.500000

bits 20, p(20)=0.250000 == p(2)*p(0):0.250000=-> p(2):0.500000 p(0):0.500000

bits 21, p(21)=0.250000 == p(2)*p(1):0.250000=-> p(2):0.500000 p(1):0.500000

bits 23, p(23)=0.250000 == p(2)*p(3):0.250000=-> p(2):0.500000 p(3):0.500000

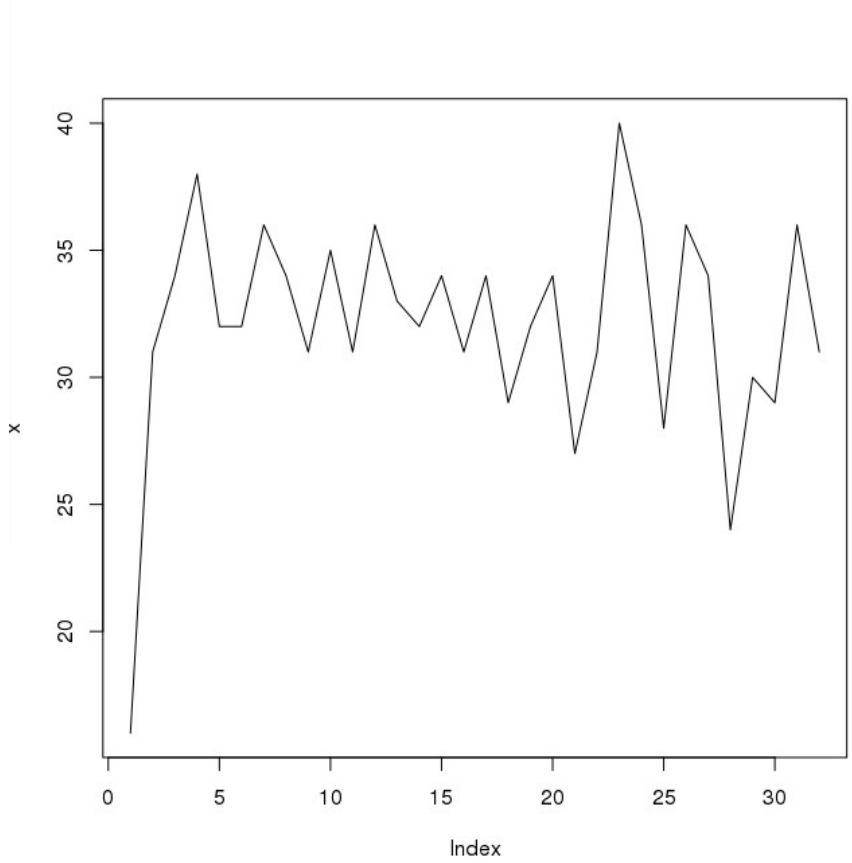
bits 30, p(30)=0.250000 == p(3)*p(0):0.250000=-> p(3):0.500000 p(0):0.500000

bits 31, p(31)=0.250000 == p(3)*p(1):0.250000=-> p(3):0.500000 p(1):0.500000

bits 32, p(32)=0.250000 == p(3)*p(2):0.250000=-> p(3):0.500000 p(2):0.500000

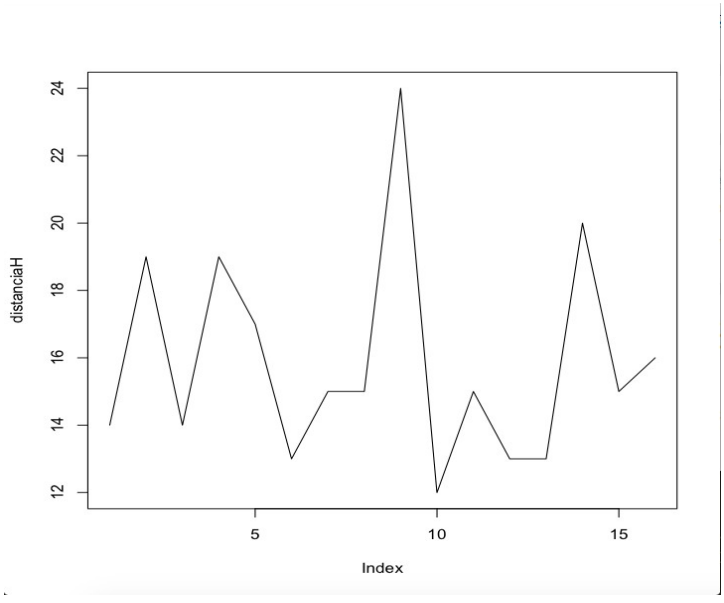
Como vemos el criterio BIC se cumple.

Adicionalmente hemos probado cual es la distancia de Hamming de la salida con la entrada original en función del numero de rondas que tiene el DES. Para ello hemos creado un pequeño programa que usa la función del DES, pudiéndole cambiar el numero de rondas que hace, y que nos devuelve el numero de cambios que ha observado en función del numero de rondas. Generamos unas 32 salidas distintas y observamos los siguientes resultados, donde el eje y representa el numero de cambios y el x el numero de rondas.



Como se puede observar en general se cambian un número mas o menos constante de bits (mas o menos 32) salvo al principio que se cambian muy pocos bits (menos de 20 de 64 posibles). Según hemos leído esta es una de las debilidades de las cajas del DES frente a las del AES ya que se necesitan varias rondas hasta que se obtiene un número aceptable de cambios en la entrada.

También lo planteamos de una segunda manera. Comprobar el número de bits que cambian únicamente para la parte de la función f del cifrado, comparando la entrada previa a la primera permutación y la salida antes de el xor con la mitad izquierda del state (64-32 bits).



Estudio de la no linealidad de las S-boxes del DES

En este apartado, comprobaremos que se cumple el criterio de no linealidad para las S-boxes del DES. Comprobaremos que se cumple que

```
f(a xor b) != f(a) xor f(b)
```

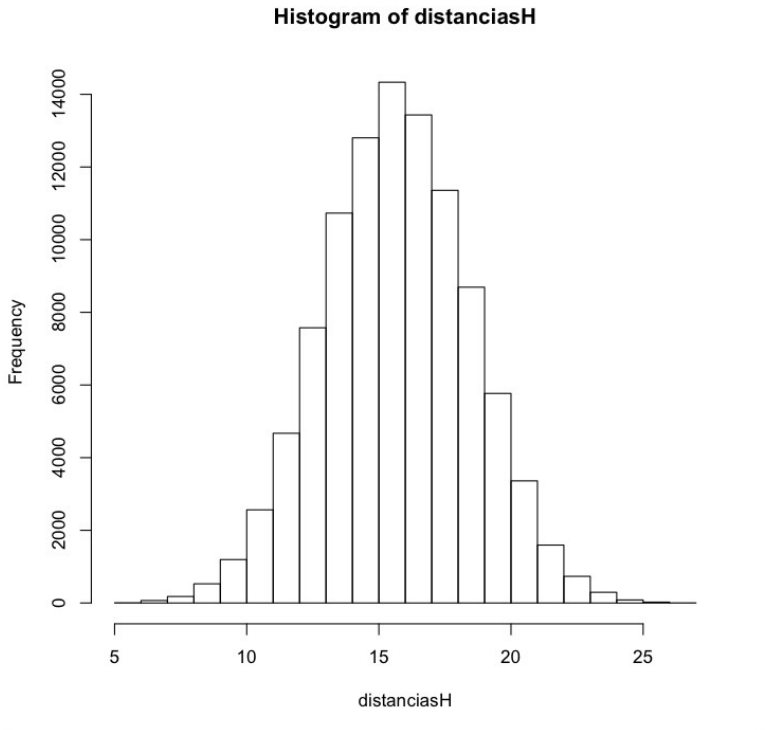
El método que hemos considerado para esto es, dados dos números aleatorios de 48 bits, a, b y c = a xor b, pasarlos por todas las sboxes.

De esta manera, obtenemos f(a xor b), f(a) y f(b). La manera de comparar que hemos utilizado es la distancia de Hamming entre f(a xor b) y f(a) xor f(b). Obtenemos esta distancia para 100k valores de a y b y representamos los datos obtenidos en un histograma. La muestra tiene los siguientes valores estadísticos:

```
> summary(distanciasH)

Min. 1st Qu.  Median   Mean    3rd Qu. Max.

5.00   14.00   16.00   16.19   18.00   27.00
```



Como cabe esperar razonable, difieren de media en 16 bits, la mitad de 32, que es la salida de las cajas. Según esto, podemos decir son suficiente confianza, que el criterio de no linealidad se cumple.

El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC) del AES

Criterio SAC

Para probar los conceptos SAC y BIC hemos creado el programa indicado en la memoria, que usa 2 funciones "sac" y "bic" y que prueba los conceptos.

La función sac genera todas las posibles entradas de las sbox, complementa uno de los bits y la inserta en la función de sustitución de sbox que se nos indique (la directa o la inversa), obtenemos el resultado y comprobamos cual es el valor de los 8 bits de la salida si 1 o 0, si es uno aumentamos el contador y por ultimo calculamos la probabilidad de ser uno de cada uno de los 8 bits.

Como ejemplo obtenemos los siguientes

bit 0 1024 1's (50.000 %)

bit 1 1024 1's (50.000 %)

bit 2 1024 1's (50.000 %)

bit 3 1024 1's (50.000 %)

bit 4 1024 1's (50.000 %)

bit 5 1024 1's (50.000 %)

bit 6 1024 1's (50.000 %)

bit 7 1024 1's (50.000 %)

Criterio BIC

La función que genera este test crea todas las posibles entradas de las sboxes, complementa uno de los bits y obtiene el valor de salida de la sbox correspondiente a esa entrada en función de la sbox que estemos probando (directa o inversa). Después como en el código del DES obtenemos las probabilidades de que 2 bits sean 1 y 1 a la vez y lo comparamos con el producto de la probabilidad de que cada bit sea 1, como muestra la siguiente salida del programa:

bits 01, p(01)=0.250000 == p(0)*p(1):0.250000=-> p(0):0.500000 p(1):0.500000

bits 02, p(02)=0.250000 == p(0)*p(2):0.250000=-> p(0):0.500000 p(2):0.500000

bits 03, p(03)=0.250000 == p(0)*p(3):0.250000=-> p(0):0.500000 p(3):0.500000

bits 04, p(04)=0.250000 == p(0)*p(4):0.250000=-> p(0):0.500000 p(4):0.500000

bits 05, p(05)=0.250000 == p(0)*p(5):0.250000=-> p(0):0.500000 p(5):0.500000

bits 06, p(06)=0.250000 == p(0)*p(6):0.250000=-> p(0):0.500000 p(6):0.500000

bits 07, p(07)=0.250000 == p(0)*p(7):0.250000=-> p(0):0.500000 p(7):0.500000

...

bits 74, p(74)=0.250000 == p(7)*p(4):0.250000=-> p(7):0.500000 p(4):0.500000

bits 75, p(75)=0.250000 == p(7)*p(5):0.250000=-> p(7):0.500000 p(5):0.500000

bits 76, p(76)=0.250000 == p(7)*p(6):0.250000=-> p(7):0.500000 p(6):0.500000

Como apuntes curiosos que tenemos en este programa:

Hemos cambiado las sboxes que se suministraban con la práctica, tomando directamente el valor en hexadecimal para manejar de una manera más sencilla estos valores y no tener que transformar de string a hexadecimal para operar.

En C la forma de indexar [][] es igual que indexar [] si el numero es el adecuado, por lo que en la función de sustitución de sbox en lugar de separar el byte en los 4 bytes mas significativos y los 4 menos hemos puesto directamente [entrada] ya que obtenemos el mismo resultado.

Estudio de la no linealidad de las S-boxes del AES

En este apartado, comprobaremos que se cumple el criterio de no linealidad para las S-boxes del AES. Comprobaremos que se cumple que

```
f(a xor b) != f(a) xor f(b)
```

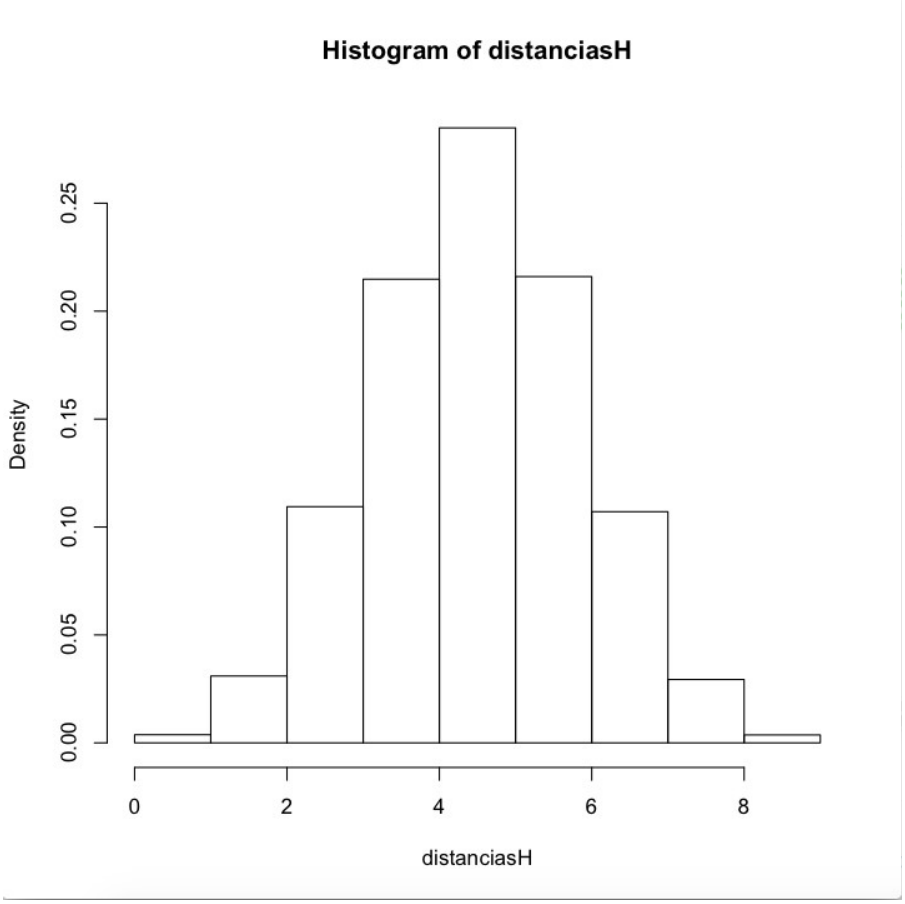
El método que hemos considerado para esto es equivalente al método para el DES, dados dos números aleatorios de 8 bits, a, b y c = a xor b, pasarlos por todas las sboxes.

De esta manera, obtenemos f(a xor b), f(a) y f(b). La manera de comparar que hemos utilizado es la distancia de Hamming entre f(a xor b) y f(a) xor f(b). Obtenemos esta distancia para 100k valores de a y b y representamos los datos obtenidos en un histograma. La muestra tiene los siguientes valores estadísticos:

```
> summary(distanciasH)

Min. 1st Qu.  Median    Mean 3rd Qu.    Max.

0.000   3.000   4.000   3.991   5.000   8.000
```



En este caso sí que obtenemos valores para la distancia de Hamming de 0, en algunas ocasiones. Para la muestra de 100k, aparecen un 0.378% de veces en las que en efecto,

son iguales esos dos valores. Ahora bien, estamos considerando solo la salida de una sola sbox, sin realizar el resto de operaciones correspondientes del byteSub en AES, y solo lo aplicamos sobre un número de 8 bits, en lugar del state completo, así que es razonable y estadísticamente, no relevante, que en pocas ocasiones encontremos distancia de Hamming de 0.