

Práctica 2: Seguridad Perfecta y Criptografía Simétrica

Introducción	2
1. Seguridad Perfecta	2
Comprobación empírica de la Seguridad Perfecta del cifrado por desplazamiento:	2
2. Implementación del DES	4
Programación del DES	4
Programación del Triple DES (OPCIONAL)	6
3. Principios de diseño del DES	7
Estudio de la no linealidad de las S-boxes del DES	7
Estudio del Efecto de Avalancha	8
El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC)	9
Construcción de S-boxes propias	12
4. Principios de diseño del AES	13
Estudio de la no linealidad de las S-boxes del AES	13
Generación de las S-boxes AES	14

Introducción

En esta segunda práctica vamos a trabajar más en detalle con algoritmos y propiedades de criptosistemas más elaborados que los de la práctica anterior.

Empezaremos comprobando empíricamente la seguridad perfecta del cifrado por desplazamiento dependiendo de la uniformidad al elegir la clave o no.

Posteriormente, deberemos elaborar el sistema DES para encriptar, así como su versión triple como ejercicio opcional que hemos implementado. Una vez codificado el programa que cifra en DES, deberemos utilizarlo para comprobar sus propiedades, tales como la no linealidad de las S-boxes, efecto avalancha y los criterios BIC y SAC de independencia de cambios en los bits. Tras esto, elaboraremos nuestras propias cajas de sustitución, generándolas aleatoriamente hasta que se cumplan las propiedades descritas.

Para terminar, trabajaremos con conceptos del sistema de encriptación AES, comprobando también la no linealidad de sus cajas de sustitución y construyéndolas nosotros mismos utilizando el algoritmo de generación visto en clase.

1. Seguridad Perfecta

a. Comprobación empírica de la Seguridad Perfecta del cifrado por desplazamiento:

La seguridad perfecta consiste en que se cumpla lo siguiente:

$$\forall x \in P \quad \forall y \in C \quad p(x|y) = p(x)$$

Es decir, que el conocimiento del texto cifrado no proporcione ninguna información acerca del mensaje en claro.

Para implementar las pruebas hemos diseñado las siguientes distribuciones de probabilidad como vemos en el siguiente fragmento de código:

```
if (equi == 1) {  
    k = (rand() % M);  
} else {  
    k = (int) randn(13, 1) % M;  
}
```

El primer caso se da si ponemos el flag -P, tras lo cual generamos una clave aleatoria y uniforme entre 0 y 25 (dentro de nuestro alfabeto).

En caso contrario, hemos decidido generar, de forma arbitraria, un valor entero determinado por una distribución normal de media 13 y desviación típica 1.

Para hacer las pruebas hemos realizado los comandos siguientes:

```
Cripto/P2/g05$ ./seg-perf -P -i quijote.txt -o quijoteP.txt
Cripto/P2/g05$ ./seg-perf -I -i quijote.txt -o quijoteI.txt
Cripto/P2/g05$ ./seg-perf -P -i QUIJOTE.txt -o QUIJOTEP.txt
Cripto/P2/g05$ ./seg-perf -I -i QUIJOTE.txt -o QUIJOTEI.txt
```

Hemos decidido usar ficheros con un texto considerable, pues necesitamos un muestreo grande si queremos obtener resultados interesantes.

En el fichero *quijote.txt* almacenamos en texto plano el primer capítulo del quijote, mientras que en *QUIJOTE.txt* tenemos el Quijote entero.

- quijoteP.txt

Aquí podemos ver la probabilidad de cada caracter en el texto original.

Para el caso de la A, vemos que las probabilidades de cada $P(A|*)$ se acercan a las de $P(A)$, pero debido a que el muestreo no parece ser suficiente, sigue habiendo un error de alrededor del 2% en probabilidades con respecto a las esperadas.

```
Pp(A) = 0.152213
Pp(B) = 0.021118
Pp(C) = 0.037252
Pp(D) = 0.051133
Pp(E) = 0.131095
Pp(F) = 0.005932
Pp(G) = 0.010203
Pp(H) = 0.011152
Pp(I) = 0.054692
Pp(J) = 0.004152
Pp(K) = 0.000000
Pp(L) = 0.059556
Pp(M) = 0.028354
Pp(N) = 0.067149
Pp(O) = 0.094199
Pp(P) = 0.018389
Pp(Q) = 0.016728
Pp(R) = 0.061336
Pp(S) = 0.067861
Pp(T) = 0.032507
Pp(U) = 0.047455
Pp(V) = 0.008067
Pp(W) = 0.000000
Pp(X) = 0.000356
Pp(Y) = 0.015067
Pp(Z) = 0.004034
Pp(A|A) = 0.142402 Pp(A|B) = 0.138186 Pp(A|C) = 0.157735 Pp(A|D) = 0.149400 Pp(A|E) = 0.141858 Pp(A|F) = 0.150560 Pp(A|G) =
0.158613 Pp(A|H) = 0.152791 Pp(A|I) = 0.153693 Pp(A|J) = 0.139981 Pp(A|K) = 0.149590 Pp(A|L) = 0.145689 Pp(A|M) = 0.150153 Pp(
A|N) = 0.158714 Pp(A|O) = 0.160734 Pp(A|P) = 0.174168 Pp(A|Q) = 0.153535 Pp(A|R) = 0.154672 Pp(A|S) = 0.151450 Pp(A|T) = 0.160955
Pp(A|U) = 0.164271 Pp(A|V) = 0.163597 Pp(A|W) = 0.132780 Pp(A|X) = 0.139939 Pp(A|Y) = 0.159229 Pp(A|Z) = 0.153292
```

- quijotel.txt

Para la distribución normal, vemos que las probabilidades se distribuyen menos equitativamente como esperábamos.

```
Pp(A|A) = 0.000000 Pp(A|B) = 0.000000 Pp(A|C) = 0.000000 Pp(A|D) = 0.000000 Pp(A|E) = 0.000000 Pp(A|F) = 0.000000 Pp(A|G) =
0.000000 Pp(A|H) = 0.000000 Pp(A|I) = 0.000000 Pp(A|J) = 0.030928 Pp(A|K) = 0.371542 Pp(A|L) = 0.739612 Pp(A|M) = 0.881447 Pp(
A|N) = 0.773239 Pp(A|O) = 0.413376 Pp(A|P) = 0.053184 Pp(A|Q) = 0.002976 Pp(A|R) = 0.001385 Pp(A|S) = 0.000000 Pp(A|T) = 0.000000
Pp(A|U) = 0.000000 Pp(A|V) = 0.000000 Pp(A|W) = 0.000000 Pp(A|X) = 0.000000 Pp(A|Y) = 0.000000 Pp(A|Z) = 0.000000
```

- QUIJOTEP.txt

El texto original era de gran tamaño, por lo que tenemos un muestreo muy considerable.

Y, como podemos ver, por ejemplo para el caso de la A, la probabilidad del caracter A y la de sus probabilidades condicionales asociadas es muy próxima para todo valor, con errores cercanos al 1%, por lo que podemos afirmar que se cumple la seguridad perfecta si escogemos claves equiprobables.

```

Pp(A) = 0.143947
Pp(B) = 0.014718
Pp(C) = 0.035208
Pp(D) = 0.051482
Pp(E) = 0.137358
Pp(F) = 0.004817
Pp(G) = 0.009977
Pp(H) = 0.011603
Pp(I) = 0.053219
Pp(J) = 0.005885
Pp(K) = 0.000001
Pp(L) = 0.053529
Pp(M) = 0.026943
Pp(N) = 0.066149
Pp(O) = 0.096355
Pp(P) = 0.020984
Pp(Q) = 0.019917
Pp(R) = 0.059776
Pp(S) = 0.073166
Pp(T) = 0.037418
Pp(U) = 0.048296
Pp(V) = 0.010552
Pp(W) = 0.000001
Pp(X) = 0.000256
Pp(Y) = 0.014595
Pp(Z) = 0.003846
Pp(A|A) = 0.143893 Pp(A|B) = 0.145279 Pp(A|C) = 0.140240 Pp(A|D) = 0.144471 Pp(A|E) = 0.146439 Pp(A|F) = 0.143548 Pp(A|G) =
0.141689 Pp(A|H) = 0.146071 Pp(A|I) = 0.146150 Pp(A|J) = 0.145595 Pp(A|K) = 0.140629 Pp(A|L) = 0.143423 Pp(A|M) = 0.142875 Pp(
A|N) = 0.145014 Pp(A|O) = 0.141469 Pp(A|P) = 0.144959 Pp(A|Q) = 0.143435 Pp(A|R) = 0.145009 Pp(A|S) = 0.145304 Pp(A|T) = 0.144148
Pp(A|U) = 0.141664 Pp(A|V) = 0.149933 Pp(A|W) = 0.143203 Pp(A|X) = 0.141031 Pp(A|Y) = 0.144065 Pp(A|Z) = 0.143015

```

- QUIJOTEI.txt

En cambio en el de la distribución normal, vemos que las probabilidades condicionadas están distribuidas entre los caracteres cifrados de la “I” a la “R”, dejando al resto de letras con probabilidades condicionadas asociadas de 0. Por lo que no se cumple la seguridad perfecta.

```

Pp(A|A) = 0.000000 Pp(A|B) = 0.000000 Pp(A|C) = 0.000000 Pp(A|D) = 0.000000 Pp(A|E) = 0.000000 Pp(A|F) = 0.000000 Pp(A|G) =
0.000000 Pp(A|H) = 0.000000 Pp(A|I) = 0.000321 Pp(A|J) = 0.033098 Pp(A|K) = 0.344874 Pp(A|L) = 0.746373 Pp(A|M) = 0.886646 Pp(
A|N) = 0.805784 Pp(A|O) = 0.419186 Pp(A|P) = 0.056878 Pp(A|Q) = 0.002507 Pp(A|R) = 0.000126 Pp(A|S) = 0.000000 Pp(A|T) = 0.000000
Pp(A|U) = 0.000000 Pp(A|V) = 0.000000 Pp(A|W) = 0.000000 Pp(A|X) = 0.000000 Pp(A|Y) = 0.000000 Pp(A|Z) = 0.000000

```

2. Implementación del DES

a. Programación del DES

Implementación

El algoritmo cifrará el texto usando una clave de 64b generada aleatoriamente, y con un bit de paridad impar en el último bit de cada byte. Esta funcionalidad la realiza la función *createKey()*.

Además generaremos también el vector de inicialización IV de forma aleatoria sin ningún tipo de restricción usando la función *createIV()*.

Estos valores generados los imprimiremos en pantalla para usarlos en el descifrado posterior, como veremos en las capturas más adelante.

Además, hemos decidido hacer un control en caso de que el fichero no tenga un tamaño múltiplo de 64b, y añadiremos 0's como padding.

Sobre la implementación en sí, cabe destacar que esencialmente, desde el main llamaremos a las funciones:

- *createSubkeys*

Esta función, a partir de la clave generada aleatoriamente, crea las 16 subclaves que se usarán en cada ronda del encriptado.

- *encode_block(uint64_t Mens, uint64_t* subkeys, int cifrar)*

Esta función, simulará las 16 rondas del algoritmo, es decir, dado un bloque de 64b del mensaje y las subclaves generadas anteriormente, lo cifrará o descifrará dependiendo de si se le pasa un 1 o un 0 al flag de cifrar. Cabe destacar que de forma interna llamará a la función f del DES que ha sido implementado en una función separada por modularidad.

Al ser el DES en modo CBC, en el main, entre cada cifrado/descifrado de bloque, habrá un xor entre el bloque cifrado anterior y el mensaje a cifrar siguiente, manteniendo una cierta dependencia de un bloque con los anteriores. Usando como primer mensaje cifrado el vector de inicialización generado aleatoriamente antes.

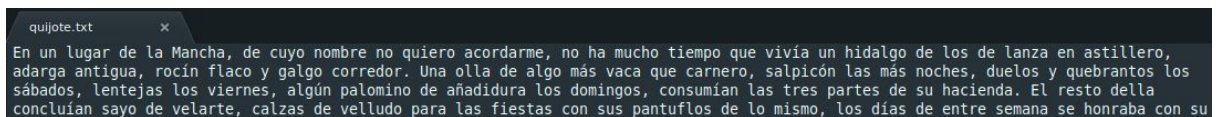
A continuación vemos un ejemplo de uso:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicass/Cripto/P2/g05$ ./desCBC -C
-i quijote.txt -o quijote2.txt
Key: e9fece83b67031ce
IV: 79468f858c4a8bf2
carlos@carlos-SVF1521N6EW:~/Escritorio/practicass/Cripto/P2/g05$ ./desCBC -D
-k e9fece83b67031ce -iv 79468f858c4a8bf2 -i quijote2.txt -o quijote3.txt
```

Vemos cómo se imprimen la clave y vector de inicialización generados de forma aleatoria, y los usamos para descifrar el texto posteriormente.

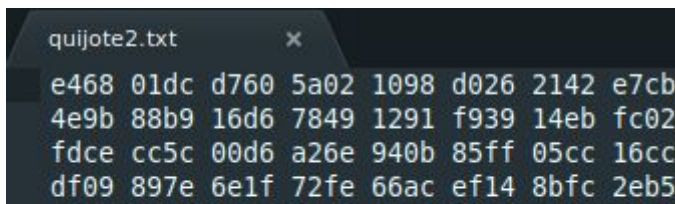
quijote.txt

El texto original a cifrar:



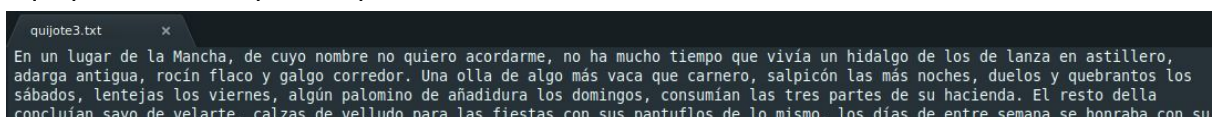
quijote2.txt

Tras cifrarlo, aparece en binario pues DES no siempre transforma los caracteres a ASCII y no se suelen representar correctamente como caracteres:



quijote3.txt

Aquí podemos comprobar que se ha descifrado correctamente:

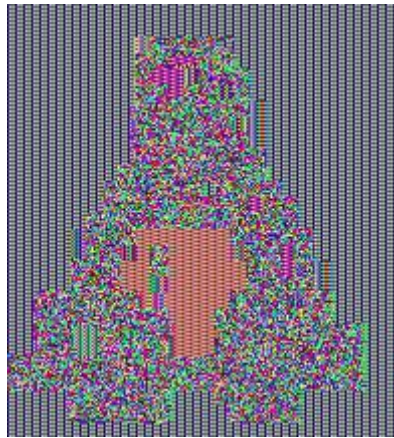


Cifrado de Imágenes

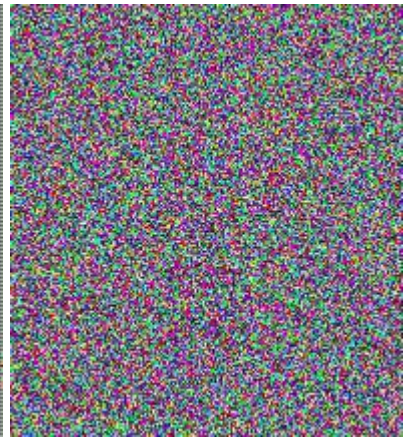
Para realizar este apartado hemos utilizado el script image.sh, que esencialmente corta la cabecera de la imagen ppm y cifra el cuerpo de la imagen para posteriormente devolverle la cabecera original y de esta forma mantenemos el formato de la imagen. El resultado de los diferentes cifrados es el siguiente:



Imagen original



Cifrada con ECB



Cifrada con CBC

Encima podemos visualizar el efecto de hacer cifrados con los cifrados de forma independiente (ECB) y aplicando dependencia de los cifrados anteriores.

Como esperábamos, cifrar con ECB no es bueno para mensajes largos y estructurados, pues aún podemos seguir apreciando la silueta y los colores de la imagen original.

Mientras que cifrando con CBC se pierde completamente el rastro de la imagen original.

b. Programación del Triple DES (OPCIONAL)

La implementación de esta versión es similar a la del DES, la diferencia radica en que ahora utilizaremos 3 claves k_1, k_2, k_3 (aparecerán concatenadas en el comando), siguiendo el esquema:

para cifrar: $y = E_{k_3}(D_{k_2}(E_{k_1}(x)))$

para descifrar: $x = D_{k_1}(E_{k_2}(D_{k_3}(y)))$

De forma análoga al caso anterior, ejecutamos el ejemplo, y vemos que nos genera textos como los del ejemplo anterior:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicac/Cripto/P2/g05$ ./tripleDESCBC -C -i
quijote.txt -o tquijote.txt
Key: f2a20831a79b2a57d3804a9e627c51d98ccbe945314f0757
IV: 3ff719e7feaf83f7
carlos@carlos-SVF1521N6EW:~/Escritorio/practicac/Cripto/P2/g05$ ./tripleDESCBC -D -k
f2a20831a79b2a57d3804a9e627c51d98ccbe945314f0757 -iv 3ff719e7feaf83f7 -i tquijote.txt
-o tquijote2.txt
```

Cabe destacar que la clave que se usa para cifrar ahora tiene el triple de tamaño como se aprecia en la captura.

quijote.txt

Ciframos el mismo texto inicial:

```
quijote.txt x
En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero,
adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los
sábados, lentejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda. El resto della
concluían sayo de velarte, calzas de velludo para las fiestas con sus pantuflos de lo mismo, los días de entre semana se honraba con su
```

tquijote.txt

Nos da otro cifrado diferente:

```
tquijote.txt x
8217 b623 db79 fc19 1ed8 8814 690d a5f2
11a1 991d 8da1 fdbf 0033 5260 adc7 eb73
2f83 c023 e51b 8819 bd8e 9ca8 25f5 c206
1434 ca9d df3f 14fd 0fab c159 fda2 d98e
```

tquijote2.txt

Y se descifra correctamente:

```
tquijote2.txt x
En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero,
adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los
sábados, lentejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda. El resto della
concluían sayo de velarte, calzas de velludo para las fiestas con sus pantuflos de lo mismo, los días de entre semana se honraba con su
```

3. Principios de diseño del DES

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

En este apartado vamos a demostrar empíricamente la propiedad de no linealidad de las S-boxes del cifrado DES. Es decir, vamos a ver que: $f(x \oplus y) \neq f(x) \oplus f(y)$ donde **f** representa la salida de la caja S-box y \oplus representa la operación xor.

Para ello, diseñamos un programa que generará cadenas binarias de 48 bits aleatorias, **x** e **y**, el número de repeticiones que consideremos necesarias para sacar evidencias de la no linealidad. Dichas cadenas aleatorias serán la entrada de las S-boxes (en bloques de 6), de las que obtendremos la salida de 32 bits (en bloques de 4). Compararemos el xor de ambas salidas con el resultado de hacer xor de **x** e **y** antes de introducirlo en la S-box, comprobando que el resultado no es el mismo.

Tendremos un contador con el número de casos lineales, es decir, en los que ambos resultados han coincidido. Si realizamos un número considerable de repeticiones del método y el contador se mantiene a 0, podremos afirmar experimentalmente la no linealidad buscada.

Ejemplo de ejecución, confirmando no linealidad:

```
deivid@lenovo-g50-80: ~/Escritorio/Cripto/P2/g05
Archivo Editar Ver Buscar Terminal Ayuda
deivid@lenovo-g50-80:~/Escritorio/Cripto/P2/g05$ ./sboxes_lin_des
Casos probados = 1000000
Numero de casos lineales = 0
deivid@lenovo-g50-80:~/Escritorio/Cripto/P2/g05$ ./sboxes_lin_des
```

b. Estudio del Efecto de Avalancha

Vamos a construir un programa que permita ver el efecto avalancha del algoritmo DES, es decir, al realizar un mínimo cambio en la clave o el bloque, la diferencia del encriptado va aumentando rápidamente a medida que se van sucediendo las rondas.

Partiendo de un cifrado **C** con el mensaje **M** y la clave **K**. El procedimiento va a consistir en dos modificaciones:

- Cambio en el bloque de mensaje: realizamos 1, 2 y 3 cambios en el mensaje que va a ser cifrado obteniendo en cada caso un **M'**, que dará como resultado **C'**.
- Cambio en la clave: realizamos 1, 2 y 3 cambios en la clave utilizada, obteniendo claves **K'**, que dará como resultado otro cifrado **C'**.

Experimentalmente observaremos que los cambios de **C_i**, es decir, el cifrado en la ronda *i*-ésima van aumentando respecto del cifrado original **C_i**, a medida que se van incrementando las rondas. Esto ocurre tanto si se ha cambiado la clave (**K'**) como si se ha cambiado el mensaje (**M'**).

Ejemplo de salida de la ejecución en el fichero *avalancha.txt*, observando el efecto avalancha:

Cambiando 1 bits del mensaje	Cambiando 1 bits de la clave
Ronda 0: bits diferentes = 1	Ronda 0: bits diferentes = 0
Ronda 1: bits diferentes = 5	Ronda 1: bits diferentes = 2
Ronda 2: bits diferentes = 20	Ronda 2: bits diferentes = 8
Ronda 3: bits diferentes = 31	Ronda 3: bits diferentes = 22
Ronda 4: bits diferentes = 29	Ronda 4: bits diferentes = 37
Ronda 5: bits diferentes = 29	Ronda 5: bits diferentes = 35
Ronda 6: bits diferentes = 31	Ronda 6: bits diferentes = 35
Ronda 7: bits diferentes = 30	Ronda 7: bits diferentes = 38
Ronda 8: bits diferentes = 26	Ronda 8: bits diferentes = 37
Ronda 9: bits diferentes = 30	Ronda 9: bits diferentes = 33
Ronda 10: bits diferentes = 31	Ronda 10: bits diferentes = 33
Ronda 11: bits diferentes = 29	Ronda 11: bits diferentes = 34
Ronda 12: bits diferentes = 29	Ronda 12: bits diferentes = 30
Ronda 13: bits diferentes = 31	Ronda 13: bits diferentes = 34
Ronda 14: bits diferentes = 33	Ronda 14: bits diferentes = 30
Ronda 15: bits diferentes = 27	Ronda 15: bits diferentes = 30
Ronda 16: bits diferentes = 27	Ronda 16: bits diferentes = 35

Como podemos observar, los cambios van aumentando y en la quinta ronda, prácticamente ya la mitad de los bits son diferentes respecto al cifrado original. En las siguientes rondas ya se va estabilizando y no llegan a aumentar más. Esto sucede en ambas modificaciones, de clave y de texto.

c. El Criterio de Avalancha Estricto (SAC) y el Criterio de Independencia de Bits (BIC)

En este apartado vamos a realizar un programa que pruebe los criterios de avalancha estricto (SAC) y de independencia de bits (BIC):

- **SAC:** nos dice que si complementamos un bit de la entrada de las S-boxes, cada uno de los bits de salida cambiará con probabilidad $\frac{1}{2}$, es decir, que será igualmente probable que cada bit sea un 1 o un 0:

$$P(b_j = 0 \mid \bar{a}_i) = P(b_j = 1 \mid \bar{a}_i) = \frac{1}{2} \quad 1 \leq j \leq 4 \quad 1 \leq i \leq 6$$

Para comprobarlo, generamos cadenas aleatorias de 48 bits que serán la entrada de las S-boxes. Nos fijamos en cada bit de salida (del total de 4) y contamos las veces que en cada posición hay un 0 y las veces que hay un 1. Esto lo repetimos el número suficiente de veces (ya que la probabilidad será más exacta cuantas más muestras tengamos) y vemos que el número de 0's y de 1's es muy próximo, confirmando la equiprobabilidad.

Comprobamos experimentalmente que dicha probabilidad se cumple, dependiendo del número de iteraciones:

Con 55 iteraciones, por ejemplo, no tenemos suficientes datos para que la probabilidad se aproxime bien al valor 0.5 deseado.

Numero de repeticiones: 55		
Resultados de SAC		
b1	P(1) = 0.472727	P(0) = 0.527273
b2	P(1) = 0.490909	P(0) = 0.509091
b3	P(1) = 0.527273	P(0) = 0.472727
b4	P(1) = 0.600000	P(0) = 0.400000
b5	P(1) = 0.472727	P(0) = 0.527273
b6	P(1) = 0.545455	P(0) = 0.454545
b7	P(1) = 0.454545	P(0) = 0.545455
b8	P(1) = 0.472727	P(0) = 0.527273
b9	P(1) = 0.618182	P(0) = 0.381818
b10	P(1) = 0.545455	P(0) = 0.454545
b11	P(1) = 0.472727	P(0) = 0.527273
b12	P(1) = 0.381818	P(0) = 0.618182
b13	P(1) = 0.381818	P(0) = 0.618182
b14	P(1) = 0.436364	P(0) = 0.563636
b15	P(1) = 0.472727	P(0) = 0.527273
b16	P(1) = 0.509091	P(0) = 0.490909
b17	P(1) = 0.436364	P(0) = 0.563636
b18	P(1) = 0.454545	P(0) = 0.545455
b19	P(1) = 0.527273	P(0) = 0.472727
b20	P(1) = 0.400000	P(0) = 0.600000
b21	P(1) = 0.527273	P(0) = 0.472727
b22	P(1) = 0.381818	P(0) = 0.618182
b23	P(1) = 0.509091	P(0) = 0.490909
b24	P(1) = 0.490909	P(0) = 0.509091
b25	P(1) = 0.509091	P(0) = 0.490909
b26	P(1) = 0.472727	P(0) = 0.527273
b27	P(1) = 0.636364	P(0) = 0.363636
b28	P(1) = 0.600000	P(0) = 0.400000
b29	P(1) = 0.454545	P(0) = 0.545455
b30	P(1) = 0.490909	P(0) = 0.509091
b31	P(1) = 0.563636	P(0) = 0.436364
b32	P(1) = 0.418182	P(0) = 0.581818

Si probamos con un número grande de iteraciones, por ejemplo 1000000 la probabilidad ya sí es muy cercana a $\frac{1}{2}$ en ambos casos, para $P(0)$ y para $P(1)$:

Numero de repeticiones: 1000000		
Resultados de SAC		
b1	P(1) = 0.500143	P(0) = 0.499857
b2	P(1) = 0.500096	P(0) = 0.499904
b3	P(1) = 0.500192	P(0) = 0.499808
b4	P(1) = 0.500171	P(0) = 0.499829
b5	P(1) = 0.499881	P(0) = 0.500119
b6	P(1) = 0.500445	P(0) = 0.499555
b7	P(1) = 0.500581	P(0) = 0.499419
b8	P(1) = 0.500151	P(0) = 0.499849
b9	P(1) = 0.500039	P(0) = 0.499961
b10	P(1) = 0.500067	P(0) = 0.499933
b11	P(1) = 0.500066	P(0) = 0.499934
b12	P(1) = 0.499580	P(0) = 0.500420
b13	P(1) = 0.500345	P(0) = 0.499655
b14	P(1) = 0.500125	P(0) = 0.499875
b15	P(1) = 0.499424	P(0) = 0.500576
b16	P(1) = 0.499628	P(0) = 0.500372
b17	P(1) = 0.500715	P(0) = 0.499285
b18	P(1) = 0.500159	P(0) = 0.499841
b19	P(1) = 0.499868	P(0) = 0.500132
b20	P(1) = 0.500413	P(0) = 0.499587
b21	P(1) = 0.499982	P(0) = 0.500018
b22	P(1) = 0.501259	P(0) = 0.498741
b23	P(1) = 0.499834	P(0) = 0.500166
b24	P(1) = 0.500096	P(0) = 0.499904
b25	P(1) = 0.499839	P(0) = 0.500161
b26	P(1) = 0.499574	P(0) = 0.500426
b27	P(1) = 0.500217	P(0) = 0.499783
b28	P(1) = 0.500245	P(0) = 0.499755
b29	P(1) = 0.499592	P(0) = 0.500408
b30	P(1) = 0.499759	P(0) = 0.500241
b31	P(1) = 0.500264	P(0) = 0.499736
b32	P(1) = 0.499345	P(0) = 0.500655

- **BIC**: este criterio dice que, tras cambiar un bit de la entrada, cada par de bits de la salida de las S-boxes cambiará independientemente, es decir, que la probabilidad de que cambien 2 bits de salida será el producto de la probabilidad de cambio de cada uno de esos 2 bits. Esto se cumple para cada S-box.

$$P(b_j, b_k | \bar{a}_i) = P(b_j | \bar{a}_i) \times P(b_k | \bar{a}_i) \quad 1 \leq j, k \leq 4 \quad 1 \leq i \leq 6$$

Intuitivamente, por el apartado anterior sabemos que la probabilidad de cada b_j , sea para 0 o para 1, era de $\frac{1}{2}$. Por tanto cabe pensar que la probabilidad conjunta será de $\frac{1}{4}$. Para simplificar el problema, podemos realizar los cálculos (ya que 0 y 1 son equiprobables) para el valor de 0, tanto de cada b_j ($P(b_j=0)$) como de la conjunta ($P(b_j=0, b_k=0)$). Es decir, calculamos las probabilidades condicionadas (al cambio de bit de entrada) contando el número de 0's de cada b_j y el número de veces que cada par (b_j, b_k) tiene en ambos un 0.

Si realizamos 55 iteraciones, por ejemplo, los resultados no son lo suficientemente cercanos a los valores esperados de $\frac{1}{2}$ y $\frac{1}{4}$, pues el muestreo no es suficiente:

```

Resultados de BIC

SB0X 1
b1 P(0) = 0.545455
b2 P(0) = 0.490909
b3 P(0) = 0.545455
b4 P(0) = 0.436364
b1,2 P(0,0) = 0.236364
b1,3 P(0,0) = 0.272727
b1,4 P(0,0) = 0.272727
b2,3 P(0,0) = 0.218182
b2,4 P(0,0) = 0.218182
b3,4 P(0,0) = 0.236364

SB0X 2
b1 P(0) = 0.436364
b2 P(0) = 0.436364
b3 P(0) = 0.527273
b4 P(0) = 0.618182
b1,2 P(0,0) = 0.163636
b1,3 P(0,0) = 0.272727
b1,4 P(0,0) = 0.236364
b2,3 P(0,0) = 0.181818
b2,4 P(0,0) = 0.272727
b3,4 P(0,0) = 0.327273

```

Sin embargo, si aumentamos las iteraciones a 1000000, los valores sí que son lo suficientemente cercanos, a $\frac{1}{2}$ en la probabilidad de cada b_j y a $\frac{1}{4}$ en la conjunta:

```

Resultados de BIC

SB0X 1
b1 P(0) = 0.500692
b2 P(0) = 0.499820
b3 P(0) = 0.499518
b4 P(0) = 0.499992
b1,2 P(0,0) = 0.250499
b1,3 P(0,0) = 0.250124
b1,4 P(0,0) = 0.249871
b2,3 P(0,0) = 0.249316
b2,4 P(0,0) = 0.249848
b3,4 P(0,0) = 0.249895

SB0X 2
b1 P(0) = 0.500389
b2 P(0) = 0.500304
b3 P(0) = 0.499308
b4 P(0) = 0.500214
b1,2 P(0,0) = 0.250455
b1,3 P(0,0) = 0.250481
b1,4 P(0,0) = 0.249879
b2,3 P(0,0) = 0.250036
b2,4 P(0,0) = 0.250311
b3,4 P(0,0) = 0.249414

```

d. Construcción de S-boxes propias

Para realizar este apartado, hemos hecho uso del código de los apartados anteriores, comprobando la linealidad, el BIC y el SAC.

Esencialmente hemos ido generando S-boxes aleatoriamente y hemos ido comprobando secuencialmente cada característica.

Para la linealidad, hemos pedido que si hay algún caso donde $f(x \oplus y) = f(x) \oplus f(y)$, se paren las pruebas siguientes y se proceda a generar nuevas S-boxes.

A continuación, hemos comprobado el SAC y luego el BIC, para ello hemos realizado las pruebas del apartado b), pero al final del todo hemos mirado si las medias de las probabilidades están bajo un error del 0.01 con respecto a las que se esperan. En caso de que superen ese umbral, se paran las pruebas siguientes y se comienza de nuevo con otras S-boxes diferentes.

Para el número de iteraciones para tomar muestreos, hemos tomado valor 1000 para que el programa no tarde excesivamente. En la salida imprimimos el número de intentos hechos antes de imprimir la S-box definitiva:

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P2/g05$ ./sboxes_des
Tras 18 iteraciones
SBOX 0:
Fila 0: 8 5 11 2 9 7 14 0 4 3 13 6 15 12 1 10
Fila 1: 3 6 14 15 9 1 11 0 10 13 4 5 7 2 12 8
Fila 2: 9 15 7 10 2 1 14 6 11 4 13 3 0 8 5 12
Fila 3: 14 1 8 12 4 10 7 0 3 9 2 11 15 6 13 5
SBOX 1:
Fila 0: 9 0 12 1 3 5 11 14 2 15 7 4 13 6 8 10
Fila 1: 2 15 12 14 3 1 4 5 7 8 11 0 13 6 10 9
Fila 2: 9 5 4 14 8 13 2 6 10 7 15 0 12 1 11 3
Fila 3: 4 14 11 13 8 1 5 6 9 3 10 7 2 15 0 12
SBOX 2:
Fila 0: 4 9 7 15 12 2 11 1 6 0 10 3 14 5 8 13
Fila 1: 13 14 8 3 9 1 11 15 0 10 5 2 6 12 4 7
Fila 2: 7 12 6 0 3 2 10 14 1 8 5 4 11 13 15 9
Fila 3: 2 8 7 11 10 14 4 12 6 0 13 5 9 3 1 15
SBOX 3:
Fila 0: 15 0 3 13 4 8 11 10 7 9 5 12 14 2 6 1
Fila 1: 9 12 2 1 10 6 0 11 8 13 14 4 15 7 5 3
Fila 2: 9 5 2 15 7 1 4 3 12 11 10 6 14 8 0 13
Fila 3: 3 4 10 2 0 7 9 13 14 1 5 12 15 11 6 8
```


4. Principios de diseño del AES

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

En este apartado debemos comprobar la no linealidad de las cajas de sustitución del AES, es decir, que $f(x \oplus y) \neq f(x) \oplus f(y)$. El procedimiento a seguir es muy similar al del caso de las cajas del DES. En este caso, los vectores aleatorios x e y los tomamos de 64 bits, ya que el AES trabaja con bloques de 128 bits, y si comprobamos la no linealidad con los 64 bits se puede extender a la linealidad de los 128 bits. La otra diferencia respecto al DES es que ahora deberemos pasar 8 bits por la misma caja de sustitución siempre, lo que nos devolverá otros 8 bits a la salida.

Realizando las iteraciones suficientes (en nuestro caso 1000000) observamos que el contador de casos lineales siempre es 0, por lo que comprobamos empíricamente la no linealidad de las cajas de sustitución del AES.

Ejemplo de ejecución:

```
deivid@lenovo-g50-80: ~/Escritorio/Cripto/P2/g05
Archivo Editar Ver Buscar Terminal Ayuda
deivid@lenovo-g50-80:~/Escritorio/Cripto/P2/g05$ ./sboxes_lin_aes
Casos probados = 1000000
Numero de casos lineales = 0
deivid@lenovo-g50-80:~/Escritorio/Cripto/P2/g05$
```

La S-box inversa del AES no hace falta comprobarla, pues al ser, precisamente, la inversa de la anterior, si la primera cumple no linealidad la segunda también, pues es simplemente una permutación de los elementos. Es decir, podemos ver la S-box del AES como una biyección. Al tener que f cumple la no linealidad, obligatoriamente su función inversa debe cumplirla.

Demostración:

Tenemos, por la prueba empírica de no linealidad de la S-box directa de AES:

$$f(x \oplus y) \neq f(x) \oplus f(y) \quad \forall x, y$$

Al ser un biyección, tomamos la función inversa, que en este caso equivale a S-box inversa:

$$f^{-1}(f(x \oplus y)) \neq f^{-1}(f(x) \oplus f(y)) \quad \forall x, y$$

Ahora tenemos, por definición de función inversa:

$$x \oplus y \neq f^{-1}(f(x) \oplus f(y)) \quad \forall x, y$$

Supongamos que existe una linealidad de f inversa, tendríamos que:

$$x \oplus y \neq f^{-1}(f(x)) \oplus f^{-1}(f(y)) \quad \text{para algún } x, y$$

Aplicando de nuevo la definición de inversa, obtenemos:

$$x \oplus y \neq x \oplus y \quad \text{para algún } x, y$$

Por tanto llegamos a una contradicción y concluimos que f inversa (S-box inversa) no puede ser lineal en ningún caso.

b. Generación de las S-boxes AES

Hay que destacar que para entender este apartado, hay que entender que los polinomios están en Z_2 , y los representamos como números en binario mediante sus coeficientes.

Para poder implementar los algoritmos de Euclides y Euclides extendido, ha sido necesario implementar las siguientes funciones:

- xtime

Esta función realiza la multiplicación de un polinomio por x módulo m(x) del AES. Y se usa para la función AES_product.

- AES_product

Esta función realiza el producto de 2 polinomios módulo m(x) haciendo uso de la función xtime. El criterio que seguimos es el criterio de realizar el menor número de xtimes, por lo que de 2 polinomios, realizaremos xtime el tamaño del polinomio de menor grado.

- pol_division

Con este algoritmo realizamos el algoritmo de división de polinomios en general en Z_2 , para obtener el cociente q y el resto r de un dividendo n y un divisor d.

A continuación adaptamos los algoritmos de Euclides y Euclides extendido a polinomios usando las funciones anteriores. La idea está en que todas las operaciones se realizan módulo m(x), por tanto, el producto se hace utilizando AES_product, que está optimizado mediante el uso de xtime. pol_division nos proporciona el cociente y resto de las divisiones. Además, las restas y sumas quedan sustituidas por la operación xor.

Una vez implementados los algoritmos anteriores, procedemos a la generación de las Sboxes del AES siguiendo los pasos siguientes para cada elemento de la S-box:

1. $S[i,j] = ij$ (concatenación de bits)
2. $S[i,j] = (S[i,j])^{-1}$ (calculado con Euclides extendido)
3. $S[i,j] = S[i,j] + (S[i,j] \ll 1) + (S[i,j] \ll 2) + (S[i,j] \ll 3) + (S[i,j] \ll 4) + C$, con $C = 63_{16}$

Para generar las Sboxes inversas hemos aprovechado el procedimiento anterior, y simplemente hemos hecho la asignación siguiente como último paso:

$$(S^{-1}[k,l] = ij) \text{ con } k = \text{"la parte izquierda de } S[i,j]\text{"}, l = \text{"la parte derecha de } S[i,j]\text{"}$$

A continuación mostramos las Sboxes generadas, y vemos que coinciden con las Sboxes del AES como esperábamos;

- Sbox directa

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P2/g05$ ./SB0X_AES -C
63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```

- Sbox inversa

```
carlos@carlos-SVF1521N6EW:~/Escritorio/practicas/Cripto/P2/g05$ ./SB0X_AES -D
52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```