

UNIVERSIDAD AUTÓNOMA DE MADRID



**FUNDAMENTOS DE CRIPTOGRAFÍA Y
SEGURIDAD INFORMÁTICA
(2020-2021)**

PRÁCTICA 2

Alba Ramos Pedroviejo
Ana María Cidoncha Suárez

Grupo 1461

Madrid, 25 de noviembre de 2020

ÍNDICE DE CONTENIDOS

Aclaraciones previas	3
Seguridad perfecta	3
Comprobación empírica	4
Implementación del DES	8
Programación del DES	8
ECB en textos largos e imágenes	11
Programación del Triple DES	11
Principios de diseño del DES	12
Estudio de la no linealidad de las S-boxes del DES	12
Estudio del Efecto de Avalancha	12
Principios de diseño del AES	13
Estudio de la no linealidad de las S-boxes del AES	13
Generación de las S-boxes AES	15

Aclaraciones previas

Seguridad perfecta: el programa está pensado para hacer estadísticas sobre un texto lo suficientemente largo como para que, al menos, haya un carácter de cada una de las 26 letras que componen el alfabeto español, sobre todo en el texto cifrado, por lo que si metemos un texto pequeño es posible que al intentar calcular la probabilidad condicionada y dividir por cero el resultado del programa sea *-nan*.

DES y triple DES:

- Tanto la clave como el vector de inicialización deben ir en hexadecimal, pues desde el principio se nos ha indicado que todo vaya en hexadecimal en estos programas.
- La representación de lo que estamos cifrando y descifrando se está mostrando en hexadecimal en vez de en binario y no sabemos bien por qué. A pesar de abrir los ficheros en modo binario, cuando comprobamos el fichero descifrado por ejemplo vemos que está todo en hexadecimal, es decir, que no abres el fichero y sale lo que saldría en el original, sino que salen cosas en hexadecimal. Aún así, los programas funcionan correctamente, y hemos comprobado que si leemos el fichero original en hexadecimal y vamos comparando con lo que desciframos nosotras tras cifrar con nuestro programa, sale igual.
- Nuestros programas funcionan bien con entradas múltiplos de 64b. Si el fichero de entrada no lo es, el último bloque que se lea no será de 64b y entonces no funcionará ese bloque. Hemos intentado arreglar esto, pero no somos capaces. Tenemos mucha carga de trabajo actualmente, no hemos sido capaces de implementar una solución a tiempo, pero creemos que podríamos lograrlo con un poco más de tiempo. La idea sería que, si el último bloque no mide 64b, rellenarlo con ceros al final y cifrar así, y para descifrar eliminar los ceros que pueda haber en el último bloque. Esto supone otra limitación: podría ser que el bloque original acabara con algún cero, y aparte nuestro padding... Entonces hay un problema. Sin embargo, no nos ha dado tiempo, hemos empezado a intentarlo como se ve en el código, pero no hemos podido dejarlo funcionando.

Seguridad perfecta

Shannon postuló que un sistema de secreto perfecto es aquel en el que se cumple que la probabilidad de que dado un mensaje cifrado c , la probabilidad de que el mensaje en claro sea m es igual a la probabilidad de que el mensaje en claro sea m sin el conocimiento del texto cifrado, es decir, que el conocimiento a priori no nos da ninguna información sobre el conocimiento a posteriori. La seguridad perfecta asume que cada elemento de texto plano se cifra con una clave distinta.

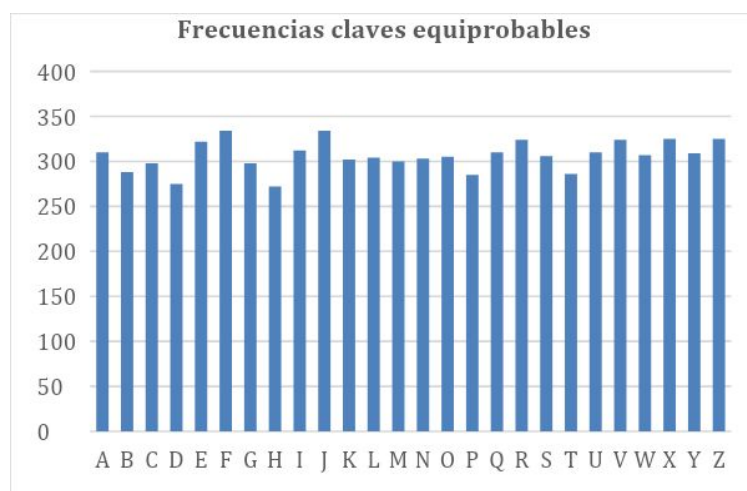
Un criptosistema tiene seguridad perfecta si $P_p(x|y) = P_p(x) \quad \forall x \in P, \quad \forall y \in C$

Comprobación empírica

Para realizar la comprobación empírica hemos realizado un programa, *seg-perf {-P | -I} [-i f ilein] [-o f ileout]* que, además de admitir los parámetros *-i* para leer el texto de un fichero y *-o* para escribir los resultados en un fichero, admite los argumentos *-P* si se utiliza el método equiprobable o *-I* si el método es no equiprobable.

Para el método de obtención de claves equiprobables (modo *-P*) hemos utilizado el generador estándar de números aleatorios de C (*rand()*). Para el método de obtención de claves no equiprobables (método *-I*) lo que hemos hecho ha sido sesgar la probabilidad de las claves hacia la 'Z'. En una de las pruebas hemos sesgado las claves 'A', 'D', 'E' y 'O' hacia la 'Z' y en otras de las pruebas, para ver más claramente la diferencia, se han sesgado todas las claves desde la 'F' hasta la 'Y' hacia la 'Z'.

Claves equiprobables:



La frecuencia de las claves, como se ve en el gráfico, ha salido bastante equiprobable utilizando la función *rand()* de C.

Hemos probado a cifrar el primer capítulo de “El Quijote” y un texto más pequeño creado por nosotras obteniendo los siguientes resultados:

Para el texto de “El Quijote”, las probabilidades condicionadas de, por ejemplo, la letra ‘A’ han sido:

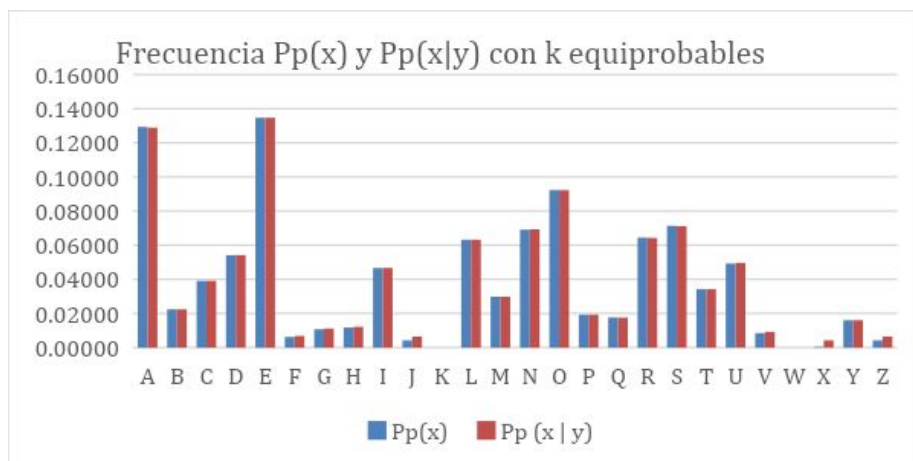
$p(A A) = 0.128378$	$p(A B) = 0.105611$	$p(A C) = 0.116667$	$p(A D) = 0.130584$
$p(A E) = 0.140893$	$p(A F) = 0.138365$	$p(A G) = 0.135314$	$p(A H) = 0.103333$
$p(A I) = 0.125000$	$p(A J) = 0.178886$	$p(A K) = 0.158416$	$p(A L) = 0.106164$
$p(A M) = 0.100694$	$p(A N) = 0.141026$	$p(A O) = 0.094156$	$p(A P) = 0.101307$
$p(A Q) = 0.119632$	$p(A R) = 0.121406$	$p(A S) = 0.131148$	$p(A T) = 0.128931$
$p(A U) = 0.155116$	$p(A V) = 0.133127$	$p(A W) = 0.118056$	$p(A X) = 0.141026$
$p(A Y) = 0.168285$	$p(A Z) = 0.130137$		

Haciendo la media obtenemos una probabilidad condicionada media de la letra ‘A’, de 0.12891 que si la comparamos con la probabilidad de dicha letra en el texto plano ($Pp(A) = 0.12925$) vemos que, salvo unos pocos decimales de redondeos, es igual.

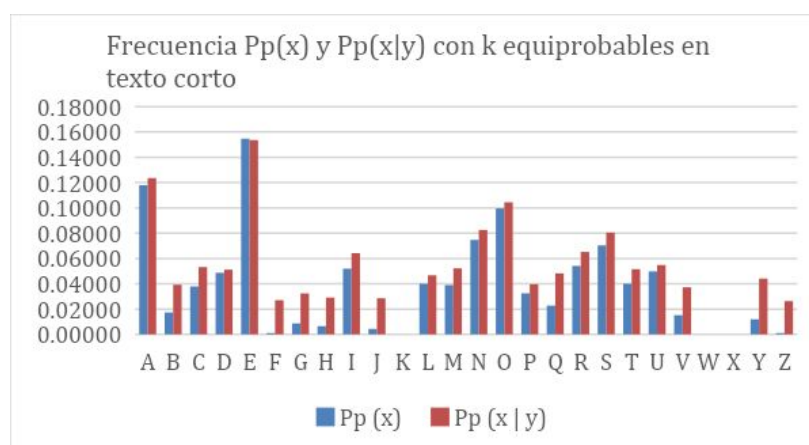
P(X)		Media Pp ()
Pp(A)	0,12925	0,12891
Pp(B)	0,02246	0,02246
Pp(C)	0,03915	0,03916
Pp(D)	0,05421	0,05426
Pp(E)	0,13465	0,13483
Pp(F)	0,00627	0,00687
Pp(G)	0,01079	0,01121
Pp(H)	0,01180	0,01222
Pp(I)	0,04668	0,04671
Pp(J)	0,00427	0,00652
Pp(K)	0,00000	0,00000
Pp(L)	0,06325	0,06325
Pp(M)	0,02987	0,02980

P(X)		Media Pp ()
Pp(N)	0,06914	0,06936
Pp(O)	0,09236	0,09233
Pp(P)	0,01933	0,01935
Pp(Q)	0,01769	0,01763
Pp(R)	0,06450	0,06431
Pp(S)	0,07140	0,07122
Pp(T)	0,03426	0,03425
Pp(U)	0,04932	0,04964
Pp(V)	0,00853	0,00928
Pp(W)	0,00000	0,00000
Pp(X)	0,00050	0,00430
Pp(Y)	0,01606	0,01608
Pp(Z)	0,00427	0,00651

Hemos procedido igual con el resto de las letras obteniendo los siguientes resultados que se aprecian mejor en el histograma.



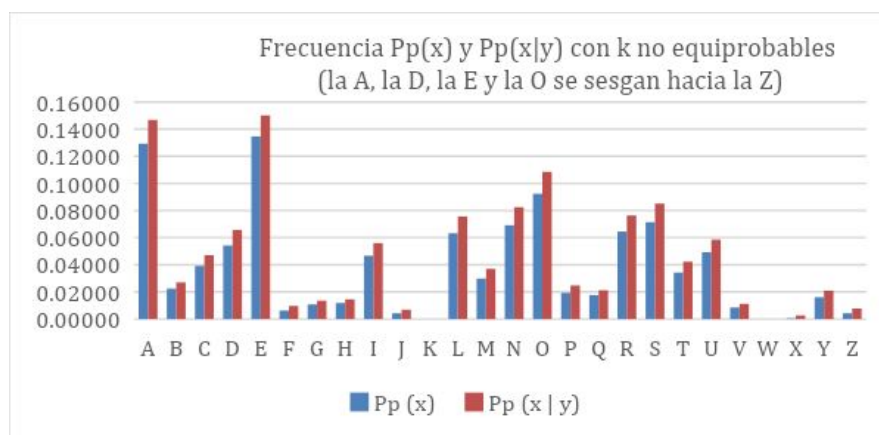
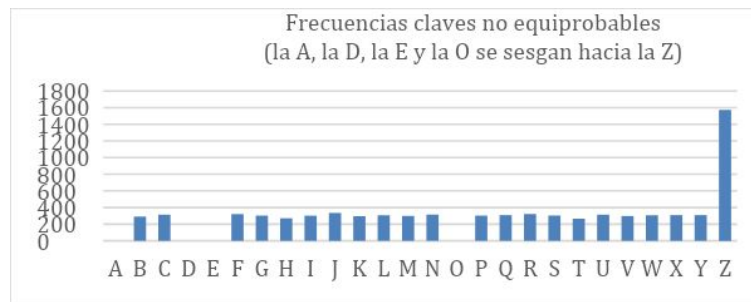
Para un texto más pequeño se observa cómo, aunque las probabilidades condicionadas se aproximan a la probabilidad de la letra en el texto plano, sobre todo en las letras más frecuentes, no son iguales.



Esto nos determina que para comprobar empíricamente nuestros resultados tenemos que obtenerlos a partir de un texto lo suficientemente largo para poder verificar la seguridad perfecta.

Claves no equiprobables:

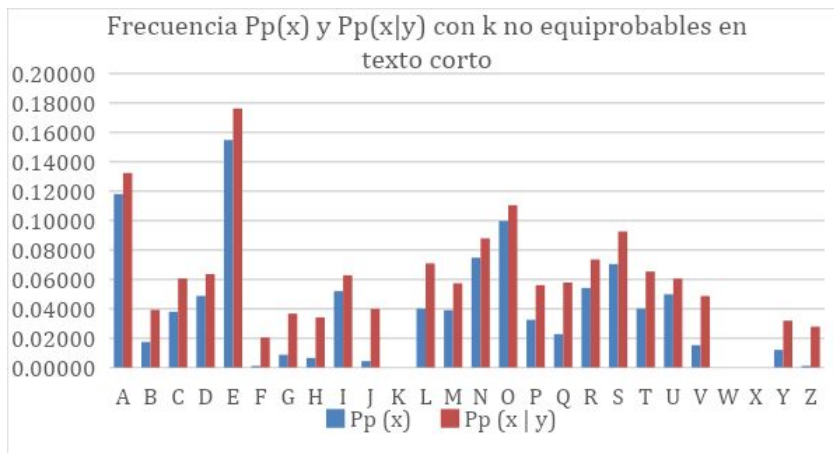
- Caso 1: Aquí hemos roto la equiprobabilidad sesgando las claves 'A', 'D', 'E' y 'O' hacia la 'Z' obteniendo los siguientes resultados:



Si se observa el histograma que tiene la media de las probabilidades condicionadas ya se observa una diferencia entre la probabilidad condicionada de cada letra y la probabilidad de la letra en el texto claro. Al ser una media parece que la diferencia no es tan grande, pero si tomamos, por ejemplo, el caso de la letra 'A' igual que antes obtenemos:

$p(A A) = 0.000000$	$p(A B) = 0.146132$	$p(A C) = 0.106628$	$p(A D) = 0.000000$
$p(A E) = 0.000000$	$p(A F) = 0.163194$	$p(A G) = 0.142857$	$p(A H) = 0.131757$
$p(A I) = 0.212121$	$p(A J) = 0.155629$	$p(A K) = 0.098551$	$p(A L) = 0.106164$
$p(A M) = 0.098901$	$p(A N) = 0.091644$	$p(A O) = 0.000000$	$p(A P) = 0.137931$
$p(A Q) = 0.105590$	$p(A R) = 0.125000$	$p(A S) = 0.131868$	$p(A T) = 0.100304$
$p(A U) = 0.148551$	$p(A V) = 0.115830$	$p(A W) = 0.140684$	$p(A X) = 0.159011$
$p(A Y) = 0.133803$	$p(A Z) = 0.476718$		

Si repetimos el proceso para un texto más corto:

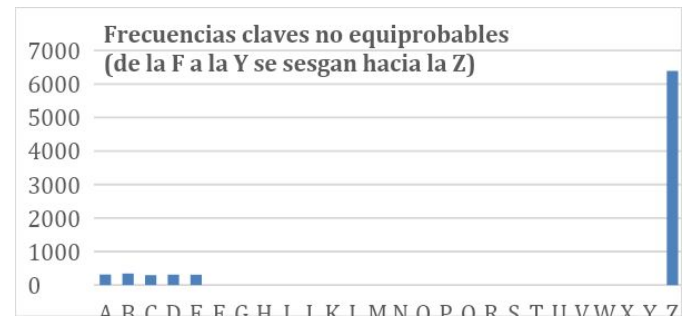
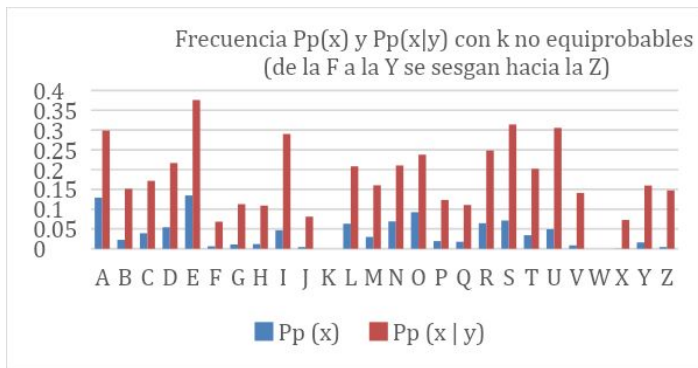


Se observa como la diferencia es mayor, es decir, no hay seguridad perfecta ninguna.

P(X)		Media Pp ()
Pp(A)	0,12925	0,14677
Pp(B)	0,02246	0,02699
Pp(C)	0,03915	0,04706
Pp(D)	0,05421	0,06563
Pp(E)	0,13465	0,15008
Pp(F)	0,00627	0,00963
Pp(G)	0,01079	0,01353
Pp(H)	0,01180	0,01453
Pp(I)	0,04668	0,05599
Pp(J)	0,00427	0,00688
Pp(K)	0,00000	0,00000
Pp(L)	0,06325	0,07564
Pp(M)	0,02987	0,03698

P(X)		Media Pp ()
Pp(N)	0,06914	0,08243
Pp(O)	0,09236	0,10855
Pp(P)	0,01933	0,02473
Pp(Q)	0,01769	0,02123
Pp(R)	0,06450	0,07637
Pp(S)	0,07140	0,08509
Pp(T)	0,03426	0,04230
Pp(U)	0,04932	0,05855
Pp(V)	0,00853	0,01121
Pp(W)	0,00000	0,00000
Pp(X)	0,00050	0,00263
Pp(Y)	0,01606	0,02095
Pp(Z)	0,00427	0,00775

- Caso 2: Aquí, para ver más claramente la diferencia entre la equiprobabilidad de las claves y la no equiprobabilidad, se han sesgado todas las claves desde la 'F' hasta la 'Y' hacia la 'Z' en el texto largo de "El Quijote". No se ha realizado la prueba para el texto corto pues ya no tiene sentido a la vista de los resultados anteriores.



Aquí se puede observar que la diferencia es mucho más considerable.

Implementación del DES

Programación del DES

Para implementar el DES en modo CBC hemos creado el programa *desCBC.c*, que lo que hace es generar una clave de cifrado si no se le pasa ninguna por argumento, comprobar la paridad de la clave recibida (porque la que genera el programa sí que aseguramos que cumpla la paridad) y realizar el proceso que haría el DES en modo CBC: si estás cifrando, se hace la XOR del primer bloque del mensaje plano con el IV, se pasa por el DES y se genera un mensaje cifrado, el cual será el IV para el siguiente bloque. Si estás descifrando, se realiza la inversa, es decir, se pasa el bloque cifrado por el DES, se realiza la XOR con el IV y esto genera el bloque plano correspondiente, y para el siguiente bloque el mensaje cifrado que entró en este será el IV del siguiente.

También hemos creado la librería *des.h*, que contiene las constantes que nos proporcionan como material de la asignatura y las tablas de permutaciones, expansiones, S-BOXES, etc.

Donde reside toda la lógica del algoritmo es en el módulo *funcionesDES.c* y la librería *funcionesDES.h* para importarlo en diferentes sitios. En este módulo se han implementado 5 funciones:

- *findParity*: esta simple función comprueba la paridad de la clave de 64b que entra al DES. Se ha obtenido de [stackoverflow](#).
- *sbox*: aquí implementamos las SBOXES del DES, que son 8 cajas que reciben 6 bits de entrada cada una y devuelven 4 bits a la salida cada una. De los 6b de entrada a1a2a3a4a5a6, la fila de la SBOX viene determinada por los bits a1a6, y la columna por los bits a2a3a4a5. Con máscaras podemos extraer y juntar estos bits para indexar con ellos en la SBOX correspondiente, asegurándonos de que la salida de dicha SBOX se guarde en la posición correcta (es decir, vamos desplazando hacia la izquierda de forma que tengamos, al acabar, la salida de la SBOX1 al principio y no al final). Después, las máscaras deben actualizarse para la siguiente SBOX, es decir, desplazarse 6 posiciones. Es importante notar que trabajamos todo el rato con representación de 64b, aunque lo que queramos guardar realmente ocupe menos. Esto nos facilita enormemente el trabajo y el padding automatizado, pero tendremos que prestar atención a que las cosas estén en la posición de menor peso de las

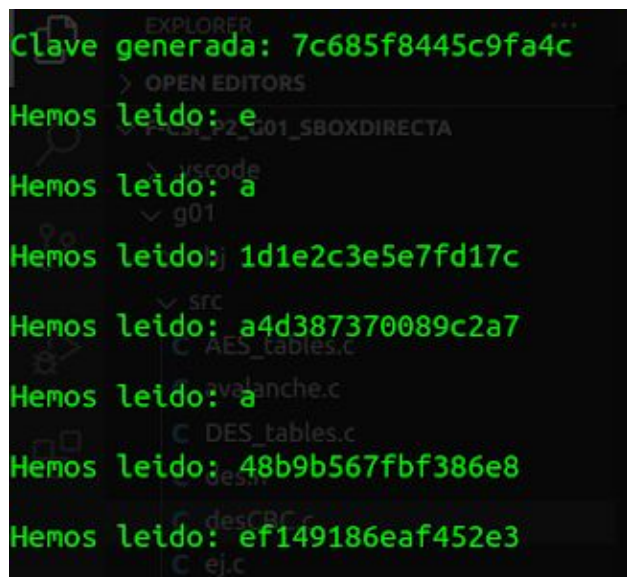
variables donde las guardemos, y luego ir desplazándolas para dejarlas como queremos.

- *generarSubclaves*: partiendo de una clave de 64b, se van a genera 16 subclaves, una para cada ronda del DES. Para ello, primero realiza una permutación y compresión de la clave de entrada para eliminar los bits de paridad, quedando una clave de 56b. Esta se divide en dos trozos de 28b cada uno, y a continuación se generan las 16 subclaves. Por cada una, se realiza un desplazamiento circular a la izquierda según la ronda que sea en ambas partes, se vuelven a combinar y esta clave se permuta y comprime a 48b.
- *permutar*: esta función realiza una permutación genérica. Como el algoritmo del DES está constantemente permutando y expandiendo, hemos decidido implementar una función que lo automatice, ya que el grueso de estas funcionalidades es el mismo. Esta función recibe el mensaje que hay que permutar, la tabla que se usará para la permutación (IP, IP_INV, E, PC1, etc.) y el tamaño de la entrada y de la salida. El mensaje a permutar vendrá en un `uint64_t`, como todo lo que estamos utilizando en el programa para mayor comodidad, pero, aunque esté guardado en este formato, podría en realidad ocupar menos bits y estar relleno de ceros. Es por ello por lo que se indica el tamaño de la entrada y de la salida deseada: así, además, sabemos cuándo hay que realizar una expansión y cuándo una compresión, cosa que realizaremos mediante desplazamientos según sea expandir o comprimir. Luego, la permutación simplemente es recorrer la tabla que nos indiquen por argumento y utilizar máscaras de bits para extraer el bit indicado en cada posición de la tabla y llevarlo a la posición correspondiente. Por ejemplo, si quieres llevar el bit 50 a la posición 1, entonces necesitas una máscara que tenga un 1 en la posición 50 y hacer la AND de la entrada con la máscara, así te habrás quedado ese bit únicamente. Luego, tendrás que desplazarlo a la posición 1, y finalmente hacer la OR del mensaje final y ese desplazamiento, así conservas lo que ya hubiera previamente y actualizas ese bit.
- *des*: es el algoritmo en sí que va a usar las funciones anteriores. Dado un mensaje y una clave, realiza el proceso de rondas Feistel visto en la asignatura: permutación inicial del mensaje, partición en dos trozos, 16 rondas en las que se pasa la parte derecha por la función F y se realiza la XOR con la parte izquierda, y swap. Al acabar las rondas, swap final y permutación inicial invertida. La función F consiste en hacer una expansión de la parte derecha, la XOR con la clave de esa ronda, pasar por las SBOXES y realizar una permutación del resultado. Para el tema de las claves se observa que el algoritmo es el mismo, pero, en el caso de que estemos descifrando, las claves se utilizarían en el orden contrario. Para esto tenemos un flag que si está a 1 significa que se debe cifrar y, si está a 0, descifrar, y entonces se invierten las claves generadas.

El programa se ha ido desarrollando siguiendo el ejemplo que nos proporcionan en Moodle “DES algorithm illustrated”, obteniendo los mismos resultados y, por tanto, comprobando que funciona correctamente según este ejemplo.

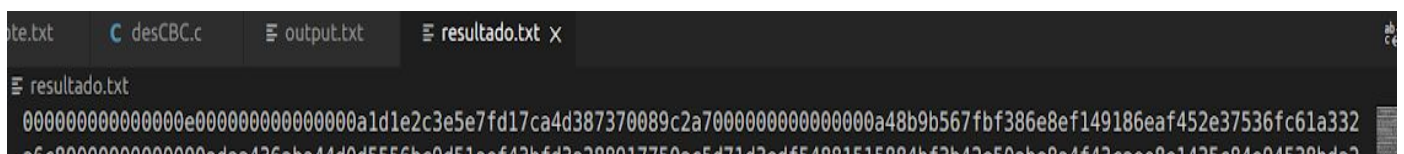
Nuestra implementación funciona correctamente para entradas que sean múltiplo de 64b. Hemos tratado de implementar el padding para el caso de entradas que no fueran múltiplos, pero no hemos sido capaces. Entendemos perfectamente cómo debería realizarse este proceso: el último bloque debería completarse con ceros hasta la longitud deseada, y al descifrar esos ceros se eliminarían de forma trivial. Sin embargo, no nos ha dado tiempo a seguir investigando este aspecto en el lenguaje C, así que si la entrada no es múltiplo de 64b el bloque final estará mal cifrado. Entendemos que esto es una limitación, pero creemos que el problema viene más bien por el lenguaje C en sí, y no tanto un problema de comprensión del problema criptográfico planteado.

Para comprobar el correcto funcionamiento del programa, cifraremos un fichero de texto largo, como el quijote. El programa permite que tú le introduzcas la clave con la que quieres cifrar (y comprobará que cumpla el criterio de paridad) o te generará una él mismo si tú no le proporcionas ninguna. En este caso, que es el que se muestra en la figura inferior de ejemplo, nos genera la clave 7c685f8445c9fa4c, que como es generada por el propio programa cumple la paridad. Hemos usado el IV 0000000000000000. Ambos parámetros, clave e IV, tendrán que estar en hexadecimal, como se nos ha indicado en diversas ocasiones. A continuación, hemos mostrado los bloques que se van leyendo del fichero de entrada para demostrar después que el descifrado funciona bien (no se muestran los ceros del comienzo, pero vamos que el tamaño de cada bloque es 64b).



```
Clave generada: 7c685f8445c9fa4c
Hemos leído: e
Hemos leído: a
Hemos leído: 1d1e2c3e5e7fd17c
Hemos leído: a4d387370089c2a7
Hemos leído: a
Hemos leído: 48b9b567fbf386e8
Hemos leído: ef149186eaf452e3
```

El resultado del cifrado del Quijote lo guardaremos en otro fichero, el cual leeremos y desciframos con la clave que nos ha generado el programa antes. El resultado del descifrado lo guardamos en un fichero final, del cual se muestra el principio en la imagen inferior. Podemos ver cómo ha descifrado la e, la a, el 1d1e2c3e57fd17c... El descifrado, por tanto, funciona correctamente.



```
00000000000000000000000000000000a1d1e2c3e5e7fd17ca4d387370089c2a70000000000000000a48b9b567fbf386e8ef149186eaf452e37536fc61a332
a6c80000000000000000000000000000adaa436aba44d0d5556bc0d51aef43bf43e288917750ac5d71d3edf54881515884bf3b42e50abe8a4f43c9e8e1435c84e94538bda2
```

Queda clara la limitación del programa: se nos está mostrando el resultado final en hexadecimal, y no en binario, como debería ser, para que al abrir el fichero viéramos el contenido tal cual estaba en el fichero original. Esto, de nuevo, tiene que ver con la incapacidad de realizar esto en el lenguaje C, a pesar de abrir los ficheros tanto de entrada como de salida en modo lectura binaria y escritura binaria respectivamente. No obstante, queda demostrado que el cifrado y el descifrado son correctos.

ECB en textos largos e imágenes

En el cifrado ECB a cada bloque se le aplica la misma clave por lo que dos bloques idénticos tendrán el mismo cifrado. En el cifrado CBC cada bloque depende del anterior por lo que dos bloques iguales tendrán cifrado distinto.

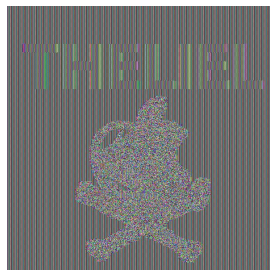
En un texto largo cifrado con ECB es muy fácil localizar patrones que se repiten por lo que es muy fácil atacarlo.

Lo mismo pasa con las imágenes que suelen tener zonas uniformes, con el mismo color, que se codificarían de la misma manera por lo que, igualmente, sería fácilmente atacado. Debido a que la mayoría de las imágenes de hoy en día como los tipos jpg, png o tiff utilizan compresión en las imágenes no podemos hacer las pruebas correspondientes con el ECB pero a continuación se incluyen unas imágenes para ver cómo serían en “texto plano”, cifrado con ECB y cifrado con CBC para ver la diferencia:

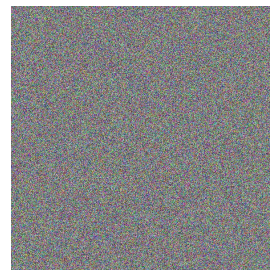
THELIEL



Original



ECB



CBC

Programación del Triple DES

El triple DES CBC es algo muy parecido al DES CBC anterior. En este caso, la clave se dividirá en 3 claves, por tanto, ocupará 192b. El programa comprobará la paridad de esas 3 claves, ya que se aplicará DES con cada una y es necesario que cumplan todas este criterio. El triple DES realiza $\text{DES}_{k3}(\text{DES}_{k2_inv}(\text{DES}_{k1}(p)))$ para cifrar (y esto sería el equivalente al DES anterior, por tanto estos tres operaciones son la caja negra que cifra. Previamente, por tanto, se realiza la XOR con el IV y después se actualiza el IV al igual que haríamos con el DES CBC) y $\text{DES}_{k1_inv}(\text{DES}_{k2}(\text{DES}_{k3_inv}(c)))$ para descifrar (esta sería la caja negra, y se actualizaría el IV igual que explicamos para el descifrado DES CBC).

Podemos comprobar que funciona correctamente igual que en caso anterior: vemos los bloques que va cifrando y después, al descifrar, podemos comprobar que coinciden. Tiene las mismas limitaciones que el DES del apartado anterior.

```
Hemos leído: e
C: funcionesDES.c
Hemos leído: a
C: funcionesDES.h
C: nonLinear.c
Hemos leído: 4099f94c1ef96213
C: nonLinearOn.c
Hemos leído: a5a843f05ed9b01c
C: seg-boy.c
C: seg-perf.c
Hemos leído: a
C: des.c
C: avalanche
Hemos leído: 96ce607790404cc1
C: des.txt
Hemos leído: a5a843f05ed9b01c
C: des.c
```

00000000000000e00000000000000a099f94c1ef96213a5a843f05ed9b01c0000000000000a96ce607790404cc1a5a843f05ed9b0
b01c000000000000ada2da0bb2398df0d03a5a843f05ed9b01ca5a843f05ed9b01ca5a843f05ed9b01ca5a843f05ed9b01ca5a843f05e
5ed9b01ca5a843f05ed9b01ca5a843f05ed9b01c0000000000000a0000000000000e0000000000000df8a9c7ef4e89f85ba5a84

Principios de diseño del DES

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

Para estudiar la no linealidad del DES se ha creado el programa *nonLinear.c*, que realiza cierta cantidad de tests para comprobar que $F(A) + F(B) \neq F(A + B)$, donde el + indica la operación XOR. Para ello se generan un A y un B de 64b como sería la entrada del DES que pasa por las diferentes rondas. A continuación, se toman los 32b inferiores, como se tomaría en una ronda del DES la parte derecha. Después se realiza una expansión de 32b a 48b de esta parte y se realiza la XOR con la clave (que se ha generado aleatoriamente para estas pruebas, pero en el DES real sería la clave de cada ronda). Más adelante se pasa este resultado por las S-BOXES y, finalmente, se permuta, obteniendo así la salida de la función F. Esto se realiza para A, para B y para A + B. Si el resultado de $F(A) + F(B)$ es distinto del resultado de $F(A + B)$ para ese test, se incrementa un contador. Al final se muestra este resultado en función del número de tests ejecutados. En la imagen inferior podemos ver cómo, efectivamente, se cumple el criterio.

```
-----COMPROBACION DE LA NO LINEALIDAD DEL DES-----
La no linealidad del DES establece que  $F(A + B) \neq F(A) + F(B)$ 
A continuacion se muestra cuantas veces se ha cumplido esto en las pruebas ejecutadas:
Diferencias: 100000 / 100000
```

Estudio del Efecto de Avalancha

Para estudiar el efecto avalancha se ha creado el programa *avalanche.c*, en el que realizamos una serie de tests para comprobar que $P(c_i=1 \mid /b_j) = P(c_i=0 \mid /b_j) = 0.5$, para cada c_i bit de salida y cada b_j bit de entrada. En cada test, generamos una entrada aleatoria de 48b para las S-BOXES. Como en cada S-BOX entran 6 bits, vamos a ir complementando cada uno de ellos cada vez, para todas las cajas. A continuación, comprobamos cada bit de la salida y vemos si tiene un valor de 1, y aumentamos un contador para cada uno de los 32b de la salida. Cuando acaban los tests, simplemente se obtiene el porcentaje de cambio de cada bit: se divide la cantidad de cambios de ese bit entre los 6 bits que hay posibles a la entrada de una caja, esto se multiplica por 100 para sacar el porcentaje y finalmente se divide entre el total de tests realizados.

Cuanto más test hagamos mejor se verá el efecto. Al principio habíamos redondeado el resultado de la probabilidad y lo habíamos mostrado sin decimales para que se viera mejor, pero después nos dimos cuenta de que redondeando estábamos perdiendo mucha información (porque los cambios en las probabilidades con cada ejecución eran muy pequeños, pero al redondear y hacerlo sin decimales no se reflejaban correctamente y parecía que siempre saliera lo mismo) y decidimos eliminar esto. Podemos observar en la siguiente figura que, efectivamente, todos los bits cambian con una probabilidad de un 50% aproximadamente.

```
-----COMPROBACION DEL EFECTO AVALANCHA DEL DES-----  
El Strict Avalanche Criterion establece que para todo i,j,  $P(c_i=1 \mid /b_j) = P(c_i=0 \mid /b_j) = 0.5$ ,  
siendo  $c_i$  un bit de la salida de la SBOX y  $b_j$  uno de la entrada.  
A continuacion se muestra la probabilidad de cambio de cada bit tras las pruebas ejecutadas:  
Bit 1 --> 50.15%  
Bit 2 --> 50.10%  
Bit 3 --> 50.24%  
Bit 4 --> 50.02%  
Bit 5 --> 50.30%  
Bit 6 --> 50.05%  
Bit 7 --> 49.85%  
Bit 8 --> 50.02%  
Bit 9 --> 49.83%  
Bit 10 --> 49.97%  
Bit 11 --> 50.02%  
Bit 12 --> 50.09%  
Bit 13 --> 49.65%  
Bit 14 --> 50.04%  
Bit 15 --> 50.02%  
Bit 16 --> 49.87%  
Bit 17 --> 50.07%  
Bit 18 --> 49.97%  
Bit 19 --> 49.92%  
Bit 20 --> 50.03%  
Bit 21 --> 46.88%  
Bit 22 --> 52.87%  
Bit 23 --> 50.17%  
Bit 24 --> 50.11%  
Bit 25 --> 49.80%  
Bit 26 --> 50.19%  
Bit 27 --> 49.96%  
Bit 28 --> 50.15%  
Bit 29 --> 49.57%  
Bit 30 --> 50.15%  
Bit 31 --> 49.89%  
Bit 32 --> 50.03%
```

Principios de diseño del AES

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

“El criptoanálisis lineal es un método de ataque basado en pares de bytes de texto en claro criptograma empleados para obtener información sobre la clave principal de cifrado”.

“El ataque lineal más efectivo contra DES se logra mediante la combinación de expresiones lineales sobre una vuelta concreta de la red Feistel con expresiones referentes a los bits de salida de alguna S-Box en concreto. En el caso del algoritmo AES-Rijndael, se ha probado que este tipo de ataque no es efectivo siempre y cuando no se implemente una versión del algoritmo con menos de cuatro rondas en la red interna”.¹

La S-Box del AES se construye usando la inversa multiplicativa en el campo finito de Rijndael y se dice que el inverso multiplicativo es altamente no lineal, es decir “altamente no afín”. Posteriormente, a ese inverso multiplicativo se le hace una transformación afín. Esto hace que se minimice la correlación entre las transformaciones lineales de bits de entrada/salida y, al mismo tiempo, minimiza la probabilidad de propagación de la diferencia.

Al igual que hemos hecho con el DES, para estudiar la no linealidad del AES se ha creado el programa *nonLinear_AES.c*, que realiza cierta cantidad de tests para comprobar que $F(A) + F(B) \neq F(A + B)$, donde el $+$ indica la operación XOR.

Se han generado números aleatorios (a y b) con la función *rand()* de C y se han pasado por las SBOXES, tanto Directa como Inversa, así como $a+b$. Si el resultado de $F(A) + F(B)$ es distinto del resultado de $F(A + B)$ para ese test, se incrementa un contador. Al final se muestra este resultado en función del número de tests ejecutados.

En la imagen inferior podemos ver cómo, efectivamente, se cumple el criterio para ambas cajas donde se consigue casi en el 100% de los casos (99,60%). Esta pequeña diferencia probablemente sea porque la función *rand()* de C no es tan aleatoria como se presupone pero en cualquier caso, este pequeño 0,4% de los casos en los que no se ha cumplido se solventarían con las posteriores pasos de difusión por medio de unas transformaciones lineales (*ShiftRow* y *MixColumn*) y un paso de mezcla con la clave (*KeyMixing*).

1


```

-----COMPROBACION DE LA NO LINEALIDAD DEL AES-----
La no linealidad del AES establece que  $F(A + B) \neq F(A) + F(B)$ 
A continuacion se muestra cuantas veces se ha cumplido esto en las pruebas ejecutadas con la caja
SBOX DIRECTA:
Diferencias: 99591 / 100000

```

```

-----COMPROBACION DE LA NO LINEALIDAD DEL AES-----
La no linealidad del AES establece que  $F(A + B) \neq F(A) + F(B)$ 
A continuacion se muestra cuantas veces se ha cumplido esto en las pruebas ejecutadas con la caja
SBOX INVERSA:
Diferencias: 99608 / 100000

```

Generación de las S-boxes AES

S-box directa.

La S-box asigna una entrada de 8 bits, c , a una salida de 8 bits, $s = S(c)$. Tanto la entrada como la salida se interpretan como polinomios sobre $GF(2)$. La generación de las S-boxes constan de dos fases:

- Primero, la entrada se asigna a su inverso multiplicativo en $GF(2^8) = GF(2)[x] / (x^8 + x^4 + x^3 + x + 1)$, el campo finito de Rijndael. El cero, como identidad, se asigna a sí mismo pues no tiene inverso. Se usa el inverso multiplicativo por tener buenas propiedades de no linealidad como hemos podido comprobar anteriormente.
- Posteriormente, el inverso multiplicativo se transforma usando la siguiente transformación afín invertible:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

donde $[b'_7, \dots, b'_0]$ es la salida de la caja S y $[b_7, \dots, b_0]$ es el inverso multiplicativo como vector.

Para la primera parte hemos implementado el algoritmo de Euclides extendido trabajando a nivel de bit.

A continuación, se muestra una salida por pantalla del cálculo de los 255 inversos multiplicativos necesarios para el cálculo de la Sbox directa (el cero no tiene inverso)

multiplicativo) aunque, en la versión final, este cálculo no se muestra por pantalla ni en fichero:

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

Para la transformación afín, la vista en clase se realiza de la siguiente manera donde b' , b y c son matrices de 8 bits, c es $01100011_2 = 63_{16}$ y los subíndices indican una referencia al bit indexado.

$$b'_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i$$

Una fórmula equivalente es la siguiente donde la transformación afín es la suma de múltiples rotaciones del byte como vector, donde la suma es la operación XOR, b representa el inverso multiplicativo, y \lll es un desplazamiento circular a nivel de bits a la izquierda, y la constante $63_{16} = 01100011_2$. Esta es la fórmula utilizada en la práctica por resultarnos más fácil de implementar a nivel de programación.

$$s = b \oplus (b \lll 1) \oplus (b \lll 2) \oplus (b \lll 3) \oplus (b \lll 4) \oplus 63_{16}$$

Un ejemplo de la salida de nuestro programa es el siguiente:

63	7c	77	7b	f2	6b	6f	c5	30	1	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
4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	0	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	2	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
e0	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	b	db
e7	32	3a	a	49	6	24	5c	c2	d3	ac	62	91	95	e4	79
a	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	8b
70	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
e1	3e	b5	66	48	3	f6	e	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	d	bf	e6	42	68	41	99	2d	f	b0	54	bb	16

S-box inversa.

Para la generación de la S-box inversa hay que hacerlo al revés, hacer primero la transformación afín y después el inverso multiplicativo. La transformación afín se realiza de la siguiente manera donde, al igual que en la directa, la b , b' y d son matrices de 8 bits, d es $00000101_2 = 05_{16}$ y los subíndices indican una referencia al bit indexado:

$$b'_i = b_{(i+2)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus d_i$$

Esta transformación se puede representar de la siguiente manera:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Al igual que con la directa, en nuestra implementación hemos utilizado la siguiente fórmula por ser equivalente.

$$b = (s \lll 1) \oplus (s \lll 3) \oplus (s \lll 6) \oplus 5_{16}$$

De igual manera, para el inverso multiplicativo hemos utilizado la implementación del algoritmo de Euclides extendido.

Un ejemplo de la salida por pantalla de nuestro programa es el siguiente:

5	9	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	b	42	fa	c3	4e
8	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	0	8c	bc	d3	a	f7	e4	58	5	b8	b3	45	6d
0	2c	1e	8f	ca	3f	f	2	c1	af	bd	3	1	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	e	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	7	c7	31	b1	12	10	59	27	80	ec	5f
60	51	7f	a9	19	b5	4a	d	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	4	7e	ba	77	d6	26	e1	69	14	63	55	21	c	7d

Ambas tablas se vuelcan también a fichero, como se solicitaba en la práctica, con la opción [-o nombre_fichero].