Nuevas ideas en criptografía simétrica

Autor: Damián Mateos Ramos (damymr@gmail.com)

Este es el texto que corresponde al trabajo de fin de carrera presentado a fin de obtener el título de Ingeniero Informático por la Universidad de León.

Deseo agradecer a Francisco Javier Cabida Fernández (strapping@gmail.com) por soportarme tanto tiempo con mis locas ideas y por animarme a publicar este trabajo.

Deseo asimismo agradecer a Kriptopolis (www.kriptopolis.com) la posibilidad de publicarlo.

1 Índice

2 Introducción

La seguridad informática es un aspecto muy importante en nuestra vida cotidiana, no siempre nos percatamos de ello, pero cada vez que utilizamos el móvil o la tarjeta de crédito, cada vez que nos comunicamos con nuestro banco por Internet, estamos haciendo uso de ella. Se nos presenta como algo transparente, buscando siempre una sencillez que contrasta con la tremenda complejidad que supone lograrla.

Actualmente existen dos vertientes en criptografía (cifrado de datos) que representan dos formas de resolver el problema: criptografía de clave pública y criptografía de clave privada, cada una con sus ventajas e inconvenientes.

Los últimos avances y estudios permiten ver la despiadada carrera existente para crear y desbaratar los algoritmos utilizados. Recientemente se ha logrado romper una clave RSA de 1024 bits, cuando se esperaba que pudieran utilizarse hasta 10 años más; se han encontrado deficiencias en los conocidos algoritmos de hashing SHA-1 y MD5, los más utilizados del mundo, con especial severidad en el MD5 en el que se ha conseguido que el hash de dos conjuntos de datos cualesquiera y totalmente diferentes coincidan. Tampoco se libran los criptosistemas de clave privada: la aparición del criptoanálisis imposible ha supuesto una tremenda revolución y algoritmos que antes se consideraban inmunes a ataques más eficientes que la fuerza bruta ya no lo sean. Otra amenaza importante es la aparición de la futura computación cuántica, mucho más poderosa que la computación actual.

En este panorama la única forma que existe actualmente para lograr mantener la seguridad es el aumento del tamaño de las claves utilizadas, y el desarrollo de nuevos algoritmos.

La intención del presente trabajo es obtener un criptosistema simétrico que pueda ofrecer un alto grado de seguridad, trabajando con claves mayores de las utilizadas actualmente y con una característica muy

interesante: el mismo algoritmo será capaz de cifrar y descifrar, y utilizará operaciones sencillas y será muy sencillo de implementar.

Sin embargo hoy día no existe ninguna *receta* que permita abordar el diseño de este tipo de criptosistemas, es más bien un proceso creativo, así pues otro de los objetivos es mostrar un conjunto de ideas y razonamientos de aplicación directa en el diseño de los mismos, proveyendo así de unas directrices y una metodología que simplifique ese diseño, junto a unas herramientas que permitan analizar posibles debilidades.

Para ello se estudiará la *Red de Feistel* y se propondrá una generalización de la misma y un método sencillo de construcción de redes que sean capaces de cifrar y descifrar los datos sin ser necesaria ninguna modificación. También se tratarán las operaciones a realizar en esa red, cuales son aconsejables y cuales no según los objetivos que nos marquemos; y por último se hablará del tratamiento y la forma de utilización de la clave para maximizar la seguridad y eliminar algunos de los riesgos conocidos.

En la segunda parte del trabajo será en la que se desarrolle el diseño del algoritmo ya mencionado, basándose en parte de las ideas de la primera parte y efectuándole una serie de test con el objetivo de determinar las características que posea. También le serán aplicadas algunas herramientas expuestas en la primera parte cuyo objetivo es detectar debilidades comunes.

Es necesario mencionar la escasez de recursos computacionales con los que se ha abordado la tarea y la gran amplitud del trabajo, lo que ha llevado a hacer mayor hincapié en las bases teóricas que en la puesta en práctica. Además es importante reseñar que no se ha encontrado bibliografía que trate el tema del diseño de criptosistemas, así pues, salvo la descripción de la red de Feistel todas las demás ideas y herramientas han sido creadas en exclusiva para el presente trabajo y se pretende que varias de ellas sean realmente innovaciones.

Sin más, disfruten de la lectura.

3 Red de Feistel

Se conoce como red de Feistel a una estructura descrita por Horst Feistel, criptógrafo de IBM, y utilizada frecuentemente en algoritmos de encriptación de datos como Lucifer, DES, FEAL, Blowfish y otros. Esta estructura presenta unas características muy interesantes entre las que destaca que la codificación y la decodificación sean muy similares o en ciertos casos idénticas (autoreversivilidad). A la hora de implementar los sistemas en hardware, esta propiedad consigue reducir la complejidad y el coste de los circuitos, siendo sólo necesario modificar la clave.

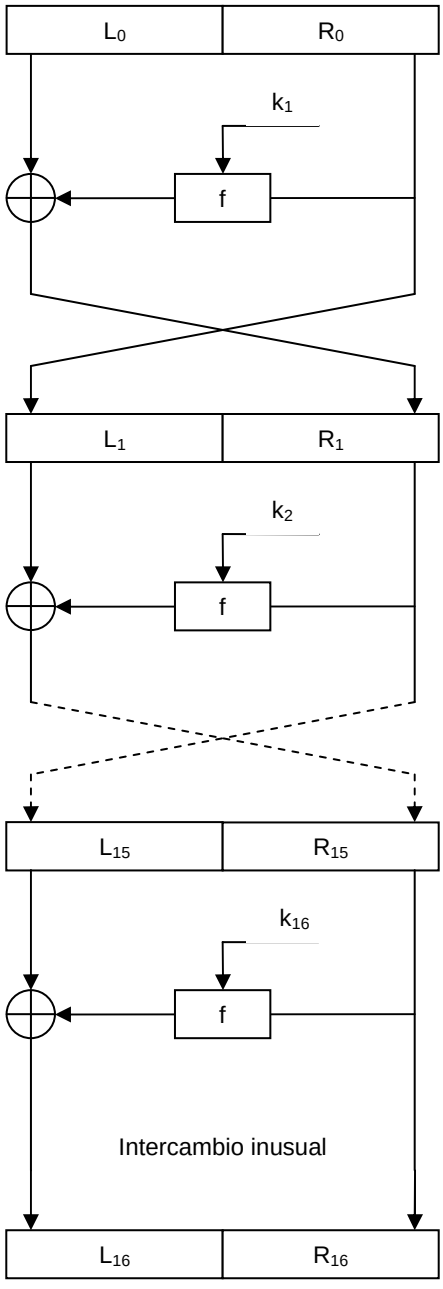


Fig. 2.1

El funcionamiento de una red de Feistel es el siguiente:

1. Se divide el bloque inicial en dos partes: izquierda (L) y derecha (R).
2. A la parte derecha se le aplica una función *f* que aporte la confusión y la difusión adecuada. En esta función un lugar importante lo ocupa la clave (k_i), ésta debe permanecer en secreto y sólo la deben conocer el emisor y el receptor del mensaje.
3. El resultado de esta función es aplicado a la parte izquierda del bloque mediante un XOR
4. Se intercambian las dos partes y se itera el proceso, esta vez con los papeles cambiados.

Otra de las características que hacen de la red de Feistel muy utilizada es que la función *f* no tiene por qué ser inversible. De hecho, para el proceso de decodificación basta con cambiar el orden de las claves y volver a aplicar el algoritmo:

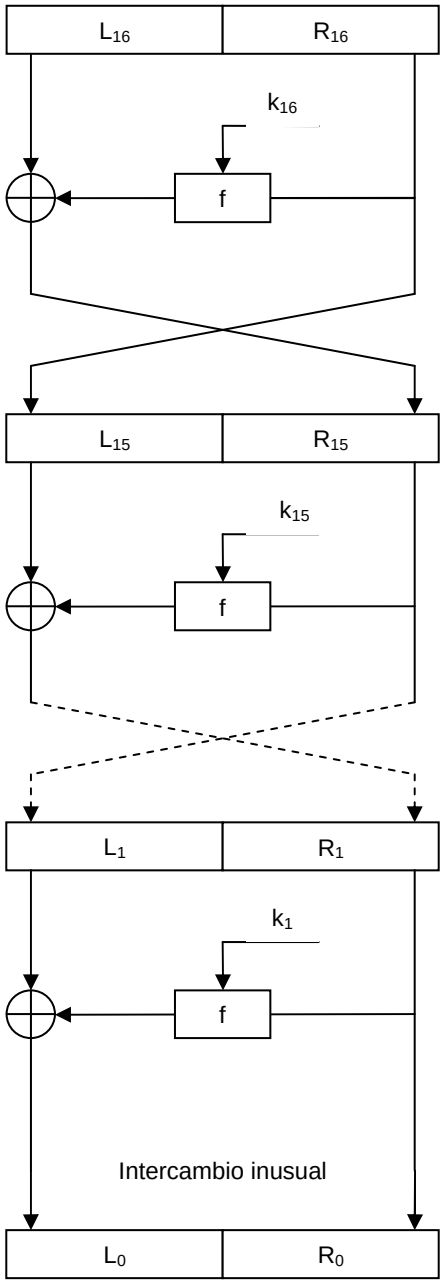


Fig. 2.2 Red de Feistel para desencriptar

La estructura de la decodificación es exactamente la misma que la de la codificación. Comprobamos que simplemente con cambiar las claves de orden podemos obtener el bloque original y en ningún momento se ha necesitado la inversa de la función f.

Con el uso de k_{16} aplicado a R_{16} podemos obtener R_{15} (y L_{15} sin hacer nada); si seguimos hacia atrás en la estructura de la encriptación se observa que se consigue la misma información (aunque los registros están intercambiados). Llegados al final de la cadena, se aplica *f* al registro R (siguiendo hacia atrás en la codificación, sería aplicar *f* al registro L) y el resultado aplicado a la otra mitad. En la decodificación no se produce un intercambio pero en la codificación hacia atrás sí: el resultado es que se invertiría el intercambio del principio del proceso de decodificación y el resultado es el mismo: se ha conseguido codificar y decodificar con la misma estructura, solamente hemos cambiado las claves.

Matemáticamente se expresa así:

$L_i = R_{i-1}$

$R_i = L_{i-1}$ **XOR** $f(R_{i-1}, k_i)$

El último intercambio no se suele hacer, o si se hace (por ejemplo con una implementación mediante iteración) se deshace. Normalmente se toma un número de rondas mayor o igual a tres y suele ser par. La red de 16 rondas mostrada corresponde a la utilizada por el sistema DES, también es la que utiliza Blowfish.

El pseudocódigo para una red de Feistel de 16 rondas es el siguiente:

1. Separar el bloque en dos mitades: L y R
2. Para i=0 hasta i=15
3. L = L **XOR** $f(R, k_i)$
4. Intercambiar L y R
5. Fin_bucle
6. Intercambiar L y R (deshacer el último intercambio)
7. Recombinar L y R

La mayoría de los algoritmos que utilizan como corazón la red de Feistel, emplean alguna forma de modificación de la misma que ayude a los propósitos para los que fue diseñado: así en DES se realizan dos permutaciones (una a la entrada de la red y otra a la salida). En Blowfish se aplica en cada ronda un XOR a R con unos valores precalculados, antes de la línea 3, y después del bucle se aplica tanto en la parte R como en la L.

4 Red de Feistel extendida

A continuación se propone una forma de crear diferentes redes con las mismas propiedades que las poseídas por las redes de Feistel: estructura idéntica para la encriptación y la desencriptación, y posibilidad de emplear funciones no inversibles pero que permitan a la red sí ser inversible.

4.1 Notación

El diseño de la red se hará mediante diagramas similares a los mostrados anteriormente. Las flechas indican por dónde circula la información, la función (que hace uso de una supuesta clave k, aunque no se represente) necesita conocer su entrada para poder calcular su salida y los intercambios se hacen mutuamente entre registros. Un registro es un conjunto de bits (por ejemplo 32 bits) en los que se divide el bloque original. Las funciones suelen tener una entrada adicional, una subclave calculada a partir de la clave secreta, que será obviada en algunos ejemplos.

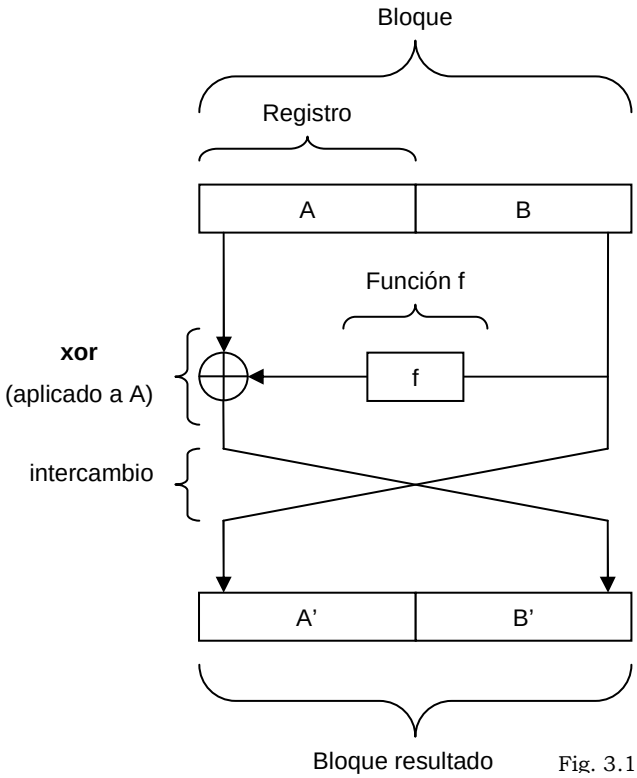


Fig. 3.1 Notación

Vamos a considerar que las claves se utilizan siempre de forma adecuada para la codificación y la decodificación, más adelante estableceremos qué orden se debe seguir.

4.2 El problema de la decodificación

La información que almacenan los registros de destino, junto con la clave) debe ser suficiente para reconstruir los registros iniciales. Las funciones no requerirán inversa, con lo cual será necesario siempre tener la información de la entrada de la función para calcular la de la salida. El camino que recorre la información en el diagrama es lo que denominamos **flujo**, se puede pensar que este flujo circula en paralelo, pero hay que diseñar el diagrama de forma que se puedan ejecutar las operaciones de forma secuencial. Esto es muy importante para que sea posible la decodificación, por ejemplo:

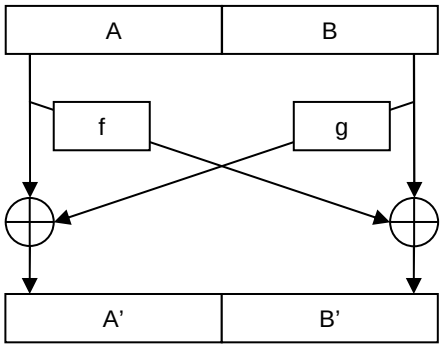


Fig. 3.2

Para poder decodificar A necesitamos conocer g(B) y por lo tanto B, y para poder decodificar B necesitamos f(A) y por lo tanto A: se entra en una situación verdaderamente difícil incluso para el receptor legítimo del mensaje. El flujo en el diagrama va en paralelo: las funciones g y f son calculadas antes de ser aplicadas mediante la operación xor a los bloques A y B. Si intentásemos secuencializar las funciones f y g de alguna forma, veríamos que es imposible puesto que cuando una es aplicada, se pierde la información de entrada para la otra. Las funciones tienen que poder ser secuencializadas, es decir, ser ejecutadas en un orden que permita disponer de los datos de entrada

necesarios para cada una cuando van a ser ejecutadas de forma atómica.

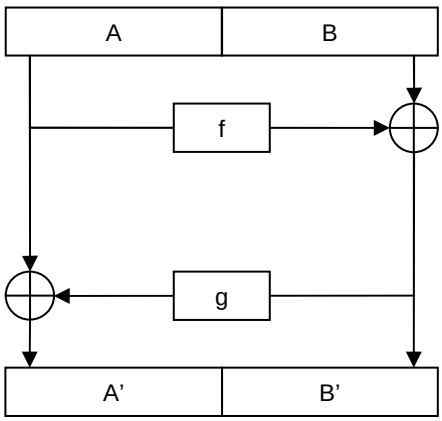


Fig. 3.3

En esta situación las funciones f y g tienen un orden de ejecución: es necesario calcular f(A) para conocer la salida del primer xor y de este modo poder calcular g(b). Existen ciertos casos en los que se pueden calcular funciones en paralelo y no interfieren con las demás: se da cuando podemos hacer la combinación mediante xor de más de dos caminos, por ejemplo:

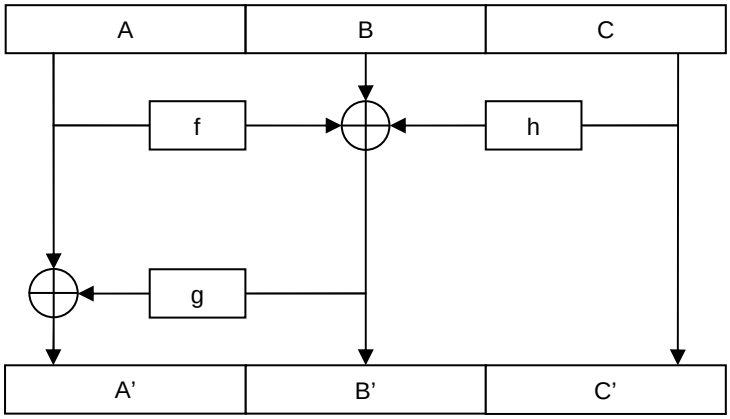


Fig. 3.4

En este caso la ejecución de f y h ha de ser en paralelo, y sin embargo la decodificación sería posible. La regla general es: **cuando a un registro se le va a aplicar una función, el valor anterior de ese registro no puede utilizarse para otras funciones incluida la actual**, porque quedará oculto tras la aplicación de la función. De todos modos, esto no viola el principio mencionado anteriormente de la seriabilidad de las funciones, ya que h y f se pueden aplicar en cualquier orden a B y el resultado es el mismo.

Otra de las posibles dificultades a la hora de decodificar puede ser la pérdida de información: si en algún lugar del diagrama se pierde el valor de un registro, aunque se cumpla la regla anterior no se podrá decodificar el bloque original.

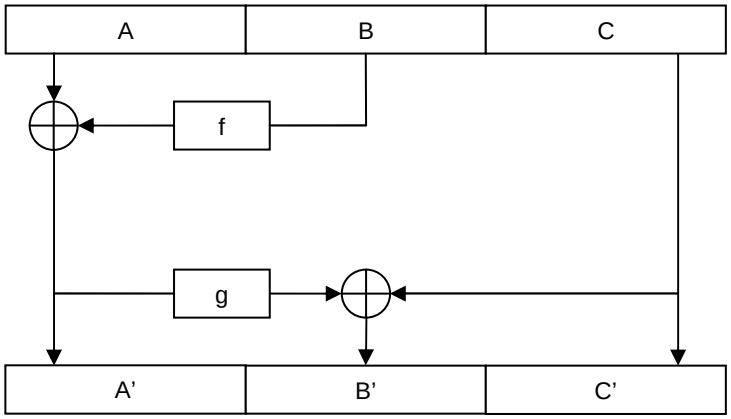


Fig. 3.5

En este caso es imposible reconstruir el valor del bloque original puesto que no hay forma de conocer el valor de B. Se puede deducir fácilmente la segunda regla: **debe de haber un camino que una de forma unívoca un registro origen y uno destino, por el cual no se atraviese ninguna función no reversible**.

Se puede comprobar fácilmente que estas dos reglas son suficientes para que un codificador de este tipo, por complicado que sea, funcione. El registro de destino al que está ligado un registro origen es el que tiene la información para descifrar este registro origen, en el camino no hay funciones no inversibles, únicamente pueden haber intercambios, xors (ambas inversibles) y funciones inversibles. El hecho de que las funciones sean serializables, nos garantiza que siempre podremos calcular la última (por la segunda condición, los datos de origen de la última función irán a un registro destino directamente) y deshacer el último xor; en este punto repetimos con la última función que no haya sido recalculada.

Una forma sencilla de comprobar que un diagrama es válido se expone a continuación.

Se crea una tabla con tantas filas como registros haya y tantas columnas como funciones e intercambios entre registros haya, más una para los registros iniciales. Ahora en cada columna se va apuntando el efecto de una función o un intercambio, teniendo siempre presente dos datos: de una fila sólo conocemos el último valor del registro apuntado y en una función se hace XOR (representado con una x) siempre con el dato presente en ese momento en la fila:

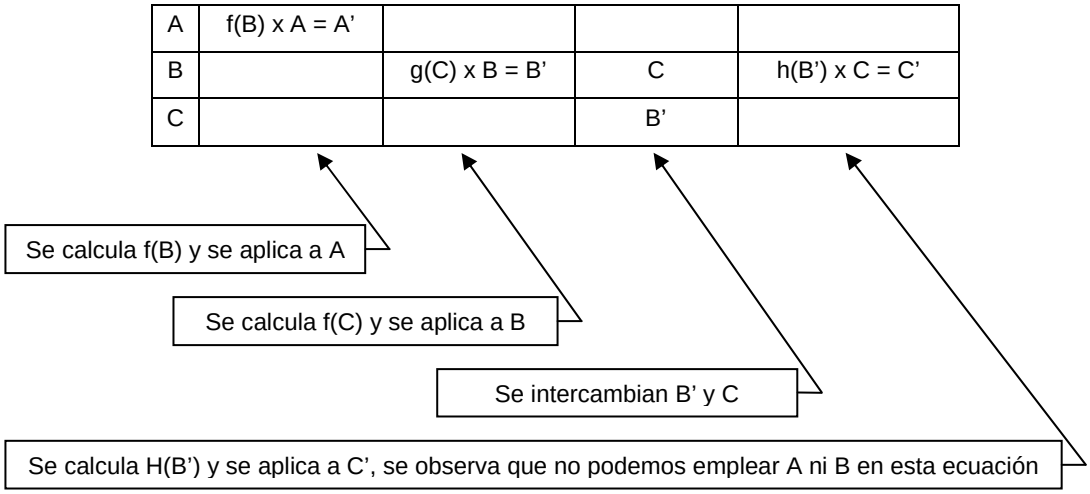


Fig. 3.6 Algoritmo para comprobar que una red está bien diseñada

Esta tabla corresponde al diagrama:

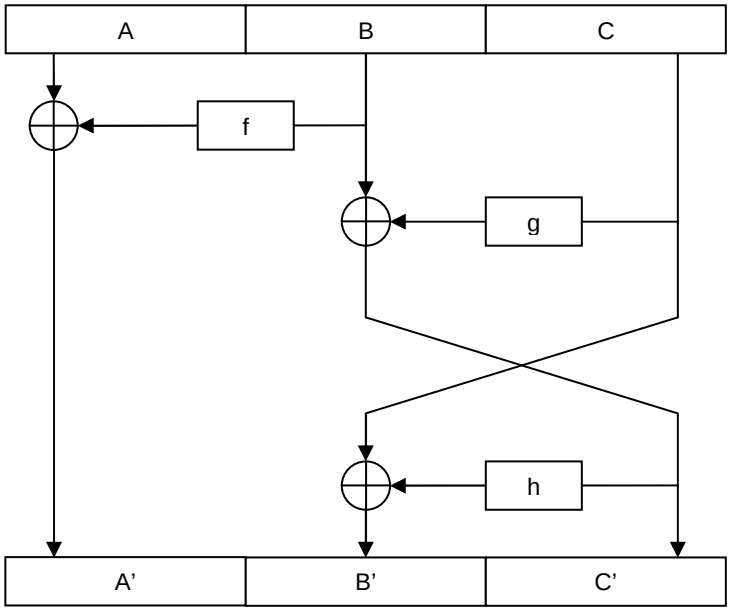


Fig. 3.7 Ejemplo de la red analizada

4.3 Reversibilidad del algoritmo

El algoritmo debe ser reversible: supongamos que se trata de una caja negra que tiene como entrada un bloque y como salida el bloque codificado. Tiene que ser posible enlazar ese bloque codificado con una caja negra idéntica en estructura (cambiando las claves lógicamente) que aplicado al bloque codificado nos devuelva el bloque original.

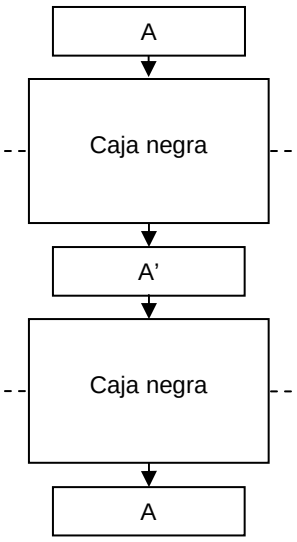


Fig. 3.8

Si nos fijamos en el flujo, es fácil adivinar que en la segunda caja se debe seguir un flujo inverso al de la primera caja, pero como son iguales, el flujo de toda caja debe ser simétrico o isomorfo a una simetría. A su vez, la composición de las dos cajas tendrá una estructura simétrica.

Por ejemplo: la red de Feistel de una ronda (en la cual no se produce intercambio, como hemos visto) sería así:

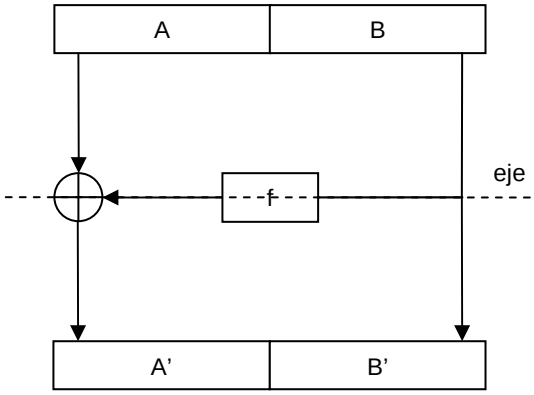


Fig. 3.9

Vemos el eje que de simetría mencionado, al dar la vuelta por ese eje al diagrama (ajustando las flechas) obtenemos el esquema de decodificación que es idéntico al de codificación.

La codificación-decodifiación quedaría:

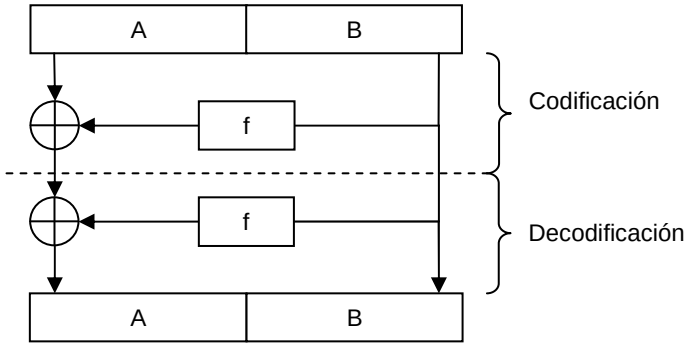


Fig. 3.10

Que como vemos, la función f al ser la misma, dará el mismo valor. Trivialmente se puede apreciar cómo al aplicar dos XOR con el mismo valor al registro A, se recupera el valor que tenía inicialmente.

Veamos cómo sería el esquema de una red de Feistel de un número par de rondas (las que se suelen utilizar). Sabemos que se efectúan n-1 intercambios de registros, si contemplamos de manera global la cadena de Feistel…

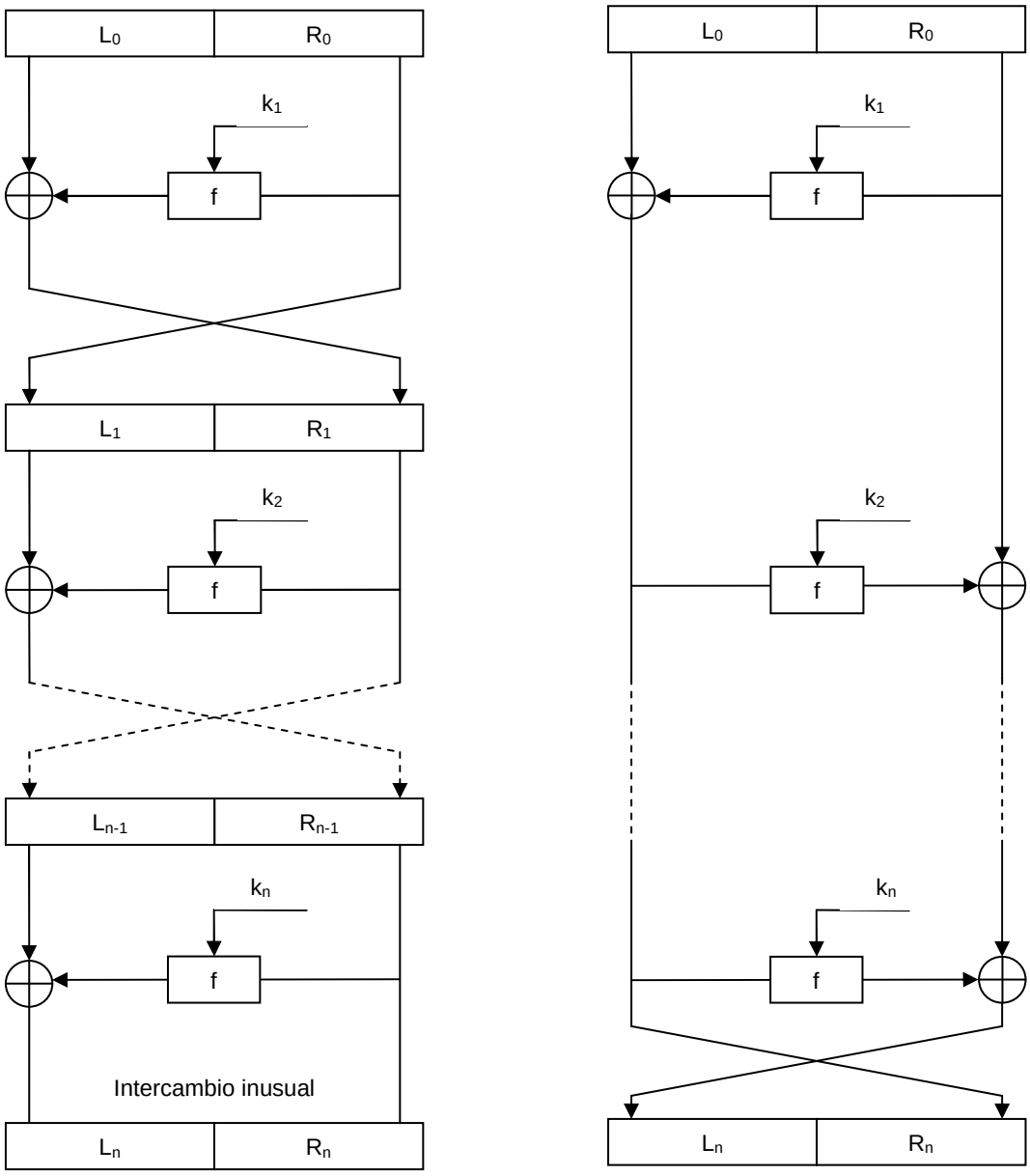


Fig. 3.11 Red de Feistel tradicional y desdoblada

Vemos que con hacer sólo un cambio al final e ir alternando las funciones llegamos a un resultado similar y computacionalmente más rápido. Sin embargo ninguna de las dos estructuras se ajusta a la condición de ser simétricas, sin embargo… ¿podemos convertirla en simétrica? Como veremos sí, estas estructuras son isomorfas a una simétrica equivalente:

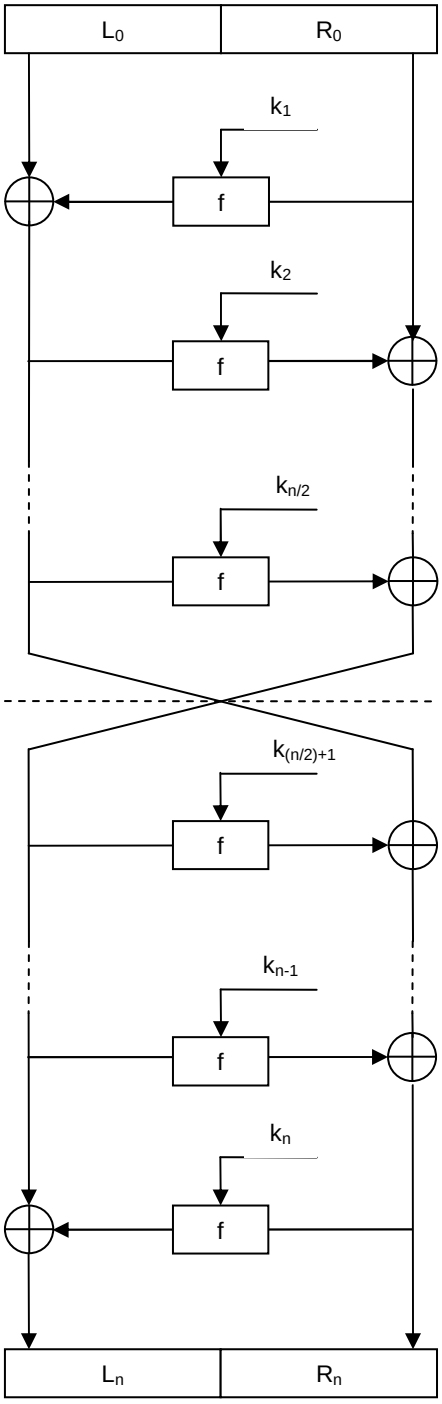
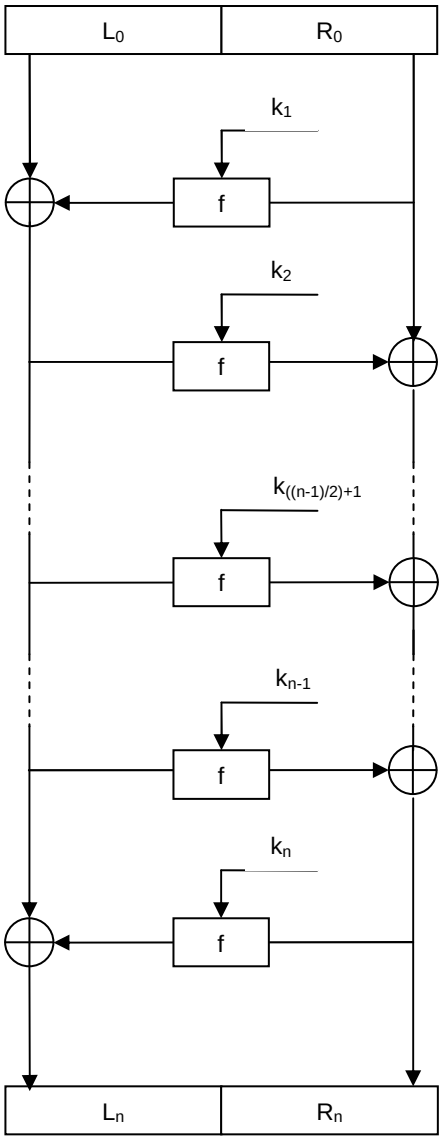


Fig. 3.12 Red de Feistel simétrica par

He aquí la red de Feistel, con número de rondas par, como ejemplo de red con un punto de simetría central, y que sirve para codificar y decodificar. Una red de Feistel con un número impar de rondas presentará una estructura similar sin el intercambio central, y, puesto que cumple la condición indicada, es también autoinversible.

Fig. 3.13 Red de Feistel simétrica impar



Aunque estas reglas definen las características que debe poseer una estructura de cifrado con las peculiaridades que deseamos, no son las únicas estructuras posibles. Con la regla de la simetría nos aseguramos que el algoritmo codificador es idéntico al decodificador y para construir una nueva red basta con diseñar la mitad de ella.

4.4 Asimetría aparente

Hasta ahora las funciones utilizadas tenían dos entradas (una para un registro y una para la clave que obviábamos) y una salida que era aplicada mediante la operación XOR a un registro que no formase parte de la entrada. Sin embargo la operación XOR es asociativa, tiene elemento neutro ($0.....0_n$) y elemento inverso (todo elemento es inverso de sí mismo). Esto nos va a permitir la construcción de funciones diferentes a las consideradas:

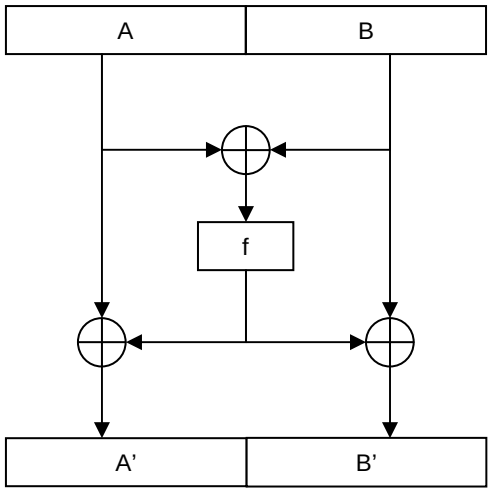


Fig. 3.14 Asimetría aparente

Analicemos con un poco de detalle la estructura mostrada: no tiene simetría vertical (aparentemente) y no se sigue la regla de que el cálculo de una función no puede ser aplicado a una de sus entradas. Sin embargo se puede codificar y decodificar con el mismo algoritmo.

¿Cómo es posible que funcione? La respuesta está en las propiedades de la operación XOR. El resultado de la ejecución de este algoritmo será:

A' = f (AxB) x A

B' = f (AxB) x B

Siendo x la operación XOR. Ahora si volvemos a introducir la salida nuevamente:

A" = f (A'xB') x A' = f((f(AxB) x A) x (f(AxB) x B)) x f(AxB) x A (1

B" = f (A'xB') x B' = f((f(AxB) x A) x (f(AxB) x B)) x f(AxB) x B (2

Considerando la asociatividad del XOR y el elemento inverso XOR(Y,Y)=0, nos queda:

1) f(f(AxB) x f(AxB) x A x B) x f (AxB) x A = f (A x B) x f (A x B) x A = A

2) f(f(AxB) x f(AxB) x A x B) x f (AxB) x B = f (A x B) x f (A x B) x B = B

Ya vemos que funciona, la clave está en que al hacer *A xor B* se obtiene el mismo resultado que con *A' xor B'*. Si llamamos X al resultado de la función, *A' xor B' = (A xor X) xor (B xor X) = (A xor B) xor (X xor X) = = (A xor B) xor 0 = A xor B*. Es decir, del bloque inicial y del bloque final, al hacer XOR se obtiene el mismo resultado, que es la entrada de la función; la función lógicamente devolverá el mismo resultado X y al aplicárselo a A' queda: A" = A' xor X = A xor X xor X = A, con B' ocurre exactamente lo mismo. La función f puede ser inversible o no, eso no tiene importancia.

A este tipo especial de función se le asigna el siguiente gráfico:

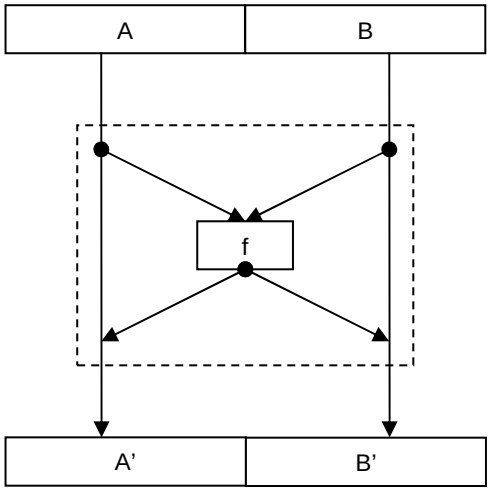


Fig. 3.15 Asimetría aparente

En este caso las consideraciones de los registros a los que puede o no afectar no es importante, ya que las entradas de la función pueden calcularse a partir de los registros de salida. Se le ha dado esta forma simétrica a propósito para que corresponda con la simetría apuntada anteriormente. Es importante tener en cuenta que la salida de la función puede ser perfectamente aplicada a cualquier otro par de registros, no necesariamente los de origen. En ocasiones pueden entrelazarse varias funciones de este estilo (el criptosistema IDEA hace uso de ello como veremos).

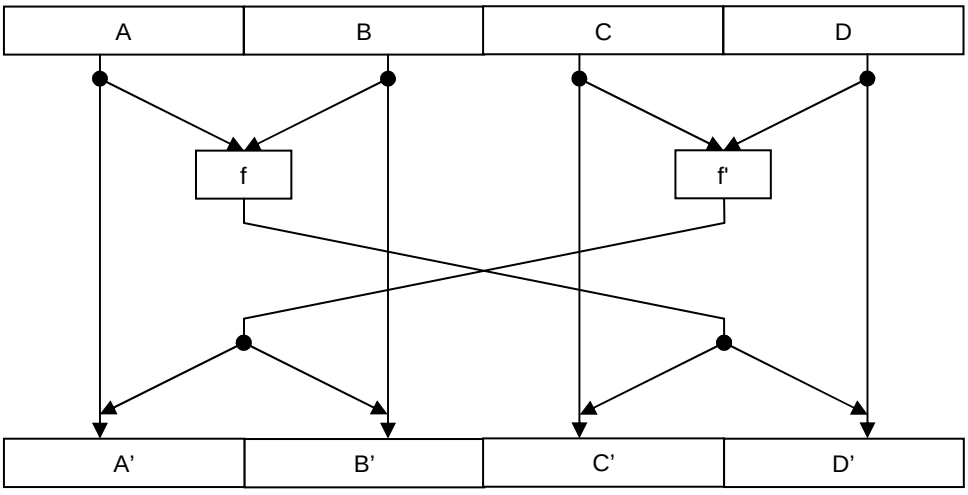


Fig. 3.16

Este ejemplo representa una red inversible y autodecodificable. A diferencia de los vistos anteriormente, en éste ninguno de los registros de salida es un dato que permita calcular una función directamente, sin embargo por lo visto anteriormente, sabemos que al aplicar *C' xor D'* el resultado es el mismo que la entrada de la función *f'* sin considerar el valor utilizado de la función *f*. Lo mismo se puede aplicar a *A' y B'*. Conociendo las entradas de las funciones podemos calcular su valor y consecuentemente decodificar los registros. En este ejemplo se han combinado dos redes más sencillas, se puede demostrar que cualquier combinación en paralelo de cualquier número de estas redes sencillas genera una red autodecodificable siempre que se cumpla:

- No se producen intercambios entre registros, la información de A pasa a A', la de B a B' y así sucesivamente.
- El resultado de una función sólo puede ser aplicado a un par de registros de forma que: o bien los dos son entrada de una función o ninguno lo es.

En este caso nos podemos saltar la regla de la simetría para las líneas que van desde el XOR antes de una función a la bifurcación del resultado de calcularla. También nos podemos saltar la regla de la secuencialidad por las características de la operación XOR.

En la forma de representación mediante tablas, esta asimetría aparente es representada de la siguiente forma:

A	A x f'(B, C) = A'
B	B x f'(B, C) = B'
C	C x f(A, B) = C'
D	D x f (A, B) = D'

Fig. 3.17

4.5 Funciones interconectadas

Los diagramas vistos hasta ahora muestran a las funciones como entes que reciben un dato de los registros y, con una o más claves obtenidas de algún modo, devuelven un resultado. Sin embargo se podría dar el caso de existir funciones que reciban más de un dato de los registros y devuelvan más de un dato de salida. Estas funciones se pueden considerar funciones simples que reciben y devuelven un solo dato, pero que entre medias pueden intercambiar información. Como es lógico, para que se puedan utilizar funciones interconectadas es necesario poder disponer de las entradas originales en el momento de la decodificación. Veamos un ejemplo:

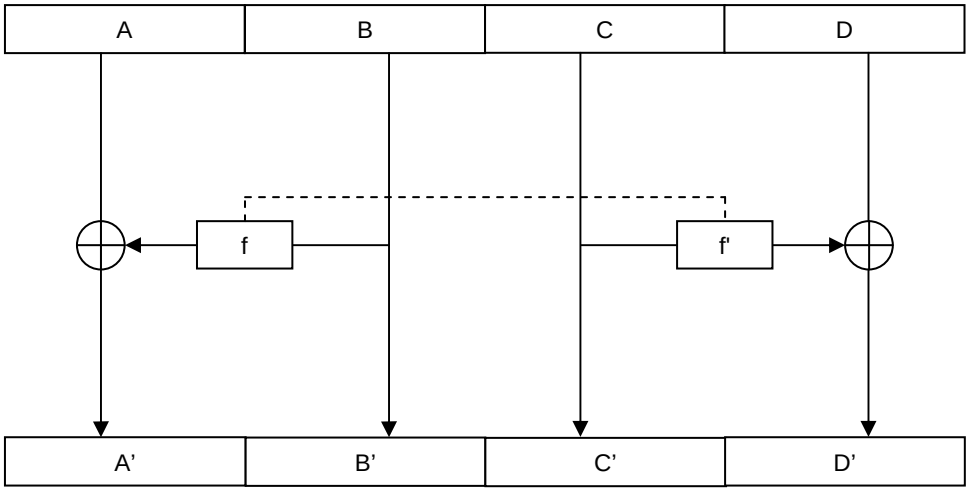


Fig. 3.18 Funciones interconectadas

La línea discontinua representa el intercambio de información pero no dice nada acerca de la información exacta que se intercambia o de la cantidad. Las funciones interconectadas son solamente un recurso gráfico para simplificar el diseño de los diagramas.

4.6 Funciones inversibles

En los diagramas anteriores nos hemos centrado en el uso de funciones sin inversa como método para crear confusión y difusión, pero es posible utilizar funciones fácilmente inversibles como la *suma módulo n* o el *producto módulo n* para ayudar a estos fines. Se da por descontado que una red no puede basarse solamente en funciones inversibles pero combinadas con los métodos ya descritos puede conferir a esa red una resistencia mucho mayor frente a ataques.

Entre las principales funciones inversibles ya se han nombrado algunas como la suma y el producto modulares, hay otras muy importantes como xor o permutaciones. Éstas funciones, salvo las permutaciones, cuentan con una entrada y una salida, y se insertan en los caminos principales que van de un registro origen a uno de destino. Han de ser involutivas porque al igual que del mensaje en claro devuelven uno cifrado, han de funcionar al revés para decodificar. Y por último deben situarse en lugares que permitan efectivamente la simetría en la que estamos insistiendo. Veamos un ejemplo:

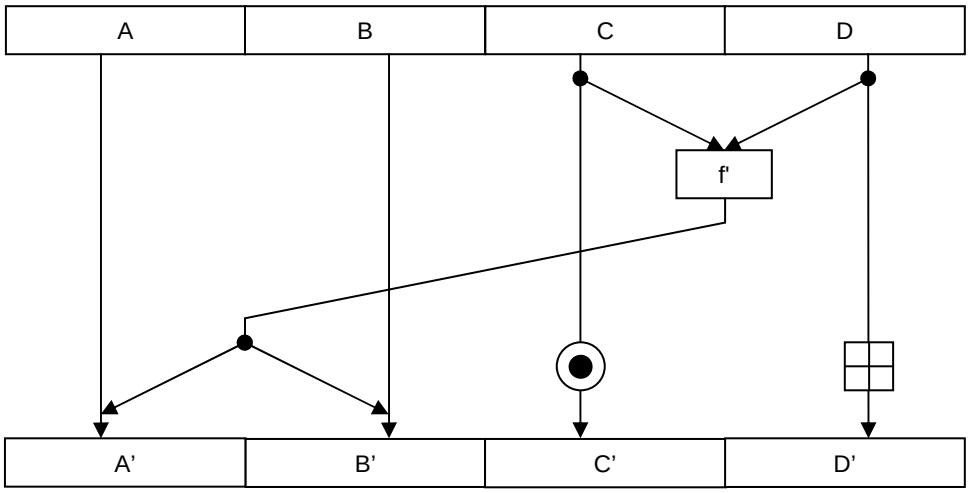


Fig. 3.12 Funciones inversibles

Esta red no es autoinversible, ello se debe a que las dos nuevas funciones inversibles añadidas no tienen simétrica. Una forma de arreglar esto pasaría por añadir las simétricas antes de la función *f*. Existe otra forma para redes más grandes y complejas que veremos más adelante.

4.7 Construcción de redes

Hemos visto hasta ahora las ideas principales que deben cumplir las redes creadas para hacer uso de las dos características que hemos mencionado: autodecodificación y utilización de funciones sin inversa. Ahora se procederá a describir una forma de construcción de dichas redes.

4.7.1 Herramientas de trabajo

Las herramientas de trabajo van a ser bloques básicos que representen subredes con las características mencionadas anteriormente. Las más sencillas ya las hemos visto:

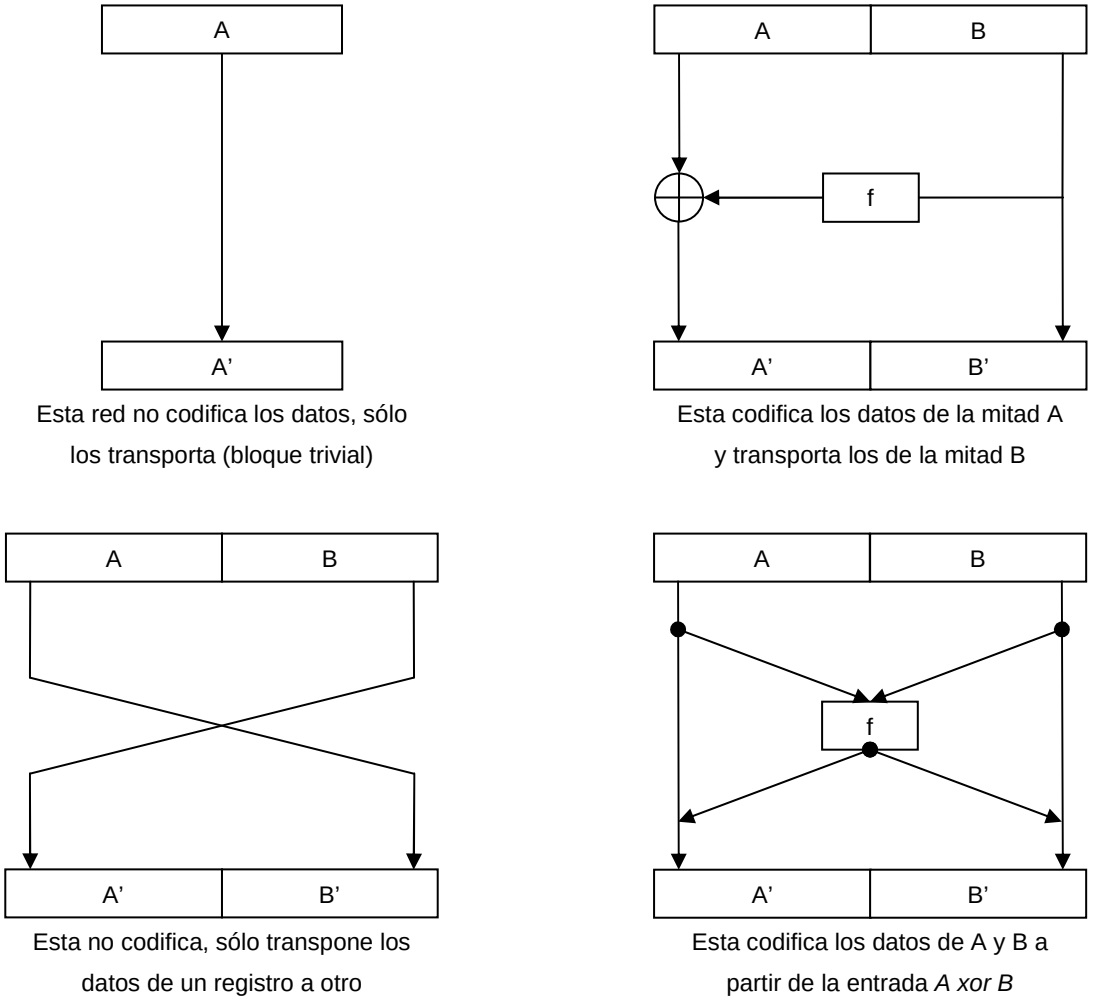


Fig. 3.20 Herramientas de trabajo

El flujo de información se supondrá que corre hacia abajo. Se van a dar unas reglas para poder combinar bloques.

Debe quedar claro que los bloques indicados no son los únicos. Se pueden crear otros bloques básicos nuevos. Los bloques que cumplan los objetivos serán bloques válidos; los válidos que no puedan derivarse de los básicos serán a su vez bloques básicos.

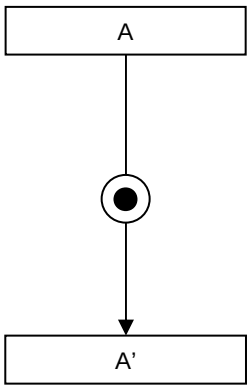


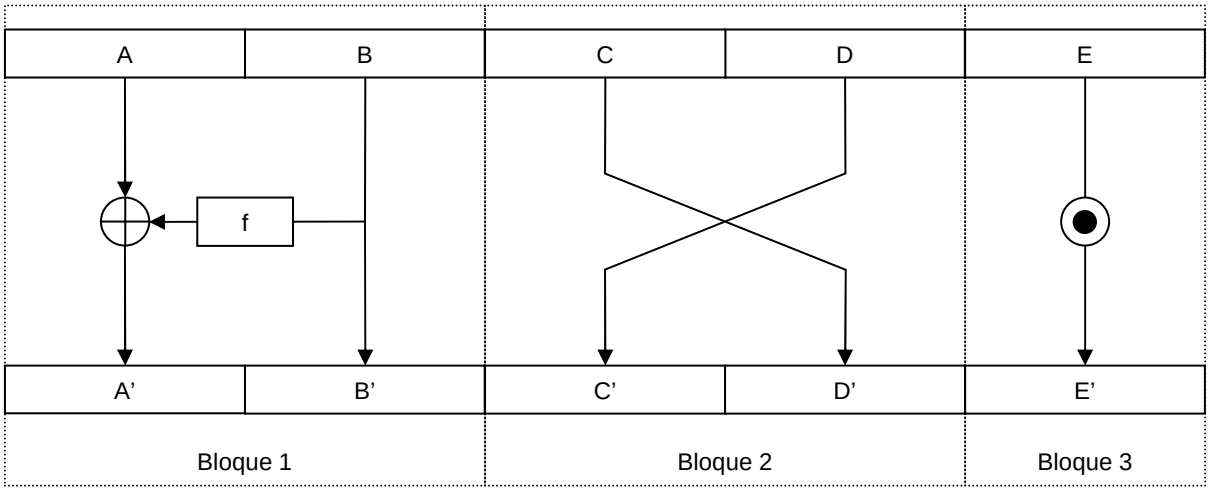
Fig. 3.21 Red que aplica una función inversible a un registro

Éste bloque en particular se utiliza para aplicar una función reversible a un registro. Esta función puede ser cualquiera, y puede intervenir clave o no. Lo más importante a tener en cuenta cuando se utilice, es que los datos al decodificarse deben seguir el mismo camino y atravesar las funciones exactamente en orden inverso al que fueron aplicadas la primera vez, esto nos permitirá salir de ciertas situaciones.

4.7.2 Construcción de redes completas

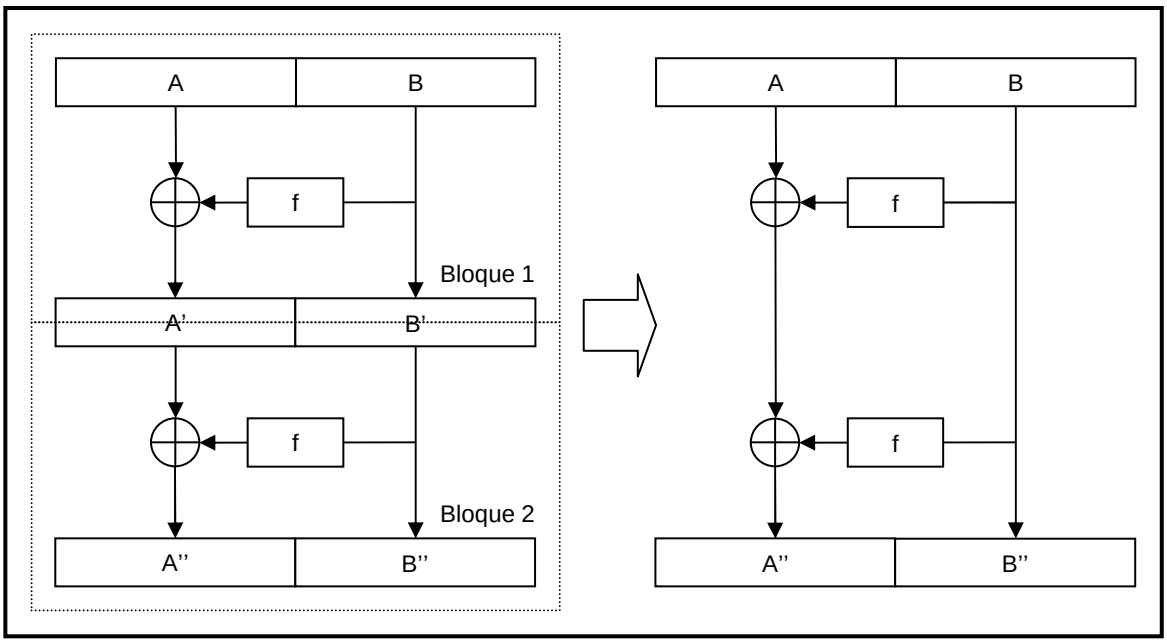
La primera regla es que cualquier combinación en horizontal de los bloques básicos es un bloque válido. Veamos un ejemplo:

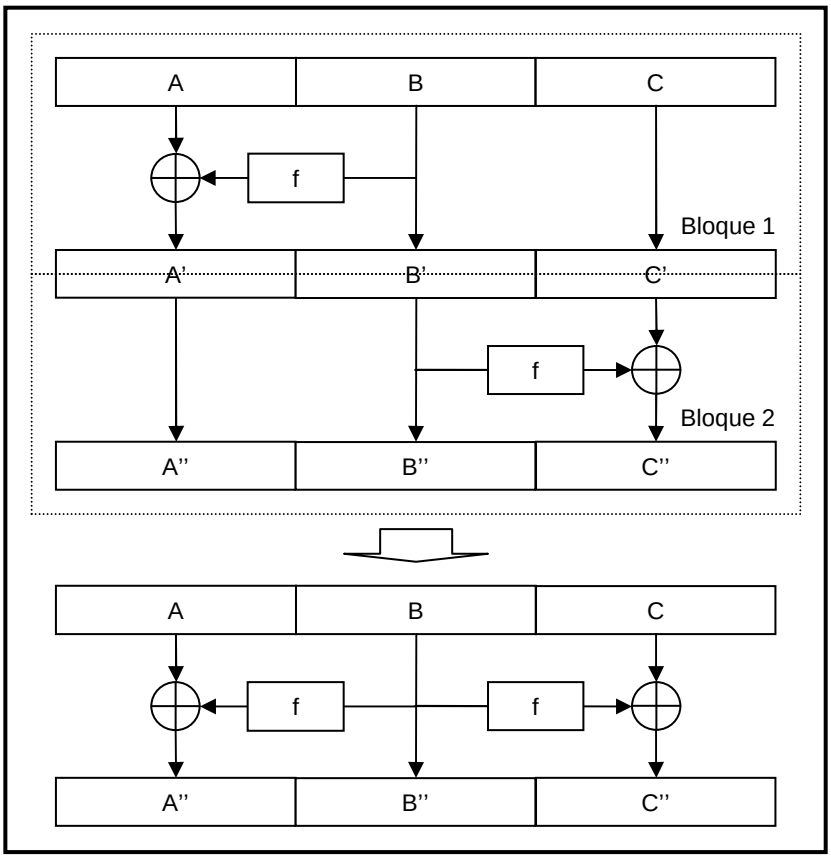
Fig. 3.22



Esta regla, que se comprueba muy fácilmente, permitirá utilizar más de uno o dos registros cuando sea necesario y consigue que no sea necesario definir todas las posibles variaciones para n registros de los bloques válidos.

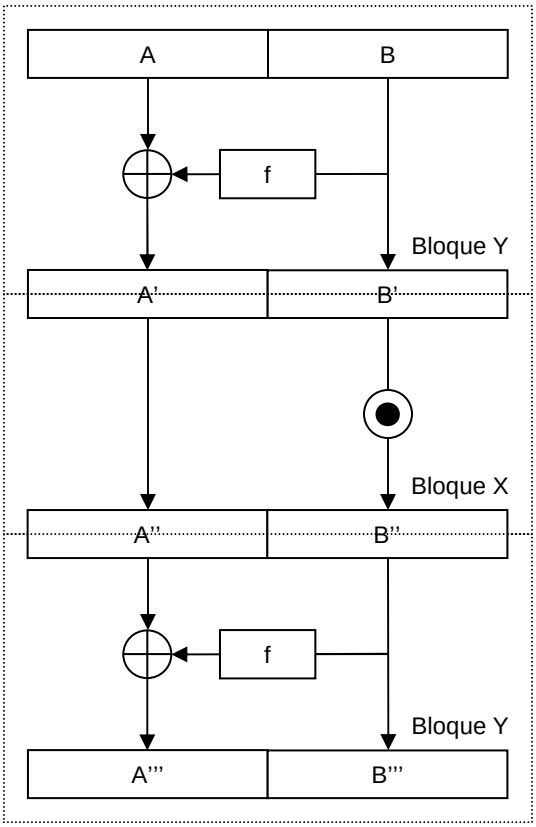
La segunda regla hace referencia a la combinación vertical cuando no se modifican los caminos principales (no hay permutaciones entre registros y no se aplica una función reversible a un registro). Se permite la combinación vertical de bloques cuando los caminos principales no son modificados y no hay interacción entre las entradas de un bloque y las salidas de otro. Es importante darse cuenta de que sí pueden compartir una entrada o afectar a una salida a la vez.





Figs: 3.23 y 3.24 Redes que muestran acoplamientos verticales válidos sin conflictos entre entradas y salidas

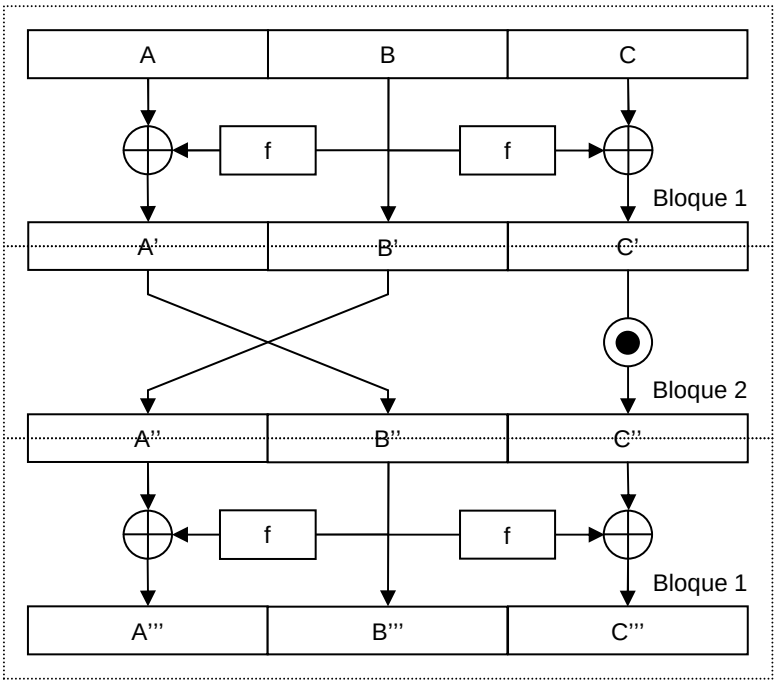
La tercera regla trata de las combinaciones verticales de los bloques cuando hay interacción de entradas y salidas entre ellos.



Figs: 3.25 y 3.26 Redes con acoplamiento vertical válido y conflictos con entradas y salidas

Se debe cumplir la simetría, si consideramos un bloque X y le anteponemos un bloque Y, hay que posponerle el mismo bloque Y. La interacción se produce en el registro B ya que el bloque X lo utiliza a la vez como entrada y como salida.

Es en este tipo de acoplamientos donde se puede hacer uso de las permutaciones, ya que ellas cambian los caminos principales. Una permutación puede hacer de bloque central o de bloque anexo, en la combinación que se quiera.



Por supuesto, los bloques pueden ser cualquier bloque obtenido a partir de las reglas anteriores. Una opción es que el bloque central sea un bloque trivial, (que no haga nada).

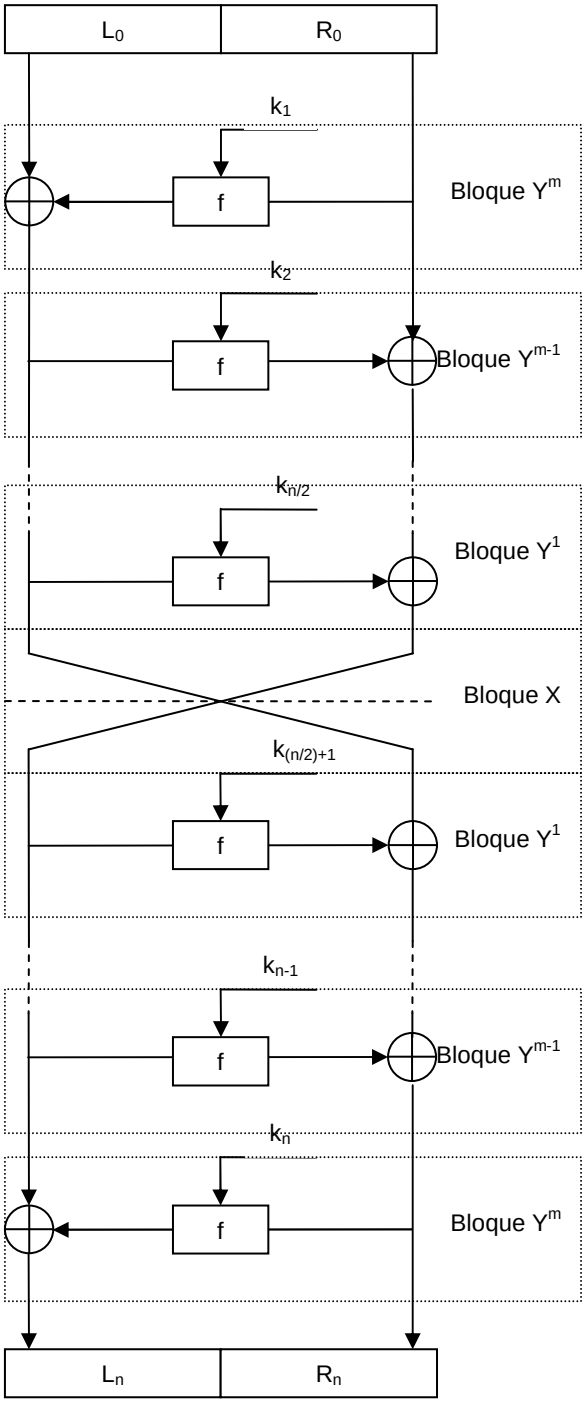


Fig. 3.27 Red de Feistel analizada dividida en bloques

Esta es una red de Feistel con un número par de rondas en las que los intercambios entre registros se han desenlazado y como consecuencia las funciones van alternativamente de derecha a izquierda. El intercambio que se debe respetar, que se podría haber situado al final, se ha situado justo en medio para apreciar la simetría, en esta red vamos a identificar los bloques que la componen. (Por simplicidad se omiten los registros en cada uno de los pasos, dependiendo de si la mitad del número de rondas es par o impar la función del bloque Y puede ir hacia un lado u otro). Por supuesto el resultado de esta operación es un bloque válido y puede utilizarse como tal para construir redes más complejas.

4.7.3 Construcción mediante eslabones

La construcción de una red como acabamos de hacer proporciona un método válido para crear un criptosistema con las características que deseamos, pero el resultado final aparece como un bloque completo. Si nuestras intenciones son que el algoritmo resultante se pueda computar en un bucle esta forma de diseño puede no ser la mejor. Pasaremos a describir ahora una forma de crear este tipo de redes de forma que sea sencillo programarlas mediante iteraciones.

Lo primero en que debemos fijarnos es: un bloque válido puede descomponerse en dos: uno en el que no haya permutaciones entre registros y otro en el que sólo haya permutaciones (puede ser un bloque trivial).

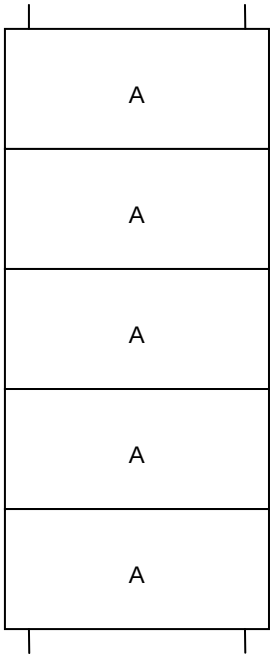


Fig. 3.28 Construcción mediante eslabones

Veamos poco a poco cómo crear la red: supongamos que tenemos un bloque “A” que es autodecodificable y sin permutaciones (el camino principal desde un registro va a parar al mismo registro). Trivialmente, cualquier secuencia de bloques “A” será a su vez autodecodificable.

Ahora imaginemos, si en el segundo bloque A se utilizasen claves inversas a las del primero este bloque decodificaría lo que codificó el primero, deshaciendo su trabajo y debilitando todo el sistema. Y aunque las claves no fuesen inversas, es probable que si no se elige con mucho cuidado el bloque “A”, la red que obtenemos como resultado tenga debilidades o incluso que tenga estructura de grupo. Para evitar esto podemos introducir permutaciones entre los registros de forma que el segundo bloque A ya no sería realmente inverso del primero y no existiría el problema mencionado. Como ya es evidente, el cuerpo del bucle realmente es el bloque A, si introducimos entre medias un pequeño bloque, surge el problema de si lo asociamos al bloque de arriba o al de abajo, ya que deberá estar dentro del bucle.

Supongamos que lo asociamos a la parte de arriba.

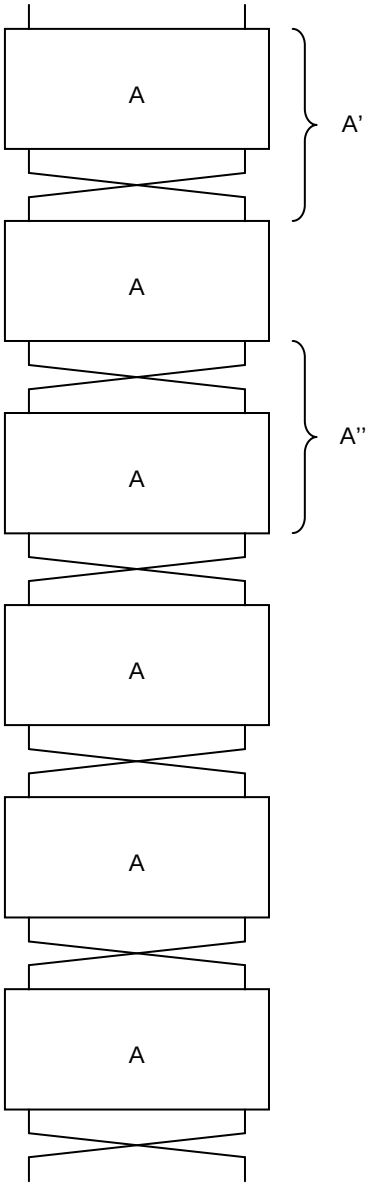


Fig. 3.29

Vemos que el nuevo bloque A' es el resultado de la combinación del bloque A con la permutación. Sin embargo podemos ver un fallo en este diseño: no es simétrico, existe una permutación al final que es necesario neutralizar. Habitualmente lo que hacen los algoritmos basados en redes de Feistel (como DES o Blowfish), o el IDEA, deshacen la última permutación.

Si asociásemos la permutación al bloque inferior (bloque A"), veríamos que el primer bloque no tiene permutación y habría que permutar antes de entrar en la red propiamente dicha.

Si se desea codificar el algoritmo mencionado (bloque A') mediante software podemos seguir la codificación tradicional:

1. Para i=0 hasta i=n hacer
2. Ejecutar A
3. Ejecutar permutación
4. Fin bucle
5. Ejecutar permutación

Una forma diferente y ligeramente más eficiente sería:

1. Ejecutar A
2. Para i=0 hasta i=n-1 hacer
3. Ejecutar permutación
4. Ejecutar A
5. Fin bucle

De esta forma nos ahorramos el cálculo de dos permutaciones, no es un gran ahorro, pero cuando la velocidad cobra importancia y siendo un cambio mínimo, se puede aplicar.

Ahora bien, imaginemos que en el bloque A queremos aplicar alguna operación para la cual no haya simétrico dentro del propio A, ya sea porque no nos interesa o porque no queremos recargar de forma inútil a la red. Este bloque podría ser por ejemplo un bloque de funciones inversibles que utilizamos para añadir confusión al sistema. Habiendo visto el caso anterior la respuesta es clara, podemos sustituir las permutaciones por bloques de este tipo, en los mismos lugares y todo funcionará igual.

El problema surge cuando queremos mantener estar permutaciones y queremos añadir las funciones inversibles. La respuesta es sencilla: hay que construir esos nuevos bloques de forma que puedan conmutar con las permutaciones. Se nos pueden dar dos casos:

- Si el bloque de funciones inversibles se considera que van al mismo lado que la permutación: en este caso es trivial demostrar que la explicación dada para el esquema anterior funciona, simplemente hay que considerar que la permutación es un nuevo bloque que incluye permutación y aplicación de funciones inversibles, siempre y cuando formen un bloque válido.
- Si el bloque de funciones inversibles va al lado contrario: en este caso hay que tener cuidado para que la división en eslabones sea consistente. Supongamos que las funciones inversible C van antes del bloque A, y la permutación B va después, como veremos a continuación B y C deben conmutar. El esquema que queda es:

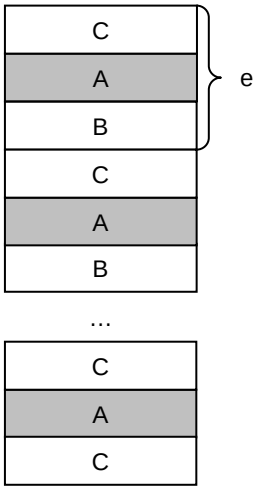


Fig. 3.30

Parece extraño y asimétrico, pero funciona. Puesto que el bloque C está por encima, para cumplir la simetría tiene que haber un bloque C por debajo del último A. Y como B está por debajo, debajo del último A no puede haber un bloque B, también por simetría. El último fragmento se puede expresar así si se desea, estando compuesto por un eslabón (e) un una parte remanente necesaria para la reversibilidad.

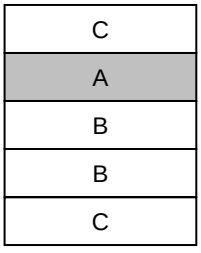


Fig. 3.31

Si se desease añadir más bloques a nuestro bloque A, hay que asegurarse de que es conmutable con los bloques ya añadidos, en ese caso no tendremos ningún problema si actuamos de forma similar: los bloques como B (internos) hay que anularlos justo después de la salida del último A. Los bloques como C (externos) hay que repetirlos al final de la red. Veamos cómo la red mostrada anteriormente sí se autodecodifica:

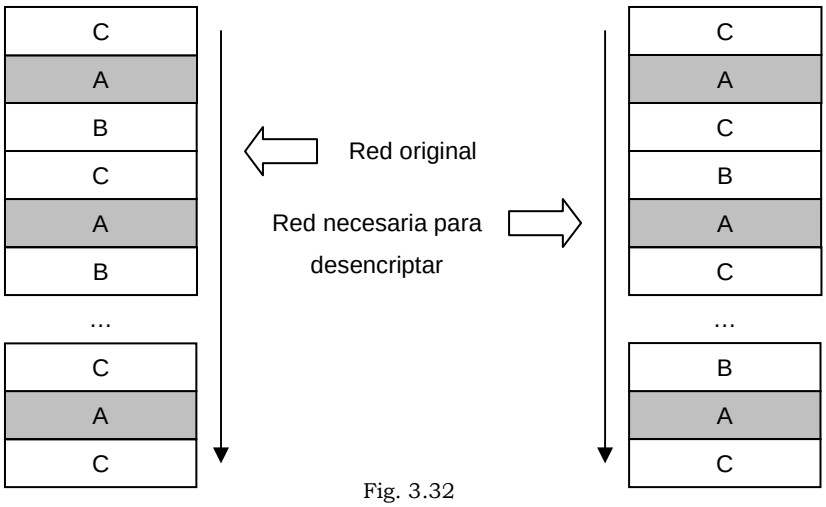


Fig. 3.32

La red necesaria para desencriptar se muestra a la derecha y es la red original dada la vuelta. Como sabemos que B y C conmutan (da igual aplicar uno y después otro) los bloques intermedios actúan del mismo modo que estuviese antes B y luego C como en la original, luego la red es autoinversible.

La permutación B no puede ser cualquiera, tiene que ser autoinversible. Aunque ya lo apuntamos anteriormente, ahora podemos justificarlo. Cuando la información encriptada pasa por el primer eslabón para ser desencriptada, debe deshacer lo que hizo el último eslabón. Para pasar de un eslabón al siguiente tiene que pasar por una y sólo una permutación, que es la misma que por la que se pasó al codificar, así que esa permutación ha de ser involutiva, es decir, que la permutación B debe ser autoinversible. Las permutaciones autoinversibles son aquellas que intercambian registros de dos en dos y un registro puede ser movido de lugar como mucho una vez.

4.7.4 Ejemplo de red construida mediante eslabones

Vamos a dar un ejemplo de una red que se puede construir mediante eslabones siguiendo las directrices marcadas. En vez de crear un nuevo tipo de red, se va a describir el funcionamiento del algoritmo IDEA cuyo esquema es el siguiente.

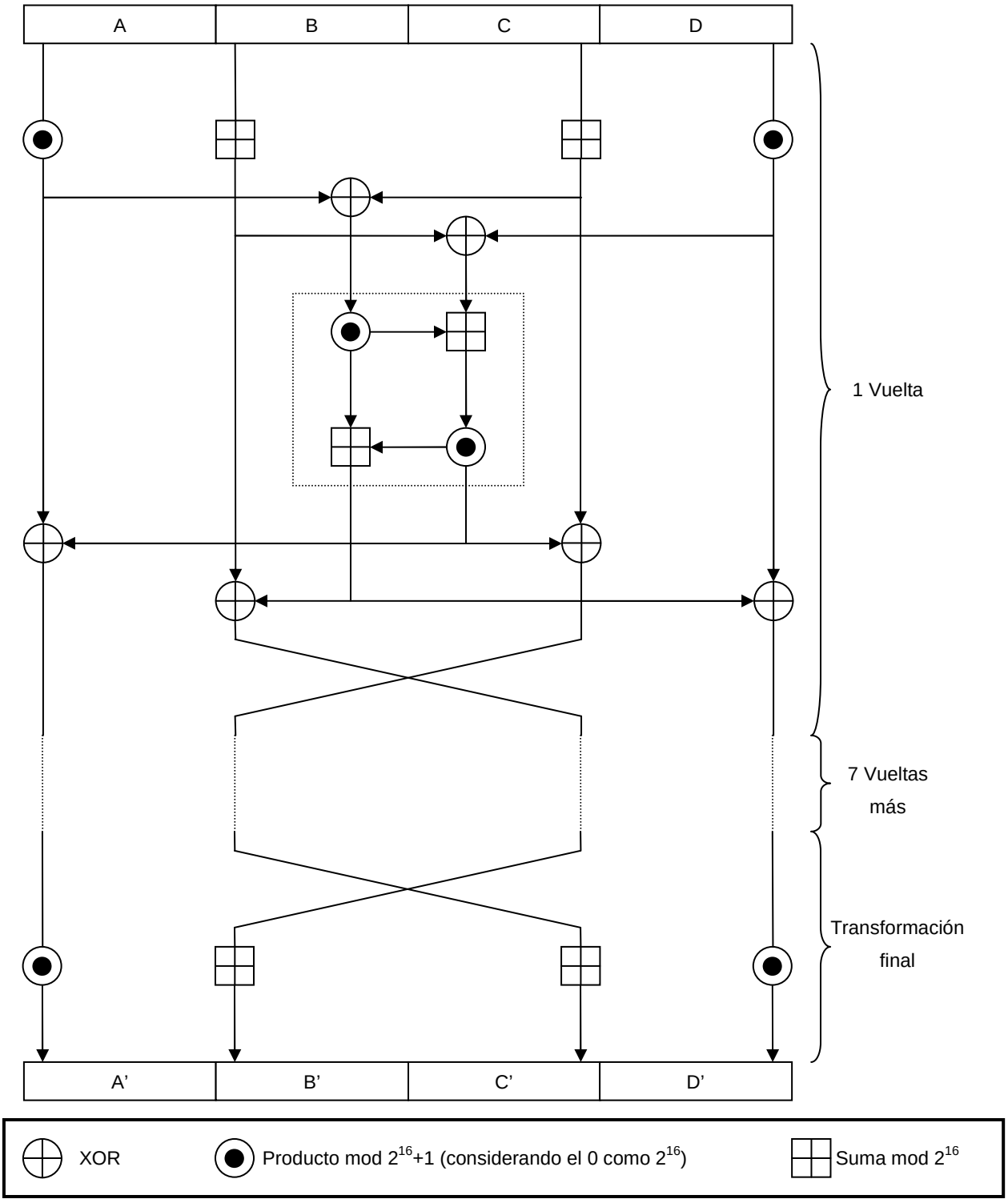


Fig. 3.33 Red del criptosistema IDEA

A primera vista parece muy complejo construir una red como esta y que además sea autodecodificable, pero vayamos analizando por partes. Primero miremos el eslabón:

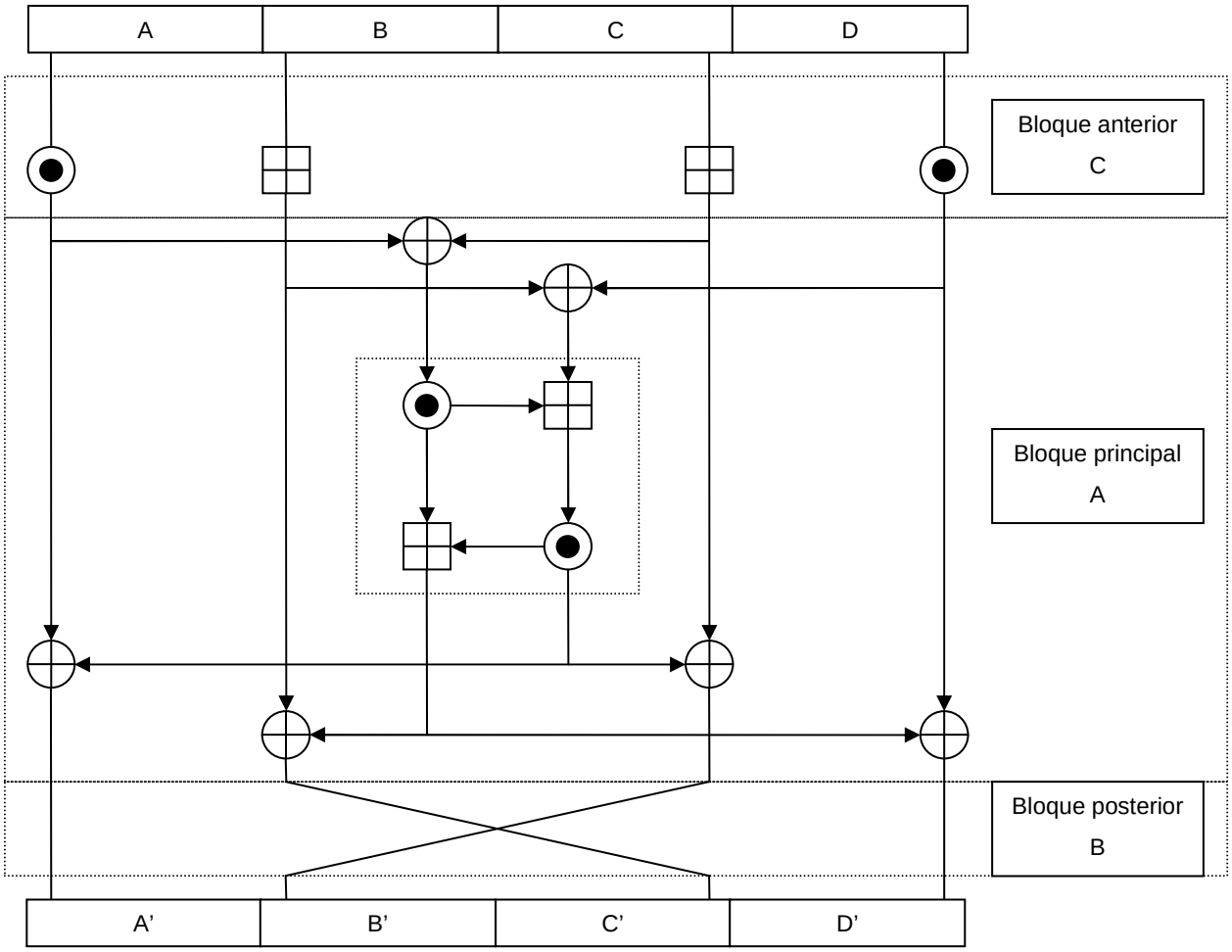


Fig. 3.34 Red IDEA analizada por bloques

Podemos ver los tres bloques que son autoinversibles. El bloque A se puede representar del siguiente modo:

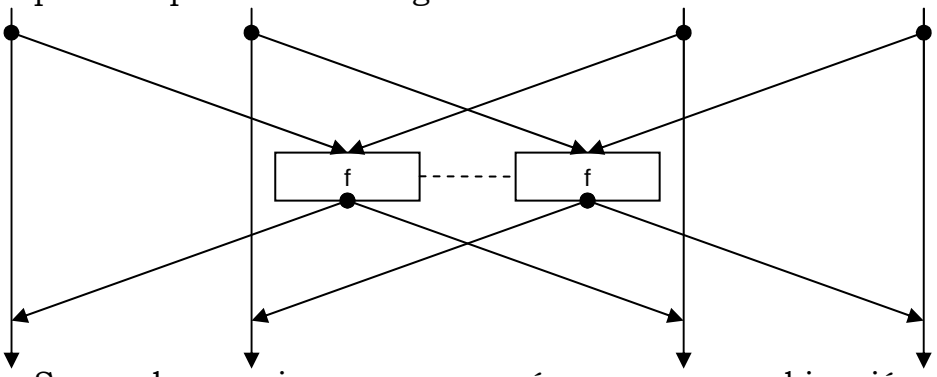
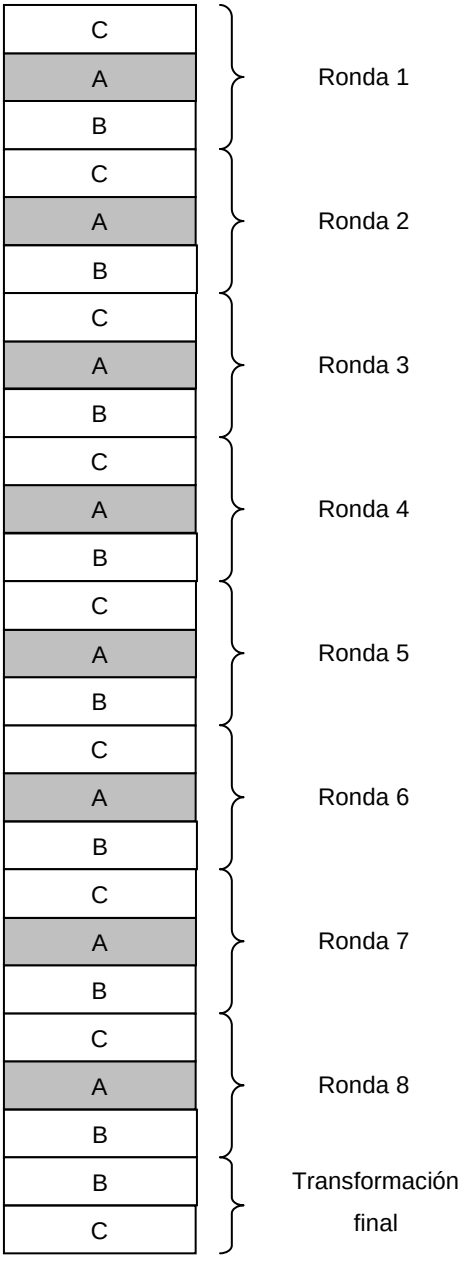


Fig. 3.35 Bloque principal de la red IDEA

Se puede apreciar que no es más que una combinación de dos bloques básicos vistos ya, y que las funciones intercambian información (en realidad, como se dijo, es un recurso gráfico, solo es una función).

En los bloques B y C es sencillo ver la composición de los mismos, el bloque B es una permutación junto con dos bloques triviales y el C es una composición de cuatro funciones inversibles. Vemos cómo los bloques B y C efectivamente conmutan.

El bloque C va antes que el principal y el bloque B va después. De esta forma, y considerando lo dicho en la construcción mediante eslabones, la siguiente red es válida y sirve para codificar y decodificar:



Y no es ni más ni menos que la red IDEA al completo.

Puede parecer que describir un sistema que ya se ha probado que funciona no es demasiado útil, sin embargo, con pequeños cambios y siguiendo las reglas y directrices propuestas, se pueden conseguir redes bastante complejas sin mucha dificultad y que cumplirán los objetivos marcados de ser autodecodificables, facilitando enormemente la construcción de dispositivos hardware, pues se puede utilizar el mismo dispositivo para codificar y para decodificar.

Fig. 3.36 Red IDEA construida mediante eslabones

3.8 Subclaves

Hasta ahora no nos hemos preocupado de las subclaves, hemos supuesto que la sunclave adecuada siempre estaba preparada al codificar y al decodificar. Se ha dejado este apartado para el final porque habiendo seguido todo el razonamiento anterior es fácil deducir cómo se deben elegir las claves para lograr la decodificación.

Para ello se debe tener una visión global de la cadena: las subclaves del último bloque en la codificación irán a parar al primer bloque de la decodificación, pero estando invertidas para que las funciones deshagan lo que hicieron. Las subclaves dentro de un bloque, visto este como una cadena completa, van a parar al otro lado del eje de simetría del bloque. La subclave de un bloque básico es la misma pero invertida. Es como vemos una forma recursiva de asignarlas. Veamos un ejemplo:

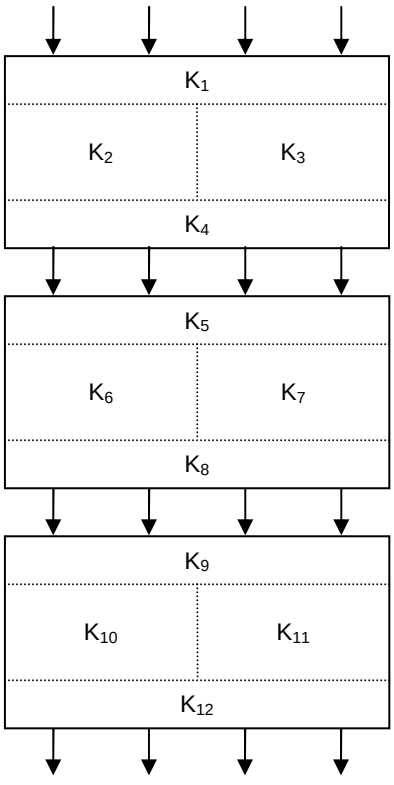


Fig. 3.37

Las claves del primer bloque serán las últimas del bloque al decodificar. Sean las K' las claves de decodificación y K^* la inversa.

$K_1, K_2, K_3, K_4 \rightarrow K'_9, K'_{10}, K'_{11}, K'_{12}$

$K_5, K_6, K_7, K_8 \rightarrow K'_5, K'_6, K'_7, K'_8$

$K_9, K_{10}, K_{11}, K_{12} \rightarrow K'_1, K'_2, K'_3, K'_4$

Ahora por bloques, la subclave superior pasa a ser la inferior, y la inferior pasa a ser la superior. No se producen cambios en izquierda derecha. Para el primer bloque las claves de decodificación serían:

$K_9, K_{10}, K_{11}, K_{12} \rightarrow K'_1, K'_2, K'_3, K'_4$

$K_9 \rightarrow K'_4 \qquad K'_4 = K^*_9$

$K_{10} \rightarrow K'_2 \qquad K'_2 = K^*_{10}$

$K_{11} \rightarrow K'_3 \qquad K'_3 = K^*_{11}$

$K_{12} \rightarrow K'_1 \qquad K'_1 = K^*_{12}$

$K'_1 = K^*_{12}$
$K'_2 = K^*_{10}$
$K'_3 = K^*_{11}$
$K'_4 = K^*_9$

K* representa el inverso de K para la función *f* que la utiliza, es decir, el valor que debe tomar para conseguir que la función deshaga lo que hizo con K. Cuando tratamos con funciones aplicadas con la operación XOR tenemos que K*=K, pero si fuera con suma modular, serían diferentes[1].

[1] Un caso típico sería es de las funciones inversibles distintas al xor en el que un elemento no es inverso de sí mismo.

5 Funciones

Una vez determinada la estructura del algoritmo, debemos decidir cuáles serán las funciones que utilizaremos. Es ésta una decisión delicada, puesto que de ésta elección se obtendrá una parte importante de la seguridad que nuestro algoritmo puede ofrecer. Algunos autores apuestan por operaciones complicadas, del estilo utilizado en criptografía de clave pública, como la exponenciación discreta con números mucho menores.

Para la elección del conjunto de funciones nos basaremos en las teorías de Shannon en las que indica que en un criptosistema debe haber confusión y difusión. Se puede decir que la confusión actúa ocultando el valor real de cada elemento (byte, palabra…) sustituyéndolo por otro valor; la difusión se encarga de mover dentro de un bloque los datos, de forma no trivial, para esconder el orden en el que aparecían.

Un ejemplo de función que genere confusión puede ser la operación XOR con valores de la clave. Cada elemento A toma el valor A'=A xor K. Esta función no aporta difusión al sistema, no desplaza los elementos de su lugar de origen sino que se conforma en ocultarlos. Una función que genere confusión sería el desplazamiento dentro del registro, reintroduciendo por un lado lo que exceda del otro; como podemos imaginar, cambia de sitio a todos los elementos pero no oculta su valor. Una forma perfeccionada de crear difusión consiste en poder realizar cualquier permutación sobre los datos.

Debido a la especial estructura de la que dispondrá la familia de algoritmos definidas por las reglas anteriormente dadas, las funciones no tienen por qué tener inversa; pero tampoco es deseable que se produzcan demasiadas colisiones. Podemos utilizar funciones muy complicadas que aporten gran cantidad de confusión y de difusión al sistema si estamos dispuestos a pagar el coste computacional asociado. Por el contrario podemos hacer uso de funciones muy sencillas con una aportación menor y arriesgarnos a que el sistema obtenido cuente con problemas de seguridad aun cuando su velocidad sea más elevada.

Todo depende del valor y la cantidad de información que va a circular por el mismo.

5.1 Ataques para la obtención de la clave

Un ataque muy peligroso a un criptosistema se da cuando el atacante dispone de parejas de texto en claro y su correspondiente cifrado. En este ataque trata de buscar la clave para poder descifrar el resto de mensajes enviados. Una característica necesaria del criptosistema es que la clave no pueda ser obtenida fácilmente a partir de parejas de texto claro y cifrado, esto no obliga a que en todas las funciones dentro del algoritmo se deba cumplir esta recomendación, pero sin duda el hecho de que en el mayor número posible de funciones sea imposible la obtención de la subclave ayuda a que el algoritmo sea resistente frente a este ataque.

Un ataque utilizado cuando no tenemos conocimientos del sistema es la fuerza bruta, en ella se procede a encriptar los textos en claro con claves diferentes, hasta que se encuentre la clave que hace corresponder el resultado con el cifrado real de ese mensaje. Es un método poco peligroso si la clave es larga y las subclaves se toman de forma inteligente, pero es un método que al fin y al cabo tendrá una probabilidad de éxito del 50% recorriendo la mitad del espacio de las claves.

No existe una función biyectiva en la que intervengan tres datos (dos variables de entrada y un resultado) y conociendo dos de ellos no sea posible obtener la tercera, es un principio que no podemos ignorar. Una solución factible sería emplear más de una subclave en cada función, de modo que lo máximo que podemos obtener sea una relación entre las dos subclaves pero no información real acerca de la clave. Para ejemplificar esto observemos los siguientes diagramas:

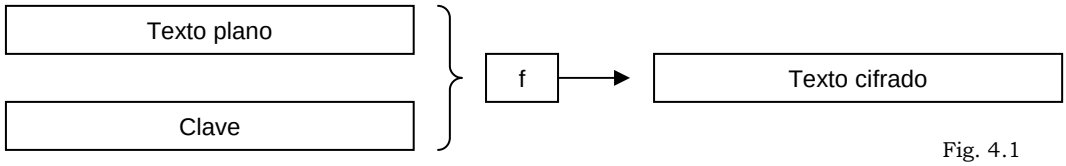


Fig. 4.1

Es en este caso, cuando tenemos el texto claro y el texto cifrado, podemos obtener la clave, sea buscando la inversa de la función f, o aplicando f repetidamente con diferentes claves hasta que conocida el resultado con el cifrado conocido. Si la función f no es biyectiva, habrá más de una coincidencia; esto puede ser deseable según el contexto: no es posible obtener la clave, pero habrá claves que den el mismo cifrado.
Si se trata del sistema global, está claro que esto no es bueno para el sistema pues dos claves diferentes dan lugar al mismo cifrado, si se trata de las funciones dentro de los bloques es posible que una correcta elección de las subclaves eviten ese problema aunque el riesgo existe y no siempre es fácil evitarlo.

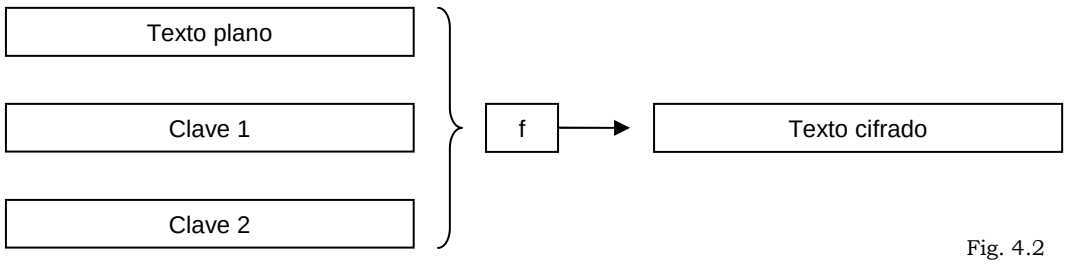


Fig. 4.2

Si entran en juego dos claves bien escogidas, lo máximo que podemos obtener es una relación entre las dos, es decir, no podremos saber nada con seguridad de ninguna de las dos subclaves y por lo tanto de la clave. No es necesario que esta característica afecte a todas las funciones del criptosistema, con afectar a un subconjunto de las mismas pueden complicar enormemente el criptoanálisis del mismo.

5.2 Ejemplos de funciones

A continuación se procede a mostrar una lista de funciones que pueden utilizarse en los bloques. En ellas se detallarán las características propias que aportan cada una de ellas. Recalcar que el uso de cualquier

función es posible pero vamos a centrarnos en algunas cuya simplicidad y características hagan su uso interesante en criptografía.

5.2.1 Funciones con una entrada que afectan a una salida

Recordemos que este tipo de funciones recibían un registro como entrada, le aplicaba una función y el resultado era aplicado mediante la operación XOR a otro registro. Debido a este comportamiento, todas estas funciones aplican confusión en mayor o menor grado.

- **Función identidad:** la función en realidad no hace nada, la codificación es simplemente el XOR aplicado. Esta función no tiene demasiada importancia en criptografía, aplica confusión pero no difusión, y no hace uso de ninguna clave.

- **XOR con la clave:** función sencilla que aplica confusión pero no difusión. Es muy secilla y rápida de calcular y puede resultar muy eficiente si se combina con una función que aplique difusión. Sin conocer la clave es imposible descifrar un texto cifrado, pero es muy sencillo obtener la clave a partir de una pareja de texto en claro y texto cifrado.

- **Desplazamiento circular:** consiste en desplazar el registro hacia la izquierda o derecha de forma circular, es decir, los bits que se van desbordando se reintroducen por el otro lado. Esta función añade difusión. Esta función tiene la debilidad de que de un texto dado sólo puede obtener n textos cifrados, siendo n la longitud del registro. La clave es de longitud log2(n).

- **Permutación:** consiste en crear una permutación (a veces mediante una S-BOX) y aplicarla a la entrada. Suelen utilizarse varias permutaciones predefinidas características de cada criptosistema; a veces se utilizan algunas técnicas para expandir

los datos y posteriormente comprimirlos con una S-BOX de forma que para cada resultado existan unas pocas posibilidades y de esa forma sea imposible decidir cual de ellas ha sido. Por otro lado, aunque hay colisiones, el algoritmo considerado globalmente no tiene por qué tener debilidades.

  - Una consideración importante: si se crea la permutación de tamaño n, hay n! permutaciones posibles. Esta cantidad de información puede obtenerse con el entero superior de log2(n!), por ejemplo, para obtener todas las permutaciones de 32 bits hacen falta al menos 118 bits que las definan. La forma de conseguir que con los 118 bits se puedan obtener todas las permutaciones ya es más complicada, y frecuentemente se optará por utilizar más bits, con las consiguientes colisiones, pero que permitan definir de forma más sencilla la permutación.

- **Operaciones aritméticas:** existen criptosistemas que emplean sumas y productos de los datos con las claves. La suma módulo n añade confusión al sistema y, debido a los acarreos, añade una pequeña cantidad de difusión. La utilización del producto no es siempre posible ya que debe existir inverso para la operación de descifrado, con lo que obliga a trabajar módulo n con n primo. Trabajando con registros de 16 bits se puede emplear el número 65537 como módulo, y considerando el 0 como el 65535 de la misma forma en que lo utilida IDEA.
- **Operaciones polinómicas:** se pueden considerar los registros como polinomios donde los dígitos binarios representan a los exponentes. En estas circunstancias, se puede hacer uso de la suma de polinomios, y del producto de polinomios en el que se divide por otro polinomio irreducible y se toma el resto. Añaden principalmente confusión, el producto añade además difusión pero su cálculo puede llevar más tiempo.

- **Operaciones fruto de la combinación de operaciones ya mencionadas:** unas operaciones con capaces de añadir confusión, otras añaden difusión. Podemos combinar dos o más funciones para obtener lo mejor de los dos mundos.

Para lograr diseñar un criptosistema seguro con éxito debemos apoyarnos en varios principios. En referencia a las funciones utilizadas, es conveniente que muchas sean direccionales y difíciles de invertir. Las colisiones se han considerado malas en el sentido de que reducen el campo de búsqueda en un algoritmo por fuerza bruta cuando afectan al criptosistema en su conjunto, sin embargo pueden resultar muy útiles cuando esas colisiones son controladas. Un algoritmo que haga uso de funciones que provocan un número pequeño y constante de colisiones para cada resultado de la función hace imposible que se sepa cuál fue la entrada y, por lo tanto, darle la vuelta.
Esta forma de añadir mayor seguridad era utilizada en el DES y fue donde residió la mayor parte de su fuerza. No olvidemos que el DES no ha sido desechado por encontrar debilidades en su estructura, sino porque utiliza una clave demasiado pequeña de 56 bits y es vulnerable a un ataque por fuerza bruta.

5.2.2 Funciones de dos entradas que afectan a dos salidas

Estas funciones trabajan con el esquema de asimetría aparente, recordemos que las dos entradas se funden en una mediante XOR. Nos vamos a centrar en el caso de que sean aplicadas a los mismos registros del origen. Los esquemas a los que dan lugar pueden ser muy atractivos y a la vez muy simples de calcular. Veremos a continuación los ejemplos más sencillos que se pueden plantear con este esquema, y las interesantes propiedades que poseen todos ellos.

- **Función identidad:** No se realizan cambios a la entrada, no se necesita clave. Su comportamiento tras aplicar ese resultado a los

registros de origen es el de intercambiar su valor; de forma que A tiene el valor de B y B el de A.

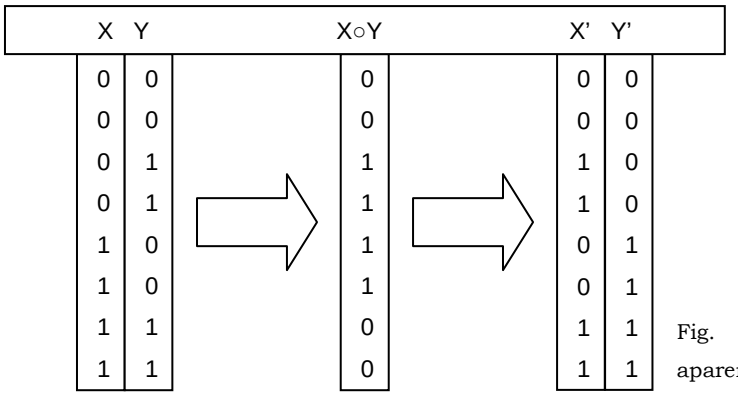


Fig. 4.3 Asimetría aparente trivial

- **Función AND:** A la entrada le es aplicada un AND con la clave, el comportamiento será muy curioso como veremos, y muy valioso para las aplicaciones criptográficas en las que queramos darle uso: consigue intercambiar de una forma excepcionalmente rápida y muy sencilla los bits de los dos registros en los que la clave valga 1, dejando los bits en los que la clave posee un valor de 0 sin modificarlos. Se permite hacer un intercambio de bits entre registros de forma que dependa totalmente de la clave, añade por lo tanto difusión a nivel de bits.

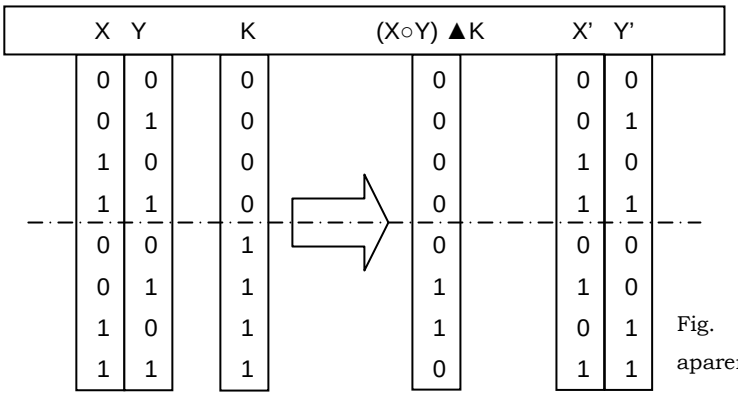


Fig. 4.4 Asimetría aparente AND

- **Función OR:** el comportamiento de la aplicación de la función OR es mucho más curioso si cabe. Se intercambian los valores en los que la clave vale 0 se niegan los valores en los que la clave vale 1 de forma que además de añadir confusión es capaz de añadir difusión:

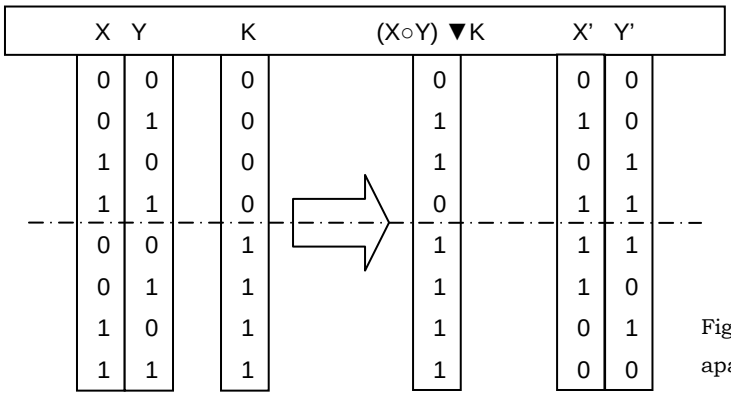


Fig. 4.5 Asimetría aparente OR

- **Función XOR:** esta función representa una vuelta más frente a las dos anteriores. En aquellos lugares donde la clave vale 0, se limita a intercambiar los valores de los registros. En los lugares en los que vale 1, además de intercambiar los valores los niega. Quizás sea demasiada vuelta de tuerca, pues el cifrado que se consigue es prácticamente un intercambio completo seguido de una negación dependiente de la clave. Si lo que deseamos es conseguir difusión es preferible hacer uso de las funciones AND y OR, siendo preferible la OR por la confusión añadida que genera.

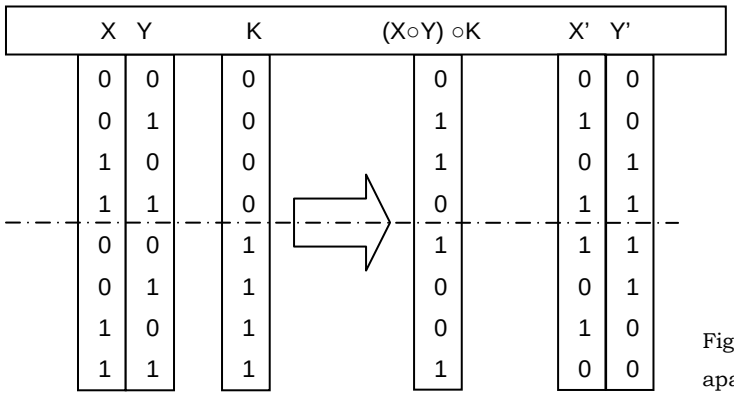


Fig. 4.6 Asimetría aparente XOR

5.2.3 Conseguir proteger las subclaves

Hemos mencionado que si en una función se hace uso de una subclave, siempre va a ser posible obtenerla con el texto fuente y el texto cifrado, pero si se hace uso de dos o más subclaves sólo se podrá obtener una relación ente ellas. Una relación entre subclaves independientes sólo aporta información acerca de la relación existente entre bits

independientes, conociendo uno de ellos podría obtener el valor del otro, el problema está en conocer el valor del primero y la relación existente.

5.2.4 Resistencia al criptoanálisis

Todos los esquemas mostrados hasta ahora y todas las operaciones (salvo las asimetrías aparentes no triviales AND y OR) determinan un flujo de datos más o menos complicado en el que en teoría pueden establecerse unas ecuaciones de comportamiento, es decir, presenta un flujo perfectamente determinista. Este determinismo consigue que se pueda averiguar cómo afecta cada bit de la entrada a la salida y sea factible la aplicación de criptoanálisis lineal. Las asimetrías aparentes AND y OR consiguen que el flujo del sistema sea dependiente de la clave y un ataque mediante criptoanálisis lineal no pueda realizarse.

Sin embargo, para una clave fija puede llevarse a cabo un ataque mediante criptoanálisis diferencial, en el que se cifran parejas de texto con diferencias conocidas y se analiza la evolución de esas diferencias a lo largo del flujo de datos. De este modo se van descartando claves por probabilidad, hasta que una clave aparece como candidata.

Una forma relativamente sencilla de evitar el criptoanálisis diferencial es conseguir que los datos puedan afectar al flujo de los mismos. De esta forma puede conseguirse que sea altamente resistente a este tipo de cripoanálisis ya que diferencias en el texto pueden representar no sólo una cascada de diferencias, sino también una modificación del flujo de bits. Existen otras técnicas, el IDEA es resistente al criptoanálisis diferencial gracias a las funciones que utiliza y a su particular estructura.

5.2.5 Modificación del flujo de datos según determinen los propios datos

Como hemos mencionado, es una forma de resistir al criptoanálisis diferencial, las diferencias entre textos no representan sólo una cascada de diferencias, sino que implican una modificación de esa cascada. Debemos tener cuidado, sin embargo, en conseguir que esa

modificación sea adecuada, es muy sencillo que se produzcan colisiones cuando unos datos se afectan a sí mismos, pero debemos mantenerlas bajo control. Las modificaciones que puede llevar a cabo los datos sobre datos deben ser de flujo, de forma que alteren la cascada de diferencias lo máximo posible. En este punto lo ideal es utilizarlos para crear permutaciones, o en su defecto, utilizarlos para crear una subfamilia de permutaciones.

Es imposible lograr que con una cadena de bits de longitud n podamos conseguir todas las posibles permutaciones de n elementos, ya que tenemos la desigualdad $2^n < n!$ para $n>4$, pero veremos que con modificaciones extremadamente sencillas pueden conseguirse protecciones fuertes.

Si tenemos una ristra de n bits, podemos escoger $\log_2(n)$ bits de cualquier forma, por ejemplo los últimos, y desplazar de forma circular toda la cadena hacia la izquierda. Esta función es no lineal y puede producir colisiones (hasta un máximo de $\log_2(n)$ colisiones para ciertas cadenas, pero que en supuestas cadenas aleatorias se reducirían enormemente. Este subgrupo de permutaciones puede alterar realmente el flujo de diferencias, en particular cuando esas diferencias afectan a los bits que se utilizan para determinar el desplazamiento. Para ser más efectiva podrían elegirse bits intermedios, de forma que la probabilidad de que no resulte afectado ninguno de los bits de desplazamiento por una modificación anterior quede reducida.

5.2.6 Criptosistema aleatorio

Recientemente se ha propuesto un método llamado “criptoanálisis imposible” que es capaz de proporcionar algoritmos mejores que la fuerza bruta para criptosistemas resistentes al criptoanálisis diferencial ordinario, como el IDEA o el Skipjack. Este nuevo método hace uso de unos diferenciales más potentes y permite ataques poderosos contra los mismos. Frente al criptoanálisis imposible presento la idea de un criptosistema en el que al menos una parte del sistema dependa de una variable aleatoria.

Aunque pueda parecer un tanto paradójico que datos aleatorios puedan afectar al flujo de los bits dentro del sistema, veremos que es posible y que permite efectivamente la decodificación de los datos.

En primer lugar debemos encontrar dónde introducir esos datos aleatorios. En la clave no, porque si esos datos se utilizan para cifrar, la fase de descifrado no podrá realizarse sin ellos; si no se utilizan son poco menos que inútiles. Debemos afectar a los datos, pero cuando los datos afecten al flujo del sistema como hemos visto anteriormente. No podemos modificar los datos en sí, pero podemos añadir fragmentos aleatorios que afectarán al flujo de la forma que ya hemos visto y que por lo tanto afectarán al sistema. Al final de la fase de decodificación esos datos serán desechados, obteniendo la información original.
Esta fase de añadir datos aleatorios al codificar, o desechar datos al decodificar, no puede realizarse en el núcleo del algoritmo que está conformado por la red elegida porque no es simétrico; estas operaciones deben hacerse a nivel intermedio entre el núcleo del algoritmo codificador y el programa que hace uso de él, como muestra el siguiente esquema:

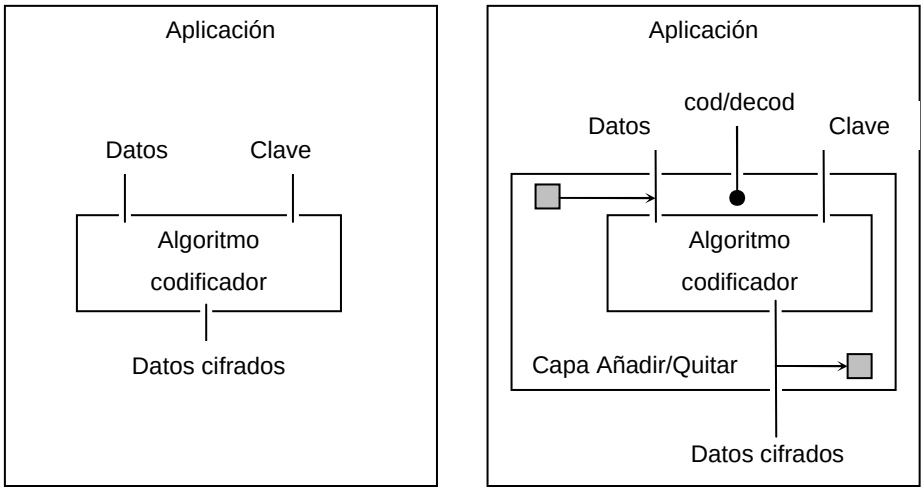


Fig. 4.7
Modificación para obtener un criptosistema aleatorio

El núcleo del algoritmo sigue siendo reversible, permite codificar y decodificar según las claves que recibe, la nueva capa actúa de diferente modo en las dos fases, de forma que si vamos a codificar añade datos al azar en posiciones determinadas antes de codificar, y si vamos a decodificar quita datos de esas mismas posiciones después de

decodificar. Esa nueva capa debe considerarse como parte del criptosistema, debe ser intocable desde fuera para no manipular los números aleatorios.

El principal inconveniente de esta técnica es que la información codificada aumenta de tamaño, un aumento definido y perfectamente controlado por otro lado. Esta técnica es especialmente recomendable para aplicaciones que requieran de una muy alta seguridad, y la seguridad ofrecida es tanto mayor cuanto mayor sea la cantidad de datos aleatorios incrustados.

Para aprovechar esta seguridad al máximo, conviene colocar esa información aleatoria en lugares que afecten al flujo en la primera ronda de la red, de esta forma nos aseguramos que la aleatoriedad inunde el flujo cuanto antes.

Una pequeña ventaja de esta técnica es que la fuerza bruta aplicada a una pareja de texto claro/texto cifrado simplemente se estrella. Supongamos que tenemos en nuestro poder un bloque de 120 bits y el bloque cifrado de 128 bits. Probar claves por fuerza bruta para codificar esos 120 bits y que el resultado dé los 128 que ya tenemos no es eficiente: probando todas las claves posibles tendríamos una probabilidad menor de 0,4% de encontrar la correcta. Es una pequeña ventaja porque la fuerza bruta aplicada en sentido inverso (descifrando con todas las claves hasta dar con la acertada) siempre tendrá éxito. Otra ventaja es que un solo mensaje podría dar 256 posibles mensajes cifrados diferentes como resultado.

Esta técnica puede se utilizada con cualquier algoritmo existente o futuro, como vemos es añadir una capa intermedia, y promete proveer de una seguridad añadida excepcional. Además, considerando el avance de las redes de comunicación y del almacenamiento experimentado recientemente, el aumentar ligeramente el tamaño de los mensajes cifrados no representa prácticamente ningún problema.

6 Claves

Hasta ahora se ha supuesto que disponemos de un método que nos proporciona subclaves a partir de la clave principal, hay muchas formas de conseguir esto pero lamentablemente hay muchas formas malas, y si nos descuidamos, muchas realmente malas para ello. Con la correcta utilización de la clave pueden evitarse muchos de los problemas que algunos criptólogos temen al diseñar un nuevo algoritmo como que el sistema presente estructura de grupo, o existan claves débiles o semidébiles. Estos temores han llevado por ejemplo a los autores del Skipjack a decantarse por una estructura no simétrica, de forma que para cifrar y descifrar se utilicen dos algoritmos diferentes.

Partamos de un supuesto ideal: si necesitamos x subclaves de longitud y, y dispusiéramos una clave de longitud $x*y$, no existirían problemas en la obtención de subclaves. Pero como esto no va a ser posible habrá bits que deban ser utilizados más de una vez, según el lema del palomar.

Desechemos lo evidente, un mismo bit no puede estar presente más de una vez en una subclave, es necesario que en una subclave todos sean independientes, ya que en caso contrario conocer un bit puede significar conocer otros.

6.1 Claves débiles y semidébiles

Vamos a centrarnos en el problema de la existencia de claves débiles y semidébiles. En estos sistemas la extracción de las subclaves se realizan en orden, es aquí donde deberemos tener el mayor cuidado. Si es posible cambiar todo el orden de aplicación a las subclaves y con ellas podemos construir una clave nueva, esa clave nueva será la pareja semidébil de la primera. Veamos con un ejemplo gráfico, utilizando una forma geométrica, para la obtención de las subclaves:

A	B	C	D	
1	0	1	1	**E**
0	1	1	0	**F**
0	0	0	1	**G**
1	0	1	0	**H**

Fig. 5.1

Con una clave de 16 bits vamos a obtener 8 subclaves de 4 bits, un total de 32 bits. Cada bit será utilizado dos veces, pero la forma de obtener las subclaves nos garantiza que no se usarán dos veces en la misma subclave.

Las 8 subclaves a utilizar serán las siguientes:

A	1001
B	0100
C	1101
D	1010
E	1101
F	0110
G	1000
H	0101

Figs: 5.2 y 5.3

Las claves E-H se toman de derecha a izquierda. Este método simple, y que puede padecernos incluso obvio, tiene escondido un pequeño defecto que puede hacer peligrar la seguridad del sistema: si reordenamos las claves en orden inverso, justo como serían aplicadas en la decodificación, podemos construir una estructura similar a la de partida.

H	G	F	E	D	C	B	A
0101	1000	0110	1101	1010	1101	0100	1001

H	G	F	E	
0	1	0	1	**D**
1	0	1	1	**C**
0	0	1	0	**B**
1	0	0	1	**A**

Fig. 5.4

Esta clave es la pareja semidébil de la clave original, sea el criptosistema que sea siempre que respete el tamaño del bloque (un criptosistema aleatorio carece de esta debilidad); lo peor de todo es que toda clave es semidébil y que este problema se presenta con cualquier tamaño del cuadrado, ya que la búsqueda de la pareja semidébil se realiza con una inversión del cuadrado según una de las diagonales.

El ejemplo mostrado sirve, además de para comprobar cómo actúa la búsqueda de claves semidébiles, para proporcionarnos ideas de cómo evitarlo.

6.2 Algoritmo para detectar claves débiles y semidébiles

Se va a proceder ahora a proporcionar un algoritmo que permitirá identificar en un criptosistema claves débiles y semidébiles. Para ello hay que hacer una consideración importante, cualquier operación que sea aplicada a las subclaves después de ser extraídas, o, en su defecto, cualquier operación que sea aplicada a una clave antes de extraer las subclaves y que alcanza el mismo resultado que una operación que se efectúe a las subclaves después de ser extraidas, no tiene interés en cuanto a evitar o provocar claves débiles y semidébiles se refiere. Se debe trabajar únicamente con la elección de los bits que forman cada subclave.

Si el algoritmo necesita que las subclaves en modo de descifrar tengan que ser modificadas para lograr los inversos de las funciones, es importante conseguir una forma de plantear esa modificación con forma de grafo para que sea útil este algoritmo. Si no es posible, sólo se podrá utilizar para comprobar si una clave es débil o semidébil, o no; aquí se va a suponer que no se realizan cálculos previos a la clave antes de ser aplicada, es decir, una clave y la que realiza la operación inversa en el algoritmo son iguales, caso típico de un XOR. En ese caso el grafo es simple: un bit de la clave de origen debe ir a un lugar determinado de la clave de destino, si esto lo podemos hacer con todos los bits entonces esa clave tiene una pareja y es débil o semidébil. Este hecho es el que ilustra el concepto de “paso siguiente” y el que debe ser modificado para hacer frente a los casos mencionados que no funcionarán bien.

Las subclaves, para descifrar, siguen un orden casi inverso (según dicta la estructura y la forma de utilizar las subclaves como ya se ha visto). Para detectar la presencia de claves débiles y semidébiles a partir de la elección de las subclaves, vamos a obtener las peculiaridades que deben de tener los bits de esas elecciones para provocarlas:

- Una clave será débil si y sólo sí, las subclaves inversas en la estructura del algoritmo son iguales.
- Una clave será semidébil si y sólo si con las subclaves inversas en orden inverso se puede reconstruir una nueva clave válida.

Dada una forma de extraer claves, vamos a ver para cada bit, cuales son los inversos, es decir, suponiendo que diésemos la vuelta a la clave, vamos a ver dónde iría a parar ese bit. Veámoslo mejor con el ejemplo anterior:

A	**B**	**C**	**D**	
1	2	3	4	**E**
5	6	7	8	**F**
9	10	11	12	**G**
13	14	15	16	**H**

Fig. 5.5

Hemos numerado los bits del 1 al 16 y, como habíamos supuesto antes, el orden es lineal: para cifrar se utiliza A-B-C-D-E-F-G-H y para descifrar H-G-F-E-D-C-B-A. De forma que si A=H y B=G y así sucesivamente podremos afirmas que la sucesión original podría invertir su efecto, sería una clave débil. Vamos a igualar los bits de los que hace uso cada una de las subclaves para ver la supuesta relación que debería haber entre ellos:

1 5 9 13	16 15 14 13
2 6 10 14	12 11 10 9
3 7 11 15	8 7 6 5
4 8 12 16	4 3 2 1
4 3 2 1	4 8 12 16
8 7 6 5	3 7 11 15
12 11 10 9	2 6 10 14
16 15 14 13	1 5 9 13

Fig. 5.6

Lo que equivale a decir que el bit que actúa en el lugar 1, en la operación de descifrado actuará en el lugar 16, el 5 en el 15 y así sucesivamente. Ahora es necesario construir un grafo dirigido con tantos nodos como bits tenga la clave y en el que cada arista partirá desde cada bit de las subclaves usadas para codificar y llegará a cada bit de las subclaves usadas para decodificar:

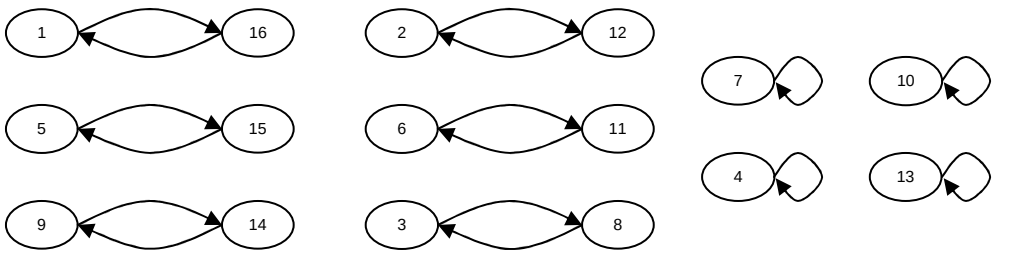


Fig. 5.7

De donde podemos obtener todas las claves débiles: si a todos los elementos de cada subgrafo conexo le asignamos el mismo valor, la clave resultante será una clave débil; de forma que las claves débiles serán de la forma: A-B-C-D-E-F-G-C-H-I-F-B-J-H-E-A representando los 16 bits de la clave en orden (1..16) y pudiendo tomar cada letra un

valor de 1 o 0. Según esta forma de selección de la clave habrá 2^{10} claves débiles, que de un conjunto de 2^{16}, significa que una de cada 64 claves posibles es débil.

De este mismo grafo podremos obtener las claves semidébiles. Olvidemos los demás bits y centrémonos sólo en 2. Supongamos que una clave tuviese en el primer bit un 1 y en su pareja un 0. Supongamos ahora que queremos descifrar con esa clave, para ello debería haber un 0 en el primer bit y un 1 en su pareja. Como consecuencia de esto, volver a cifrar con la misma clave no podrá descifrar el mensaje como ocurría con las claves débiles, pero volver a cifrar con una clave diferente que tuviese en el primer bit un 0 y en su pareja un 1 descifraría el mensaje. Por supuesto esto debería ser extendido a todos los subgrafos conexos.

Esa clave descrita sería la pareja semidébil de la clave original. Como consecuencia podemos calcular el número de claves semidébiles existentes. Primero calculamos el número de claves que tienen una pareja (débiles y semidébiles) y posteriormente le restamos la cantidad de claves débiles que hemos calculado, con ello obtenemos las claves semidébiles que existen.

Para cada subgrafo conexo de dos nodos existen 4 posibilidades, y para cada nodo aislado sólo 2: de forma que el número de claves que tienen una pareja es de $4^6 \cdot 2^4 = 2^{16}$; todas las claves tienen pareja, es decir, todas las claves son débiles o semidébiles y habrá $2^{16} - 2^{10} = 64512$ claves semidébiles.

Los cálculos han sido sencillos por la sencillez de los grafos, pero para realizar cálculos con los subgrafos conexos complejos vamos a definir el "paso siguiente". El "paso siguiente" de un grafo sería un nuevo grafo resultado de asignar a cada nodo destino de cada arista el valor del origen de la arista; si se producen colisiones (un nodo quiere valer 0 y 1 a la vez) diremos que no existe el paso siguiente. Veamos un ejemplo para ilustrar este concepto:

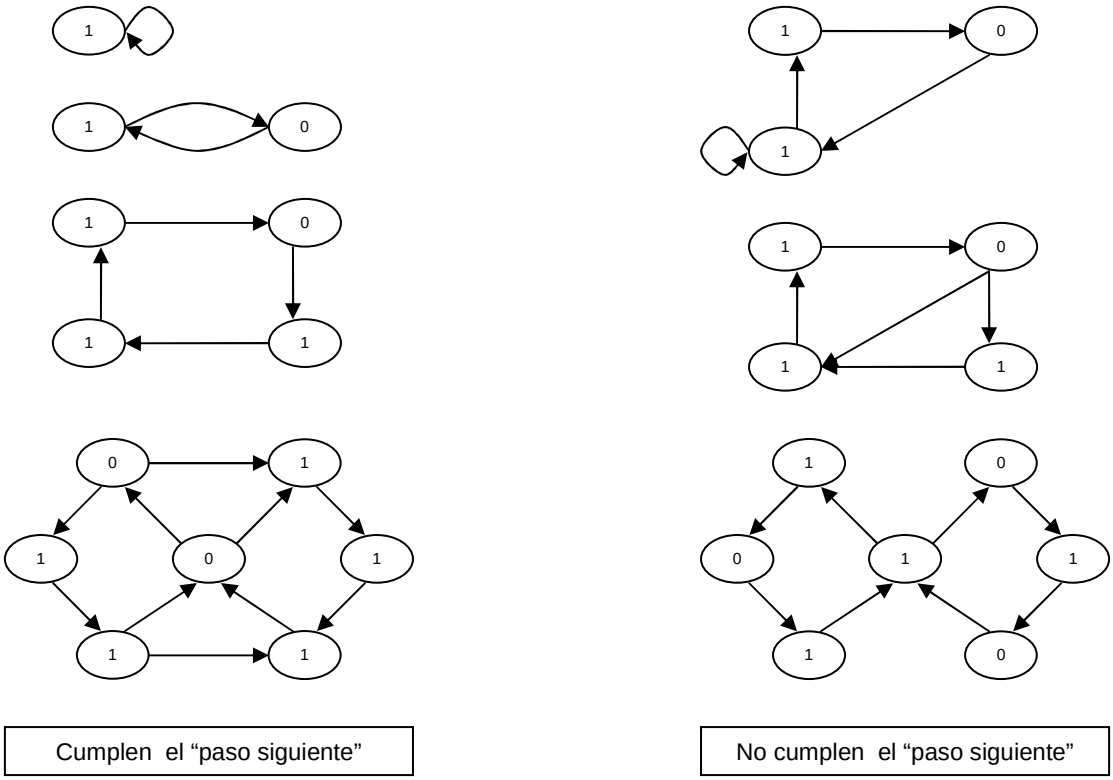


Fig. 5.8 Paso siguiente

Para que un grafo pueda provocar la existencia de claves con pareja es necesario que exista el grafo “paso siguiente”, esto se puede justificar porque la clave original en modo de decodificación se ajustará a este grafo.

Si nos detenemos a pensar un poco descubriremos que si un bit se relaciona con otro, por la particular construcción de los grafos que vamos a obtener, el segundo bit se debe relacionar obligatoriamente al primero. Para representarlo de forma más sencilla se utilizarán grafos no dirigidos en el que dos nodos están interrelacionados si están unidos por una arista. Aunque esto pueda parecernos trivial la verdad es que nos dota de algo más que de la ventajosa representación: los subgrafos conexos deberán grafos bipartidos y contarán como mucho con cuatro formas posibles de repartir los valores que cumplan el “paso siguiente”. Otra consecuencia inmediata es que las claves forman realmente una pareja: lo que se cifra con una se puede descifrar con la otra, son conmutativas.

Comprobando el “paso siguiente” podemos saber si una clave dada es débil o semidébil. Para calcular la cantidad de claves con pareja que hay debemos analizar los subgrafos conexos. Un subgrafo conexo es capaz de aportar un factor multiplicativo X al cómputo total, donde X son todas las posibles formas de dar valores a los nodos y posean “paso siguiente”. Un subgrafo conexo como mínimo aportará 2 posibles soluciones que cumplan la regla del “paso siguiente” que son las soluciones triviales (todos los nodos poseen un 0 o todos poseen un 1); de modo que como mínimo existirá el mismo número de claves con parejas como claves débiles haya. El máximo es de 4, pues se trata con grafos bipartidos.

Así por ejemplo, un grafo que posea 8 subgrafos conexos de los cuales hay 4 que admiten cuatro formas de asignar a los nodos un valor y otros 4 admiten 2 formas, existirán 4096 claves con pareja, de las cuales 256 serían débiles y el resto semidébiles.

Como consecuencia de lo anterior podemos obtener dos resultados importantes:

- Es mejor que el grafo tenga la menor cantidad de subgrafos conexos posibles, idealmente el grafo debería ser conexo; además es mejor que los grafos no sean bipartidos (porque admitirían una solución de alternancia de 0 y 1, y el factor con el que contribuirían sería de 4 en vez de 2).
- En todo criptosistema que utilice el mismo procedimiento para codificar y decodificar (sólo cambiando las subclaves e orden) existirán dos claves débiles, que son las claves triviales.

Si un dato de la clave no fuese utilizado aparecería como un nodo aislado en la representación, esto significaría que no importa su valor en absoluto y una clave podría tener más de una pareja por su culpa.

6.3 Ejemplo de búsqueda de claves débiles y semidébiles

Como ejemplo del citado algoritmo se va a proceder a buscar estas claves en el DES. Es un ejemplo idóneo porque las claves se utilizan para realizar la operación XOR y no son modificadas desde la extracción

hasta la aplicación. El DES divide la clave original en dos subconjuntos que serán disjuntos desde el primer momento. A esos subconjuntos se

les aplican las mismas modificaciones de forma independiente, luego se cogen unos bits determinados de cada uno. El hecho de que estos conjuntos sean tratados por igual nos permite tener ya de antemano dos cosas: hay al menos dos subgrafos conexos y los dos son isomorfos.

Suponiendo la clave original de 64 bits de las que se desechan los bits múltiplos de 8 por ser de paridad, las 16 subclaves en orden utilizarán los siguientes bits:

10	51	34	60	49	17	33	57	2	9	19	42	3	35	26	25	44	58	59	1	36	27	18	41
2	43	26	52	41	9	25	49	59	1	11	34	60	27	18	17	36	50	51	58	57	19	10	33
51	27	10	36	25	58	9	33	43	50	60	18	44	11	2	1	49	34	35	42	41	3	59	17
35	11	59	49	9	42	58	17	27	34	44	2	57	60	51	50	33	18	19	26	25	52	43	1
19	60	43	33	58	26	42	1	11	18	57	51	41	44	35	34	17	2	3	10	9	36	27	50
3	44	27	17	42	10	26	50	60	2	41	35	25	57	19	18	1	51	52	59	58	49	11	34
52	57	11	1	26	59	10	34	44	51	25	19	9	41	3	2	50	35	36	43	42	33	60	18
36	41	60	50	10	43	59	18	57	35	9	3	58	25	52	51	34	19	49	27	26	17	44	2
57	33	52	42	2	35	51	10	49	27	1	60	50	17	44	43	26	11	41	19	18	9	36	59
41	17	36	26	51	19	35	59	33	11	50	44	34	1	57	27	10	60	25	3	2	58	49	43
25	1	49	10	35	3	19	43	17	60	34	57	18	50	41	11	59	44	9	52	51	42	33	27
9	50	33	59	19	52	3	27	1	44	18	41	2	34	25	60	43	57	58	36	35	26	17	11
58	34	17	43	3	36	52	11	50	57	2	25	51	18	9	44	27	41	42	49	19	10	1	60
42	18	1	27	52	49	36	60	34	41	51	9	35	2	58	57	11	25	26	33	3	59	50	44
26	2	50	11	36	33	49	44	18	25	35	58	19	51	42	41	60	9	10	17	52	43	34	57
18	59	42	3	57	25	41	36	10	17	27	50	11	43	34	33	52	1	2	9	44	35	26	49

22	28	39	54	37	4	47	30	5	53	23	29	61	21	38	63	15	20	45	14	13	62	55	31
14	20	31	46	29	63	39	22	28	45	15	21	53	13	30	55	7	12	37	6	5	54	47	23
61	4	15	30	13	47	23	6	12	29	62	5	37	28	14	39	54	63	21	53	20	38	31	7
45	55	62	14	28	31	7	53	63	13	46	20	21	12	61	23	38	47	5	37	4	22	15	54
29	39	46	61	12	15	54	37	47	28	30	4	5	63	45	7	22	31	20	21	55	6	62	38
13	23	30	45	63	62	38	21	31	12	14	55	20	47	29	54	6	15	4	5	39	53	46	22
28	7	14	29	47	46	22	5	15	63	61	39	4	31	13	38	53	62	55	20	23	37	30	6
12	54	61	13	31	30	6	20	62	47	45	23	55	15	28	22	37	46	39	4	7	21	14	53
4	46	53	5	23	22	61	12	54	39	37	15	47	7	20	14	29	38	31	63	62	13	6	45
55	30	37	20	7	6	45	63	38	23	21	62	31	54	4	61	13	22	15	47	46	28	53	29
39	14	21	4	54	53	29	47	22	7	5	46	15	38	55	45	28	6	62	31	30	12	37	13
23	61	5	55	38	37	13	31	6	54	20	30	62	22	39	29	12	53	46	15	14	63	21	28
7	45	20	39	22	21	28	15	53	38	4	14	46	6	23	13	63	37	30	62	61	47	5	12
54	29	4	23	6	5	12	62	37	22	55	61	30	53	7	28	47	21	14	46	45	31	20	63
38	13	55	7	53	20	63	46	21	6	39	45	14	37	54	12	31	5	61	30	29	15	4	47
30	5	47	62	45	12	55	38	13	61	31	37	6	29	46	4	23	28	53	22	21	7	63	39

Fig. 5.9 Bits utilizados por DES en cada una de las subclaves

Los números representan el bit de la clave original que se usa. Construiremos el grafo sabiendo que el bit 10 de la clave original debe ser igual al bit 18 de su pareja, del mismo modo el bit 18 de la clave original debe ser igual al bit 10 de la pareja. Se utilizará una matriz de incidencias para representar uno de los subconjuntos conexos, el correspondiente a la primera mitad.

La matriz de incidencia es la siguiente:

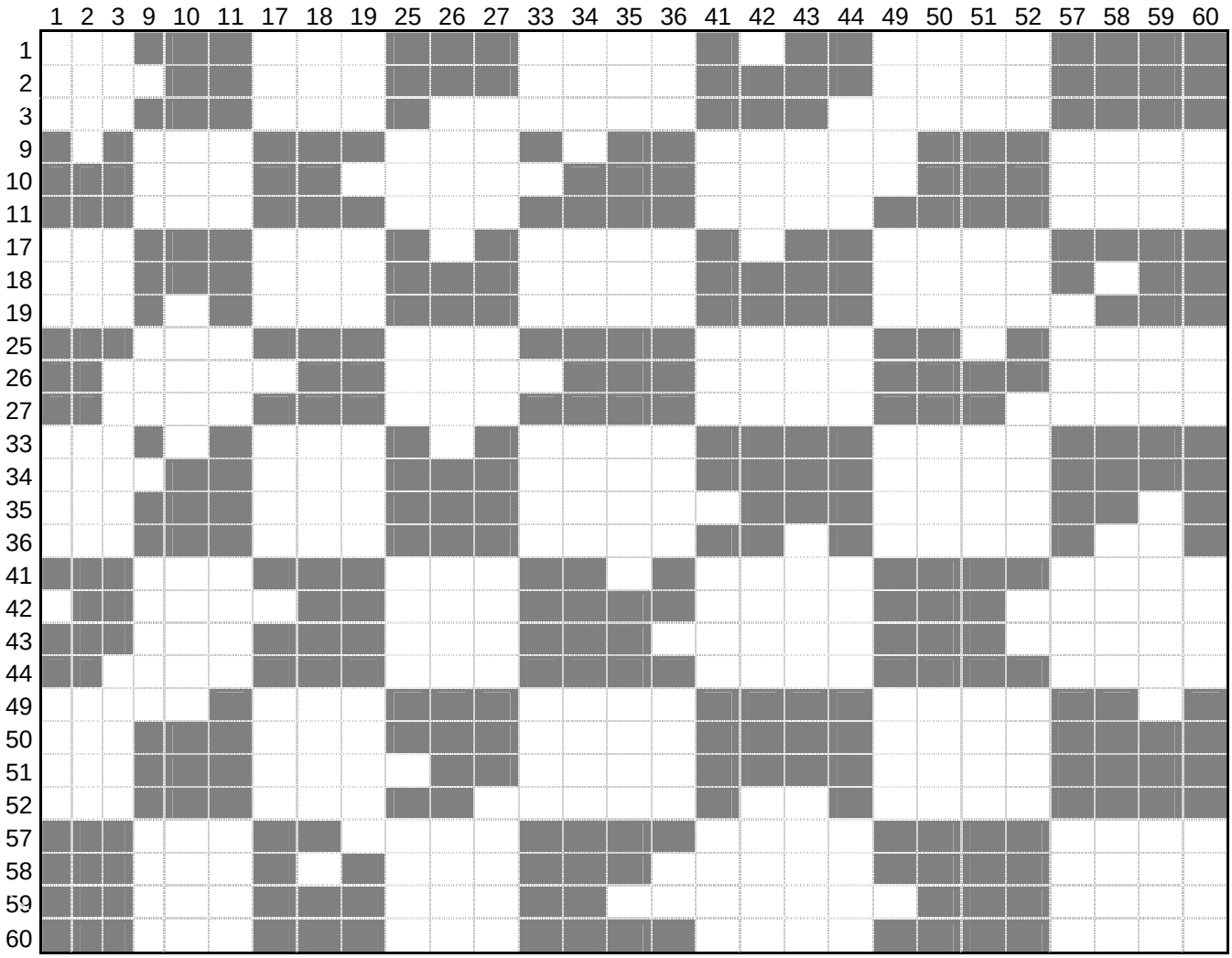


Fig. 5.10 Matriz de adyacencias de los bits utilizados en la primera mitad del DES

Ahora hay que determinar dos cosas: primero si el grafo es conexo, y segundo si es bipartido. Para ello hagamos una tabla con dos columnas que iremos rellenando. En una de ellas colocaremos un nodo al azar, en la otra aquellos en los que incide y no estén ya allí, a continuación haremos lo mismo con cada uno de los nodos que hayamos añadido, colocando los nodos en los que incide en la otra parte de la tabla. Si al acabar el algoritmo están todos los nodos, el grafo es conexo, y si no hay ningún nodo repetido en las dos columnas, es bipartido.

Procedamos a analizar cómo se comporta el grafo:

1	3	9	10
17	18	11	25
19	33	26	27
35	36	41	43
50	51	44	57
52	2	58	59
34	49	60	42

Fig. 5.11 Análisis del grafo obtenido para la primera mitad del DES

Como vemos, el grafo es conexo y bipartido de forma que ya podemos calcular cuantas claves débiles y semidébiles presenta el DES: al haber dos subgrafos bipartidos hay 2^2=4 claves débiles porque son dos los subgrafos conexos. El número de claves con pareja sería de 4^2=16, de las que 4 son débiles como ya hemos visto; de modo que quedan 12 claves semidébiles. El propio algoritmo podría buscarlas, dando a cada nodo del subgrafo todo ceros, todo unos, todo ceros y unos (es bipartido) y al revés. Las claves serían:

Débiles

01 01 01 01 01 01 01 01
FE FE FE FE FE FE FE FE
1F 1F 1F 1F 0E 0E 0E 0E
E0 E0 E0 E0 F1 F1 F1 F1

Semidébiles

01FE 01FE 01FE 01FE, FE01 FE01 FE01 FE01
1FE0 1FE0 0EF1 0EF1, E01F E01F F10E F10E
01E0 01E0 01F1 01F1, E001 E001 F101 F101
1FFE 1FFE 0EFE 0EFE, FE1F FE1F FE0E FE0E
011F 011F 010E 010E, 1F01 1F01 0E01 0E01
E0FE E0FE F1FE F1FE, FEE0 FEE0 FEF1 FEF1

Fig. 5.12 Claves débiles y semidébiles obtenidas

Contando ya con los bits de paridad.

6.4 Evitando codificaciones débiles

Existe un problema que puede ser más o menos grave según qué criptosistemas y consiste en que una clave con abundancia de 0 o de 1 puede provocar una codificación muy débil de los datos. En casos extremos podría llegar incluso a que la única codificación de los datos proviniese de los propios datos. Este es un problema menor pero real: en el algoritmo IDEA se ha encontrado una familia de claves que puede ser identificada si se utilizan por la debilidad del cifrado que provocan. Estas claves son desechadas por diseño, sin embargo se va a proporcionar un procedimiento que consigue que incluso estas claves den una mayor seguridad en la codificación.

Para ello se va a hacer uso de los dos valores que puede almacenar un bit, utilizando a veces el 0 y otras veces el 1. Se utilizarán "filtros" encargados de modificar los valores de cada subclaves, las reglas para construirlos se darán después de ver un ejemplo en el que se aplica:

A	B	C	D	
0	0	0	0	**E**
0	0	0	0	**F**
0	0	1	0	**G**
0	0	0	0	**H**

Clave

A	B	C	D	
1	0	0	1	**E**
1	0	1	0	**F**
1	1	1	0	**G**
1	0	0	0	**H**

Filtro 1

A	B	C	D	
0	0	1	1	**E**
0	1	1	0	**F**
1	0	0	1	**G**
1	0	1	0	**H**

Filtro 2

Fig. 5.13 Ejemplos de filtros

El hecho de utilizar otra vez esta forma de seleccionar las subclaves, a pesar de haberla desechado ya, es por la utilidad que posee con fines didácticos. La técnica se realiza de la siguiente manera: para las cuatro primeras subclaves se toman XOR el primer filtro y las 4 segundas se toman XOR el segundo filtro. Estos filtros son definidos en el diseño. De esta forma las subclaves que se aplicarían serían las siguientes:

A	B	C	D	E	F	G	H
1111	0010	0100	1000	0011	0110	1011	1010

Fig. 5.14

Vemos que las subclaves obtenidas presentan mucha menos uniformidad que las que se obtendrían son los filtros. Además con los filtros se aprovecha la información de la clave al máximo siempre, un bit de la clave que varíe hará variar el valor de ese bit en cualquier sitio donde sea utilizado.

Es muy importante utilizar más de un filtro, de hecho es mejor utilizar tantos filtros como veces puede ser elegido un bit para formar parte de una clave. El diseño de los filtros debería ser el siguiente:

- Tanta cantidad de filtros al menos como la mitad de veces pueda ser elegido un bit.
- Un 50% de ceros y la misma cantidad de unos.
- Entre cada pareja de filtros debería de haber alrededor de un 50% de diferencias.

Con los filtros nos garantizamos de que una clave no pueda ser dada especialmente para provocar un cifrado débil, se puede utilizar para anular el efecto de un filtro pero nunca de todos ellos; esa es además la razón de que un filtro sea inútil, sólo cambia de sitio el problema. Cuanto mayor sea el parecido de la clave con uno de los filtros, más se acercará el parecido con el resto de filtros al 50%.

Como ya se dijo, los filtros no pueden proteger contra claves débiles y semidébiles. Cuando se produce la decodificación hay que realizar la inversión de filtros para que esta operación sea posible; en el ejemplo es sencillo verlo pues basta con cambiar de orden los filtros y aplicar simetría a todas las matrices por la diagonal de 45º, pero en un ejemplo más complejo puede ser un poco más complicado. Es preferible desglosar los filtros grandes en pequeños filtros, cada uno de los cuales se aplica a una subclave, y cuando las subclaves se inviertan para decodificar, invertir exactamente de la misma forma a los filtros. Utilizar filtros iguales para claves inversas en el algoritmo revierte en mayor rapidez de ejecución, pues los filtros están asociados a las funciones, pero se podrían asociar a las subclaves, siendo computados sólo una vez.

7 Propuesta de algoritmo simétrico de cifrado

Con todo lo expuesto anteriormente, se va a diseñar un algoritmo de cifrado que posea las siguientes características:

1. Ofrecer un nivel elevado de seguridad.
2. Rápido.
3. Operaciones poco complejas.
4. Resistente a los ataques actuales.
5. Fácil de implementar.
6. Algoritmo de encriptación y desencriptación idénticos.
7. Utilizará una clave de un mínimo de 512 bits sin más claves débiles y semidébiles que las triviales.

Su nombre será NICS (Nuevas ideas en criptografía simétrica).

7.1 Estructura

El algoritmo dividirá el mensaje en fragmentos de 128 bits. Cada bloque se dividirá en 4 partes de 32 bits, para facilitar su implementación en procesadores actuales. Su construcción seguirá el modelo de eslabones, compuestos por dos bloques: principal y accesorio detallados a continuación:

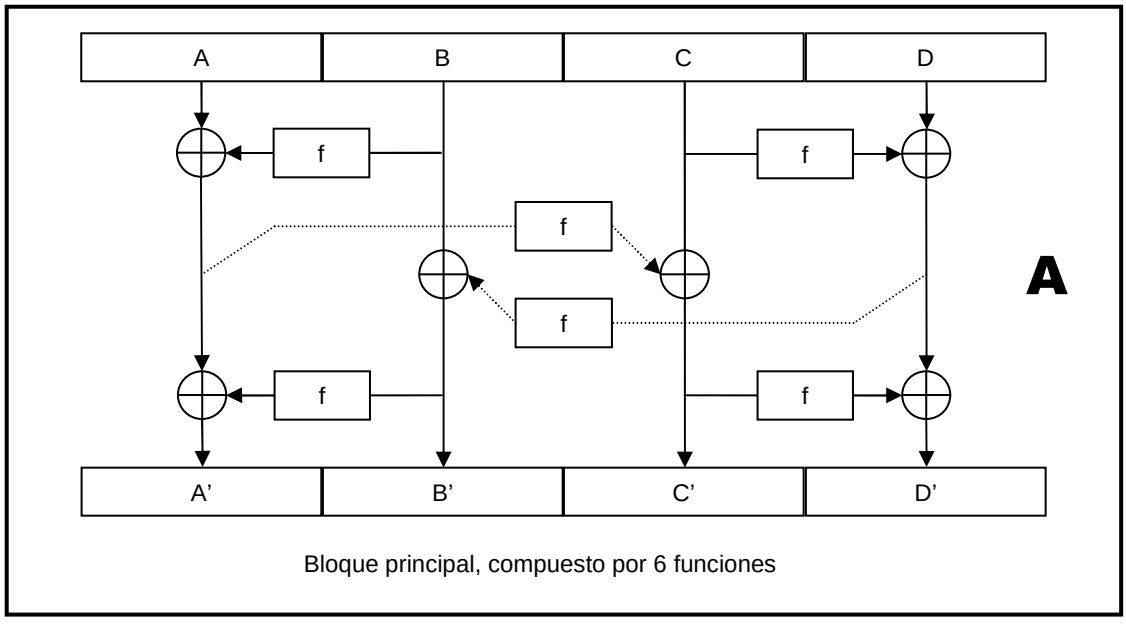


Fig. 6.1 Bloque principal del NICS

El bloque “accesorio” tiene esta forma:

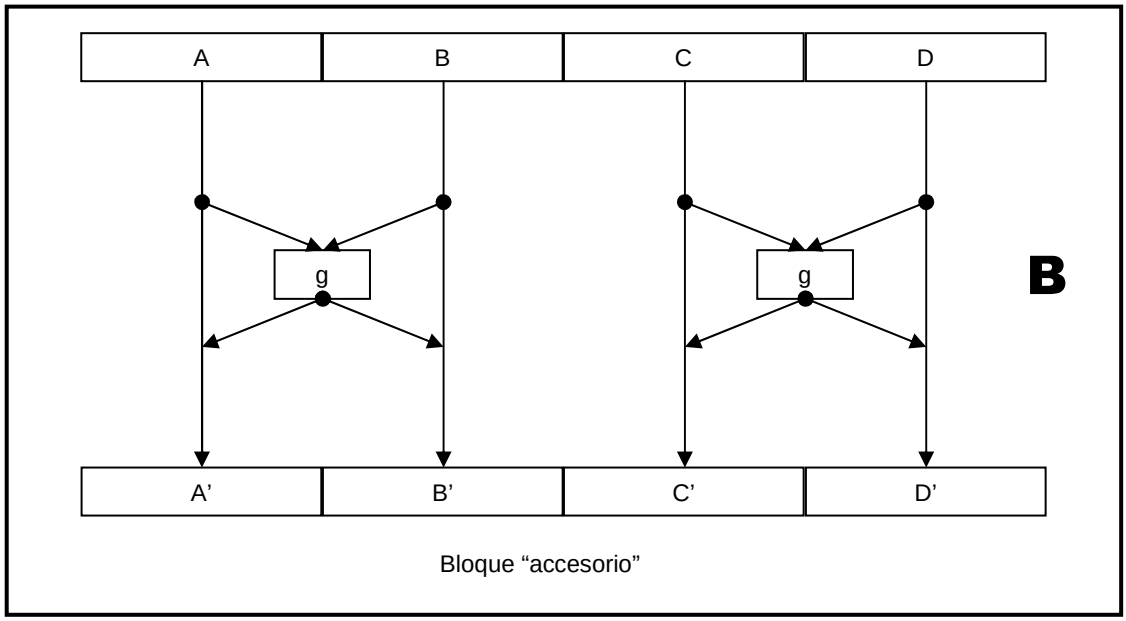


Fig. 6.2 Bloque accesorio del NICS

La construcción será de la siguiente manera:

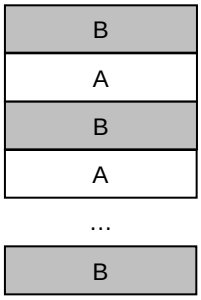


Fig. 6.3

El bloque A (principal) se encargará principalmente de añadir confusión, el bloque B se encargará de la difusión. No existen permutaciones explícitas entre registros, para realizar las permutaciones se hará uso de la asimetría aparente en el bloque B con la operación OR que proporciona además un poco de confusión.

7.2 Las funciones

Las funciones que se utilizarán (*f* y *g*) serán sencillas pero fuertes. La función *f* hará uso de dos subclaves diferentes y funcionará del siguiente modo: (imaginemos que *f* se aplica desde B hasta A, K_1 y K_2 son las dos subclaves de 32 bits)

1. Se calcula Z = B xor K_1.
2. Se toman los 5 últimos bits de Z y con ellos se desplaza Z de forma circular hacia la izquierda para calcular Z’.
3. Se calcula Z” = Z’ xor K_2.
4. Se aplica a A, A = A xor Z”.

La función *g* será una de las mostradas en el los ejemplos de asimetría aparente, es la asimetría aparente OR que utiliza una subclave de 32 bits y funciona intercambiando aquellos bits donde la clave vale 0 y negando los bits en los que la clave vale 1. Los pasos son los siguientes: (Utilizando los registros A y B, y la subclave K).

1. Calcular Z = A xor B.

2. Calcular Z' = Z or K.

3. Se aplica a A. A=A xor Z'.

4. Se aplica a B. B=B xor Z'.

Se han escogido estas funciones por varias razones, entre ellas porque los propios datos son capaces de alterar el flujo de modificaciones. La función *f* utiliza los datos para modificarse a si mismos, es una función no inversible gran parte de las veces, con un número de colisiones variable pero pequeño, y que hace que las dos claves utilizadas tengan una relación variable: su relación depende de los datos y es más difícil obtener información de ellas.

La función *g* también se muestra muy útil: en la codificación que realiza la función *f* un cambio en uno de los bits se transmite de forma desigual en cada vuelta, un bit en el registro B afectará mucho más al bloque A que al B; también es cierto que según pasen las vueltas el bloque A afectará al B. Aun así *g* consigue que las diferencias entre los registros A y B, y C y D se igualen ronda a ronda.

Para determinar el número de vueltas, habría que buscar el número mínimo de rondas que nos garantizan que la modificación de un bit en la entrada puede afectar a la salida en un 50% pero que todos los bits puedan ser afectados. El número de rondas es directamente proporcional a la seguridad del sistema, e inversamente proporcional a la velocidad del mismo. Para este sistema, cuatro rondas pueden en el mejor de los casos cumplir las expectativas, pero se van a utilizar seis para aumentar la seguridad.

Se necesitarán 12 subclaves de 32 bits para cada bloque A y 2 claves de 32 bits para cada bloque B. Existen 6 bloques A y 7 bloques B. En total haremos uso de 86 subclaves de 32 bits lo que hace un total de 2752 bits necesarios.

En cada ronda compuesta por las funciones *f* y *g* se utilizan 448 bits, se desea que en cada una de esas rondas cada bit sea utilizado una sola vez y que no sean utilizados todos ellos. Las claves que se van a utilizar tendrán una longitud de 512 bits, cada bit será utilizado 5, 6 o 7 veces, las claves atravesarán filtros simétricos y se comprobará la ausencia de claves débiles y semidébiles con el algoritmo descrito en la primera parte.

El procedimiento para la obtención de las subclaves es el siguiente:

1. A partir de la clave original se obtienen las 14 subclaves necesarias para la primera ronda. Se selecciona para ello los 32 primeros bits para la primera subclaves, los 32 siguiente para la segunda subclave, y así hasta obtener las 14 subclaves de la primera ronda. Con un total de 448 bits utilizados.
2. Se aplica un desplazamiento circular hacia la izquierda de 67 posiciones.
3. Se obtienen las siguientes 14 claves desde el principio, claves para todos los bloques excepto para el B'. Se regresa al paso 2 y se repite cuantas veces sea necesario.

A continuación se presenta una tabla con los bits que se emplean para cada una de las subclaves, será nuestra base para demostrar que el grafo que se obtendría es conexo y no es bipartido.

Que es conexo es sencillo e intuitivo: Estableciendo las relaciones entre las primeras claves y las últimas, sólo con una ronda, se obtiene un camino en el que participan casi todos los nodos. Al ir añadiendo más relaciones, esos nodos descolgados se van uniendo hasta que se forma un grafo conexo. No se repiten relaciones, como ocurría en el DES.

Fig. 6.4 Listado de los bits que se usan en cada subclave

1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
3	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
4	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
5	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
6	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
7	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
8	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
9	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
10	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319
11	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351
12	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383
13	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415
14	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447
15	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98
16	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130
17	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162
18	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194
19	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226
20	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258
21	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290
22	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322
23	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354
24	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386
25	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418
26	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450
27	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482
28	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	0	1	2
29	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165
30	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
31	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229
32	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261
33	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293
34	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325
35	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357
36	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389
37	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421
38	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453
39	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485
40	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	0	1	2	3	4	5
41	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
42	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
43	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232
44	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264
45	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296
46	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328
47	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360
48	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392
49	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424
50	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456
51	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488
52	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	0	1	2	3	4	5	6	7	8
53	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
54	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
55	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104
56	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136
57	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299
58	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331
59	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363
60	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395
61	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427
62	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459
63	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491
64	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	0	1	2	3	4	5	6	7	8	9	10	11
65	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
66	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
67	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107
68	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139
69	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171
70	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203
71	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366
72	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398
73	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430
74	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462
75	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494
76	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
77	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
78	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78
79	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110
80	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142
81	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174
82	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206
83	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238
84	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270
85	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433
86	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465

Demostrar que no es bipartido conlleva mucho más trabajo. Hay que lograr representar el grafo de alguna forma y la gran cantidad de nodos dificulta esta tarea bastante. Evidentemente no resulta práctico una representación tradicional con nodos y aristas, y tampoco es manejable una matriz de adyacencias como se hizo en el caso del DES. Se ha optado por listar los nodos y a su derecha indicar cuales son sus relaciones con los demás nodos. No sirve para tener una visión global, pero es muy práctico para poder aplicar el algoritmo descrito anteriormente para descubrir si el grafo es bipartido o no. A continuación se presenta la lista sobre la que se ha trabajado, indicando para cada nodo aquellos nodos con los que se relaciona:

0	402	457	323	445	311	433
1	403	458	324	446	312	434
2	404	459	325	447	313	435
3	405		326	448	314	436
4	406		327	449	315	437
5	407		328	450	316	438
6	408		329	451	317	439
7	409		330	452	318	440
8	410		331	453	319	441
9	411		332	198	320	442
10	412		333	199	321	443
11	413		334	200	322	444
12	414		335	201	323	445
13	415		336	202	324	446
14	416		337	203	325	447
15	417		338	204	326	192
16	418		339	205	327	193
17	419		340	206	328	194
18	420		341	207	329	195
19	421		342	208	330	196
20	422		343	209	331	197
21	423		344	210	332	198
22	424		345	211	333	199
23	425		346	212	334	200
24	426		347	213	335	201
25	427		348	214	336	202
26	428		349	215	337	203
27	429		350	216	338	204
28	430		351	217	339	205
29	431		352	218	340	206
30	432		353	219	341	207
31	433		354	220	342	208
32	434		355	221	343	209
33	435		356	222	344	210
34	436		357	223	345	211
35	437		358	224	346	212
36	438		359	225	347	213
37	439		360	226	348	214

38	440		361	227	349	215
39	441		362	228	350	216
40	442		363	229	351	217
41	443		364	230	352	218
42	444		365	231	353	219
43	445		366	232	354	220
44	446		367	233	355	221
45	447		368	234	356	222
46	448		369	235	357	223
47	449		370	236	358	224
48	450		371	237	359	225
49	451		372	238	360	226
50	452		373	239	361	227
51	453		374	240	362	228
52	454		375	241	363	229
53	455		376	242	364	230
54	456		377	243	365	231
55	457		378	244	366	232
56	458		379	245	367	233
57	459		380	246	368	234
58	460		381	247	369	235
59	461		382	248	370	236
60	462		383	249	371	237
61	463		384	250	372	238
62	464		385	251	373	239
63	465		386	252	374	240
64	143		387	253	375	241
65	144		388	254	376	242
66	145		389	255	377	243
67	146	335	390	256	378	244
68	147	336	391	257	379	245
69	148	337	392	258	380	246
70	149	338		259	381	247
71	150	339		260	382	248
72	151	340		261	383	249
73	152	341		262	384	250
74	153	342		263	385	251
75	154	343		264	386	252

76	155	344	265	131	253
77	156	345	266	132	254
78	157	346	267	133	255
79	158	347	268	134	256
80	159	348	269	135	257
81	160	349	270	136	258
82	161	350	271	137	259
83	162	351	272	138	260
84	163	352	273	139	261
85	164	353	274	140	262
86	165	354	275	141	263
87	166	355	276	142	264
88	167	356	277	143	265
89	168	357	278	144	266
90	169	358	279	145	267
91	170	359	280	146	268
92	171	360	281	147	269
93	172	361	282	148	270
94	173	362	283	149	271
95	174	363	284	150	272
96	175	364	285	151	273
97	176	365	286	152	274
98	177	366	287	153	275
99	178	367	288	154	276
100	179	368	289	155	277
101	180	369	290	156	278
102	181	370	291	157	279
103	182	371	292	158	280
104	183	372	293	159	281
105	184	373	294	160	282
106	185	374	295	161	283
107	186	375	296	162	284
108	187	376	297	163	285
109	188	377	298	164	286
110	189	378	299	165	287
111	190	379	300	166	288
112	191	380	301	167	289
113	192	381	302	168	290
114	193	382	303	169	291
115	194	383	304	170	292
116	195	384	305	171	293
117	196	385	306	172	294
118	197	386	307	173	295
119	198	387	308	174	296
120	199	388	309	175	297
121	200	389	310	176	298
122	201	390	311	177	299
123	202	391	312	178	300
124	203	392	313	179	301
125	204	393	314	180	302
126	205	394	315	181	303
127	206	395	316	182	304
128	207	396	317	183	305
129	208	397	318	184	306
130	209	398	319	185	307
131	210	76	320	186	308
132	211	77	321	187	309
133	212	78	322	188	310
134	213	79	268	323	189	311
135	214	80	269	324	190	312
136	215	81	270	325	191	313
137	216	82	271	192	314
138	217	83	272	193	315
139	218	84	273	194	316
140	219	85	274	195	317
141	220	86	275	196	318
142	221	87	276	197	319
143	222	88	277	198	64
144	223	89	278	199	65
145	224	90	279	200	66
146	225	91	280	201	67
147	226	92	281	202	68
148	227	93	282	203	69
149	228	94	283	204	70
150	229	95	284	205	71
151	230	96	285	206	72
152	231	97	286	207	73
153	232	98	287	208	74
154	233	99	288	209	75
155	234	100	289	210	76
156	235	101	290	211	77
157	236	102	291	212	78
158	237	103	292	213	79
159	238	104	293	214	80
160	239	105	294	215	81
161	240	106	295	216	82
162	241	107	296	217	83
163	242	108	297	218	84
164	243	109	298	219	85
165	244	110	299	220	86
166	245	111	300	221	87
167	246	112	301	222	88
168	247	113	302	223	89
169	248	114	303	224	90
170	249	115	304	225	91
171	250	116	305	226	92
172	251	117	306	227	93
173	252	118	307	228	94
174	253	119	308	229	95
175	254	120	309	230	96
176	255	121	310	231	97
177	256	122	311	232	98
178	257	123	312	233	99
179	258	124	313	234	100
180	259	125	314	235	101
181	260	126	315	236	102
182	261	127	316	237	103
183	262	128	317	238	104
184	263	129	318	239	105
185	264	130	319	240	106
186	265	131	320	241	107
187	266	132	321	242	108
188	267	133	322	243	109
189	268	134	323	244	110

190	269	135	324		245	111
191	270	136	325		246	112
192	15	137	326		247	113
193	16	138	327		248	114
194	17	139	328		249	115
195	18	140	329		250	116
196	19	141	330		251	117
197	20	142	331		252	118
198	21	143	9		253	119
199	22	144	10		254	120
200	23	145	11		255	121
201	24	146	12	201	256	122
202	25	147	13	202	257	123
203	26	148	14	203	258	124
204	27	149	15	204		125
205	28	150	16	205		126
206	29	151	17	206		127
207	30	152	18	207		128
208	31	153	19	208		129
209	32	154	20	209		130
210	33	155	21	210		131
211	34	156	22	211		132
212	35	157	23	212		133
213	36	158	24	213		134
214	37	159	25	214		135
215	38	160	26	215		136
216	39	161	27	216		137
217	40	162	28	217		138
218	41	163	29	218		139
219	42	164	30	219		140
220	43	165	31	220		141
221	44	166	32	221		142
222	45	167	33	222		143
223	46	168	34	223		144
224	47	169	35	224		145
225	48	170	36	225		146
226	49	171	37	226		147
227	50	172	38	227		148
228	51	173	39	228		149
229	52	174	40	229		150
230	53	175	41	230		151
231	54	176	42	231		152
232	55	177	43	232		153
233	56	178	44	233		154
234	57	179	45	234		155
235	58	180	46	235		156
236	59	181	47	236		157
237	60	182	48	237		158
238	61	183	49	238		159
239	62	184	50	239		160
240	63	185	51	240		161
241	64	186	52	241		162
242	65	187	53	242		163
243	66	188	54	243		164
244	67	189	55	244		165
245	68	190	56	245		166
246	69	191	57	246		167

247	70	192	58	247		168
248	71	193	59	248		169
249	72	194	60	249		170
250	73	195	61	250		171
251	74	196	62	251		172
252	75	197	63	252		173
253	76	198	64	253		174
254	77	199	65	254		175
255	78	200	66	255		176
256	79	201	67	256		177
257	80	202	68	257		178
258	81	203	69	258		179
259	82	460	70	259		180
260	83	461	71	260		181
261	84	462	72	261		182
262	85	463	73	262		183
263	86	464	74	263		184
264	87	465	75	264		185
265	88	466	76	454		186
266	89	467	77	455		187
267	90	468	78	456		188
268	91	469	79	457	134	189
269	92	470	80	458	135	190
270	93	471	81	459	136	191
271	94	472	82	460	137	
272	95	473	83	461	138	
273	96	474	84	462	139	
274	97	475	85	463	140	
275	98	476	86	464	141	
276	99	477	87	465	142	
277	100	478	88	466	143	
278	101	479	89	467	144	
279	102	480	90	468	145	
280	103	481	91	469	146	
281	104	482	92	470	147	
282	105	483	93	471	148	
283	106	484	94	472	149	
284	107	485	95	473	150	
285	108	486	96	474	151	
286	109	487	97	475	152	
287	110	488	98	476	153	
288	111	489	99	477	154	
289	112	490	100	478	155	
290	113	491	101	479	156	
291	114	492	102	480	157	
292	115	493	103	481	158	
293	116	494	104	482	159	
294	117	495	105	483	160	
295	118	496	106	484	161	
296	119	497	107	485	162	
297	120	498	108	486	163	
298	121	499	109	487	164	
299	122	500	110	488	165	
300	123	501	111	489	166	
301	124	502	112	490	167	
302	125	503	113	491	168	
303	126	504	114	492	169	

304	127	505	115	493	170	
305	128	506	116	494	171	
306	129	507	117	495	172	
307	130	508	118	496	173	
308	131	509	119	497	174	
309	132	510	120	498	175	
310	133	511	121	499	176	
311	134	0	122	500	177	
312	135	1	123	501	178	
313	136	2	124	502	179	
314	137	3	125	503	180	
315	138	4	126	504	181	
316	139	5	127	505	182	
317	140	6	128	506	183	
318	141	7	129	507	184	
319	142	8	130	508	185	
320	399	9	131	509	186	
321	400	10	132	510	187	
322	401	11	133	511	188	
323	402	12	134	0	189	
324	403	13	135	1	190	
325	404	14	136	2	191	
326	405	15	393	3	192	
327	406	16	394	4	193	
328	407	17	395	5	194	
329	408	18	396	6	195	
330	409	19	397	7	196	
331	410	20	398	8	197	
332	411	21	399	9	387	
333	412	22	400	10	388	
334	413	23	401	11	389	
335	414	24	402	12	390	67
336	415	25	403	13	391	68
337	416	26	404	14	392	69
338	417	27	405	15	393	70
339	418	28	406	16	394	71
340	419	29	407	17	395	72
341	420	30	408	18	396	73
342	421	31	409	19	397	74
343	422	32	410	20	398	75
344	423	33	411	21	399	76
345	424	34	412	22	400	77
346	425	35	413	23	401	78
347	426	36	414	24	402	79
348	427	37	415	25	403	80
349	428	38	416	26	404	81
350	429	39	417	27	405	82
351	430	40	418	28	406	83
352	431	41	419	29	407	84
353	432	42	420	30	408	85
354	433	43	421	31	409	86
355	434	44	422	32	410	87
356	435	45	423	33	411	88
357	436	46	424	34	412	89
358	437	47	425	35	413	90
359	438	48	426	36	414	91
360	439	49	427	37	415	92

361	440	50	428	38	416	93	
362	441	51	429	39	417	94	
363	442	52	430	40	418	95	
364	443	53	431	41	419	96	
365	444	54	432	42	420	97	
366	445	55	433	43	421	98	
367	446	56	434	44	422	99	
368	447	57	435	45	423	100	
369	448	58	436	46	424	101	
370	449	59	437	47	425	102	
371	450	60	438	48	426	103	
372	451	61	439	49	427	104	
373	452	62	440	50	428	105	
374	453	63	441	51	429	106	
375	454	64	442	52	430	107	
376	455	65	443	53	431	108	
377	456	66	444	54	432	109	
378	457	67	445	55	433	110	
379	458	68	446	56	434	111	
380	459	69	447	57	435	112	
381	460	70	448	58	436	113	
382	461	71	449	59	437	114	
383	462	72	450	60	438	115	
384	463	73	451	61	439	116	
385	464	74	452	62	440	117	
386	465	75	453	63	441	118	
387	466	332	454	64	442	119	
388	467	333	455	65	443	120	
389	468	334	456	66	444	121	
390	469	335	457	67	445	122	
391	470	336	458	68	446	123	
392	471	337	459	69	447	124	
393	472	338	460	326	448	125	
394	473	339	461	327	449	126	
395	474	340	462	328	450	127	
396	475	341	463	329	451	128	
397	476	342	464	330	452	129	
398	477	343	465	331	453	130	
399	478	344	466	332	454	320	
400	479	345	467	333	455	321	
401	480	346	468	334	456	322	
402	481	347	469	335	457	323	0
403	482	348	470	336	458	324	1
404	483	349	471	337	459	325	2
405	484	350	472	338	460	326	3
406	485	351	473	339	461	327	4
407	486	352	474	340	462	328	5
408	487	353	475	341	463	329	6
409	488	354	476	342	464	330	7
410	489	355	477	343	465	331	8
411	490	356	478	344	466	332	9
412	491	357	479	345	467	333	10
413	492	358	480	346	468	334	11
414	493	359	481	347	469	335	12
415	494	360	482	348	470	336	13
416	495	361	483	349	471	337	14
417	496	362	484	350	472	338	15

418	497	363	485	351	473	339	16
419	498	364	486	352	474	340	17
420	499	365	487	353	475	341	18
421	500	366	488	354	476	342	19
422	501	367	489	355	477	343	20
423	502	368	490	356	478	344	21
424	503	369	491	357	479	345	22
425	504	370	492	358	480	346	23
426	505	371	493	359	481	347	24
427	506	372	494	360	482	348	25
428	507	373	495	361	483	349	26
429	508	374	496	362	484	350	27
430	509	375	497	363	485	351	28
431	510	376	498	364	486	352	29
432	511	377	499	365	487	353	30
433	0	378	500	366	488	354	31
434	1	379	501	367	489	355	32
435	2	380	502	368	490	356	33
436	3	381	503	369	491	357	34
437	4	382	504	370	492	358	35
438	5	383	505	371	493	359	36
439	6	384	506	372	494	360	37
440	7	385	507	373	495	361	38
441	8	386	508	374	496	362	39
442	9	387	509	375	497	363	40
443	10	388	510	376	498	364	41
444	11	389	511	377	499	365	42
445	12	390	0	378	500	366	43
446	13	391	1	379	501	367	44
447	14	392	2	380	502	368	45
448		393	3	381	503	369	46
449		394	4	382	504	370	47
450		395	5	383	505	371	48
451		396	6	384	506	372	49
452		397	7	385	507	373	50
453		398	8	386	508	374	51
454		399	265	387	509	375	52
455		400	266	388	510	376	53
456		401	267	389	511	377	54
457		402	268	390	0	378	55
458		403	269	391	1	379	56
459		404	270	392	2	380	57
460		405	271	393	259	381	58
461		406	272	394	260	382	59
462		407	273	395	261	383	60
463		408	274	396	262	384	61
464		409	275	397	263	385	62

465	410	276	398	264	386	63	
466	411	277	399	265	387		
467	412	278	400	266	388		
468	413	279	401	267	389		
469	414	280	402	268	390		
470	415	281	403	269	391		
471	416	282	404	270	392		
472	417	283	405	271	393		
473	418	284	406	272	394		
474	419	285	407	273	395		
475	420	286	408	274	396		
476	421	287	409	275	397		
477	422	288	410	276	398		
478	423	289	411	277	399		
479	424	290	412	278	400		
480	425	291	413	279	401		
481	426	292	414	280	402		
482	427	293	415	281	403		
483	428	294	416	282	404		
484	429	295	417	283	405		
485	430	296	418	284	406		
486	431	297	419	285	407		
487	432	298	420	286	408		
488	433	299	421	287	409		
489	434	300	422	288	410		
490	435	301	423	289	411		
491	436	302	424	290	412		
492	437	303	425	291	413		
493	438	304	426	292	414		
494	439	305	427	293	415		
495	440	306	428	294	416		
496	441	307	429	295	417		
497	442	308	430	296	418		
498	443	309	431	297	419		
499	444	310	432	298	420		
500	445	311	433	299	421		
501	446	312	434	300	422		
502	447	313	435	301	423		
503	448	314	436	302	424		
504	449	315	437	303	425		
505	450	316	438	304	426		
506	451	317	439	305	427		
507	452	318	440	306	428		
508	453	319	441	307	429		
509	454	320	442	308	430		
510	455	321	443	309	431		
511	456	322	444	310	432		

Fig: 6.4 Relaciones entre bits que se forman en el grafo

Esta lista de adyacencias se ha obtenido del siguiente modo: cada columna indica las relaciones que crea cada una de las rotaciones que se practican en la clave, por ejemplo: la primera columna de relaciones (con fondo blanco) indica las relaciones entre los bits de las primeras 14

subclaves y sus inversas, según la estructura. Las casillas en blanco son los bits que no se han utilizado en esas 14 primeras subclaves.

A partir de estos datos es fácil aplicar el algoritmo descrito, sin embargo lo modificaremos para hacerlo más intuitivo: vamos a crear una tabla con dos partes iguales, en cada una de ellas insertamos todos los bits. Empezamos por uno al azar, en este caso será el 0, y lo asignamos a la izquierda sombreándolo, y todos los nodos de su lista de relaciones irán relacionados a al otro lado. Cada nodo es apuntado en una lista, cuando acabemos con el 0, iremos al siguiente nodo de esa lista:

0	64	128	192	256	320	384	448
1	65	129	193	257	321	385	449
2	66	130	194	258	322	386	450
3	67	131	195	259	323	387	451
4	68	132	196	260	324	388	452
5	69	133	197	261	325	389	453
6	70	134	198	262	326	390	454
7	71	135	199	263	327	391	455
8	72	136	200	264	328	392	456
9	73	137	201	265	329	393	457
10	74	138	202	266	330	394	458
11	75	139	203	267	331	395	459
12	76	140	204	268	332	396	460
13	77	141	205	269	333	397	461
14	78	142	206	270	334	398	462
15	79	143	207	271	335	399	463
16	80	144	208	272	336	400	464
17	81	145	209	273	337	401	465
18	82	146	210	274	338	402	466
19	83	147	211	275	339	403	467
20	84	148	212	276	340	404	468
21	85	149	213	277	341	405	469
22	86	150	214	278	342	406	470
23	87	151	215	279	343	407	471
24	88	152	216	280	344	408	472
25	89	153	217	281	345	409	473
26	90	154	218	282	346	410	474
27	91	155	219	283	347	411	475
28	92	156	220	284	348	412	476
29	93	157	221	285	349	413	477
30	94	158	222	286	350	414	478
31	95	159	223	287	351	415	479
32	96	160	224	288	352	416	480
33	97	161	225	289	353	417	481
34	98	162	226	290	354	418	482
35	99	163	227	291	355	419	483
36	100	164	228	292	356	420	484
37	101	165	229	293	357	421	485
38	102	166	230	294	358	422	486
39	103	167	231	295	359	423	487
40	104	168	232	296	360	424	488
41	105	169	233	297	361	425	489
42	106	170	234	298	362	426	490
43	107	171	235	299	363	427	491
44	108	172	236	300	364	428	492
45	109	173	237	301	365	429	493
46	110	174	238	302	366	430	494
47	111	175	239	303	367	431	495
48	112	176	240	304	368	432	496
49	113	177	241	305	369	433	497
50	114	178	242	306	370	434	498
51	115	179	243	307	371	435	499
52	116	180	244	308	372	436	500
53	117	181	245	309	373	437	501
54	118	182	246	310	374	438	502
55	119	183	247	311	375	439	503
56	120	184	248	312	376	440	504
57	121	185	249	313	377	441	505
58	122	186	250	314	378	442	506
59	123	187	251	315	379	443	507
60	124	188	252	316	380	444	508
61	125	189	253	317	381	445	509
62	126	190	254	318	382	446	510
63	127	191	255	319	383	447	511

En esta tabla se ve que hemos procesado el nodo 0, habiendo obtenido las relaciones con 402, 457, 323, 445, 311 y 433. Ahora sería necesario procesar el elemento 402 cuyas relaciones (que irían marcadas a la izq.) serían: 481, 347, 469, 335, 457, 323 y 0.

Es importante observar que el 457 y el 323, que ya están marcados en la parte derecha, habría que volver a marcarlos en la parte izquierda: y cuando se marca al menos un nodo en las dos partes, según el algoritmo, se puede afirmar que el grafo no es bipartido.

Fig: 6.5 Algoritmo para detectar si el grafo es bipartido

7.3 Filtros

Para evitar que ciertas claves puedan producir codificaciones débiles, como una clave en la que casi todos sus bits son 0, se utilizarán filtros de forma simétrica, con el fin de que no sea necesario cambiar el orden de los filtros al decodificar.

7.4 Detalle de la distribución de claves

Cifrado	
K_1	K_2
K_3, K_4	K_5, K_6
K_7, K_8	K_9, K_{10}
K_{11}, K_{12}	K_{13}, K_{14}
K_{15}	K_{16}
K_{17}, K_{18}	K_{19}, K_{20}
K_{21}, K_{22}	K_{23}, K_{24}
K_{25}, K_{26}	K_{27}, K_{28}
K_{29}	K_{30}
K_{31}, K_{32}	K_{33}, K_{34}
K_{35}, K_{36}	K_{37}, K_{38}
K_{39}, K_{40}	K_{41}, K_{42}
K_{43}	K_{44}
K_{45}, K_{46}	K_{47}, K_{48}
K_{49}, K_{50}	K_{51}, K_{52}
K_{53}, K_{54}	K_{55}, K_{56}
K_{57}	K_{58}
K_{59}, K_{60}	K_{61}, K_{62}
K_{63}, K_{64}	K_{65}, K_{66}
K_{67}, K_{68}	K_{69}, K_{70}
K_{71}	K_{72}
K_{73}, K_{74}	K_{75}, K_{76}
K_{77}, K_{78}	K_{79}, K_{80}
K_{81}, K_{82}	K_{83}, K_{84}
K_{85}	K_{86}

Descifrado	
K_{85}	K_{86}
K_{81}, K_{82}	K_{83}, K_{84}
K_{77}, K_{78}	K_{79}, K_{80}
K_{73}, K_{74}	K_{75}, K_{76}
K_{71}	K_{72}
K_{67}, K_{68}	K_{69}, K_{70}
K_{63}, K_{64}	K_{65}, K_{66}
K_{59}, K_{60}	K_{61}, K_{62}
K_{57}	K_{58}
K_{53}, K_{54}	K_{55}, K_{56}
K_{49}, K_{50}	K_{51}, K_{52}
K_{45}, K_{46}	K_{47}, K_{48}
K_{43}	K_{44}
K_{39}, K_{40}	K_{41}, K_{42}
K_{35}, K_{36}	K_{37}, K_{38}
K_{31}, K_{32}	K_{33}, K_{34}
K_{29}	K_{30}
K_{25}, K_{26}	K_{27}, K_{28}
K_{21}, K_{22}	K_{23}, K_{24}
K_{17}, K_{18}	K_{19}, K_{20}
K_{15}	K_{16}
K_{11}, K_{12}	K_{13}, K_{14}
K_7, K_8	K_9, K_{10}
K_3, K_4	K_5, K_6
K_1	K_2

Este es el orden con el que deben utilizarse las claves para poder cifrar y descifrar con el mismo algoritmo. Los bloques pequeños representan a los bloques “B” y los grandes a los “A”.

Fig: 6.6 Detalles de las subclaves al encriptar y desencriptar

7.5 Detalle de los filtros

C56359C6	C56359C6
C56359C6	C56359C6
C56359C6	C56359C6
C56359C6	C56359C6
A9569C63	A9569C63
A9569C63	A9569C63
A9569C63	A9569C63
A9569C63	A9569C63
6A95A53C	6A95A53C
6A95A53C	6A95A53C
6A95A53C	6A95A53C
6A95A53C	6A95A53C
5A6C3559	5A6C3559
6A95A53C	6A95A53C
6A95A53C	6A95A53C
6A95A53C	6A95A53C
6A95A53C	6A95A53C
A9569C63	A9569C63
A9569C63	A9569C63
A9569C63	A9569C63
A9569C63	A9569C63
C56359C6	C56359C6
C56359C6	C56359C6
C56359C6	C56359C6
C56359C6	C56359C6

Fig: 6.7 Detalles de los filtros

Estos son los filtros concretos que serán aplicados a cada una de las subclaves que se obtengan. Su misión es la que ya ha sido comentada, evitar codificaciones débiles, y se utiliza la operación XOR para aplicarlos, una subclave K_n pasa a ser $K_n = K_n$ **xor** F_n.

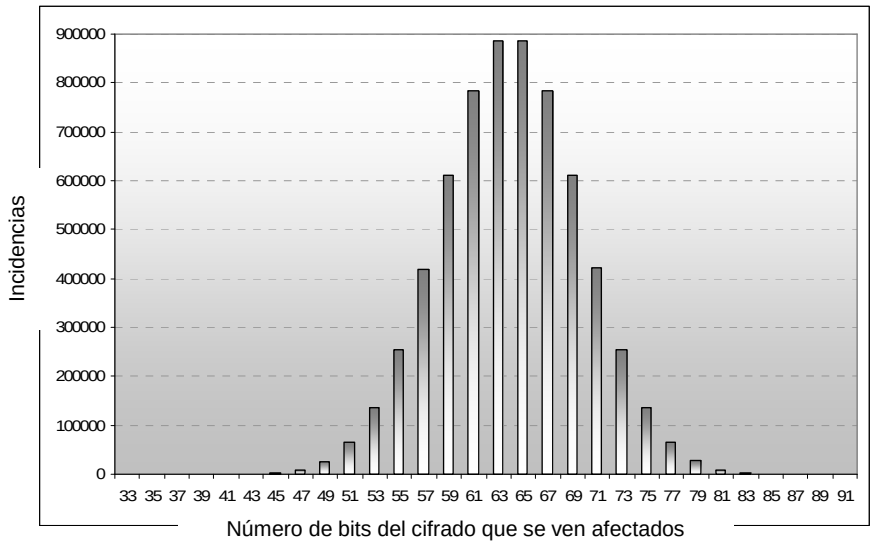
Todos ellos tienen la particularidad de que en una subclave en la que todos los bits sean iguales, se obtendrán subclaves con un 50% de ceros y un 50% de unos. Es posible que una clave sea capaz de saltarse el filtro en una de las rondas, y dar para esa ronda unas subclaves malas, sin embargo no podrá hacerlo con todos los filtros, y es ahí donde reside la fortaleza de esta forma tan particular de tratar a las subclaves.

7.6 Análisis realizados

Primer análisis: el primer análisis realizado es el siguiente: se cifra un texto con una clave, y posteriormente se cifra el mismo texto cambiando uno sólo de los bits y con la misma clave. El resultado del cifrado del texto modificado se compara con el resultado cifrado del texto original, contando el número de bits diferentes entre ambos. El resultado ideal es que con variar un solo bit de la entrada cambien el 50% de la salida. El test se realiza para cada uno de los bits del mismo texto.

Se ha aplicado la prueba mencionada utilizando 50.000 textos al azar de 128 bits se longitud, y 50.000 claves al azar, una para cada texto. A cada uno de los textos se le han aplicado 128 comparaciones, con un total de 6.400.000 comparaciones diferentes.

Fig. 6.9 Representación gráfica de las diferencias



Nº de diferencias	Nº de casos
33	1
35	1
37	7
39	40
41	201
43	866
45	3082
47	9779
49	26646
51	64543
53	136868
55	255914
57	420234
59	611020
61	782772
63	887223
65	886812
67	782840
69	611440
71	421873
73	255643
75	136839
77	64495
79	26905
81	9715
83	3151
85	831
87	211
89	41
91	7

Se ha comprobado experimentalmente que el número de diferencias variando uno solo de los bits de la entrada sigue una distribución normal. La media poblacional ha sido de 64,0025975 diferencias, la varianza de 32,0028233 y la desviación típica de 5,65710379. Puede considerarse que la media es de 64 y la varianza 32, unos valores óptimos.

Fig: 6.8 Tabla con las veces que se ha dado ese número de diferencias

Resultados de distintas ejecuciones de la misma prueba:

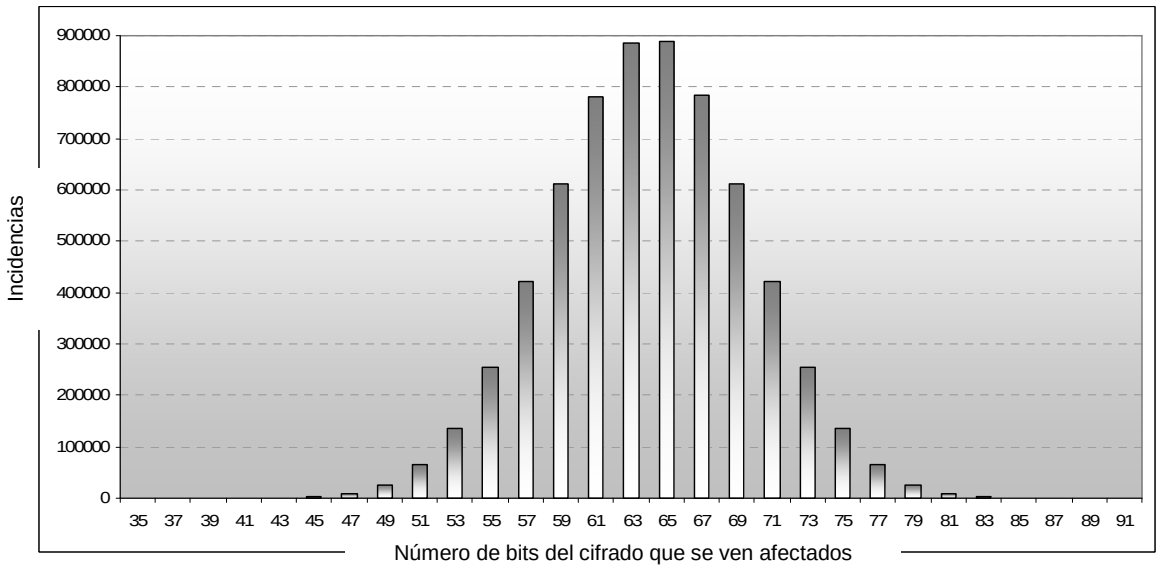


Fig: 6.10

Media 64,0012388, varianza 31,980722

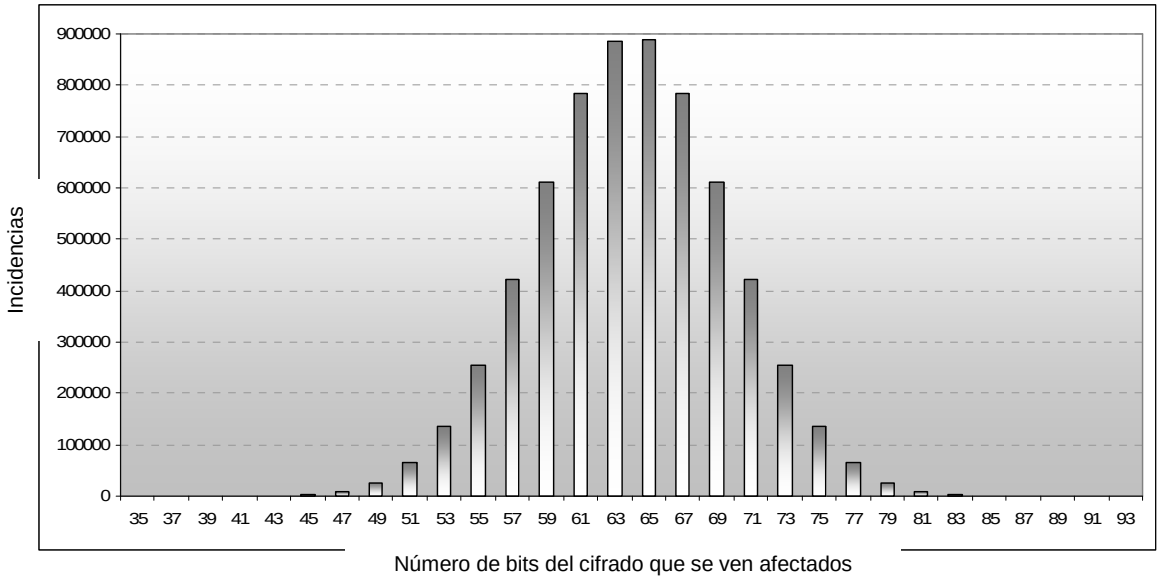


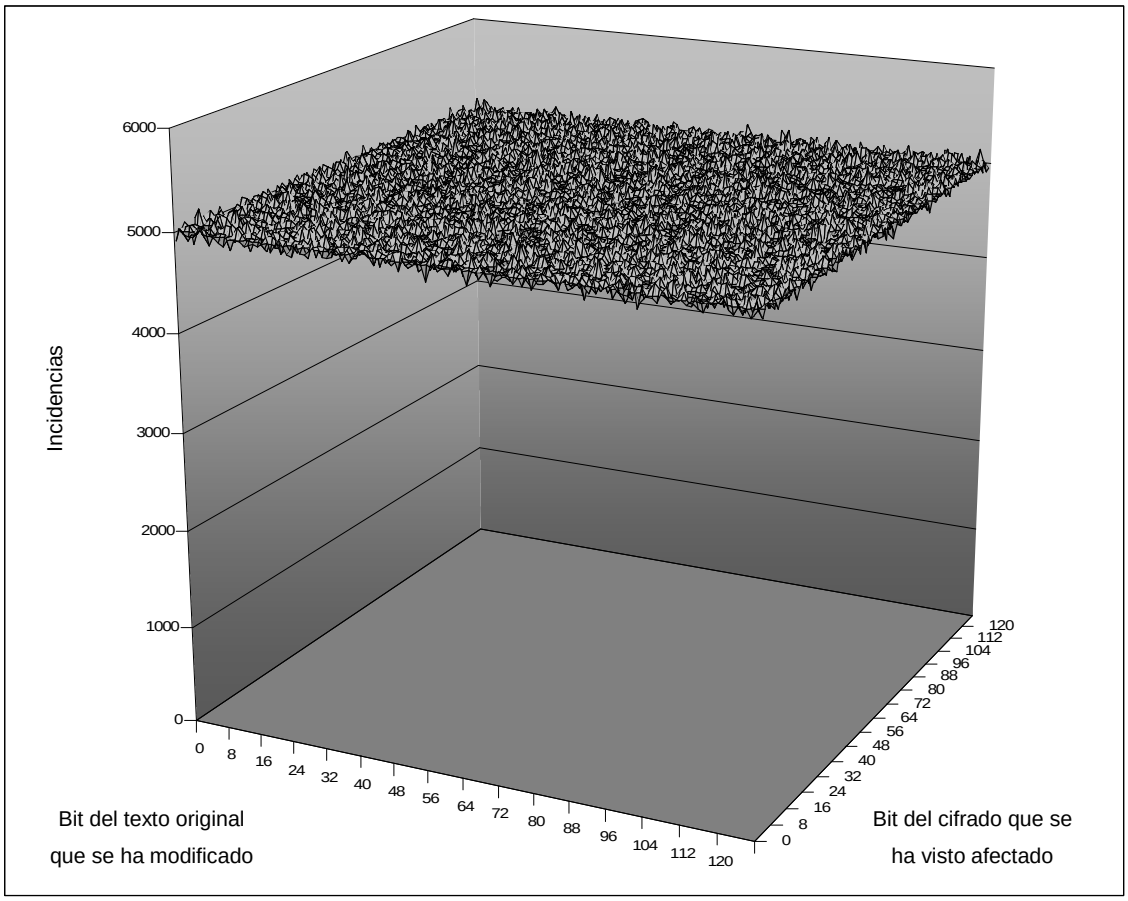
Fig: 6.11

Media 64,0025744, varianza 31,9702458

Segundo análisis: La segunda prueba trata de comprobar que el análisis diferencial no supone una amenaza seria. Esta prueba consiste en modificar el mismo bit en multitud de textos diferentes con claves de cifrado diferentes y ver si afecta más a unos bits de la salida que a otros. La solución óptima es que afecte a cada uno alrededor de un 50%, obteniendo una distribución plana.

El test se ha realizado modificando uno de los bits en 10.000 textos diferentes con 10.000 claves diferentes, y esto para todos los bits, analizando qué bits resultan afectados. Resultando un total de 1.280.000 comparaciones. Los resultados, como podemos ver, son excepcionales y el algoritmo no resulta débil frente a un ataque por criptoanálisis diferencial.

Fig: 6.12 Análisis de la frecuencia con la que un bit del cifrado es modificado al modificar cada bit del origen

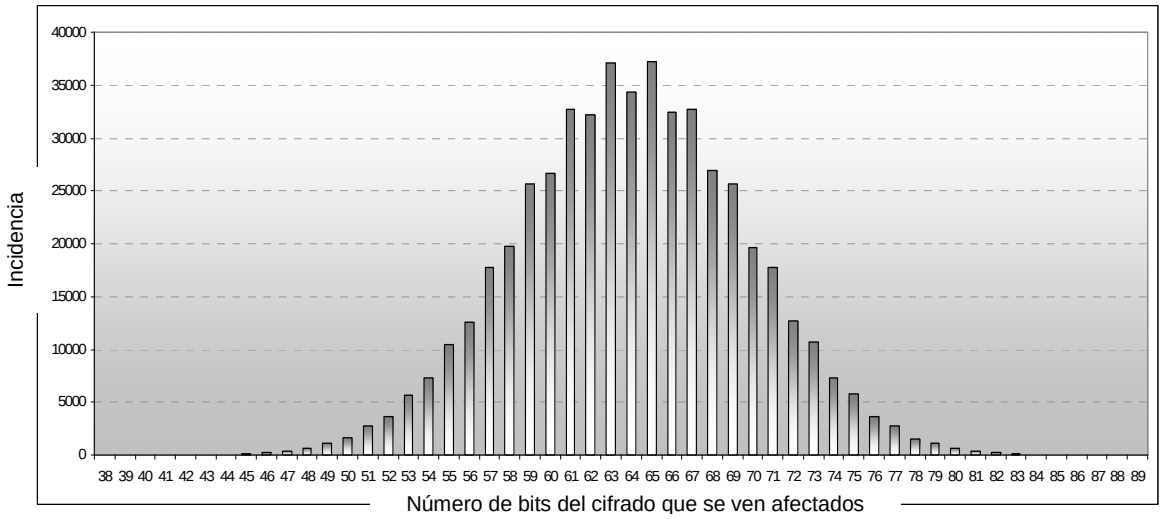


El eje X indica el bit modificado antes de cifrar, el eje Y indica los bits a los que ha afectado esa modificación después de cifrar, el eje Z indica cuantas veces ha sido modificado el bit Y al modificar el bit X.

La media obtenida es de 4999,72351, la desviación típica es de 49,8778363.

Tercer análisis: al igual que con los análisis basados en modificaciones del texto, se pueden hacer análisis basados en modificaciones de la clave. Este tercer análisis ha consistido en codificar un texto con una clave y después codificarlo con una clave idéntica en todos los bits salvo uno. Se han medido las diferencias producidas en los bits del resultado. Para efectuar este análisis se han escogido al azar 1000 textos y 1000 claves, para cada clave se han comprobado las diferencias que se producían entre el cifrado que producen y el cifrado que se produce alterando cada uno de sus bits, en total 5.120.000 comparaciones. Si se produjese el mismo caso que en análisis similar en el texto plano, en el que solamente se producían diferencias impares, nos encontraríamos frente a un problema de seguridad: conocidos un texto plano y su cifrado podríamos averiguar si el número de bits "1" es par o impar en la clave. Los resultados han sido los siguientes:

Fig: 6.13 Análisis del número de diferencias producidas al modificar un bit de la clave



Esta vez no se distribuye como una normal pero se acerca. Las diferencias impares tienden a ser más frecuentes que las pares, de hecho un 47'66% son pares y un 52'34% son impares. Sin embargo la distribución tiende a una media de 64 y una varianza de 32, en concreto 64,0155469 diferencias como promedio y 31,987563 de varianza, los valores son similares a los obtenidos en el otro análisis. De este resultado se concluye que aunque las diferencias impares sean ligeramente más frecuentes que las pares, según los datos esto no constituye una vulnerabilidad.

Cuarto análisis: este análisis es complementario al anterior y determina si realmente existe relación entre bits de la clave y bits codificados. La idea es determinar si un bit de una posición dada afecta más a unos bits del resultado que a otros.

Para ello se ha procedido del siguiente modo: se elige un texto y una clave, y se cifra. Después ciframos ese texto con una clave en la que varíe el bit de una determinada posición y comparamos con el primer cifrado. Según el análisis anterior se producen alrededor de la mitad de variaciones, pero ahora se trata de analizar el impacto sobre cada bit del cifrado.

El resultado ha sido el siguiente:

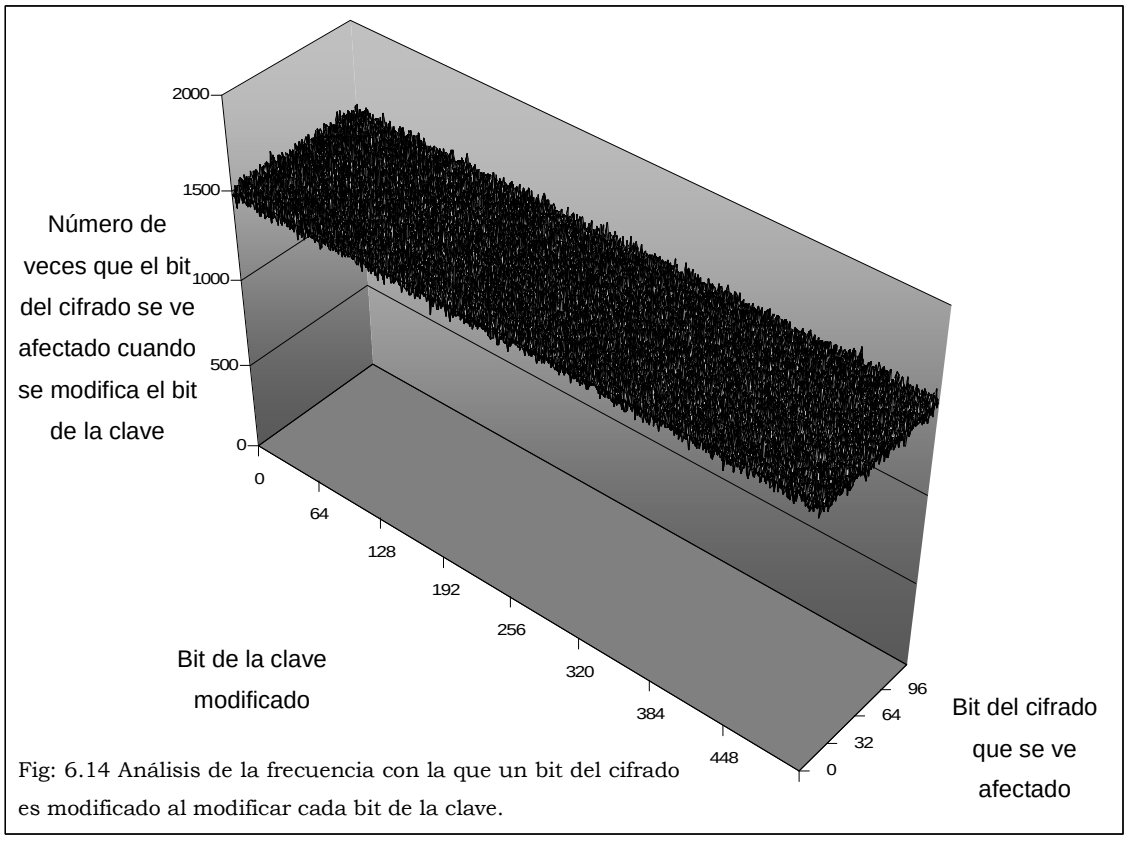


Fig: 6.14 Análisis de la frecuencia con la que un bit del cifrado es modificado al modificar cada bit de la clave.

Como se ve, el variar cualquier bit de la clave afecta con una probabilidad de un 50% a todos los bits del texto cifrado, el promedio de cambios ha sido de 1500,04114 de los 3000 producidos para cada bit de la clave, la desviación típica ha sido de 27,3628357 cambios. Con estos datos podemos afirmar que no existen relaciones estadísticas entre los bits de la clave y los bits que resultan afectados.

7.7 Codificación, esquema de relleno

El algoritmo trabaja con bloques de 128 bits (16 bytes) y todos los datos que se deseen codificar deben ser múltiplos de esta cantidad. Existen ciertas aplicaciones que no permiten tratar con un tamaño tan estricto en los datos, un ejemplo muy típico son los archivos. Un archivo puede tener cualquier longitud en bytes, para poder utilizar el criptosistema propuesto hay que lograr que su longitud sea múltiplo de 16 de forma que sea posible recuperar siempre el contenido original del mismo. Es lo que se conoce como esquema de relleno.

Para ello se utiliza una técnica de expansión: un archivo será expandido hasta rellenar lo que le falte para tener una longitud múltiplo de 16, salvo casos excepcionales. Se ha elegido la siguiente técnica por varios motivos:

- Decodificación efectiva: todos los datos codificados recuperarán su longitud original después del decodificado.
- Bajo impacto: permite aplicarla solamente al último de los bloques, permitiendo optimizaciones para varias aplicaciones.
- Extensión mínima: los datos han de experimentar la menor extensión posible, garantizando siempre la seguridad.
- Ausencia de debilidades en concepto de seguridad: se debe evitar que un atacante pueda conocer información útil de un bloque para poder llevar a cabo un ataque.

Se han evaluado varias posibilidades y se ha decidido actuar de la siguiente forma:

- El origen ha de ser extendido hasta una cantidad múltiplo de 16 bytes.
- Si no se alcanza esta cantidad se rellena con el byte 0xff, excepto cuando el último byte de la información sea 0xff, en cuyo caso se rellena con el byte 0x00.

- Si el origen tuviese ya una longitud múltiplo de 16 bytes, si no acaba en 0x00, 0x01, o 0xff no se le añadirá nada. Si acaba en uno de estos bytes se añadirá un bloque de información aleatoria que contendrá el byte 0x01 al final de forma obligatoria y representa la marca de que todo el bloque es un añadido.

La decodificación se realizará de la siguiente manera:

- Si el último byte es 0x01 se desechan los últimos 16 bytes.
- Si el último byte es 0xff se desechan los bytes por el final que sean 0xff hasta encontrar el primero que no lo sea.
- Si el último byte es 0x00 se desechan los bytes por el final que sean 0x00 hasta encontrar el primero que no lo sea.

De estar forma se persiguen varios objetivos entre los que destaca la sencillez y la localidad de la modificación, además del aumento mínimo del tamaño que será de 16 bytes como máximo en casos muy particulares.

La inclusión del byte 0x01 como byte de control se justifica por lo siguiente: con dos caracteres de control ya se podría codificar de una forma muy sencilla, sin embargo si ciertos datos tuviesen una longitud múltiplo de 16 y acabasen con uno de esos caracteres de control, se rellenaría un nuevo bloque con repeticiones del otro. Esto puede ser un problema de seguridad pues tener conocimiento de que un mensaje ha sido aumentado en 16 bytes, y obtener el cifrado, puede poner en marcha un ataque por fuerza bruta para localizar la clave. La codificación mostrada asegura que nunca podremos conocer el 100% del contenido del último bloque, asegurando su resistencia frente un ataque utilizando fuerza bruta.

8 Implementación del algoritmo (NICS)

Para la implementación del algoritmo se ha utilizado el lenguaje JAVA 1.5 por varios motivos:

- Es un lenguaje sencillo y muy flexible.
- Es multiplataforma: este algoritmo puede funcionar en cualquier dispositivo para el que exista una máquina virtual (MV) de Java.

Sin embargo el adoptar esta solución conlleva una desventaja a tener en cuenta. Los programas en JAVA no pueden igualar en velocidad a otros lenguajes como C.
Sopesando ventajas y desventajas, y considerando que el objetivo principal es conseguir un prototipo funcional para el mayor número posible de plataformas, el lenguaje seleccionado se vuelve ideal. En caso de necesitar una solución más rápida y optimizada se puede llevar a cabo la programación a un nivel mucho más bajo, incluso ensamblador para la rutina principal.

A pesar de la inherente lentitud del lenguaje utilizado, se ha hecho especial esfuerzo en la velocidad, para no sacrificarla aún más, y obtener de ese modo un sistema al que realmente se le pueda dar uso.

Se ha procedido a programar una librería con el criptosistema, para poder utilizarla de forma general, y posteriormente un programa de ejemplo que la utilice. La funcionalidad del mismo es la de cifrar y descifrar archivos.

8.1 Optimizaciones aplicadas

La librería consta de dos clases fundamentales: una representa al criptosistema propiamente dicho y otra a la clave. Cuando se crea un objeto *Key* las subclaves son calculadas una sola vez y almacenadas en memoria para todos los usos futuros. Debido a la especial estructura del algoritmo, emplear la clave para cifrar o descifrar no supone más

que un reordenamiento de las subclaves ya calculadas, con un impacto mínimo en la velocidad. Un objeto NICS (cifrador propiamente dicho) es capaz de realizar dos tipos de cifrado:

- Cifrado de un fragmento de información (array de bytes) cuya longitud sea múltiplo de 16 bytes. Se encuentra altamente optimizado: se utilizan 5 variables de tipo entero para realizar los cálculos (expresados como operaciones a nivel de bit) y se trabaja sobre el propio array de entrada, para evitar el consumo de tiempo y recursos que supone la copia de los datos.
- Cifrado de un fragmento de información (array de bytes) cuya longitud puede ser cualquiera. En este caso se aplica el esquema de relleno mencionado en la descripción del algoritmo. Es un método mucho más flexible pero que conlleva la copia de los datos a un nuevo array, con el consiguiente gasto de recursos.

Sin embargo ambos métodos son compatibles, el primero proporciona velocidad y el segundo flexibilidad, y se pueden combinar para obtener lo mejor de ambos. El procedimiento es el siguiente:

- Dividir los datos iniciales en dos fragmentos: uno cuya longitud sea la máxima posible siendo múltiplo de 16 bits, otro con el resto, el segundo fragmento debe tener al menos 16 bits (si es posible, a pesar de que la longitud del primero resulte de 0 bits).
- El primer fragmento se procesa con el método rápido y el último con el flexible para el cifrado, si la cantidad de información es muy grande, el primer fragmento puede dividirse en otros más pequeños.

El hecho de reservar 16 bytes al menos para el segundo fragmento se hace para que al descifrar puedan descartarse esos 16 bytes cuando se emplea la máxima expansión en el esquema de relleno. La combinación de ambos métodos puede aumentar espectacularmente la velocidad.

En un principio se planteó la posibilidad de utilizar archivos mapeados en memoria para agilizar las operaciones de entrada y salida, pero después de comprobar el rendimiento obtenido se ha optado por el procedimiento estándar para la entrada y salida.

8.2 Código

No se va a incluir todo el código de la aplicación pues no es el cometido del trabajo. Únicamente se va a mostrar el corazón del algoritmo, las líneas más importantes de todas y las que llevan a cabo realmente la codificación y decodificación de los datos. Es una función que recibe un array de 4 registros de 32 bits y los modifica adecuadamente según se ha visto, el posible bucle se ha deshecho para aumentar la velocidad, además se expresa todo con operaciones a nivel de bit para, si fuese necesario, sirviese de guía en una posible implementación el lenguaje ensamblador. Las subclaves se suponen que ya están listas para su utilización en un array, y solamente se hace uso de un registro auxiliar. El código es el siguiente:

Fig. 7.1 Código fuente del núcleo del algoritmo, programado en JAVA

```
    private void procesarRegistro(int[] registro){
        int x;
        ////// BLOQUE B-1 //////
        x = (registro[0] ^ registro[1]) | key.subclave[0];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[1];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-1 //////
        x = registro[1] ^ key.subclave[2];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[3];
        x = registro[2] ^ key.subclave[4];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[5];
        x = registro[0] ^ key.subclave[6];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[7];
        x = registro[3] ^ key.subclave[8];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[9];
        x = registro[1] ^ key.subclave[10];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[11];
        x = registro[2] ^ key.subclave[12];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[13];
        ////// BLOQUE B-2 //////
        x = (registro[0] ^ registro[1]) | key.subclave[14];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[15];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-2 //////
```

```
        x = registro[1] ^ key.subclave[16];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[17];
        x = registro[2] ^ key.subclave[18];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[19];
        x = registro[0] ^ key.subclave[20];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[21];
        x = registro[3] ^ key.subclave[22];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[23];
        x = registro[1] ^ key.subclave[24];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[25];
        x = registro[2] ^ key.subclave[26];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[27];
        ////// BLOQUE B-3 //////
        x = (registro[0] ^ registro[1]) | key.subclave[28];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[29];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-3 //////
        x = registro[1] ^ key.subclave[30];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[31];
        x = registro[2] ^ key.subclave[32];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[33];
        x = registro[0] ^ key.subclave[34];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[35];
        x = registro[3] ^ key.subclave[36];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[37];
        x = registro[1] ^ key.subclave[38];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[39];
        x = registro[2] ^ key.subclave[40];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[41];
        ////// BLOQUE B-4 //////
        x = (registro[0] ^ registro[1]) | key.subclave[42];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[43];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-4 //////
        x = registro[1] ^ key.subclave[44];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[45];
        x = registro[2] ^ key.subclave[46];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[47];
        x = registro[0] ^ key.subclave[48];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[49];
        x = registro[3] ^ key.subclave[50];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[51];
        x = registro[1] ^ key.subclave[52];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[53];
        x = registro[2] ^ key.subclave[54];
```

```
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[55];
        ////// BLOQUE B-5 //////
        x = (registro[0] ^ registro[1]) | key.subclave[56];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[57];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-5 //////
        x = registro[1] ^ key.subclave[58];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[59];
        x = registro[2] ^ key.subclave[60];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[61];
        x = registro[0] ^ key.subclave[62];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[63];
        x = registro[3] ^ key.subclave[64];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[65];
        x = registro[1] ^ key.subclave[66];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[67];
        x = registro[2] ^ key.subclave[68];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[69];
        ////// BLOQUE B-6 //////
        x = (registro[0] ^ registro[1]) | key.subclave[70];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[71];
        registro[2] ^= x;
        registro[3] ^= x;
        ////// BLOQUE A-6 //////
        x = registro[1] ^ key.subclave[72];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[73];
        x = registro[2] ^ key.subclave[74];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[75];
        x = registro[0] ^ key.subclave[76];
        registro[2] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[77];
        x = registro[3] ^ key.subclave[78];
        registro[1] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[79];
        x = registro[1] ^ key.subclave[80];
        registro[0] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[81];
        x = registro[2] ^ key.subclave[82];
        registro[3] ^= ((x << (x & 0x1F)) | (x >>> 32-(x & 0x1F))) ^ key.subclave[83];
        ////// BLOQUE B-7 //////
        x = (registro[0] ^ registro[1]) | key.subclave[84];
        registro[0] ^= x;
        registro[1] ^= x;
        x = (registro[2] ^ registro[3]) | key.subclave[85];
        registro[2] ^= x;
        registro[3] ^= x;
    }
```

8.3 Ejecución

El programa de ejemplo (NICS.jar) se ejecuta en la consola. La sintaxis es la siguiente:

```
java –jar NICS.jar <archivo_clave> <archivo_a_procesar>
```

Donde `<archivo_clave>` es un archivo que contiene la clave para cifrar/descifrar. Su contenido debe ser una ristra de 128 caracteres hexadecimales, sin saltos de línea. A modo de ejemplo se incluye el archivo `key.key`.

Por otro lado `<archivo_a_procesar>` es el archivo que queremos cifrar o descifrar. Cuando un archivo se cifra, el resultado se almacena en un archivo cuyo nombre es el nombre y la extensión del archivo original, y la extensión es ".nics"; cuando se descifra, se restaura el nombre original. Por ejemplo `datos_bancarios.dat` se cifraría como `datos_bancarios.dat.nics`.

Si un archivo ya existiera, ya sea al cifrar o al descifrar, se avisaría al usuario para que confirme si desea sobrescribir el contenido del mismo. Todos los errores que pudieran aparecer se avisan a través de un mensaje que detalla lo que haya podido ocurrir.

8.4 Resultados obtenidos

Las pruebas se han realizado en un equipo dotado de un procesador Intel Pentium IV, 512 Mb de RAM DDR, HD de 60Gb a 7200 rpm y bus a 800Mhz.

Se han hecho pruebas con un archivo binario de 732.934.144 bytes utilizando `hprof` para obtener estadísticas de utilización de métodos dentro del programa. La línea de ejecución ha sido

```
java –Xrunhprof:cpu=samples –jar NICS.jar key.key prueba.avi
```

Los resultados obtenidos son los siguientes:

Tiempo total	Porcentaje	Tiempo estimado	Velocidad estimada
57360 ms	26,77%	15,36 seg	364,14 Mbits/seg
60516 ms	25,93%	15,69 seg	356,35 Mbits/seg
56625 ms	28,99%	16,41 seg	340,64 Mbits/seg
58594 ms	27,58%	16,16 seg	346,03 Mbits/seg
58234 ms	26,41%	15,38 seg	363,57 Mbits/seg
56453 ms	27,56%	15,56 seg	359,41 Mbits/seg
61156 ms	26,36%	16,12 seg	346,87 Mbits/seg
56203 ms	27,56%	15,49 seg	361,01 Mbits/seg
55031 ms	28,21%	15,52 seg	360,20 Mbits/seg
56265 ms	26,69%	15,02 seg	372,36 Mbits/seg

Fig. 7.2 Resultados obtenidos

La primera columna muestra el tiempo que ha tardado el programa en procesar el archivo, medido en milisegundos. La segunda columna muestra el porcentaje de procesador que se ha utilizado para el algoritmo, es de destacar que entre un 67% y un 69% se utiliza para las operaciones de entrada y salida, en concreto, para la lectura de archivos en un array de bytes y que no puede optimizarse más, ya que es la máquina virtual de java la encargada de llevarlo a cabo.

La columna del tiempo estimado representa el tiempo que aproximado que tardaría el algoritmo en procesar esa cantidad de datos sin el lastre que suponen todas las operaciones de entrada y salida, se obtiene multiplicando los valores de las dos primeras columnas en segundos.

Por último, la columna de velocidad estimada indica la velocidad aproximada del algoritmo en esa ejecución. Se obtiene dividiendo el tamaño del archivo entre el tiempo estimado, obteniendo la velocidad que se alcanzaría si olvidásemos la entrada/salida. Está expresado en megabits/segundos.

A la vista de los resultados, la velocidad media se sitúa alrededor de 357’058 Megabits/segundo.

Es importante señalar que del porcentaje de tiempo empleado en el algoritmo propiamente dicho, la mayoría es empleado en la recogida de los datos del array de bytes y en depositarlos nuevamente en su sitio, lo que hace entrever que la velocidad de la rutina de codificación es mucho mayor, pero es complicado afinar hasta tal punto y obtener resultados fiables.

9 Conclusiones

El proceso mostrado no ha sido más que los razonamientos que he seguido para llegar a diseñar y construir un algoritmo de encriptación bastante aceptable y con los objetivos iniciales.

Un algoritmo de encriptación simétrico consta de las tres partes que se tratan en la primera sección de este documento: una estructura, unas funciones y un método de obtención de subclaves. Cada una de esas partes tiene unas funciones muy determinadas y que aportan unas características propias al algoritmo. No debemos olvidar tampoco la sinergia entre esas partes, principalmente el estrecho vínculo que une la estructura con las funciones: elegir ambas de una forma lógica consigue que se complementen y den como resultado un nivel de seguridad mucho mayor.

Con estos principios se gestó el NICS (Nuevas Ideas en Criptografía Simétrica). Se ha buscado velocidad, sencillez, auto-reversibilidad y, como no, seguridad. Para su diseño se ha hecho uso de la construcción mediante eslabones propuesta en el apartado que trata la estructura, utilizando dos módulos simples: uno para añadir principalmente confusión y otro para añadir principalmente difusión. Respecto a las funciones, se han utilizado las ideas de utilización de varias subclaves, la de permitir a los propios datos que afecten al cifrado, y el empleo de operaciones de simetría aparente que añaden difusión a nivel de bit utilizando operaciones a nivel de registro. En el método utilizado para obtener las subclaves se hace uso de las ideas en cuanto a número de veces que un simple bit es seleccionado, la utilización de filtros para evitar codificaciones débiles y la aplicación del algoritmo mostrado para comprobar que efectivamente carece de claves semidébiles y débiles (excepto las triviales que no pueden evitarse).

La velocidad, a pesar de haber sido implementado en JAVA, ha sido excepcional, y las pruebas indicadas señalan que será resistente a varios de los ataques conocidos. Por lo tanto se puede afirmar que se ha conseguido todo lo que se había propuesto en un principio.

10 Bibliografía

Aunque la mayor parte del trabajo se basa en el trabajo personal, las bases y los conocimientos previos provienen de los siguientes textos:

Handbook of Applied Cryptography

A. Menezes, P. van Oorschot, S. Vanstone

Ed. CRC Press

Libro electrónico de seguridad informática y criptografía

Dr. Jorge Ramió Aguirre, Dr. Josep María Miret Biosca

Diversa información disponible en la red, páginas, artículos y foros de debate.

- www.schneier.com/blowfish.html
- www.kriptopolis.com
- www.rsasecurity.com
- www.monografias.com/trabajos20/cifrado-en-bloques/cifrado-en-bloques.shtml