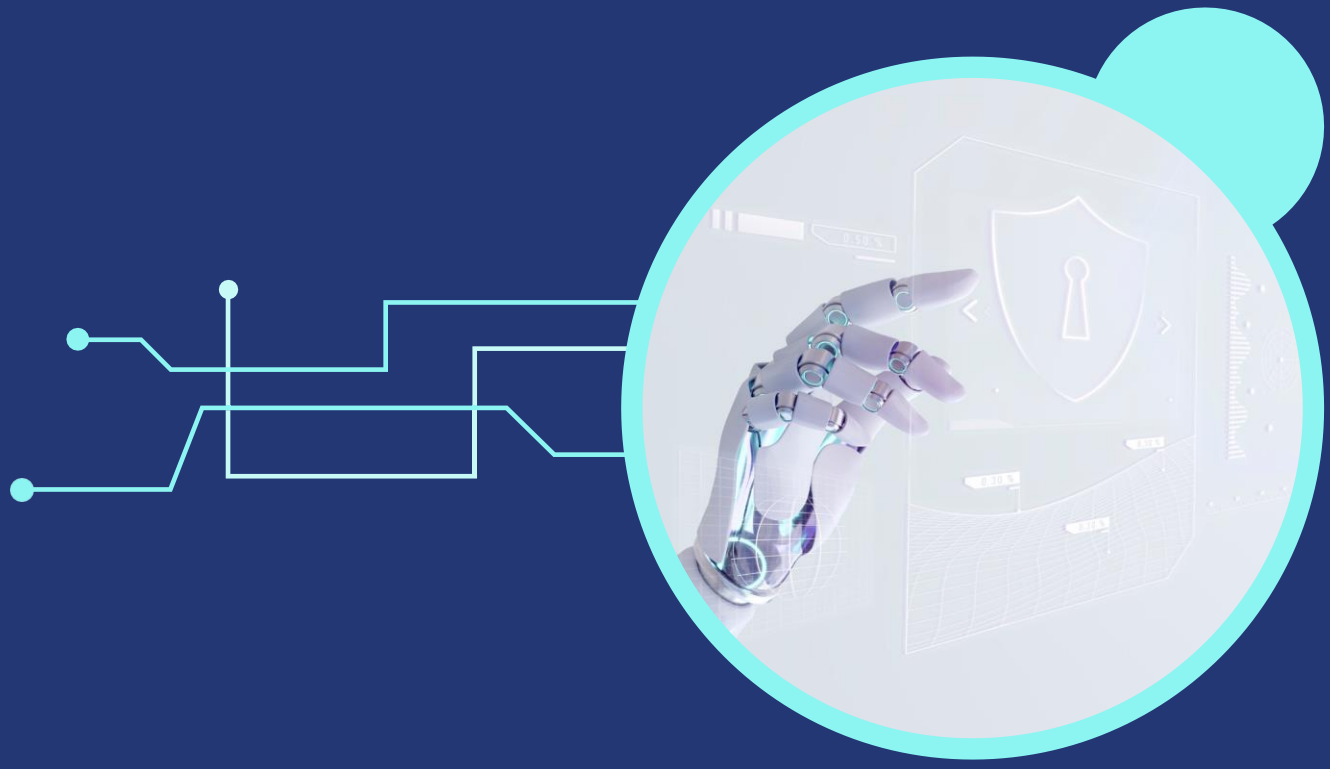
MÓDULO 1. CRIPTOGRAFÍA (UNIDAD 2)

PREMÁSTER EN

ANÁLISIS DE MALWARE, REVERSING Y BUG BOUNTY



Campus Internacional CIBERSEGURIDAD

ENIIT INNOVA IT BUSINESS SCHOOL

UCAM UNIVERSIDAD CATÓLICA DE MURCIA

ÍNDICE DE CONTENIDOS

1. Algoritmos de clave simétrica y funciones hash

En este tema se estudian los algoritmos de clave simétrica, también llamados de clave secreta, que abarcan tanto el cifrado en flujo como el cifrado en bloque. Además, se estudian las funciones hash.

1.1. Cifrado en flujo

1.1.1. Cifrado de Vernam

Antes de meternos a estudiar el cifrado en flujo vamos a hablar de uno de sus antecedentes: el cifrado de Vernam, que fue propuesto en 1917.

Como principales novedades de este algoritmo destacan:

- La utilización de un alfabeto binario. Hasta entonces los algoritmos utilizados usaban símbolos alfanuméricos.
- Utilización de una clave secreta que se hacía llegar a emisor y receptor.
- La clave se utiliza una sola vez (One Time Pad, **OTP**).
- La clave es una secuencia perfectamente aleatoria y es, al menos, tan larga como el texto claro.

A continuación, se muestra un ejemplo de cifrado de un texto con el cifrado de Vernam. En este ejemplo se ha elegido el texto “OTP”, trasladando a binario su código ASCII (la O se corresponde con 79 en el código ASCII, T es el carácter 84 y P es 80).

Se asume que la clave se ha generado de una forma aleatoria (no pseudoaleatoria).

La operación de cifrado para la obtención del criptograma consiste en aplicar la función XOR bit a bit entre el texto claro y la clave de cifrado.

Para descifrar habría que aplicar la función XOR al criptograma y la clave, obteniendo el texto original.

Mensaje	0	1	0	0	1	1	1	1	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	0
Clave	0	0	1	1	1	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0	1	1
Cripto-grama	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	1	0	1	0	1	1

Algunas características del cifrado de Vernam es que se trata de un **cifrado recíproco o involutivo**. Es decir, F (F (texto)) = texto.

Cuando se explique el algoritmo DES se darán más detalles sobre este concepto.

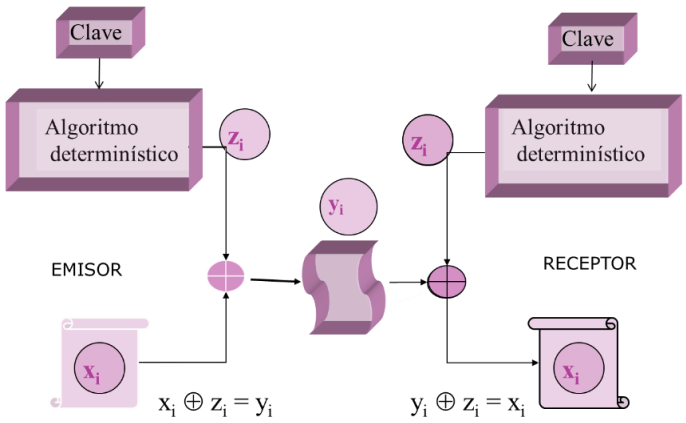
El cifrado de Vernam cumple condiciones de secreto perfecto de Shannon. Es importante señalar que este cifrado tiene una seguridad matemáticamente demostrable y es el único procedimiento conocido que proporciona una seguridad incondicional.

Desde el punto de vista de la seguridad este algoritmo es perfecto, sin embargo, presenta algunos inconvenientes al llevarlo a la práctica: para cada nuevo mensaje que se transmita, hay que hacer llegar clave a emisor y receptor por canal seguro, una clave tan larga como el propio texto claro. Por tanto, el problema se traduce de la transmisión del texto claro a la transmisión de una clave de la misma longitud, lo cual no soluciona la situación. Esto hace que en la práctica la implementación de este algoritmo resulte casi inviable.

La generación de números puramente aleatorios, que es una de las claves que garantiza la seguridad incondicional de este algoritmo, es un problema complejo de resolver. Por ello se optó por sustituir la secuencia aleatoria por una pseudoleatoria y utilizarla como clave. Esta variante es lo que se conoce como cifrado en flujo.

1.1.2.Características del cifrado en flujo

A continuación, se muestra un esquema del funcionamiento del cifrado en flujo.



Para el procedimiento de cifrado se genera una clave utilizando algoritmo determinístico. Utilizando una función XOR se aplica bit a bit sobre la clave (z_i) y el texto claro (x_i), obteniendo el criptograma (y_i). Esto se envía al receptor. Para descifrar el criptograma desde el lado del receptor se aplica la misma función sobre el criptograma y la clave, obteniendo el texto claro. Nótese que la clave que se utiliza para descifrar es la misma que la que se utiliza en el proceso de cifrado (se trata de criptografía simétrica y por tanto emisor y receptor deben compartir la clave).

Como características del cifrado en flujo cabe destacar que es un método cómodo, sencillo y además, rápido. Este método es más seguro cuanto más se acerque la secuencia cifrante a una secuencia aleatoria. Más adelante se muestran cuáles son las características de las secuencias aleatorias. Es importante señalar que este cifrado se realiza bit a bit.

Es importante tener en cuenta que el cifrado en flujo no cumple las condiciones de secreto perfecto de Shannon (es una aproximación al cifrado Vernam).

Para garantizar la seguridad de este tipo de cifrado, las secuencias cifrantes deben cumplir una serie de condiciones. Éstas se consideran condiciones necesarias, aunque no suficientes. Se trata de condiciones referentes a:

- Periodo.
- Distribución de bits.
- Imprevisibilidad.

Estos puntos se ven a continuación:

1.1.2.1. PERIODO

El periodo es la porción de la secuencia que no se repite. Recordemos que es necesario generar secuencias cifrantes tan largas como la longitud del texto a cifrar. En la práctica se pueden generar secuencias con periodo alrededor de 10^{38}.

1.1.2.2. DISTRIBUCIÓN DE BITS

Vamos a dar algunas definiciones en relación con la distribución de bits.

Rachas: Se llama rachas de longitud *k* a la sucesión de *k* dígitos iguales entre dos dígitos distintos.

Gaps: Se llama así a las rachas de ceros.

Blocks: Son las rachas de unos.

Veremos ahora el concepto de autocorrelación, que mide el grado de similitud de una secuencia desplazada *k* posiciones y viene dado por la siguiente fórmula:

$$AC(k) = (C - D)/T$$

Siendo *C* el número de coincidencias, *D* el número de desavenencias con la secuencia desplazada *k* posiciones, *T* el periodo de la secuencia.

A continuación, se muestra un ejemplo. En la primera línea se muestra la secuencia original. En la segunda se muestra la secuencia desplazada cuatro posiciones hacia la izquierda. Se compara bit a bit viendo si hay coincidencia o no en cada una de las posiciones. En este caso el periodo de la secuencia es 16.

Sec.original	1	0	1	1	0	0	1	0	1	0	0	0	0	1	1	1
Sec. desplazada	0	0	1	0	1	0	0	0	0	1	1	1	1	0	1	1

AC(4) = (6-10)/16.

Se llama autocorrelación en fase a AC(0), es decir la secuencia no desplazada o desplazada un número de posiciones que coincide con el periodo, por lo que la secuencia original y la desplazada

$$AC(0) = 1$$

coinciden completamente. En ese caso,

La autocorrelación fuera de fase se da cuando el desplazamiento se realiza un número de posiciones que no hace coincidir la secuencia original y la desplazada. En ese caso

$$AC(k) \in [-1, 1]$$

1.1.2.2.1. POSTULADOS DE ALEATORIEDAD DE GOLOMB

Solomon Golomb, de origen estadounidense y nacido en 1932 planteó en 1967 las propiedades de aleatoriedad que debían seguir las secuencias cifrantes.

- **Primer postulado (G1)**: En cada periodo de la secuencia considerada el número de 1's tiene que ser igual al número de 0's (la diferencia no debe ser > 1).
- **Segundo postulado (G2)**: En cada periodo de la secuencia considerada:

la mitad de las rachas, del número total de rachas observadas, tiene longitud 1,

la cuarta parte longitud 2,

la octava parte longitud 3 … etc.

Para cada longitud habrá el mismo número de rachas de 0's que de 1's.

- **Tercer postulado (G3)**: G3: La autocorrelación fuera de fase tiene que ser constante para todo valor de *k*.

Los postulados se pueden interpretar de esta manera:

- **G1**: El número de 1's y 0's tiene que aparecer a lo largo de la secuencia con la misma probabilidad.
- **G2**: En cada periodo de la secuencia diferentes *n*-gramas deben aparecer con la probabilidad correcta.
- **G3**: El cálculo de coincidencias entre una secuencia y su versión desplazada no debe aportar información sobre el periodo de la secuencia.

Una secuencia binaria finita que verifique G1, G2 y G3 se denomina PN-secuencia.

1.1.2.2.2. IMPREVISIBILIDAD

Una secuencia tiene propiedades de imprevisibilidad cuando no es posible predecir el siguiente dígito con una probabilidad mayor que la que la aleatoriedad, es decir, es imprevisible cuando la probabilidad de acierto es ≤ ½.

1.1.2.3. TEST DE PSEUDOALEATORIEDAD

Como hemos dicho el cifrado en flujo emplea una secuencia pseudoaleatoria (a diferencia del cifrado de Vernam que emplea una aleatoria) como secuencia cifrante. Cuanto más cercanas a la aleatoriedad sean las propiedades de la secuencia pseudoaleatoria, más seguro será el sistema.

Para medir estas propiedades se han diseñado los test de pseudoaleatoriedad.

Se trata de test estadísticos creados con el objetivo de ver cuánto distan las propiedades de una secuencia pseudoaleatoria de una perfectamente aleatoria.

Es importante señalar que las condiciones de estos tests son necesarias, pero no suficientes, es decir, aunque la secuencia supere los test no es posible asegurar que sus propiedades sean suficientes para su uso criptográfico. El criterio es que, si la secuencia no verifica alguno de los tests, ésta debe ser rechazada ya que se considera que no cumple con las características apropiadas para garantizar cierta seguridad en el cifrado.

En este apartado vamos a estudiar tres tests, cada uno más exigente que el anterior.

- **Tests del NIST**. Este test fue publicado en 2010. Se aplica a secuencias obtenidas a partir de un generador conocido, puesto que algunos test necesitan información o parámetros relativos al generador. Se utiliza para secuencias con longitud entre 10^3 y 10^7 bits. Está compuesto por un compendio de 15 tests. Si la secuencia no cumple alguno de los tests la recomendación es que sea descartada.
- **Tests de Diehard**. Esta batería de test estadísticos fue creada por George Masaglia a lo largo de varios años y se publicó en 1995. Se aplica a secuencias de al menos 10^8 bits.

Más detalles sobre el significado de estos tests se puede consultar aquí:

https://en.wikipedia.org/wiki/Diehard_tests

- **Tests de Tuftest** (2011). Estos tests son más exigentes que los anteriores, incluyendo los tests de Diehard y otros adicionales, que Marsaglia consieraba más exigentes que todos los incluidos en el conjunto de Diehard. Como en el caso

anterior, se aplica a secuencias de al menos 10^8 bits. Una secuencia que pase estos tests se podría considerar aceptable.

1.1.3.Generadores de secuencias cifrantes

Una vez vistas algunas de las propiedades que deben cumplir las secuencias cifrantes surge la pregunta: ¿cómo generar estas secuencias? En este apartado se estudian lo generadores de secuencias cifrantes. Los registros de desplazamiento realimentados linealmente son estructuras que se utilizan habitualmente con este fin. En inglés se denominan Linear Feedback Shift Register (LFSR).

Estas estructuras permiten generar secuencias binarias $\{a_n\}$.

Están compuestos por dos elementos:

- *L* celdas de memoria interconectadas entre sí.
- Una función *F* que permite expresar cada elemento de la secuencia generada basándose en los elementos anteriores.

En el caso de los LFSR, la función *F* es lineal, de ahí el nombre que reciben esos registros.

La función *F* se define como:

$$F(a_0, a_1, \dots, a_{L-1}) = \ c_0 a_0 \ \oplus \ c_1 a_1 \oplus \dots \oplus \ c_{L-1} a_{L-1}$$

Los coeficientes *c* representan los coeficientes de retroalimentación, *a* son los elementos de la secuencia binaria, $\oplus$ es la suma módulo 2. Cuando el coeficiente tiene valor 1 significa que esa celda está conectada a la puerta XOR del LFSR.

El polinomio característico del LFSR se representa como:

$$p(x) = x^L + c_{L-1}x^{L-1} + \dots + c_1 x + c_0 = x^4 + x + 1$$

La relación de recurrencia lineal que permite crear los sucesivos elementos de la secuencia viene dada por esta expresión:

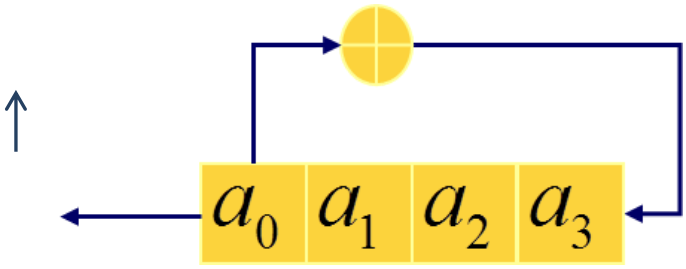
$$a_{n+L} = \ c_0 a_n \oplus \ c_1 a_{n+1} \oplus \cdots \oplus c_{L-1} a_{L+n-1}, \ n \ \geq 0$$

Los parámetros que definen a un LFSR son la longitud del registro (*L*) y el polinomio característico que indica cómo se calcula el elemento de realimentación.

El estado del registro en un momento determinado refleja el contenido binario del registro entre sucesivos pulsos de reloj. Habitualmente se proporciona un estado inicial para el registro, que representa su contenido cuando comienza el proceso de generación, es decir, serán los primeros bits generados.

El contenido del registro va variando en cada pulso de reloj, esto es, su contenido se va desplazando y de ahí el nombre de estos registros. De esta forma, a cada pulso de reloj el contenido de las celdas se desplaza un lugar, a la izquierda en nuestro caso. Esto provoca que el contenido de la celda situada más a la izquierda salga al exterior y se convierta en un elemento de la secuencia generada. La celda situada más a la derecha recibe un nuevo input, siendo éste el resultado de aplicar la función F. Por ello se llaman registros realimentados.

Vamos a ilustrar este funcionamiento con un ejemplo:



En el ejemplo el polinomio es:

$$p(x) = x^4 + x + 1$$

Y la relación de recurrencia lineal viene dada por la expresión

$$a_{n+4} = a_n \oplus a_{n+1}$$

Dado el estado inicial 1000, la secuencia generada es 1000100110101111 (con periodo T=15).

Veamos cómo se han generado los primeros elementos de la secuencia:

El registro tiene el estado inicial

1	0	0	0

Tras el pulso de reloj, se desplaza el contenido a la izquierda

0	0	0	1

1 sale y es el primer elemento de la secuencia generada. El elemento de más a la derecha se rellena con $a_4 = a_0 \oplus a_1$ = 1 ⊕ 0 = 1

Siguiente pulso de reloj, el estado del registro es

0	0	1	0

Sale 0 siendo el segundo elemento de la secuencia generada. $a_5 = a_1 \oplus a_2$ = 0 ⊕ 0 = 0

Siguiente pulso de reloj:

0	1	0	0

{a_n} = 100. $a_6 = a_2 \oplus a_3$ = 0 ⊕ 0 = 0

Siguiente pulso de reloj:

1	0	0	1

{a_n} = 1000. $a_7 = a_3 \oplus a_4$ = 0 ⊕ 1 = 1

Siguiente pulso de reloj:

0	0	1	1

{a_n} = 10001. $a_8 = a_4 \oplus a_5$ = 1 ⊕ 0 = 1

Y así se continuaría hasta obtener la secuencia 1000100110101111 (con periodo T=15).

Las secuencias cifrantes generadas con LFSR con polinomios primitivos cumplen buenas características (periodo largo, buena distribución estadística de 0's y 1's, buena correlación, fácil implementación), sin embargo, no cumplen los criterios de imprevisibilidad, ya que conociendo L bits de la secuencia y el polinomio característico se puede conocer la secuencia completa.

Para solucionar esto se introducen elementos de no linealidad, puesto que en general en criptografía se considera que lo lineal es susceptible de ser criptoanalizado. En función cómo se introduzca la no linealidad se dará origen a distintos tipos de generadores:

- **Generadores basados en combinación no lineal de LFSR**. Se utilizan varios LFSR. En cada pulso de reloj, las salidas de los diversos LFSR son las variables de entrada a una función no lineal.
- **Generadores de secuencia multivelocidad**. En este caso no todos lo LFSR se desplazan síncronamente en cada pulso de reloj. Los factores de velocidad de cada generador se utilizan como nuevos elementos de la clave.
- **Generadores de secuencia con desplazamiento irregular**. Se caracterizan porque la salida de un registro controla el funcionamiento del reloj de los registros siguientes. Generalmente la secuencia cifrante es la suma de las secuencias generadas por varios LFSR que se desplazan irregularmente.
- **Generadores de secuencia con decimación**. La salida de un registro escoge (decima) de forma aleatoria los bits de la PN secuencia generada por otro LFSR que formarán parte de la secuencia final.

En todos los casos la clave es el estado inicial del registro. En presencia de parámetros es posible que éstos también formen parte de la clave.

1.2. Cifrado en bloque

El cifrado más usado en la historia es el cifrado en flujo y su uso estaba reservado fundamentalmente para el gobierno e instituciones militares. El cifrado en bloque rompe en cierta manera con ese estilo más tradicional e incorpora estándares para

aplicaciones civiles.

1.2.1.Principios de confusión y difusión

Como se ha mencionado Shannon fue una figura clave en la historia de la criptografía. Estableció conceptos como la confusión y la difusión, los cuales están relacionados con la teoría de la información y de la comunicación.

El **principio de confusión** dice lo siguiente:

> *"La estadística del texto cifrado depende de la del texto claro de una forma tan complicada que no aporta información al criptoanalista".*

A continuación, se enuncia el **principio de difusión**:

> *"Cada dígito del texto claro debe afectar a tantos dígitos del texto cifrado como sea posible y/o cada dígito de la clave debe afectar a tantos dígitos del texto cifrado como sea posible".*

1.2.2.Características del cifrado en bloque

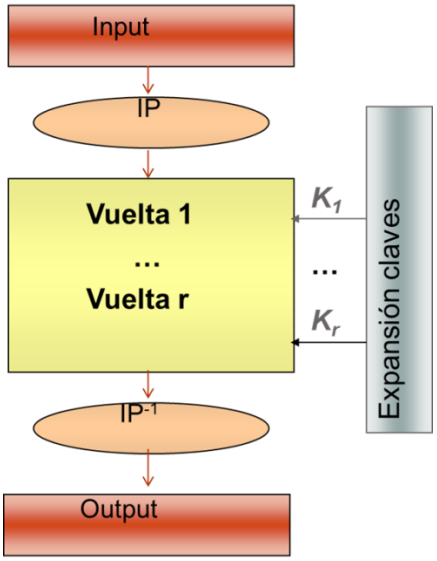
En este apartado se estudian cuáles son los aspectos que caracterizan al cifrado en bloque:

- El mensaje se cifra en grupos o bloques, a diferencia del cifrado en flujo que actúa bit a bit.
- Es necesario que la clave se distribuya con mecanismo seguro (correo seguro u otro cifrado).
- Dependencia entre bits: los bits de texto cifrado son una función compleja de todos bits de clave y de texto claro.
- Cambio de bits de entrada: el cambio en un bit del mensaje original produce un cambio de aproximadamente el 50% de los bits del mensaje cifrado.
- Cambio de bits de clave: el cambio en un bit de la clave produce un cambio de aproximadamente el 50% de los bits en el mensaje cifrado.

- Un error en transmisión de texto cifrado se propaga a todo el bloque. Tras el descifrado aproximadamente un 50% de los bits serán erróneos.
- Cada símbolo se cifra de manera dependiente de los adyacentes.
- Cada bloque se cifra de igual manera independientemente del lugar que ocupe dentro del texto claro.
- Dos mensajes originales iguales, cifrados con la misma clave, producen el mismo criptograma.

1.2.3.Arquitectura del cifrado en bloque

La arquitectura del cifrado en bloque se puede ver en la siguiente figura:



El esquema cuenta con varias fases.

1. Primeramente, se aplica una transformación inicial a la entrada.
2. Se aplica una función *F* iterada *r* veces. El número de repeticiones de esta función se conoce como el número de vueltas del algoritmo.

3. Generalmente es necesario un algoritmo de expansión de claves. Como el algoritmo da *r* vueltas, es necesario expandir la clave inicial y generar subclaves para las diferentes rondas.
4. Finalmente se aplica una transformación final, que puede ser la inversa de la transformación inicial.

Hay varios algoritmos de cifrado en bloque. En este apartado nos centraremos en DES, el cual fue un estándar de cifrado.

1.2.4.Algoritmo de cifrado DES

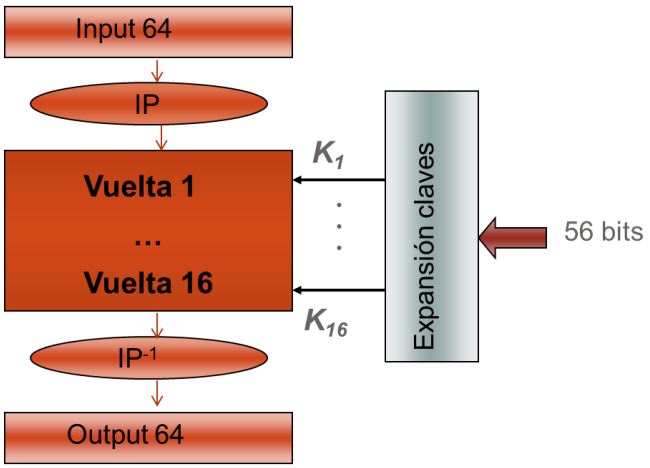
Las siglas DES significan Data Encryption Standard, puesto que efectivamente este algoritmo se llegó a convertir en un estándar de cifrado.

Repasemos brevemente sus orígenes.

- En 1973 el National Bureau of Standards (NBS) difundió la convocatoria solicitando "un algoritmo de cifrado para protección de datos durante su transmisión y almacenamiento".
- En 1974 IBM presentó un algoritmo basado en Lucifer que posteriormente dió lugar al DES.
- La aprobación y modificación se hizo bajo la supervisión de la NSA, que impusó la longitud de la clave (56 bits).
- En 1976 DES fue aprobado como estándar y al año siguiente es publicado por el Federal Information Processing Standards (FIPS) como estándar.
-

1.2.4.1. ESTRUCTURA DE DES

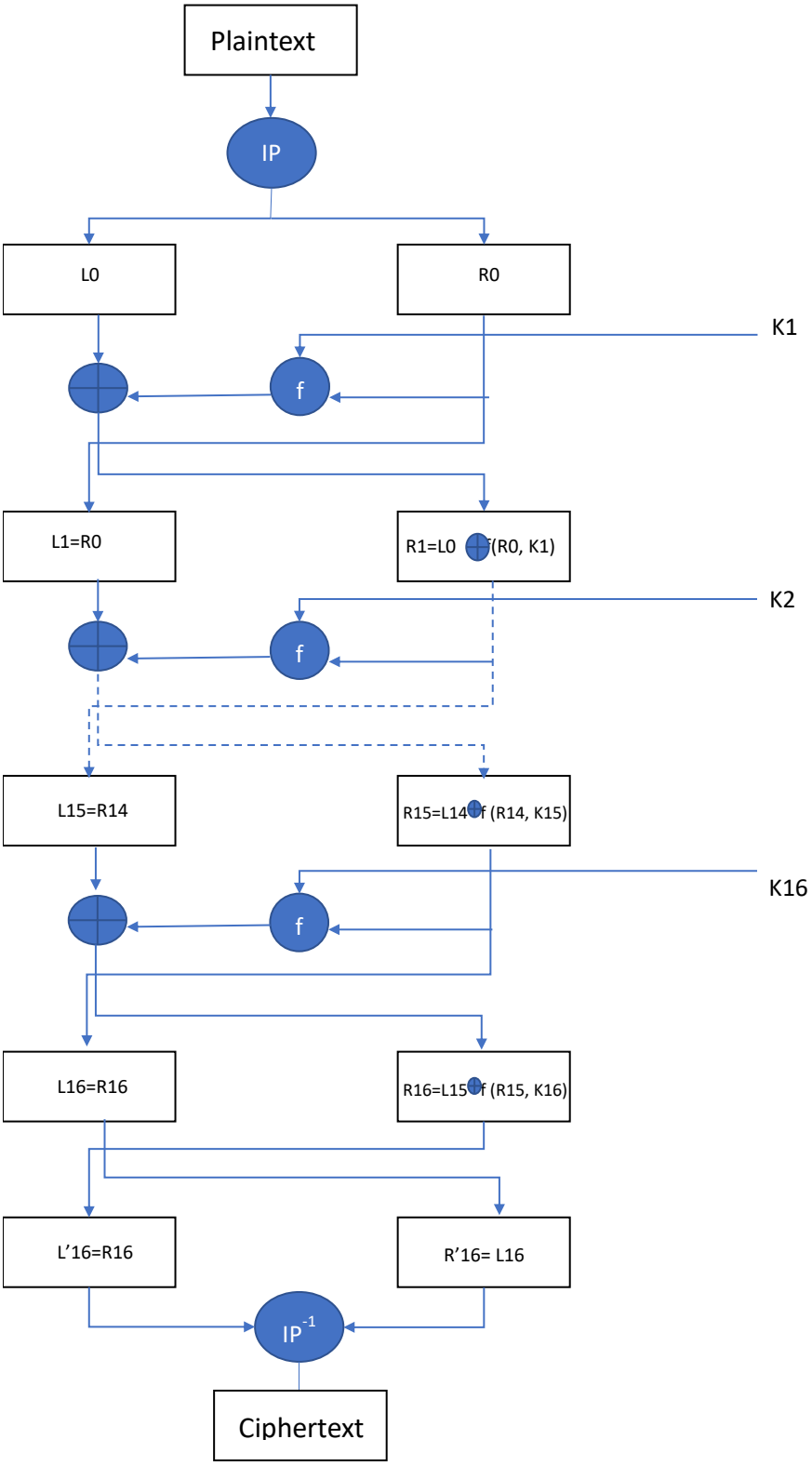
El algoritmo DES sigue el esquema de cifrado en bloque:



Trabaja con **bloques de 64 bits** del mensaje a cifrar. Realiza una transformación inicial. DES da **16 vueltas** en las que aplica una función de transformación. Además, cuenta con un algoritmo de expansión de claves que a partir de la **clave de 56 bits** genera 16 subclaves, una para cada una de las vueltas del algoritmo. Por último, aplica una transformación final que es la inversa de la transformación inicial. Genera un bloque de salida del criptograma de tamaño 64 bits.

El hecho de que este algoritmo tenga una clave de 56 bits implica que trabaja con un espacio de claves de $2^{56} = 7{,}2 \times 10^{16}$ claves. Como apunte mencionar que de los 64 bits de la clave, 8 se utilizan para calcular la paridad, por lo que realmente quedan 56 bits útiles.

DES presenta una **estructura tipo Feistel**. Esta estructura se puede ver en la siguiente figura.


Plaintext
IP
L0
R0
K1
f
L1=R0
R1=L0 f(R0, K1)
K2
f
L15=R14
R15=L14 f (R14, K15)
K16
f
L16=R16
R16=L15 f (R15, K16)
L'16=R16
R'16= L16
IP-1
Ciphertext

Dado el texto a cifrar, en primer lugar, se realiza la transformación inicial.

El bloque de 64 bits se divide por la mitad y se trabaja por un lado con el sub-bloque izquierdo *L0* (de 32 bits) y por otro con el derecho *R0* (32 bits).

Tras esto daría comienzo la primera de las 16 vueltas.

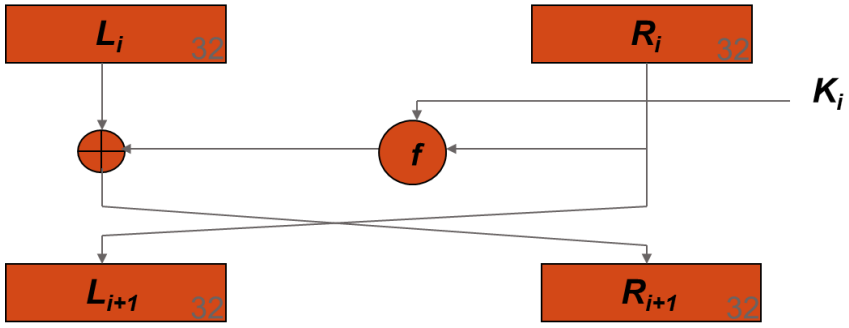
La salida de cada vuelta corresponde a las siguientes fórmulas:

$$I_n = D_{n-1}$$
$$D_n = I_{n-1} \oplus F\,(K_n, D_{n-1})$$

Nótese que para calcular el sub-bloque izquierdo de la siguiente vuelta se utiliza el sub-bloque derecho de la vuelta anterior.

En el caso del sub-bloque derecho de la siguiente vuelta, se calcula haciendo una suma bit a bit módulo 2 del sub-bloque izquierdo de la vuelta anterior y de aplicar la función F a la subclave de la vuelta correspondiente (K_n) y el sub-bloque derecho de la vuelta anterior.

Esto se puede ver con más detalle en la siguiente figura:



En las siguientes subsecciones se explicará la función F y el algoritmo de expansión para la generación de las subclaves.

La transformación final tiene dos etapas:

En primer lugar, se realiza un cruce, intercambiando los sub-bloques de modo que

L'16=R16 y *R'16=L16*.

Después se realiza la permutación final, que es la inversa de la inicial.

1.2.4.2. TRANSFORMACIÓN INICIAL

La transformación inicial en DES consiste en una permutación fija de los bits.

La permutación aplicada es

{ 58 50 42 34 26 18 10 02
60 52 44 36 28 20 12 04
62 54 46 38 30 22 14 06
64 56 48 40 32 24 16 08
57 9 41 33 25 17 09 01
58 51 43 35 27 19 11 03
61 53 45 37 29 21 13 05
63 5 47 39 31 23 15 07}

1.2.4.3. TRANSFORMACIÓN FINAL

Tras el cruce de bloques, se realiza una permutación fija de los bits, que es la inversa de la inicial.

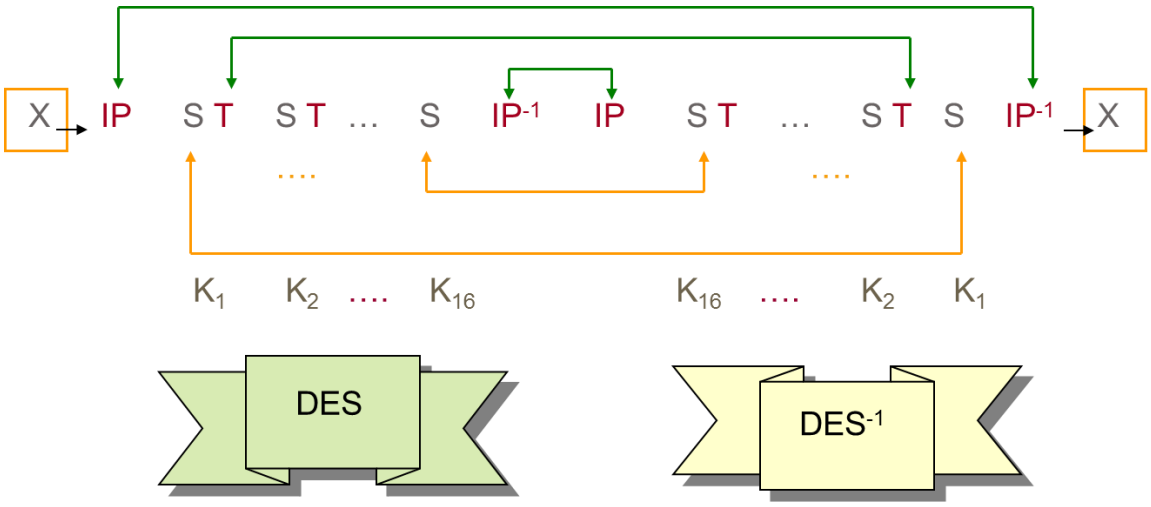
La permutación es la siguiente:

{ 40 08 48 16 56 24 64 32
39 07 47 15 55 23 63 31
38 06 46 14 54 22 62 30
36 04 44 12 52 20 60 28
35 03 43 11 51 19 59 27
34 02 42 10 50 18 58 26
33 01 41 09 49 17 57 25}

1.2.4.4. DESCIFRANDO DES

Una vez que se recibe el criptograma, para descifrarlo y obtener el mensaje original debe aplicarse DES, repitiendo todas las operaciones realizadas, pero utilizando las subclaves en el orden inverso: K_{16}, K_{15}, ..., K_1.

En la siguiente figura se muestra el proceso de cifrado y descifrado de DES:



En el proceso de cifrado: dado el mensaje X se realiza la transformación inicial. Luego se realizan las 16 vueltas, en cada una de las cuáles se realiza una sustitución (S) y una transposición (T) y finalmente la transformación final.

Para descifrar se realiza el proceso inverso, como deshaciendo las operaciones que se han llevado a cabo en el cifrado. Se comienza haciendo la transformación inversa a la transformación final (la transformación inicial). A continuación, se realizan las operaciones de las 16 vueltas, pero aplicando las subclaves en el orden inverso. Finalmente se aplica la transformación final para obtener el mensaje en claro X.

1.2.4.5. INVOLUCIÓN

En este apartado se explica el concepto de involución.

Se dice que la función F es una involución si $F(F(x)) = x$

Es decir, si al aplicar F al resultado de aplicar F a un mensaje x se vuelve a obtener el mensaje x.

En DES se cumple la propiedad involutiva, ya que la aplicación de dos operaciones

XOR consecutivas en las que uno de los sumandos no varía es como si no se hubiese realizado ninguna operación. Es decir:

$(x \oplus y) \oplus y = x \oplus (y \oplus y) = x \oplus 0 = x$

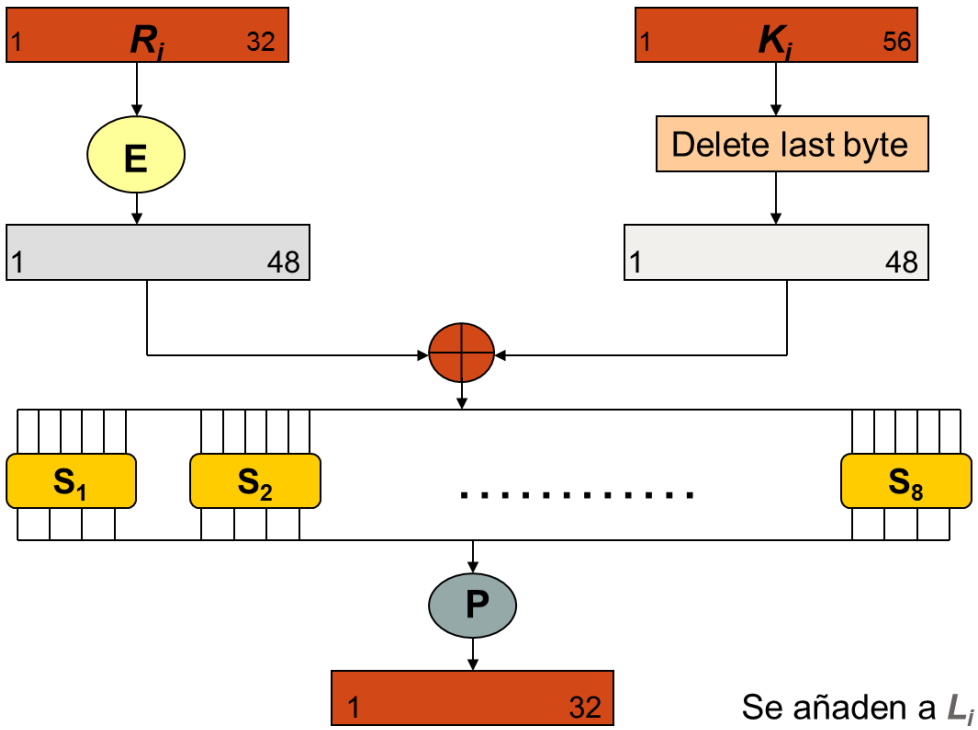
Es precisamente por esta propiedad por la que, en DES, la última operación de cifrado y la primera de descifrado se neutralizan mutuamente, tal y como se ha indicado en el esquema del apartado anterior.

1.2.4.6. FUNCIÓN F

Se ha hablado de la función F, que se utiliza para generar el sub-bloque derecho de la siguiente vuelta. Pero ¿cómo funciona F? Vamos a verlo con más detalle en este apartado.

En DES se aplica F como sigue: $F\ (K_n, D_{n-1})$.

Un esquema general del funcionamiento de la función F se puede ver en la siguiente figura:



Vamos a ver que dentro de este proceso se realizan varias operaciones.

La primera de ellas es la expansión. Consiste transformar los 32 bits de R_i en 48 bits.

A continuación, se elimina el último byte de la sub-clave correspondiente a la ronda *i* de modo que se pasa de 56 a 48 bits.

El resultado de los dos bloques generados de 48 bits anteriores se combina mediante una suma módulo 2 bit a bit y los 48 bits resultantes se agrupan en 8 grupos de 6 bits.

Cada uno de estos grupos de bits se utiliza como entrada para cada una de las 8 "**cajas S**". Más adelante se explicarán estas cajas con más detalle.

Finalmente se aplica al resultado una transformación de la "**caja P**", obteniendo una salida de 32 bits, que son los que forman el sub-bloque izquierdo de la siguiente vuelta.

1.2.4.6.1. EXPANSIÓN

Como hemos dicho el proceso de expansión amplía de 32 a 48 los bits de R_i. Esto se realiza utilizando los 32 bits y además, repitiendo la mitad de ellos.

Viéndolo en un ejemplo, se puede ver que los bits en color azul se añaden a los originales (en color negro) y repiten algunos de esos bits, hasta obtener 48.

32	1	2	3	4	5	4	5	6	7	8	9	8	9	10	11	12	13	12 …

1.2.4.6.2. CAJAS S

Se denominan **cajas S** (S-box) porque realizan sustituciones.

Estos elementos son muy importantes en DES puesto que son las responsables de la no linealidad del algoritmo. Hay 8 cajas S en DES. Están elegidas de modo que la sustitución que realizan no sea una función lineal ni afín de la entrada.

Cabe destacar que los principios de diseño de las S-boxes no fueron revelados y se considera información clasificada de los EEUU.

En la figura se puede ver un ejemplo de caja S. Consta de 2^2 filas y 2^4 columnas.

Las sustituciones en la caja S se realizan de esta forma: el primer y último bit del bloque se utilizan como índice de las filas y los cuatro centrales como índice de las columnas. Por ejemplo, si S1 recibe los bits 101110, se buscaría en la tabla en la fila 2 (10) y columna 7 (0111), de forma que se sustituye por los 4 bits de salida correspondientes al contenido de la caja en esa posición: 11 (1011).

CR	CL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	1	0	15	7	4	14	2	13	1	10	6	12	11	4	5	3	8
1	0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1	1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Las cajas S aportan buenas propiedades de confusión y difusión al algoritmo DES. Además, se cumple que cambiando uno de los 6 bits de entrada cambian al menos dos de los bits de salida.

La salida de todas las cajas S forman la entrada de la caja P. La caja P realiza una permutación lineal fija, que es la siguiente:

{ 16 7 20 21 29 12 28 17
1 15 23 26 5 18 31 10
2 8 24 14 32 27 3 9
19 13 30 6 22 11 4 25 }

Esta sustitución aporta una máxima difusión a los bits.

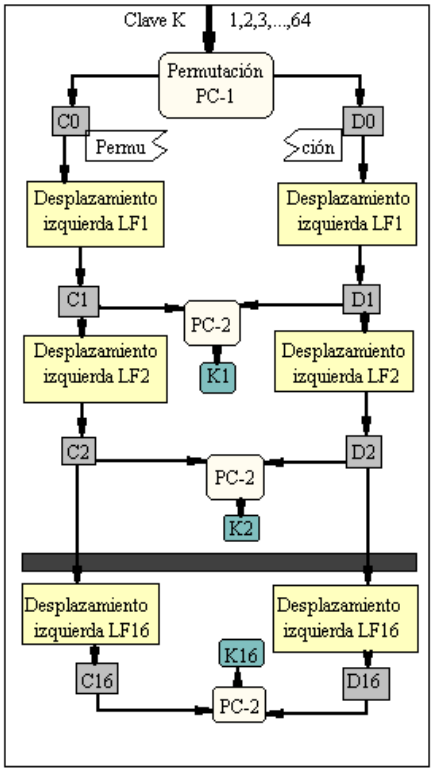
Por tanto, F realiza operaciones de sustitución y permutación.

1.2.4.7. EXPANSIÓN DE CLAVES

La clave del DES tiene 56 bits efectivos, puesto que 8 de ellos utilizan para detección de errores.

En cada vuelta se necesitan 48 bits. Como hay 16 vueltas, son necesarios 16*48= 768 bits de clave. Sin embargo, la clave tiene solo 56 bits. Para ello se utiliza un algoritmo de expansión de claves.

Este algoritmo tiene la siguiente estructura:



Los pasos del algoritmo son los siguientes:

1. Se realiza una permutación fija PC-1 de los 56 bits de clave.

2. Los 56 bits se divide en dos bloques.

3. Cada mitad se rota a la izquierda un número de posiciones dependiendo del número de vuelta en el que se encuentre el algoritmo:

   - 1 posición en las vueltas 0, 1, 8, 15.
   - 2 posiciones en el resto de vueltas.

   Esto se hace durante 16 vueltas.

4. Se unen las dos mitades.

5. De cada una de las claves de 56 bits, se seleccionan 48 bits siempre en la misma posición (24 bits del bloque izquierdo y 24 bits del bloque derecho).

6. Se permutan los 48 bits de cada subclave, siempre en el mismo orden (PC-2).

De esta manera se generan 16 subclaves de 56 bits.

Las 16 subclaves son permutaciones de los 56 bits, de los que se eligen 48 bits diferentes en cada vuelta. Cada bit se utiliza en aproximadamente 14 de las 16 subclaves.

1.2.4.8. CLAVES DÉBILES Y SEMI-DÉBILES

DES tiene cuatro claves débiles conocidas. Éstas son aquellas en las que todas las subclaves K_1 hasta K_{16} coinciden.

$$DES_k(.) = DES_k^{-1}(.)$$

El cifrado con una clave es igual al descifrado con la otra.

Las cuatro claves débiles son:

01 01 01 01 01 01 01 01
FE FE FE FE FE FE FE FE
E0 E0 E0 E0 E0 E0 E0 E0
1F 1F 1F 1F 1F 1F 1F 1F

Hay otros casos en los que solo hay dos subclaves K_i diferentes

$$DES_k(.) = DES_{k'}^{-1}(.) \iff DES_{k'}(.) = DES_k^{-1}(.)$$

Hay 12 claves semi-débiles conocidas:

01 FE 01 FE 01 FE 01 FE	FE 01 FE 01 FE 01 FE 01
01 E0 01 E0 01 E0 01 E0	E0 01 E0 01 E0 01 E0 01
01 1F 01 1F 01 1F 01 1F	1F 01 1F 01 1F 01 1F 01
1F E0 1F E0 1F E0 1F E0	E0 1F E0 1F E0 1F E0 1F
1F FE 1F FE 1F FE 1F FE	FE 1F FE 1F FE 1F FE 1F
E0 FE E0 FE E0 FE E0 FE	FE E0 FE E0 FE E0 FE E0

Hay otras 48 claves semi-débiles que producen sólo 4 subclaves diferentes en lugar de 16.

Se recomienda asegurarse de no elegir claves débiles o semi-débiles, aunque su probabilidad de aparición sea pequeña.

1.2.4.9. SEGURIDAD

Hasta el día de hoy, no se ha conseguido criptoanalizar (romper) el algoritmo DES.

Sin embargo, su espacio de claves es excesivamente reducido para la tecnología actual: **56 bits es insuficiente.**

Como técnicas de criptoanálisis de DES se han utilizado técnicas diferenciales, lineales y fuerza bruta. La última resulta la opción más eficaz. De hecho, en 1988 se desarrolló el DES cracker, que conseguía el texto original tras 9 días de cómputo.

1.2.4.10. CIFRADO MÚLTIPLE

Para mayor seguridad, es posible emplear varios cifradores de modo consecutivo. De hecho, éste es el único modo de aumentar el espacio de claves en cifrado en bloque.

Cuando se hace esto, y fuera de lo que se pudiera pensar, la longitud efectiva de la clave no se incrementa de modo lineal, sino que sigue la siguiente fórmula:

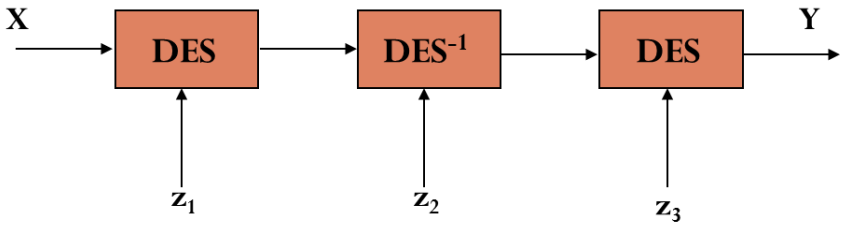
$$l = L \cdot \left\lceil \frac{n}{2} \right\rceil$$

Siendo *L*: longitud clave, *n:* número de cifradores.

Dado que DES no es un grupo, se puede aplicar el cifrado múltiple. Si fuese un grupo daría el mismo resultado aplicar varios cifrados que uno solo y por tanto, el cifrado sería inútil.

De los cifrados múltiples, el más extendido es TripleDES (TDES) en el que se encadenan tres cifrados DES consecutivos.

El esquema es el siguiente:



Hay tres variantes de este esquema:

- En el que $z_1=z_2=z_3$. La seguridad proporcionada es de 56 bits y la longitud real de la clave es 56 bits (más 8 de paridad). Se utiliza por compatibilidad con DES.
- En el que $z_1=z_3 \neq z_2$. Equivale a un sistema con clave de 80 bits. La longitud real de la clave es 112 bits (más 16 de paridad). Se puede emplear cuando el nivel de seguridad necesario es mínimo.
- En el que $z_1 \neq z_2 \neq z_3$. En este caso se proporciona la seguridad de un sistema de 112 bits de clave. La longitud real de la clave es de 168 bits (más 24 de paridad). Se puede usar para sistemas que requieran una seguridad media.

El inconveniente es que requiere el triple de operaciones de cifrado.

1.2.4.11. OTROS CIFRADORES DE BLOQUE

Hay otros cifradores de bloque conocidos. En la tabla se muestra un resumen de algunos de ellos junto con algunas de sus características principales.

Algoritmo	Bloque (bits)	Clave (bits)	Vueltas
Twofish	128	variable	variable
Khufu	64	512	16, 24, 32
Khafre	64	128	más vueltas
Gost	64	256	32
RC5	variable	variable	variable
SAFER 64	64	64	8
Akelarre	variable	variable	variable
FEAL	64	64	32

1.2.4.12. ASPECTOS PRÁCTICOS

Vamos a plasmar los conceptos que hemos visto anteriormente y a ver el funcionamiento de este algoritmo en la práctica. Es muy probable que en el desarrollo de nuestro trabajo nos encontremos en situaciones en las que sea necesario cifrar ciertos datos. Vamos a ver cómo se puede hacer esto utilizando la librería Cryptography de Python. En los ejemplos sucesivos vamos a usar Python 3.

Esta librería recoge múltiples algoritmos de cifrado como AES, DES, RSA, además de funciones de hash seguras, tales como SHA256 y RIPEMD160. La información sobre el cifrado simétrico que proporciona esta librería se puede consultar en https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/.

En este apartado vamos a ver de qué manera es posible cifrar y descifrar un mensaje con el algoritmo DES.

El proceso de cifrado con la librería Cryptography de Python se realiza de la siguiente manera:

```
import os
    from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
    from cryptography.hazmat.backends import default_backend
    backend = default_backend()
    key = os.urandom(24)
    cipher = Cipher(algorithms.TripleDES(key), modes.ECB(), backend=backend)
    encryptor = cipher.encryptor()
    cryptogram = encryptor.update(b"a secret message") + encryptor.finalize()
    print(cryptogram)
```

Ya que la clave es aleatoria, uno de los resultados que se pueden obtener de la ejecución de este código es:

```
b'\xab\t\xce7zl|g\xe8\xd4[\xc2\xdd\x14#\x8c'
```

Nótese que el resultado está expresado en bytes.

El backend proporciona interfaces para realizar operaciones como el cifrado simétrico o hashing.

El primer paso es configurar el cifrador que se va a utilizar, en este caso se está usando un algoritmo TripleDES con una clave de 192 bits de longitud.

Cuando se llama a encryptor() en un objeto Cipher el resultado se ajusta a la interfaz CipherContext.

A continuación, se puede llamar al método update(data) con datos hasta que se haya alimentado todo en el contexto. Por último, se invoca a finalize() para acabar la operación y obtener el resto de los datos.

Los cifradores de bloque requieren que el texto claro (el caso de cifrado) o el criptograma (en caso de descifrado) tenga una longitud que sea múltiplo del tamaño de bloque. En caso de que no lo sea, habrá que aplicar padding para que el mensaje tenga la longitud adecuada, sin embargo, CipherContext no aplica padding automáticamente sino que es el usuario el que debe tener en cuenta estos aspectos. El padding recomendado para cifradores de bloque es PKCS7.

Tras esto, el proceso de descifrado es sencillo

```
decryptor = cipher.decryptor()
clearText=decryptor.update(cryptogram) + decryptor.finalize()
print(clearText)
```

Como se puede ver, decryptor() funciona de manera similar a encryptor().

El resultado de la ejecución de este código nos devuelve el texto original:

```
b'a secret message'
```

- Vamos a analizar el comportamiento de DES estudiando algunos de sus aspectos interesantes, como la propiedad de difusión (o efecto avalancha), lo que denominábamos cambio en los bits de entrada y de clave al estudiar las características del cifrado en bloque.

Para comprobar el cambio en los bits de entrada vamos a cifrar el texto “a secret message” con la clave “123456789012345678901234”. Conviene recordar que la clave que se le pasa al cifrador debe estar expresada en bytes.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
backend = default_backend()
key=bytes(str('123456789012345678901234'), 'ascii')
cipher = Cipher(algorithms.TripleDES(key), modes.ECB(), backend=backend)
encryptor = cipher.encryptor()
cryptogram = encryptor.update(b"a secret message") + encryptor.finalize()
print(cryptogram)
```

Nótese que este caso es determinístico; al ser la clave fija, el resultado de sucesivas ejecuciones no varía.

El criptograma obtenido es:

```
b'7\xf8z\x92\xe4\xf3\x17\xd7\x19\xcb\xce\xa1*\xbbo\xef'
```

Ahora ciframos el texto “a secreu message”, en el que solo varía un bit respecto del texto anterior, con la misma clave. El resultado en este caso es:

```
b'\xcb\xfc\x9bO\x96s\xb6=\x19\xcb\xce\xa1*\xbbo\xef'
```

Con este ejemplo se puede apreciar que el resultado obtenido es muy diferente al anterior. De este modo se comprueba que cuando en el mensaje cambia un solo carácter, es más, cambia un solo bit del mensaje, ésto afecta al menos a la mitad de los bits del criptograma, produciendo un criptograma que difiere en gran medida del anterior. Esta es una propiedad deseable en criptografía.

- Vamos a hacer la misma comprobación variando la clave.

Ciframos el texto “a secret message” con la clave “12345578901234567890123 4”. En este caso hay un 5 donde antes había un 6.

El criptograma obtenido es:

```
b"\xc1z\x91\xef@0a<>\xf1\x859\xa4'W\xaf"
```

Nuevamente se puede apreciar la diferencia en el resultado obtenido respecto del criptograma anterior.

1.2.5.Algoritmo de cifrado AES

Las siglas AES significan Advanced Encryption Standard.

A continuación, se repasa brevemente cuál es el origen de este algoritmo.

En 1996 el National Institute of Standards and Technology (**NIST**) inicia el proceso de selección del AES para proporcionar un nuevo estándar.

El objetivo era encontrar un algoritmo de cifrado sustitutivo del DES que usasen varios actores:

- El Gobierno de EEUU (para proteger información sensible pero no clasificada).
- El sector privado en EEUU.
- Por extensión, el resto de los países del mundo.

La intención era seleccionar un algoritmo que no solo tuviera vigencia en el presente sino también hasta bien entrado el S XXI (≈ 2060).

Había ciertos requisitos requeridos:

- Longitud de bloque: 128 bits.
- Longitud de clave: **128, 192 y 256** bits.

Algunos de los criterios para la selección del algoritmo eran los siguientes:

- Seguridad (esfuerzo criptoanalítico).
- Eficacia computacional.
- Adecuación hardware y software.
- Simplicidad de diseño.
- Flexibilidad.
- Requisitos de licencia (no patentado ni patentable).

En octubre de 2000 el NIST anunció el ganador: el algoritmo RINJDAEL propuesto por Vincent Rijmen y Joan Daemen.

En mayo 2002 AES entra en funcionamiento.

Este algoritmo fue el sustituto de DES, que era el estándar hasta el momento y es el estándar a día de hoy.

1.2.5.1. OPERACIONES EN GF

Ya que AES utiliza operaciones en grupos de Galois (GF), antes de analizar el algoritmo vamos a ver cómo se realizan operaciones en GF(2^8).

Para ello tendremos dos factores en hexadecimal, a modo de ejemplo:

$(57)_{16}$. En binario es $(0101\ 0111)_2$. Tomándolo como coeficientes, dan lugar al polinomio $x^6+x^4+x^2+x+1$

$(83)_{16}$. En binario es $(1000\ 0011)_2$, que da lugar al polinomio x^7+x+1

1.2.5.2. SUMA

Veamos un ejemplo de suma, $(57)_{16} + (83)_{16}$.

Tomando los polinomios anteriores, se suman los factores:

$x^6+x^4+x^2+x+1$

$+ x^7+ \quad x+1$

$x^7+ x^6+x^4+x^2$

Hay que tener en cuenta que en este cuerpo 1 + 1 =0.

Tomando los coeficientes en binario se representa como $(1101\ 0100)_2$, que corresponde en hexadecimal a $(D4)_{16}$.

1.2.5.2.1. MULTIPLICACIÓN

Vamos a multiplicar ahora los factores anteriores a modo de ejemplo de multiplicación.

$(57)_{16} * (83)_{16}$.

Con los polinomios sería $x^6+x^4+x^2+x+1 * x^7+x+1 = x^{13} + x^7+x^6+x^{11}+x^5+x^4+x^9+x^3+x^2+x^8+x^2+x+x^7+x+1 = X^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1$

Como el grado es mayor de x^8 se debe dar el resultado mod $x^8+x^4+x^3+x+1$

Por tanto, debe dividirse

$$x^{13}+ x^{11} + x^9 + x^8 + \quad x^6+ x^5+ x^4+ x^3+ 1 : x^8+ x^4+ x^3+ x+ 1= x5 +x3$$
$$+ x^{13}+ \quad x^9+ x^8+ \quad x^6+ x^5$$

$$x^{11} \quad + x^4+ x^3 \quad + 1$$
$$+ \quad x^{11} \quad + x^7+ x^6 \quad + x^4+ x^3$$

$$x^7+ x^6 \quad +1$$

El resultado es x^7+x^6+1 que corresponde en binario con $(1100\ 0001)_2$ y en hexadecimal $(C1)_{16}$.

1.2.6. Diseño

El algoritmo está diseñado para emplear N_b palabras de 32 bits pudiendo tomar N_b los valores 4, 6 u 8

La longitud de la clave es de N_k palabras de 32 bits, pudiendo tomar N_k los valores 4, 6 u 8.

De esta forma el algoritmo puede trabajar con claves de tamaños 128, 192 o 256 bits.

Por simplicidad, aquí se va a mostrar la versión que establece $N_b = 4$, $N_k = 4$.

1.2.6.1. ESTRUCTURA

AES está diseñado como un **cifrador iterado que usa funciones invertibles**. Opera con **bloques enteros** (a diferencia de las redes de Feistel que lo hacen con mitades de bloque).

Al resultado de cada paso del algoritmo se le llama **estado**, que es un conjunto de tantos bits como el bloque.

El estado inicial, estados intermedios y estado final se almacenan en una tabla de 4x4,

que contiene los 128 bits del bloque, como se muestra en la figura.

Bytes de entrada

en_0	en_4	en_8	en_{12}
en_1	en_5	en_9	en_{13}
en_2	en_6	en_{10}	en_{14}
en_3	en_7	en_{11}	en_{15}

Estado

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Bytes de salida

sal_0	sal_4	sal_8	sal_{12}
sal_1	sal_5	sal_9	sal_{13}
sal_2	sal_6	sal_{10}	sal_{14}
sal_3	sal_7	sal_{11}	sal_{15}

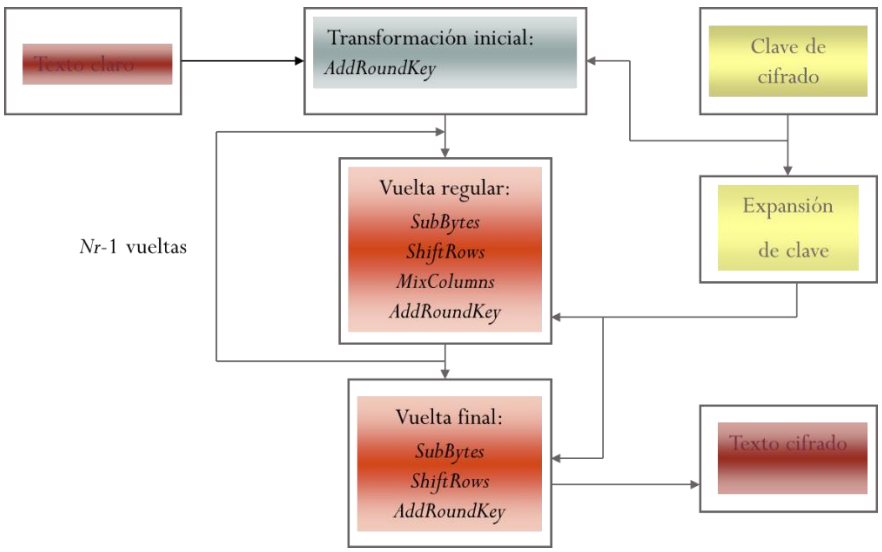
Al ejecutarse el algoritmo los bytes de entrada se convierten en estado y éste en la salida.

El algoritmo AES sigue la siguiente estructura:

- Transformación inicial.
- *Nr-1* vueltas regulares.
- Vuelta final.

Donde *Nr* = 10 para claves de 128 bits. *Nr*= 12 para claves de 192 bits. *Nr* = 14 para claves de 256 bits.

El siguiente esquema refleja el funcionamiento general del algoritmo Rinjdael:



En primer lugar, se realiza una transformación inicial del texto claro aplicando *AddRoundKey*. A continuación, se realizan Nr-1 vueltas (vuelta regular) que comprenden cuatro transformaciones: ***SubBytes***, ***ShiftColumns***, ***MixColumns***, ***AddRoundKey***.

En cada vuelta se utiliza una subclave que se genera con el algoritmo de expansión de clave a partir de la clave de cifrado.

Por último, se realiza la vuelta final que comprende tres transformaciones (las cuatro de una vuelta regular a excepción de *MixColumns*): *SubBytes*, *ShiftColumns*, *AddRoundKey*.

A continuación, se explican con más detalle los pasos del algoritmo.

1.2.6.2. TRANSFORMACIÓN INICIAL

En primer lugar, se aplica la transformación *AddRoundKey*. Ésta consiste en aplicar la suma módulo 2, bit a bit, de los 128 primeros bits de clave con el texto claro, es decir:

Texto claro $\oplus$ clave (128 bits).

1.2.6.3. TRANSFORMACIÓN SUBBYTES

Esta sustitución que se aplica a todos y cada uno de los bytes de estado. Esto se hace por medio de una única caja S (en el caso de AES sí se reveló su diseño).

La caja S introduce elementos de no linealidad en el proceso.

La caja S está orientada a

- Minimizar correlación de entrada y salida.
- Minimizar la probabilidad de propagación de diferencias.
- Maximizar la complejidad de la transformación.

En la figura se muestra un ejemplo de caja S:

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

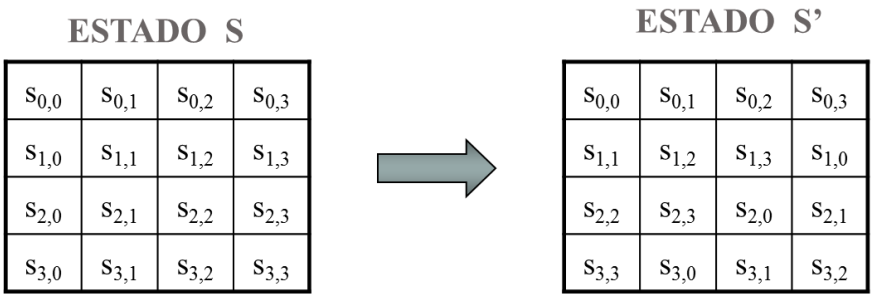
En la tabla se puede consultar el valor por el que sustituir el byte *xy*.

1.2.6.4. TRANSFORMACIÓN SHIFTROWS

En esta transformación se permutan cíclicamente los contenidos de las filas de estado, lo cual contribuye a lograr difusión. Esto es interesante para ser más resistente frente al criptoanálisis diferencial.

En función de la fila, se permuta un número de posiciones diferente, realizando un desplazamiento cíclico a izquierdas de tantas posiciones como indique el número de la fila. Es decir, en la fila se desplaza 0 posiciones, en la fila 1 se desplaza 1 posición a la izquierda, en la fila 2 se desplaza 2 posiciones y 3 posiciones en la fila 3.

En la figura se refleja el estado anterior y posterior a esta transformación.



1.2.6.5. TRANSFORMACIÓN MIXCOLUMNS

En esta transformación se multiplica cada columna del estado por una matriz, lo cual permite maximizar la difusión.

Hay que tener en cuenta que cada byte se trata como un polinomio en GF(2^8).

La multiplicación quedaría de esta manera:

$$\begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} \quad \forall columna\ j$$

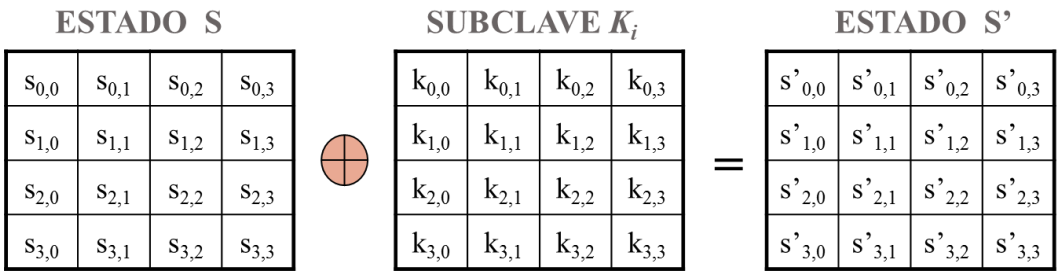
Los coeficientes de la matriz están expresados en hexadecimal.

Para realizar la multiplicación debe tenerse en cuenta que al multiplicar un byte por 01 da como resultado el mismo byte. Multiplicar por 02 equivale a realizar un desplazamiento del byte un lugar a la izquierda rellenando con ceros a la derecha. Multiplicar por 03 implica desplazar un lugar a la izquierda el byte y sumarlo módulo 2 consigo mismo.

En todos los casos, si el resultado obtenido tiene más de 8 bits, éste se reduce aplicando la operación de mod ($x^8+x^4+x^2+x+1$).

1.2.6.6. TRANSFORMACIÓN ADDROUNDKEY

En esta transformación se suman módulo 2, bit a bit, los bits del estado y los bits de la subclave de la vuelta correspondiente K_r.



1.2.6.7. EXPANSIÓN DE CLAVES

Este algoritmo genera, a partir de la clave inicial, la subclave inicial + *Nr* subclaves de vuelta.

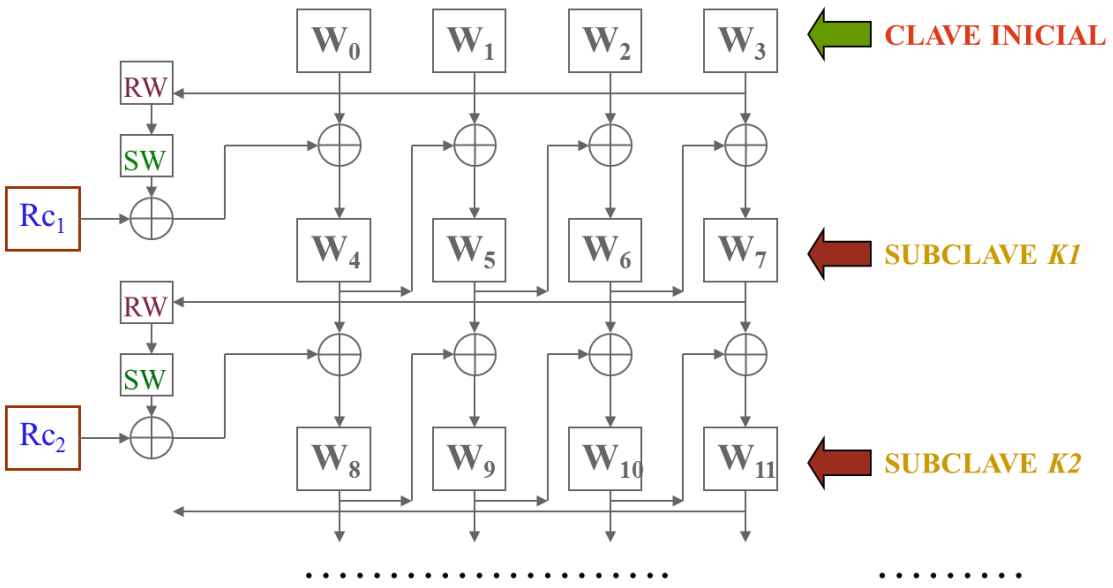
Ya que la clave que admite AES puede ser de longitud 128, 192 o 256 bits, el esquema varía en función de la longitud de la clave de cifrado.

En la expansión de claves se construye una secuencia (*Wi*) de 4(*Nr*+1) palabras de 32 bits.

Es decir, en función del número de vueltas, las palabras generadas son

- 128 bits clave -> 44 palabras
- 192 bits clave -> 52 palabras
- 256 bits clave -> 60 palabras

El esquema del algoritmo de expansión de claves para caso de clave de 128 bits (N_k =4) es el que se refleja en la figura:



En el esquema solo se muestran las primeras palabras. Por simplicidad no se muestran todas las palabras. El esquema seguiría de la misma forma hasta completar las 44 palabras totales.

En la figura se puede ver que el algoritmo de generación de palabras funciona de modo que las N_k palabras de la clave constituyen las primeras N_k palabras de la clave expandida.

El esquema de funcionamiento es diferente en función de si se trata de la primera columna o del resto. Las palabras de la segunda columna en adelante se generan realizando una operación XOR de la palabra que está justo encima y de la palabra anterior.

Sin embargo, en el caso de la primera columna el comportamiento es diferente. En este caso la palabra es el resultado de realizar XOR entre la palabra que se encuentra encima y una función de la palabra anterior (que se encuentra en la última columna de la fila anterior).

Veamos con más detalle en qué consiste esta función. Primeramente, se aplica RW sobre la palabra anterior, que realiza rotación cíclica a izquierdas de los 4 bytes de entrada. Es decir,

$$RW(b_0, b_1, b_2, b_3) = (b_1, b_2, b_3, b_0)$$

Posteriormente se realiza SW, aplicando la caja S a cada uno de los 4 bytes de entrada (transformación SubBytes)

$$SW(b_1, b_2, b_3, b_0) = (b_1', b_2', b_3', b_0')$$

Finalmente se realiza una XOR de este resultado con Rc_i, que es una constante de 32 bits que depende del número de vuelta.

Por tanto, la generación de las palabras de la primera fila queda como sigue:

$$W(i) = RC_i \oplus (RW\ (SW\ (W_{i-1})))$$

1.2.6.8. DESCIFRADO

Para descifrar AES se realiza la inversa de cada operación, en el orden inverso.

En el caso de *SubBytes*, habría que buscar el valor en la caja S y obtener los bytes xy. Para mayor agilidad se suele utilizar una caja que hace la transformación inversa a la caja S.

En *ShiftRows* se realizaría la permutación inversa, para volver al estado inicial.

Respecto de *MixColumns*, se invierte la transformación multiplicando cada columna por esta matriz:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

AddRoundKey se invierte restando módulo 2, bit a bit.

1.2.6.9. ASPECTOS PRÁCTICOS

Vamos a seguir analizando los aspectos prácticos del cifrado simétrico con la librería Cryptography, en este caso con el algoritmo AES.

El esquema es muy similar al anterior.

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
backend = default_backend()
key = os.urandom(32)
cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=backend)
encryptor = cipher.encryptor()
cryptogram = encryptor.update(b"a secret message") + encryptor.finalize()
print(cryptogram)
```

Se está utilizando una clave de 256 bits.

Uno de los resultados de la ejecución es el siguiente:

```
b'\xdcfwk\x001\xc88p\x8aO\x1f\x97Se\x9c'
```

Para descifrar se hace de esta manera:

```
decryptor = cipher.decryptor()
clearText=decryptor.update(cryptogram) + decryptor.finalize()
print(clearText)
```

Con lo cual volvemos a obtener el texto original

```
b'a secret message'
```

- En el caso de AES también se cumple el cambio en el criptograma al cambiar los bits de entrada.

Cifrando el texto "a secret message" con la clave "12345678901234567890123456789012" se obtiene

```
b'\xb6b\x18\xd7G0~\xf7\xd6EP\xf5o\xa5\xdc\x16'
```

Cifrando el texto "a secreu message" con la misma clave el resultado es

```
b'\xf1\xd6\xab\xf4\xa0\xce\x08H\\\xb3H\xb8\x08\x8e\x82\xb5'
```

- Hacemos la misma comprobación cambiando un bit en la clave.

Al cifrar el texto "a secret message" con la clave "12345678901234567890123456789011" el criptograma es

```
b'g\xc7\xa2Y\xc2#\xc0\nG\xf3H\xf2\xde.@;'
```

Puede verse cómo en ambos casos el resultado varía respecto del criptograma anterior. Pueden observarse también las diferencias con los criptogramas obtenidos con el algoritmo DES.

1.3. Modos del cifrado en bloque

Es importante tener en cuenta que solo se debe usar cifrado en bloque directamente para cifrar información reducida. Esta práctica es inadecuada cuando se trabaja con grandes cantidades de datos. La razón es que en mensajes largos se darán muchas repeticiones, lo cual facilita un criptoanálisis en el que mediante técnicas estadísticas se pueda deducir el mensaje y comprometer el criptosistema.

Para mensajes largos el método de cifrado admisible es el cifrado en flujo, ya que el criptograma es diferente, aunque haya repeticiones en el texto claro.

Para poner solución a esta situación referente al cifrado en bloque lo que se hace es construir una arquitectura que se asemeje al esquema de cifrado en flujo y en la que se tome el bloque como elemento básico.

El NIST define estos cinco modos de cifrado en bloque:

- Electronic Code Book (ECB).

- Cipher Block Chaining (CBC).
- Cipher Feedback (CFB).
- Output Feedback (OFB).
- Counter mode (CTR).

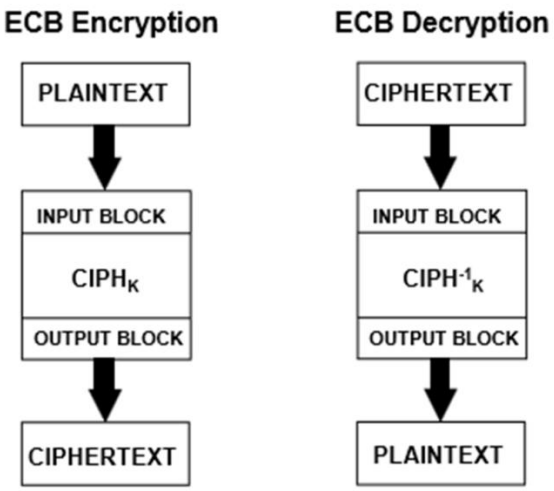
A continuación, se explica cada uno de estos modos de cifrado.

1.3.1.Electronic Code Book (ECB)

De esta manera se denomina a este modo, que realiza el cifrado de modo directo.

Por tanto, está reservado para conjuntos pequeños de datos. Por lo comentado anteriormente no debe usarse para mensajes largos.

El esquema de este modo puede verse en la siguiente figura.



Como se observa en la figura, es un modo de cifrado **directo**, es decir, en el cifrado se aplica la función de cifrado directa e independientemente sobre cada bloque del texto plano, obteniendo como bloque de salida resultante el bloque del criptograma correspondiente.

En el descifrado sucede lo contrario, se aplica la función de descifrado al criptograma, obteniendo el correspondiente texto claro.

Este modo tiene las siguientes características:

- Si sucediera un error de transmisión, éste afectaría a un solo bloque (aproximadamente la mitad de los bits cambiarían de valor).
- Se puede cifrar y descifrar en paralelo.
- Se puede descifrar un bloque sin descifrar el anterior o posterior.
- Si el texto claro tiene bloques idénticos, el texto cifrado tiene los bloques correspondientes idénticos.
- Sólo se pueden cifrar bloques completos (en caso de que no estuvieran completos, éstos se completarían con ceros).

Los siguientes modos de cifrado están diseñados para evitar una de las desventajas de este modo: que bloques iguales queden cifrados de la misma forma.

1.3.2.Cipher Block Chaining (CBC)

En el modo CBC se divide el texto claro en *N* bloques. La salida de un bloque se usa para alimentar la entrada del siguiente. Este modo requiere de un vector de inicialización (VI) para combinarlo con el primer bloque de texto claro. VI no necesita ser secreto, pero sí ser impredecible, por lo que se le asignar un valor elegido de forma aleatoria. VI debe ser del mismo tamaño que el bloque (*b* bits).

En la figura se puede ver que el cifrado se realiza así:

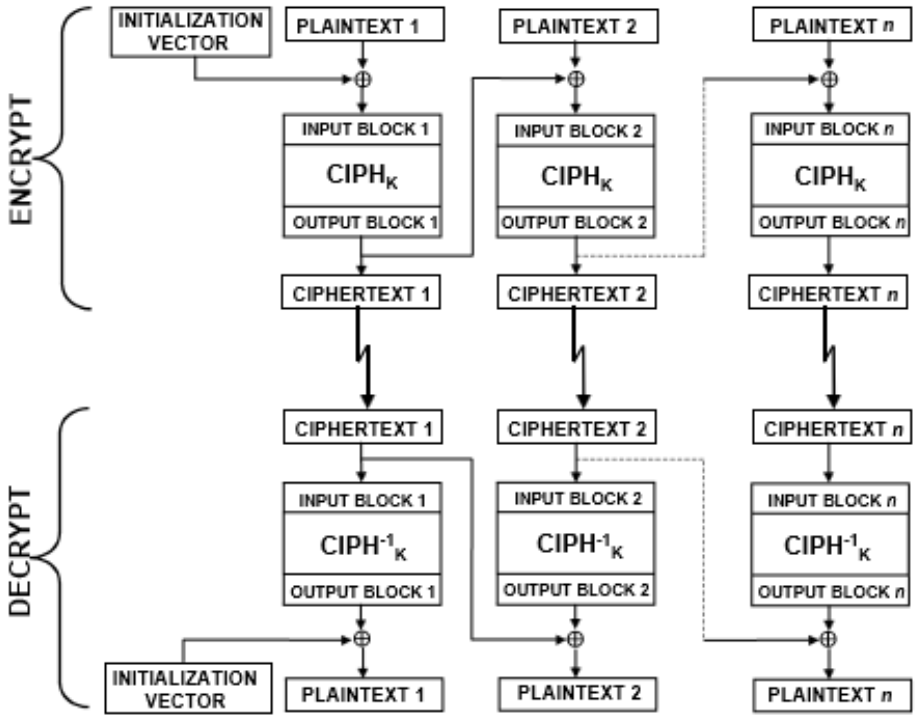
En el primer paso, la entrada a la función de cifrado está compuesta por el resultado de aplicar la función XOR al primer bloque de texto claro y el vector de inicialización. El resultado de cifrar este bloque da como resultado el primer bloque del criptograma.

En los subsiguientes pasos, la entrada a la función de cifrado está formada por la operación XOR del correspondiente bloque de texto plano y el bloque del criptograma obtenido en el paso anterior. Es decir, el criptograma de un paso se utiliza para alimentar la entrada del siguiente. Tras aplicar la función de cifrado sobre este bloque se obtiene el bloque del criptograma correspondiente al paso en el que se encuentre

el algoritmo.

El descifrado del primer bloque del criptograma se lleva a cabo aplicando, en primer lugar, la inversa de la función de cifrado sobre dicho bloque. A continuación, se realiza la XOR entre el vector de inicialización y el resultado, para deshacer la operación del cifrado y obtener el bloque del texto original.

En los siguientes pasos, el descifrado se realiza aplicando la inversa de la función de cifrado sobre el bloque del criptograma correspondiente. Al resultado de esta operación se le aplica la operación XOR junto con el bloque del criptograma del paso anterior, de modo que se recupera el bloque del texto claro correspondiente.



El modo CBC tiene las siguientes características:

- Un error afecta al bloque y al siguiente.

- Es un cifrado autosincronizante. Puede empezarse a descifrar a partir de cualquier punto.
- El resultado de cifrar un mismo mensaje es diferente si se cambia el VI.
- No se puede cifrar en paralelo.
- Se puede descifrar en paralelo.
- El último bloque es función de todos los bloques en claro. Esto es importante ya que se puede usar para autenticar el mensaje. Por eso al último bloque se le denomina Message Authentication Code (MAC).

El MAC puede enviarse junto con el mensaje para autenticarlo. El esquema sería el siguiente:

El emisor envía un mensaje junto con su MAC. Cuando el receptor lo recibe, realiza también el cálculo del MAC y puede comprobar si coincide con el que le envió el receptor. Si hay coincidencia es una garantía de que el mensaje no fue modificado durante el envío (garantiza la integridad del mensaje). Como estamos en un esquema de cifrado simétrico, es necesario que emisor y receptor compartan la clave.

1.3.3.Cipher Feedback (CFB)

El modo CFB, a diferencia del anterior que divide el mensaje en bloques de b bits, lo divide en segmentos de s bits, siendo $1 \leq s \leq b$. Los tamaños recomendados para s son 1, 8, 64, 128 bits. De hecho, a veces se hace referencia al valor de s de esta manera: por ejemplo, modo 8-bit CFB.

En este caso el criptograma anterior también se toma como entrada para la siguiente etapa, pero la operación XOR se realiza sobre el bloque de salida y no en el de entrada.

En este modo también se utiliza un VI. Como en el caso anterior, el VI no necesita ser secreto, pero sí ser impredecible, por lo que se usa un valor aleatorio.

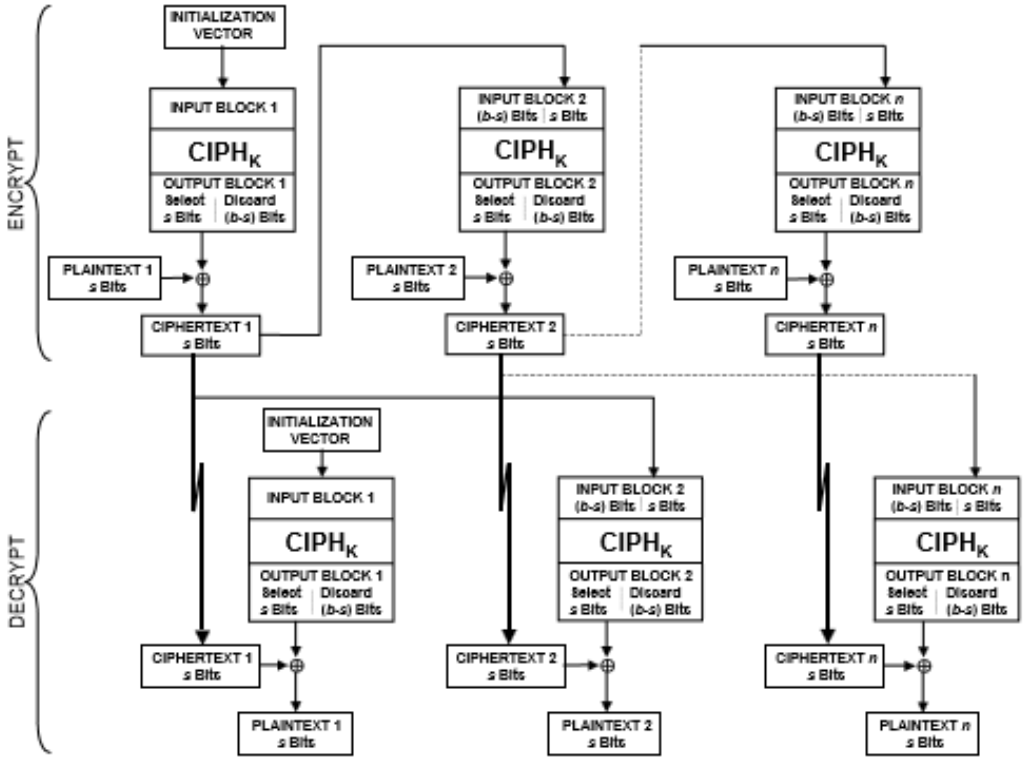
La figura refleja el modo de funcionamiento del modo CFB. En el cifrado, para generar

el primer segmento cifrado se cifra el vector de inicialización. De éste, se seleccionan los *s* bits más significativos y el resto se descarta. Estos *s* bits seleccionados se operan mediante una operación XOR con los *s* primeros bits del mensaje en claro.

Este segmento de criptograma sirve como input para la siguiente etapa, de modo que en adelante es este segmento el que se cifra, en lugar del VI. Para obtener los subsiguientes segmentos cifrados se procede de la misma manera que en la primera etapa, escogiendo los s bits más significativos y aplicando una XOR con los *s* bits del segmento correspondiente del texto claro.

El descifrado deshace los pasos realizados en el proceso de cifrado. Para descifrar el primer segmento, se cifra el VI, del cual se toman los *s* bits más significativos, y se realiza una XOR con los *s* bits del primer segmento del criptograma.

En las siguientes etapas, en lugar del VI, la entrada a la función de cifrado es el segmento del criptograma de la etapa anterior. Los *s* bits más significativos se combinan mediante una XOR con los *s* bits del segmento del criptograma correspondiente.



Las propiedades de este modo son:

- Un error de transmisión afecta al segmento y a los *b/s* siguientes
- Se puede empezar a descifrar en cualquier punto.
- El mismo mensaje se cifra diferente cambiando el VI.
- No se puede cifrar en paralelo.
- Se puede descifrar en paralelo.

1.3.4.Output Feedback (OFB)

La idea de este modo de cifrado es imitar la estructura del cifrado en flujo. Es decir, aplicar una operación XOR del mensaje y la clave de cifrado. Para ello, el VI se cifra *N* veces, siendo ésta utilizada como la secuencia cifrante.

Por esta razón, en este modo es importante que el VI se utilice únicamente una vez para cada ejecución del modo con la clave dada. Si se usara el mismo VI para cifrar varios mensajes la confidencialidad del mensaje podría quedar comprometida. De hecho, la confidencialidad podría quedar comprometida si cualquiera de los bloques de entrada a la función de cifrado coincide con el VI para el cifrado de otro mensaje bajo la misma clave.

La figura representa el funcionamiento de este modo. A diferencia los anteriores, que usaban el criptograma para alimentar la siguiente etapa, en este caso es el VI cifrado lo que se usa como entrada de la función de cifrado, de modo que se vuelve a cifrar en el siguiente paso.

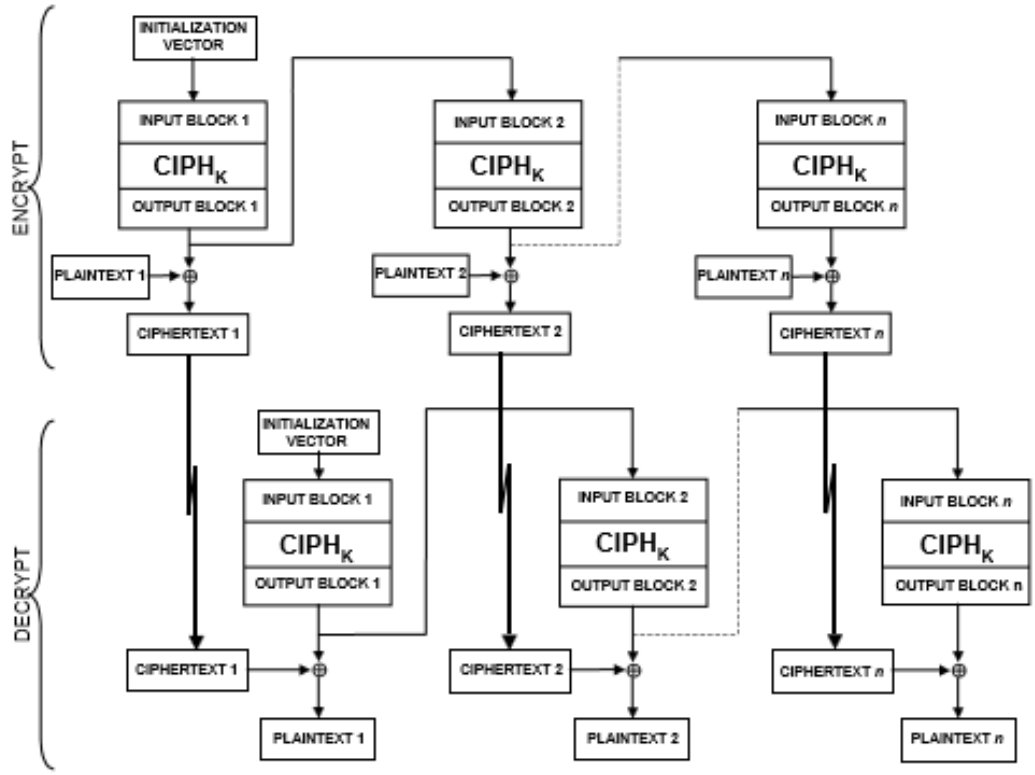
En el primer paso del cifrado se hace una XOR del VI cifrado con el primer bloque del texto claro. La salida de la función de cifrado se utiliza para alimentar a la función de la siguiente etapa. De ahí en adelante el criptograma se obtiene como resultado de la operación XOR aplicada sobre el bloque correspondiente del texto claro con el resultado de la función de cifrado de esa etapa. La entrada de la función de cifrado de una etapa es la salida del bloque cifrado en la etapa anterior.

En el caso de la última etapa, se escogen los *u* bits más significativos de la secuencia cifrante para realizar la operación XOR.

Para descifrar y obtener de nuevo el texto claro, en el caso del primer bloque, se realiza una operación XOR con el texto cifrado y el VI cifrado. En las siguientes etapas, lo que se cifra es la salida de la función de cifrado de la etapa anterior. Y se utiliza como entrada de la función XOR junto con el texto cifrado para obtener el texto claro. Se puede observar que el descifrado también sigue el esquema de descifrado en flujo.

De esta forma, el proceso de cifrado y descifrado son similares, en un caso haciendo la XOR con el texto plano (junto con la secuencia cifrante) para obtener el criptograma y en el otro utilizando el criptograma para obtener el texto claro.

Igual que en el cifrado, en la última etapa se escogen los *u* bits más significativos de la secuencia cifrante para realizar la operación XOR.



El modo OFB tiene las siguientes particularidades:

- Los errores de transmisión no se propagan.
- No es autosincronizante.
- El mismo mensaje se cifra diferente cambiando VI.
- No se puede cifrar ni descifrar en paralelo.
- El proceso de cifrado y descifrado usan la misma operación, no la inversa.
- La estructura emula la del flujo, pero en este caso no se usa una secuencia pseudoaleatoria, sino una permutación aleatoria (carece de repeticiones).
- Dado que se trata de una permutación aleatoria, la longitud de la secuencia debe limitarse a $n \leq \sqrt[4]{m}$, siendo $m = 2^b$.

1.3.5.Counter mode (CTR)

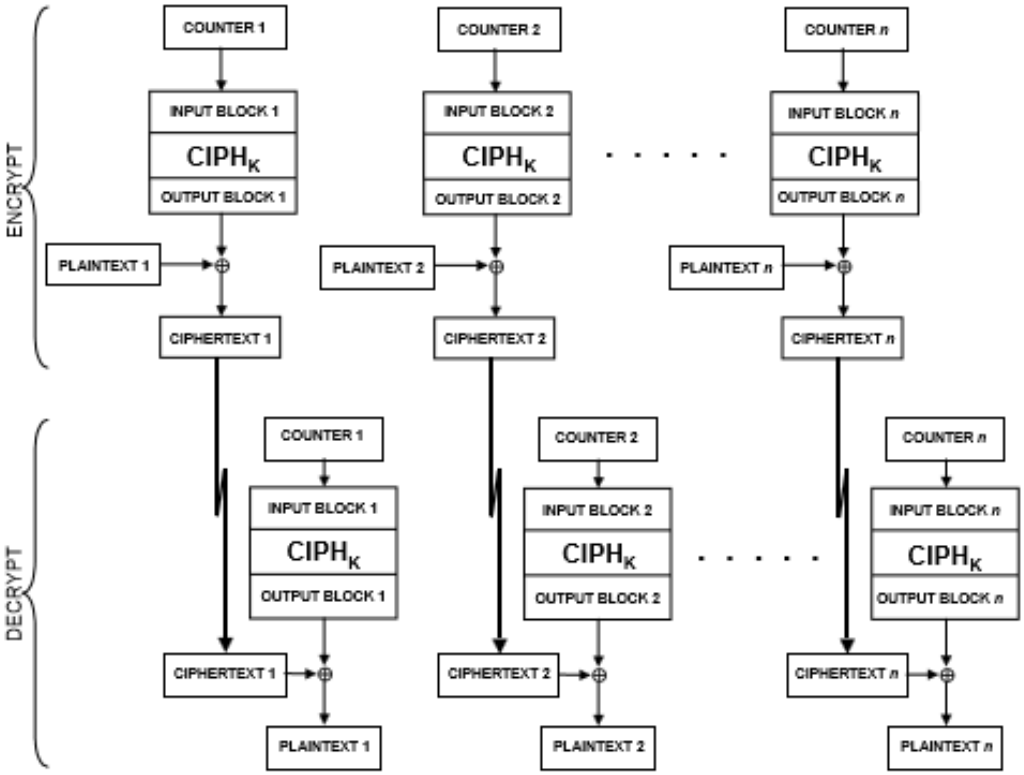
De forma similar que, en el caso anterior, el modo CTR sigue un esquema que lo asemeja al cifrado en flujo. En este caso se utiliza una secuencia de contadores para generar la secuencia cifrante, en la cual todos los contadores deben ser diferentes. Es más, para todos los mensajes cifrados bajo la misma clave, todos los contadores deben ser diferentes.

Una posible implementación de los contadores es escoger contador 1 como un número de un solo uso y estableciendo los sucesivos contadores como incrementos del primer contador.

El funcionamiento de CTR se puede ver en la figura. Para cifrar, en cada etapa se realiza XOR del contador de la etapa actual cifrado y el texto claro. La longitud de la palabra del contador tiene el mismo tamaño que el bloque.

En el caso de la última etapa, se escogen los *u* bits más significativos de la secuencia cifrante y el resto se descartan.

El descifrado se realiza la operación XOR del contador correspondiente a la etapa con el bloque del criptograma. Igual que en el cifrado, en la última etapa se escogen los *u* bits más significativos de la secuencia cifrante y el resto se descartan.



Éstas son las características del modo de cifrado CTR:

- Los errores no se propagan.
- No es autosincronizante.
- El mismo mensaje se cifra diferente cambiando el primer contador.
- Se puede cifrar y descifrar en paralelo.
- Es equivalente al cifrado en flujo, donde la secuencia cifrante son los números del contador cifrados.
- Al ser la secuencia cifrante una permutación aleatoria, la longitud de la secuencia debe limitarse a $n \leq \sqrt[4]{m}$, siendo $m = 2^b$.

Más detalles sobre los modos de cifrado se pueden consultar en la publicación especial del NIST 800-38, que se puede encontrar en los materiales complementarios de este módulo.

1.3.6. Aspectos prácticos

Vamos a estudiar varios aspectos sobre los modos de cifrado.

- En primer lugar, vamos a ver la diferencia en el resultado al utilizar los diferentes modos de cifrado.

En la librería Cryptography, el modo de cifrado se indica cuando se define el cifrador.

Como veíamos en el caso de AES, al aplicar el modo directo (ECB) para cifrar el texto "a secret message" con la clave "12345678901234567890123456789012" se obtiene:

```
b'\xb6b\x18\xd7G0~\xf7\xd6EP\xf5o\xa5\xdc\x16'
```

Como se vio, este modo no es el más recomendado y solo debe usarse para textos cortos. Vamos a ver la diferencia al cifrar el mismo texto con la misma clave, pero con otro modo de cifrado, por ejemplo, con CBC.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
backend = default_backend()
key = bytes(str('12345678901234567890123456789012'), 'ascii')
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
encryptor = cipher.encryptor()
cryptogram = encryptor.update(b"a secret message") + encryptor.finalize()
print(cryptogram)
```

El VI se ha inicializado con un valor aleatorio.

Se observa que el resultado obtenido con los diferentes modos varía.

- Observa que, en el modo CBC, si se cifra varias veces el mismo texto con la misma clave, el criptograma es diferente en cada ejecución. Es debido a que el vector de inicialización tiene un valor aleatorio.

Estos son ejemplos de criptogramas obtenidos en varias ejecuciones

```
b'N\xc5\xe4\xdd\xc8\xa6\xf1\x0b\xf5\xe3\x00\xf8\r\xd5)\xf5'

b'\xf7gj\xf3\x1f\xc0\xc9\xeb\xb3\x0f\xfcnH\xfe\xf9x'

b'\x0f\xf6\x95171\xf5\x0cU\xd2\xdb\x87\xbc\x98\xb0\xa8'
```

Sin embargo, al descifrar todos ellos se sigue obteniendo el mismo texto original.

```
b'a secret message'
```

Hay que mencionar que, si se usa un cifrador de flujo, o se usa un método de cifrado como CTR, no es necesario preocuparse por que el texto se ajuste al tamaño del bloque, por lo que no es necesario aplicar padding.

1.4. Funciones hash

En este apartado se estudian las funciones hash, revisando las más destacadas y analizando su nivel de seguridad.

1.4.1. Conceptos

Antes de entrar en qué son las funciones hash, se definen algunos conceptos.

Las funciones unidireccionales se definen como

$f: X \rightarrow Y,\ f(x) = y.$

Tienen la propiedad de que $f(x)$ es fácil de calcular y $f^{-1}(y)$ difícil, siendo computacionalmente intratable el recuperar el mensaje original.

x FÁCIL → f(x)
x DIFÍCIL ← f(x)

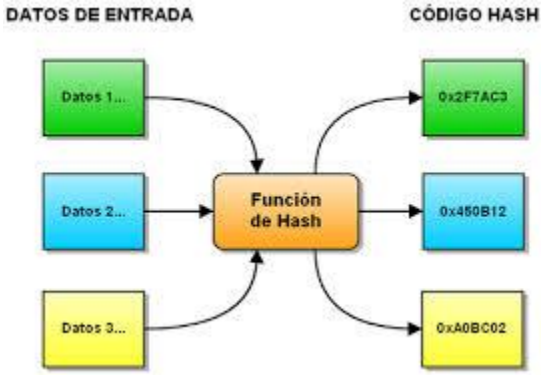
Las funciones unidireccionales con trampilla son aquellas en las que, si se conoce alguna información adicional *t*, se puede calcular en poco tiempo un $x \in X$ tal que $x = f_t - 1(y)$, para cualquier $y \in Y$.

Una función hash es una función unidireccional con trampilla que se aplica a un mensaje *M* de tamaño variable y da un resumen que es siempre del mismo tamaño. Es decir, una de las propiedades fundamentales de una función hash es que el resumen o salida que proporcionan es de longitud fija, independientemente de la longitud del mensaje de entrada.

$H(M) = m.$

Una función hash criptográfica es una función hash a la que se le piden requisitos extra que la hagan comportarse como una función aleatoria, determinista y eficiente.

Dado un mensaje, las funciones hash hacen corresponder un mensaje con un código de longitud fija. La longitud del resumen que proporcionan las funciones hash es uno de los parámetros que las definen.



Además, para que las funciones hash sean seguras deben cumplir la propiedad de que no sea fácil computacionalmente encontrar dos mensajes diferentes cuyo resumen sea el mismo, es decir, que no tengan colisiones. Este concepto se explica en el siguiente apartado.

1.4.2.Condiciones de las funciones hash

Para que las funciones hash sean seguras y se puedan utilizar convenientemente deben cumplir una serie de requisitos:

- **Dependencia de bits**: el resumen debe depender de todos los bits de la clave. Si se cambia un bit, cambian de media la mitad de los bits.

- **Resistencia a la preimagen**: dado un resumen *m*, debe ser computacionalmente difícil obtener *M* de modo que $H(M) = m$.
- **Resistencia a la segunda preimagen**: dado un mensaje cualquiera *M*, debe ser computacionalmente difícil encontrar otro mensaje *N* cuyos resúmenes coincidan ($H(M) = H(N)$).
- **Resistencia a colisiones**: debe ser computacionalmente difícil encontrar una colisión, es decir, determinar dos mensajes cualesquiera *M* y *N* cuyos resúmenes coincidan.

Analicemos con un poco más de detalle las dos últimas condiciones. Aunque se parezcan, no son iguales. La condición de resistencia a colisiones es más débil que la de resistencia a la segunda preimagen. La diferencia entre ellas es que en la segunda puede tratarse de dos mensajes cualesquiera, sin embargo, en la prmera, conocido un mensaje, se trata de encontrar otro diferente con el mismo resumen.

1.4.3.Longitud del resumen

Hemos comentado que la longitud del resumen que proporciona la función hash es un parámetro característico de la misma, pero cabe preguntarse, ¿cuál es la longitud adecuada para el resumen? La longitud del resumen debe seleccionarse adecuadamente de modo que sea usable y no excesivamente largo, pero al mismo tiempo seguro, evitando colisiones y cumpliendo con los requisitos de las funciones hash.

Para responder a esta cuestión vamos a realizar algunos cálculos.

Si un año tiene 365 días, ¿cuántas personas debe haber en una sala para que la probabilidad de que el cumpleaños de una de ellas sea el mismo que el mío supere el 50%?

Haciendo unos cálculos probabilísticos sencillos sabemos que si hay una persona en la sala la probabilidad es

$$P = \frac{1}{365}$$

Cuando hay dos personas, se calcula la probabilidad de que la fecha de cumpleaños no coincida con el de la primera persona ni con el de la segunda. De modo que se tendría

$$P = 1 - \frac{364}{365} * \frac{364}{365}$$

Extrapolándolo a *n* personas se tendría:

$$P = 1 - \left(\frac{364}{365}\right)^n$$

El resultado es que para que P sea ≥ 50%, debe haber al menos 253 personas en la sala.

Sin embargo, reformulando la pregunta, ¿cuántas personas debe haber en una sala para que la probabilidad de que el cumpleaños de dos de ellas sea el mismo día supere el 50%?

Cuando hay dos personas en la sala, la probabilidad de no cumplir años el mismo día es

$$P = \frac{364}{365}$$

Cuando hay tres personas, la probabilidad de no compartir fecha con ninguna de las otras dos es

$$P = \frac{364}{365} * \frac{363}{365}$$

La fórmula general para *n* personas sería

$$P = 1 - \left(\frac{364 * 363 * \ldots * (365 - n + 1)}{365^{n-1}}\right)$$

Curiosamente el resultado en este caso es que P ≥ 50% cuando *n* es al menos 23. El resultado es sorprendentemente menor que el caso anterior. Con este ejemplo se ve que es más fácil encontrar colisiones de lo que inicialmente se pueda pensar. Esto se conoce como la **paradoja del cumpleaños**. Esta paradoja puede extrapolarse a las colisiones en una función hash y da lugar a lo que se conoce como el ataque de la paradoja del cumpleaños.

De esta forma puede reducirse la complejidad de la búsqueda de colisiones.

Lo que se hace es generar aleatoriamente muchos valores y luego se busca entre ellos una pareja tal que *H(M)=H(N)*.

Para resúmenes muy largos, la cantidad se puede reducir en su raíz cuadrada, reduciendo drásticamente la capacidad de cálculo necesaria.

Para evitar esto es por lo que se pide a las funciones hash la condición de que sean resistentes a colisiones.

1.4.4.Estructura

En general, la estructura de las funciones hash es iterativa. Habitualmente la entrada en el paso *i* es función del *i*-ésimo bloque del mensaje y de la salida del paso anterior. Es frecuente incluir en algún bloque la longitud del mensaje para evitar colisiones.

1.4.5.Seguridad de las funciones hash

Dado que el resumen que proporcionan las funciones hash es de tamaño fijo, y los mensajes que puede recibir como entrada son infinitos, es difícil conseguir que en ningún momento se dé una colisión (dos mensajes con el mismo resumen). Si esto sucediera se daría la situación de que un valor de la función hash no se correspondería con un único mensaje.

La seguridad de las funciones hash se basa en la dificultad de, dado M, se pueda encontrar un N tal que *H(M)=H(N)*.

Un atacante podría llevar a cabo varias estrategias para intentar encontrar colisiones:

- De **preimagen**: Dado un mensaje M, el atacante calcula otro tal que $H(M) = H(N)$. De esta forma, es posible falsificar el mensaje usando *N* en lugar de *M*. Lo que es más complicado conseguir es que *N* sea un mensaje con sentido y bien construido, ya que lo más probable es que esté formado por valores aleatorios, lo cual podría llegar a delatar el ataque.
- De **colisión** propiamente dicha: El atacante busca dos mensajes que colisionen. En este caso no se parte de un mensaje dado, lo cual hace la falsificación más complicada.

A continuación, se hace un repaso de algunas de las funciones más importantes.

1.4.6.MD5

Esta función fue diseñada por Rivest en 1992. Es la sucesora de MD4. Esta función proporciona una salida de 128 bits, manipulando bloques de 512 bits.

Una de las razones de su popularidad es que se utilizó en las primeras versiones de PGP.

La estructura de MD5 es la siguiente:

- Se añaden los bits de relleno. Se añade un "1" y tantos 0 como sea necesario a M hasta

que la longitud e congruente con 448 mod 512.

- Se añade la longitud del mensaje. Se añade la longitud del mensaje representada con un entero de 64 bits. Si la longitud del mensaje es mayor que 2^64, solo se usan los bits de menor peso.

El resultado de estas operaciones tiene una longitud de múltiplo de 512 bits (y múltiplo de 16 palabras de 32 bits).

- Inicialización de cuatro registros de 32 bits.

palabra A: 01 23 45 67
palabra B: 89 ab cd ef
palabra C: fe dc ba 98

palabra D: 76 54 32 10

- Funciones. Se procesa el mensaje en bloques de 16 palabras. Se calculan cuatro funciones que manejan palabras de 32 bits.

$F(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$
$G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$
$H(X, Y, Z) = X \oplus Y \oplus Z$
$I(X, Y, Z) = Y \oplus (X \vee (\neg Z))$

Rondas. En la ronda 1 se hacen estas 16 operaciones:
[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
Donde
[abcd k s i] representa la operación a = b + ((a + F (b, c, d) + X [k] + T [i]) <<< s).
Se realizan cuatro rondas. El resultado de la función hash es la concatenación de A, B, C, D.

Esta función lleva rota desde 1996, por lo que su uso está desaconsejado en la mayoría de las aplicaciones.

1.4.7.SHA-1

Fue propuesta por el NIST en 1995. La versión actual es del año 2002. Proporciona una salida de 160 bits, con bloques de 512 bits.

Es la sucesora de SHA-0, creada por el NIST en 1993 pero fue retirada por fallos de diseño.

SHA-1 fue bastante utilizada hace uños años, por ejemplo, en certificados digitales, aunque no tanto hoy en día por motivos de seguridad, ya que se han encontrado algunas debilidades en la función ya que es posible encontrar colisiones en menos de 2^{69} operaciones.

Su diseño es similar al de MD5, con algunas diferencias:

- Utiliza 5 registros en lugar de 4. Añade E = C3D2E1F0.
- Cada ronda tiene 20 operaciones, utilizando *F, H* y J,

J(X, Y, Z) = (X ∧ Y) ∨ (X ∧ Z) ∨ (Y ∧ Z) .

F se utiliza en la primera ronda, H en a segunda y cuarta, J en la tercera.

- Utiliza una constante para cada ronda

K0 = 5A827999
K1 = 6ED9EBA1
K2 = 8F1BBCDC
K3 = CA62C1D6

- El bloque de mensaje m se divide en 16 partes de 32 bits (m_0 ... m_{15}) y se expande a 80 trozos de 32 bits (w_0 ... w_{79}) así:

wt = mt para t = 0 a15
wt = (wt-3 ⊕ wt-8 ⊕ wt-14 ⊕ wt-16) << 1 para t = 16 a 79

- Aplica 80 vueltas a cada bloque de 512 bits del mensaje original (en vez de 64).

El bucle principal es
FOR t = 0 TO 79
i = t div 20
Tmp = (a << 5) + A(b, c, d) + e + wt + Ki
e = d
d = c
c = b << 30
b = a
a = Tmp

siendo *A* la función *F*, *H* o *J* según el valor de *t* (*F* para t ϵ [0, 19], *H* para t ϵ [20, 39] y [60, 79], *J* para t ϵ [40, 59]).

Los valores de *a* a *e* se suman a los registros *A* a *E* y el algoritmo continúa con el siguiente bloque de datos. El resultado de la función resumen es la concatenación

de los registros *A* a *E* resultantes de procesar el último bloque del mensaje.

- Utiliza *big endian* en lugar *de little endian*.
- La longitud máxima para SHA-1 debe ser menor de 2^{64} bits.

1.4.8.SHA-2

Se trata en realidad de una **familia** de algoritmos y no de uno solo.

Fue propuesta tras conocerse ataque a SHA- 1 con 2^{63} operaciones.

Fue propuesta por el NIST y es la sucesora de SHA-1.

Está compuesta por un conjunto de cuatro subalgoritmos: **SHA-224, SHA-256, SHA-384, SHA-512**. Donde los números indican la longitud del resumen que proporcionan como salida.

Una de las dificultades de adopción de esta función es que, aunque proporciona mayor seguridad que SHA-1, se usa menos por la falta de compatibilidad con muchos protocolos.

Hoy en día SHA-2 se considera segura a medio y largo plazo.

1.4.9.SHA-3

A diferencia de lo que pasó con otras funciones, el nacimiento de SHA-3 no fue el de reemplazar a SHA-2, ya que no se han encontrado ataques significativos a esta función. Sin embargo, el NIST lanzó un concurso para ofrecer una alternativa a esta función.

En 2006 el NIST lanzó este concurso, con los siguientes requisitos:

- Buen rendimiento independientemente de la implementación.
- Mínimo consumo de recursos incluso para textos largos.
- Proporcionar una buena seguridad.

- Ser resistente a ataques conocidos.
- Generar resúmenes de 224, 256, 384 y 512 bits. Incluso proporcionar tamaños más largos si fuera necesario.

En octubre 2012 Keccak se seleccionó como el ganador de la competición. Este algoritmo fue diseñado por Guido Bertoni, Joan Daemen, Michaël Peeters, y Gilles Van Assche.

En 2014 el NIST publicó SHA-3. Y en agosto de 2015 el NIST anunció SHA-3 como un estándar de hashing.

1.4.10. Notas sobre seguridad

Dado que se han encontrado colisiones usando las técnicas de colisiones en MD5, SHA-0, SHA-1, una de las medidas de seguridad que se puede aplicar cuando se use estas funciones es utilizar la concatenación de dos resúmenes obtenidos con algoritmos diferentes, de modo que para vulnerarlo sería necesario encontrar dos mensajes que colisionaran en ambos casos, lo cual reduce en gran medida la probabilidad de ataque.

1.4.11. Funciones HMAC

El cifrado garantiza la confidencialidad, es decir, que los datos solo pueden ser leídos y modificados por las personas autorizadas. Sin embargo, solo cifrando el mensaje no se puede garantizar su autenticidad o integridad, esto es, que los datos estén completos y no hayan sido modificados durante la transmisión, como podría suceder en un ataque Man-in-The-Middle (MiTM), en el que el atacante se sitúe en medio de la comunicación y modifique los datos.

Para aportar autenticidad se usan los códigos de autenticación de mensajes o MAC.

Los hay de varios tipos:

- **Basados en cifrado en bloque**. El MAC es la última parte del criptograma cuando todos los bits del criptograma son función de todos los bits del mensaje original. En el apartado referente a los modos del cifrado en bloque se vio que el modo CBC se puede usar como MAC ya que el último bloque es función de todos los anteriores.
- **HMAC**. Para el cálculo del MAC se usa una función resumen. Antes de aplicar la función hash se calculan una serie de bits, dependientes de la clave a utilizar, y se añaden al mensaje. La seguridad de la función hash utilizada influye en la seguridad de HMAC.

Uno de los HMAC más utilizados es el ANSI 9.71 (ANSI y NIST). Esta función utiliza una clave de 512 bits. El HMAC se calcula como

$HMAC_k(M) = H((k \oplus opad) || (k \oplus ipad) || M)$,

donde *k* es la clave (alargada con ceros por la derecha hasta tener 64 bytes de longitud), *opad* es el byte con valor hexadecimal 5C repetido 64 veces, *ipad* es el valor hexadecimal 36 repetido 64 veces, M es el mensaje, y || representa la concatenación.

1.4.12. Otras funciones hash

Hay otras funciones hash, como pueden ser:

- RIPEDM-160
- Panama
- Tiger
- CRC32
- Etc.

1.4.13. Aplicaciones

Las funciones hash tienen muchas e importantes aplicaciones. A continuación, se mencionan algunas de ellas:

- Firmas digitales. En lugar de firmar un mensaje directamente, se firma un hash del mismo.

- Certificados digitales. X509 v3 incluyen MD5 y SHA-1 para las firmas digitales.
- DNIe. SHA-1, SHA-2. Sólo operativa la primera por razones de compatibilidad.
- Integridad de datos. Calcular resumen de ficheros y guardarlo fuera del PC. Contrastar si coinciden los resúmenes.
- Antivirus.
- Protocolos de seguridad.

1.4.14. Aspectos prácticos

En este apartado vamos a ver algunos aspectos prácticos en relación a las funciones hash, incluyendo el cómo calcularlas y algunas de sus aplicaciones.

1.4.14.1. CÁLCULO DE FUNCIONES HASH

Vamos a ver de qué manera es posible emplear las funciones hash mediante la librería Cryptography de Python. Más información sobre esta librería se puede encontrar aquí:

https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/

Éste es un ejemplo de cómo se calcula la función hash SHA256 del mensaje “a secret message”.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
digest.update(b"a secret message")
hash=digest.finalize()
print('hash: ' , hash)
```

En primer lugar, se define la función hash a utilizar y se establece el backend (en este caso el de por defecto). Con update() se indica el texto (en bytes) del que quiere calcular

la función hash. Y por último se llama a finalize() para finalizer el conexto actual y devolver el hash calculado (en bytes).

El resultado de la ejecución del código anterior es:

```
hash:  b'\x8e=\x8e"\x10k\xbe\xed\xb5\xfd\xab5\xcfeT\x037X\xfb+\xfc\xa2\xe2S\x03\x88\x05\x9aLC\xff\x8a'
```

Vamos a estudiar algunas de las propiedades de las funciones hash:

- La longitud del texto devuelto por una función hash tiene una longitud fija, independiente de la longitud del texto a cifrar. Para ello calculamos el hash del texto “a”.

El resultado es:

```
hash:  b'\xca\x97\x81\x12\xca\x1b\xbd\xca\xfa\xc21\xb3\x9a#\xdcM\xa7\x86\xef\xf8\x14|Nr\xb9\x80w\x85\xaf\xeeH\xbb'
```

Como se puede ver, aunque el texto de entrada era más corto, el resultado sigue teniendo la misma longitud.

- Vamos a comprobar también el efecto avalancha en las funciones hash. Introduciendo el texto “a secreu message”, en el que solo cambia un carácter (en este caso un bit) respecto del anterior, se puede observar que el resultado de la función hash varía notablemente:

```
hash:  b'\xed\\A\x184\xe3\xcd\x0e\xcc\xb7\xfeR\xa0z\xb0Q\x04\x1b\x96\x00\x14\xe6\xac\x11\x00\x0e-\xdb\x8f\xe6\x8c%'
```

1.4.14.2. CÁLCULO DE HASH DE UN FICHERO

Una de las aplicaciones de las funciones hash es comprobar la integridad de los datos, lo cual es muy útil por ejemplo para comprobar que los ficheros que hemos descargado son correctos. Esto se suele hacer habitualmente calculando el hash md5 del fichero y comprobando el resultado con el proporcionado por la fuente.

En el caso de que nuestro cálculo no coincida con el proporcionado para el fichero podemos tener certeza de que el fichero que hemos bajado es incorrecto. Esto puede deberse a que el fichero se haya corrompido o modificado durante la transmisión o

ha podido ser suplantado por un fichero con virus.

1.4.14.3. MÉTODOS DE HASHING EN WINDOWS

Otra de los usos importantes de las funciones hash es su aplicación en el almacenamiento de contraseñas, para evitar almacenarlas en claro y protegerlas frente a posibles ataques.

En el caso de Windows, las contraseñas se almacenan en el archivo SAM. Windows tiene varios métodos para crear el hash, siendo éstos los principales:

- Método LM (Lan Manager). Este método fue desarrollado por IBM y Microsoft y Windows lo adoptó en los años 80. Este método tiene varias debilidades. Una de ellas es que se transforman todas las letras a mayúsculas, lo cual reduce el alfabeto y por tanto facilita los ataques de fuerza bruta. Además, la longitud de la contraseña está limitada a 14 caracteres. Quizás la mayor debilidad es que el algoritmo rellena con ceros hasta completar los 14 caracteres y divide la contraseña en dos bloques de 7 caracteres. Esto reduce ampliamente la complejidad de averiguar la contraseña. Es decir, una contraseña de 14 caracteres sería tan segura como dos de longitud 7, facilitando nuevamente la tarea a los atacantes. Cada uno de estos bloques se cifra junto con una constante conocida (4b47532140232425) y los concatena.

Si la contraseña es menor de 7 caracteres, la segunda mitad será siempre 0xAAD3B435B51404EE, lo cual facilita identificar este hecho.

A pesar de sus debilidades, la razón de mantenerla es por razones de compatibilidad con versiones anteriores, o en caso de contraseñas con longitud menor de 15 caracteres. Sin embargo, en Windows Vista no se almacena el hash LM por defecto.

- Método NTLM (NT Lan Manager). Algunas de las diferencias con el anterior es que añade caracteres Unicode y elimina la conversión a mayúsculas. Amplía la longitud de las contraseñas a 128 caracteres. Además, no se

divide la contraseña en dos partes, haciendo más complicados los ataques por fuerza bruta.

Cuando se establece o se cambia la contraseña de un usuario a una que contiene menos de 15 caracteres, Windows genera los hashes LM y NT. Dependiendo de la versión, LM puede almacenarse por defecto o no.

Hay que mencionar que, aunque se utilice NTLM, mientras se almacene también el hash LM, el sistema será tan débil como éste último método. Es decir, el sistema es tan seguro como su eslabón más débil.

En el caso de que la contraseña tenga más de 14 caracteres, no se almacena el hash LM ya el algoritmo no lo soporta. En ese caso se almacena la constante aad3b435b51404eeaad3b435b51404ee en su lugar. Esto podría ser una medida para ser más resistente a ataques de fuerza bruta. También es posible indicarle a Windows que no almacene el hash LM.