Práctica 2 Fundamentos de Criptografía y Seguridad Informática: Seguridad Perfecta y Criptografía Clásica

Respuesta a las cuestiones de la práctica 1 de Fundamentos de Criptografía y Seguridad Informática.

GRUPO-1462

Ignacio Núñez

Nicolás Victorino

Índice:

0. Introducción a la práctica

Antes de comenzar, cabe mencionar que en la explicación y discusión de cada método de encriptación usado se sigue una estructura común. En primer lugar, nos encontramos con el apartado "*Previo a la implementación*", donde se comentan los diferentes puntos o funciones a desarrollar previo a la implementación del método. Después nos encontramos con el apartado "*Implementación del algoritmo*", donde se explica la implementación del método. Finalmente, nos encontramos con el apartado "*Criptoanálisis y resultados*", donde se habla más a fondo de la seguridad del cifrado, y de cómo podemos ejecutarlos.

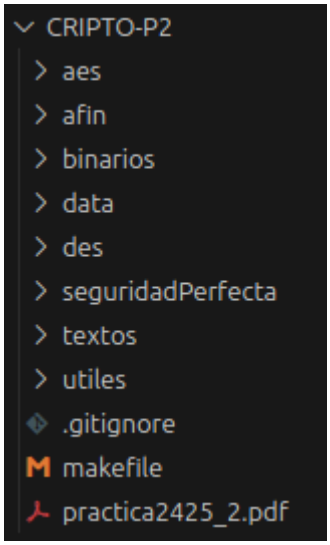
Sin embargo, en caso de no necesitarse se puede prescindir de alguno de ellos.

a. Lenguaje de programación escogido

Para realizar esta práctica, hemos decidido hacer uso del lenguaje de programación C, debido a su eficiencia y velocidad (se compila a código maquina optimizado) y a que es un lenguaje de bajo nivel, que permite mucha flexibilidad a la hora de implementar los algoritmos criptográficos y da un mayor control sobre la gestión de la memoria.

b. Organización de la práctica

La práctica está estructurada de la siguiente forma:



Como se puede observar, hemos decidido separar cada algoritmo de cifrado en carpetas, haciendo uso de un fichero *"utils.c"* general para todos los programas. Además, contamos con un makefile que se encarga de compilar y ejecutar todos ellos.

Hay 4 directorios extra:

- *Textos:* Contiene todos los textos que se usan de base para probar los algoritmos de cifrado. Además, aquí también podemos encontrar los archivos donde se guardan los textos cifrados, que por lo general se guardan en el archivo *"adios.txt"*.
- *Obj:* Contiene todos los archivos "*.o", necesarios para la compilación de los distintos programas.
- *Data:* Almacena los scripts para generar las gráficas y los datos generados por los programas.
- *Binarios:* Contiene los archivos binarios que vas a ser codificados por el algoritmo de cifrado DES.

c. Patrones seguidos en los distintos programas

- Función printExeInfo():

```
void printExeInfo() {
    printf("./afin {-C|-D} {-m |Zm|} {-a N+} {-b N+} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-a N+: clave de cifrado\n");
    printf("-b N+: clave de cifrado\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

En caso de haber un fallo a la hora de ejecutar un programa, todos ellos contienen esta función, que imprime el método correcto para hacerlo. En este caso, estamos viendo a modo de ejemplo la función del programa afín.

1. Seguridad Perfecta

a. Comprobación empírica de la Seguridad Perfecta del cifrado Afín:

Previo a la implementación:

Para el desarrollo de este apartada, hemos desarrollado múltiples funciones y estructuras auxiliares, además de reutilizar algunas de la práctica anterior.

Estructuras auxiliares y sus métodos:

- Cálculo de claves:

Hemos desarrollado la siguiente estructura, para almacenar las distintas claves posibles para un cifrado afín, además del número total de claves:

```
typedef struct {
    int *keys;
    int size;
} Keys;
```

En relación con esta estructura, hemos desarrollado las siguientes funciones:

- Función *'generateKeysado':* Dado el tamaño de un lenguaje (en nuestro caso *'LANGUAGE_SIZE'*), calcula las claves que sirven para ejercer como parte '*a*' de la clave en un cifrado afín, haciendo uso del algoritmo de Euclides. También guara el número total de claves útiles.

- Función *'generateKey*:' Escoge una de las claves obtenidas con el método anterior. Si el modo es 'P', llama a la función *'generateEquiprobableKeys',* y si el modo es 'I', llama a la función *'generateNonEquiprobableKeys'*.

```
void generateKey(Keys *keys, int*a, int*b, char *mode) {

    if (strcmp(mode, "-P") == 0) {
        generateEquiprobableKey(keys, a, b);
    } else {
        generateNonEquiprobableKey(keys, a, b);
    }
}
```

- Función '*generateEquiprobableKeys*': Dadas unas claves, elige aleatoriamente entre una de ellas.

```
*a = keys->keys[random_num(0, keys->size-1)];
*b = random_num(0, LANGUAGE_SIZE-1);
```

- Función '*generateNonEquiprobableKeys*': Dadas unas claves, le asigna un peso diferente a cada una de ellas. Esto lo hacemos dándole un peso 1 a la primera clave, y el resto de los pesos será igual al doble del peso de la clave anterior. Además, también calculamos la suma total de los pesos.

```
pesos[0] = 1;
int total = 1;

for (int i = 1; i < keys->size; i++) {
    pesos[i] = pesos[i-1] * 2;
    total += pesos[i];
}
```

Posteriormente, elegimos una clave teniendo en cuenta su peso. Esto lo hacemos generando un número aleatorio entre 0 y la variable 'total'. Con este valor, elegimos el índice del peso que más se aproxime.

```
/* Ahora uso estos pesos para elegir una clave aleatoriamente*/
int random = random_num(0, total);

int i = 0;
while (random > pesos[i]) {
    random -= pesos[i];
    i++;
}

*a = keys->keys[i];
*b = i;
```

Los pesos de cada clave con ambos métodos quedarían de la siguiente manera:

EQUIPROBABLE

KEYS	Pk(0)	Pk(1)	...	Pk(|k|)
PROBABILITY	1/|k|	1/|k|	...	1/|k|

NO EQUIPROBABLE

KEYS	Pk(0)	Pk(1)	...	Pk(|k|)
PROBABILITY	2^0	2^1	...	2^k

- Cálculo de probabilidades:

Hemos desarrollado la siguiente estructura, para almacenar la frecuencia absoluta, relativa y número total de caracteres de los elementos de un texto:

```
typedef struct {
    int *absoluta;
    float *relativa;
    int total;
} Probabilty;
```

No entraremos demasiado en detalle sobre el desarrollo, ya que lo consideramos trivial, pero contamos con métodos para crear esta estructura *(createProbability)*, liberar la memoria de la estructura *(freeProbability)* y calcular los valores mencionados dado un texto *(probabilityText)*.

- Cálculo de probabilidad condicional:

Hemos desarrollado la siguiente estructura, para almacenar la frecuencia absoluta, relativa y número total de caracteres de los elementos de un texto:

```
typedef struct {
    float **prob;
} ConditionalProbabilty;
```

Como se puede ver, simplemente encapsulamos un doble puntero de floats, para una mayor claridad en el código. Para esta estructura también contamos con un método para crearla *(createConditionalProbability)* y otro para liberar la memoria de la estructura *(freeConditionalProbability)*.

Para calcular estas probabilidades condicionades, hemos creado el método *'conditionalProbability'*, que dados 2 textos, calcula la probabilidad de tener un carácter en el primer texto dado otro carácter en el segundo texto. Es decir, calcula según lo visto en teoría $P_p(x|y)$.

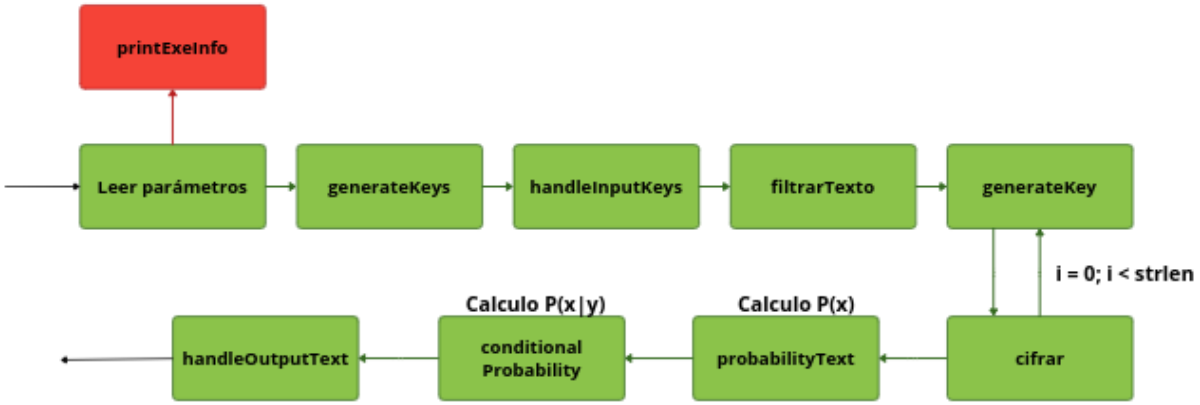
Funciones de la práctica anterior:

A lo largo de este programa, hacemos uso de los siguientes métodos, explicados y utilizados en la práctica anterior:

- 'cifrar': Método de cifrado afín.
- 'euclides2': Método de euclides que funciona con enteros.
- 'random_num': Generación de números aleatorios con límites.
- 'filtrarTexto': Limpia el texto de entrada.
- 'handleInputText': Recoge el texto de entrada dado el modo de entrada.
- 'handleOutputText': Extrae la salida del programa dado el modo de salida.

Implementación del algoritmo:

Ahora, vamos a explicar paso a paso como hemos integrado este método, siguiendo el siguiente esquema:



1. Parámetros necesarios para la ejecución del algoritmo:

```c
void printExeInfo() {
    printf("./seguridad_perfecta {-P|-I} [-i filein] [-o fileout]\n");
    printf("-P|-I: -P para equiprobable, -D para no equiprobable\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Hemos decidido mantener los parámetros de entrada propuestos por el enunciado. Como se puede ver, contamos con 1 parámetros obligatorios (“*-P|-I*”) y 2 opcionales *(“-i”, “-o”)*.

2. El siguiente paso es generar las posibles claves (‘*keys*’) para el tamaño de lenguaje especificado, 26 en nuestro caso:

```c
/*Generate possibles keys for the afin method*/
Keys *keys = generateKeys(LANGUAGE_SIZE);
```

3. Después, obtenemos el texto a cifrar, y lo filtramos para eliminar los espacios y todos los símbolos externos al lenguaje:

```c
/* Obtain text to encrypt*/
texto = handleInputText(filein);
if (texto == NULL) {
    return 1;
}

filtrarTexto(texto);
```

4. Cifro cada carácter del texto con una clave. La elección de claves puede ser equiprobable o no equiprobable, dependiendo del modo con el que se haya inicializado el programa:

```
/* Encrypt each character with a different key, depending on the mode it will equiprobable keys or not*/
for (int i = 0; i < strlen(texto); i++) {

    temp[0] = texto[i];

    generateKey(keys, &a_i, &b_i, modo);

    mpz_set_si(a, (long)a_i);
    mpz_set_si(b, (long)b_i);

    cifrar(temp, m, a, b);

    texto_cifrado[i] = temp[0];

}
```

5. Calculo la probabilidad de cada carácter en el texto original y las guardo en una variable (*'output'*) para mostrar en la salida del programa:

```
/*Get the probability of each character in the text*/
Probabilty *prob_original = probabilityText(texto);

/*Print the probabilities in the given output*/
for (int i = 0; i < LANGUAGE_SIZE; i++) {
    if (prob_original->relativa[i] == 0) {
        snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "Pp(%c)= %.2lf\n", i + 'a', 0.00);
        continue;
    }
    snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "Pp(%c)= %.2lf (%d / %d)\n", i + 'a',
    prob_original->relativa[i], prob_original->absoluta[i], prob_original->total);
}
```

6. Calculo la probabilidad condicional de cada carácter en el texto original dado un carácter del texto cifrado:

```
/*Get the conditional probability of each character in the text given its pair in the cipher text*/
ConditionalProbabilty *cond_prob = conditionalProbability(texto, texto_cifrado);

/*Print the conditional probabilities in the given output*/
for (int i = 0; i < LANGUAGE_SIZE; i++) {
    for (int j = 0; j < LANGUAGE_SIZE; j++) {
        snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "Pp(%c|%c)=%.2lf  ", i + 'a', j + 'a', cond_prob->prob[i][j]);
    }
    snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "\n");
}
```

7. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función *"handleOutputText"*, mencionada previamente.

```
/*Output the result of the program*/
if (handleOutputText(fileout, output) == -1) {
    return 1;
}
```

Criptoanálisis y resultados

Para probar este método, podemos hacer uso del makefile (‘*make run_seg-perf_P*’ para usar claves equiprobables, y ‘*make run_seg-perf_I*’ para claves no equiprobables). Por defecto, estos comandos utilizan el archivo de entrada ‘hamlet.txt’, y salida.txt’ como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta ‘textos’ si se desea, o probar con textos propios.

Ahora vamos a comentar la salida del programa con los 2 modos de generación de claves, además de con distintas longitudes de texto. En lugar de adjuntar la salida del programa en este documento, lo cual sería complicado, hemos decidido incluir en la carpeta *‘seguridadPerfecta’* la salida de algunas ejecuciones:

- Texto largo (hamlet.txt) y claves equiprobables:

Como podemos observar en el fichero *‘largo_P.txt’*, se cumple la condición de seguridad perfecta, donde: $P_p(x) = P_p(x|y)$, donde independientemente del carácter del texto cifrado, el carácter del texto original mantiene la misma probabilidad. Podemos ver que tener un margen de error de máximo un 1% arriba o abajo, algo dentro de lo normal.

- Texto largo (hamlet.txt) y claves no equiprobables:

Como se puede observar en el fichero *‘largo_I.txt’,* al contrario que con las claves equiprobables, no se cumple la condición de seguridad perfecta, no siendo independiente la P(x) de la P(y). Los valores tienden a valores más extremos al ser cifrados más veces le mismo carácter con las mismas claves.

- Texto corto (hola.txt) y claves equiprobables:

Como se puede observar en el fichero *‘corto_P.txt’*, no se cumple la condición de seguridad perfecta con claves equiprobables y un texto corto. Esto se debe a que no todos los caracteres se cifran con todas las claves, resultando en un cifrado sesgado.

- Texto corto (hola.txt) y claves no equiprobables:

Como se puede observar en el fichero *‘corto_I.txt’*, el resultado no difiere demasiado del resultado con el texto largo y claves no equiprobables. En este caso el sesgo que se genera al tener un texto corto no empeora el sesgo creado con la no equiprobabilidad de las claves.

2. Implementación del DES

a. Programación del DES

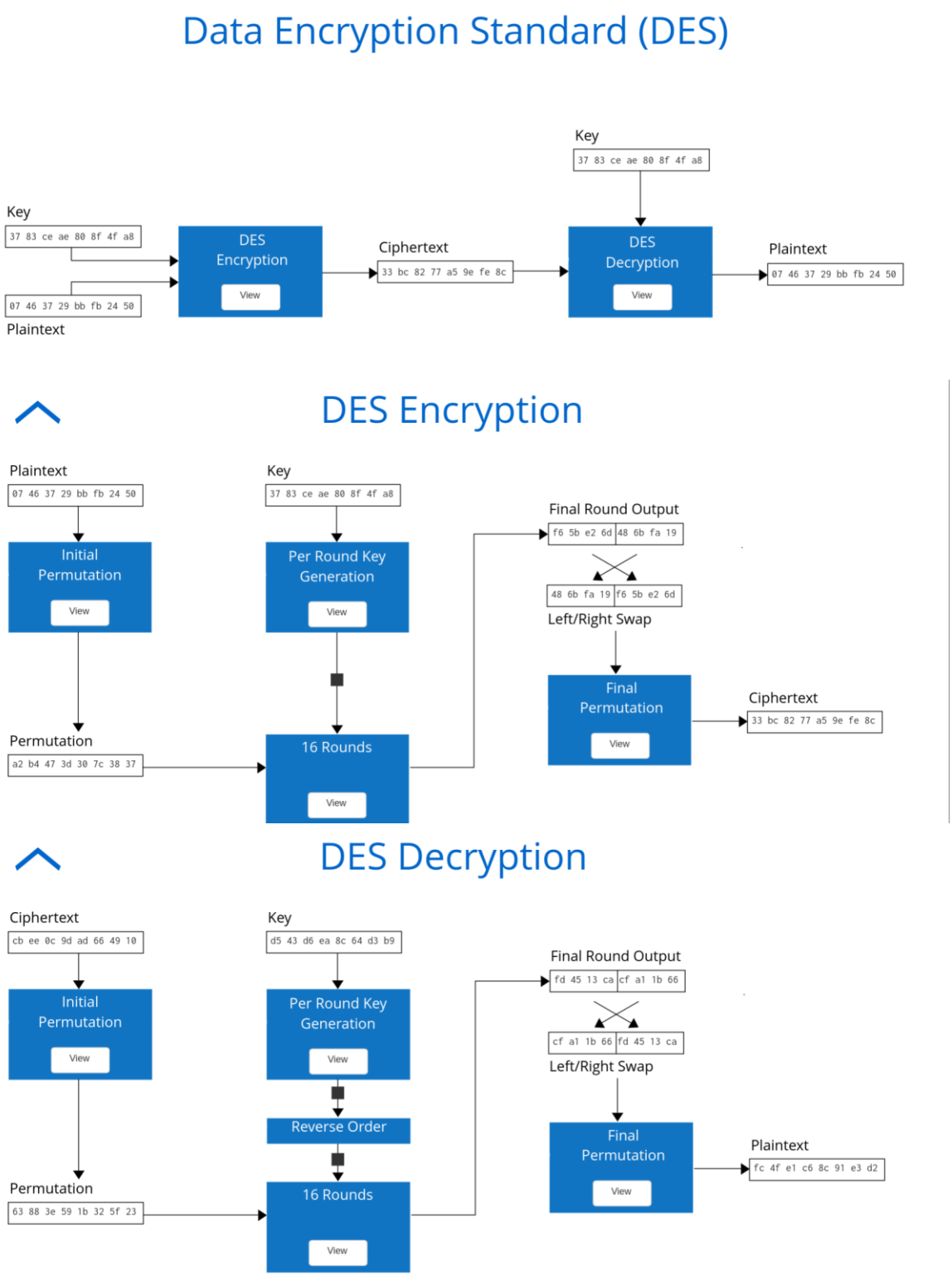
Previo a implementación

Ahora deberemos programar el método de cifrado DES. Este método es particularmente engorroso y deberemos programarlo con sumo cuidado, puesto que detectar un error puede ser sumamente costoso. Recordamos que DES cifra y descifra a nivel de byte, por lo que depurar el código no será una tarea fácil.

Para realizar la codificación correctamente nos apoyaremos con el siguiente material. Primero, usaremos la guía ofrecida en la documentación de "The DES Algorithm Illustrated" que muestra de forma muy detallada y visual cómo funciona el DES. Utilizaremos además el código proporcionado para almacenar los datos de las distintas tablas de DES. También, usaremos la aplicación desarrollada por el usuario BrantNielsen en DesVisualizer para poder observar si nuestro código cifra correctamente cada byte de ejemplo. También, se han creado varios archivos python, muy simples, que calculan el resultado de cifrar bloques por DES tanto en modo CBC como ECB. Estos se han usado también para comprobar que el resultado de nuestro algoritmo era correcto. Por último, a la hora de cifrar archivos, nos apoyaremos del comando xxd, que nos permitirá ver el contenido de los archivos a nivel de byte para verificar que, al cifrar y descifrar un archivo este quede absolutamente idéntico.

Por último, debemos mencionar que únicamente programaremos el método DES de CBC. Esta decisión se toma puesto que, si luego queremos cifrar usando ECB, será tan sencillo como pasar el bloque adicional de CBC para el XOR siempre a 0. Así, el xor inicial que se hace con el texto plano no variará nada y se estará cifrando por CBC a grandes rasgos.

Implementación del algoritmo



(Esquemas muy simplificados y útiles para depurar por BrantNielsen en DesVisualizer *de cómo cifra y descifra DES-ECB. Para el modo CBC, únicamente deberemos realizar un XOR antes de la initial permutation en cifrado y después de la final permutation en descifrado con la cade CBC que reciba el programa).*

DES cifra por bloques de 64 bits, por lo que si queremos cifrar un archivo deberemos asegurarnos de que el archivo que mandemos tenga un número de bytes múltiplo de 8 (2^8 = 64). Evidentemente, no todos los archivos cumplen esa función, así que deberemos añadir unos bytes adicionales hasta que el número de bytes del archivo sea múltiplo de 8. Estos bytes se añaden como 0s en el último bloque a cifrar. Es decir, si el último bloque contiene únicamente 5 bytes, por ejemplo, se cifrará como si fuera un bloque de 8 bytes (rellenando los bytes más significativos a 0) y poniéndolo en el archivo de salida. Además, al final del archivo pondremos como último byte el número de bytes adicionales que hemos añadido durante el cifrado (padding) para luego al descifrar saber cuántos bytes del último bloque se deben descartar.

Obviamente, existen múltiples métodos de padding mucho más sofisticados y seguros, como por ejemplo "PKCS#5 Padding". Sin embargo, se ha descartado la implementación de estos por simplificación del trabajo.

Para la implementación de todas las funciones relacionadas con DES se han creado los ficheros "des.c"" y des.h "que cubrirán todas las necesidades para el cifrado de un bloque por DES a través del método CBC. Como se puede apreciar, se ha apostado por una programación muy funcional por el hecho de que facilitaba ampliamente la depuración de la implementación. El tipo de variable más utilizada son los enteros con diferente número de bits de la biblioteca "stdint.h", que nos permitirá usar valores de enteros precisamente en el tamaño que necesitemos en cada momento. Se usarán con el tipo sin signo (unsigned) para evitar que el complemento a 2 nos interfiera en algún caso, ya que siempre trabajaremos con operaciones lógicas. Luego, se ha creado el archivo principal "des_cypher.c" que recibirá el archivo que quiere cifrar. El bloque de inicialización CBC (IV) está declarado al principio del archivo con un valor aleatorio. Por supuesto, en cualquier momento puede ser alterado si así se desea. También, existe una macro que permitirá cifrar con el método ECB si es deseado, omitiendo el vector extra para las operaciones CBC.

Criptoanálisis y resultados

Podremos ejecutar nuestro cifrador mediante el comando del make “run_des_c” para cifrar y “run_des_d” para descifrar. En el propio archivo makefile podemos editar la clave inicial con la que queremos cifrar y los archivos de entrada y salida del algoritmo. Se proporcionan algunos archivos para probar en la carpeta binarios/ del proyecto. También, se recomienda usar el comando xxd con el archivo original y el archivo descifrado y verificar que la salida de ambos es la misma.

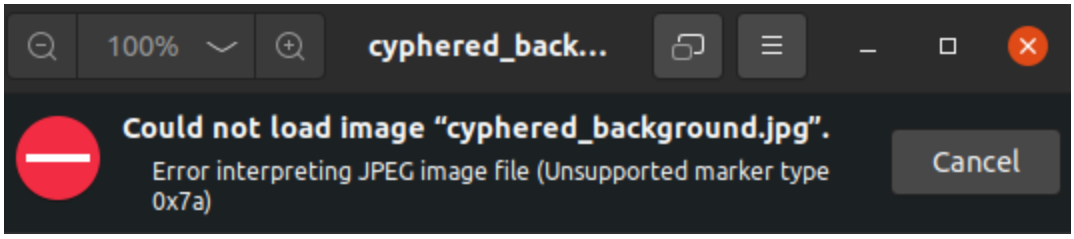
A modo de conclusión sobre el DES podemos decir que es un algoritmo de cifrado por bloques que genera salidas altamente aleatorias, con una relación mínima entre las claves y los datos cifrados. Además, es relativamente rápido, ya que se basa en operaciones lógicas simples y accesos a memoria, lo que permite su optimización para alcanzar altos niveles de eficiencia. Sin embargo, su espacio de claves (56 bits) es insuficientemente grande frente a la capacidad de cómputo actual, lo que lo hace vulnerable a ataques de fuerza bruta. Por este motivo, DES se considera inseguro para aplicaciones modernas y ha sido reemplazado en gran medida por algoritmos más robustos, como AES.

A continuación, responderemos a las preguntas propuestas por la práctica:

Comprueba que el modo de operación ECB (Electronic Code-Book) no es bueno para mensajes largos y estructurados, como por ejemplo imágenes. Comparar con la función diseñada desCBC, y explica porque existen las diferencias con el modo ECB en los textos cifrados.

La mejor prueba para demostrar esto, sería cifrar una imagen usando el DES-ECB y el DES-CBC que acabamos de crear. Sin embargo, para que la imagen que se va a cifrar sea legible una vez cifrada debemos ser muy cuidadosos con mantener la estructura del archivo JPEG, en caso contrario se convertirá en un archivo corrupto ilegible. Los archivos JPEG cuentan con una cabecera que contiene metadatos esenciales para la lectura del archivo, si estos son cifrados el ordenador no será capaz de interpretar el archivo. Además, existen ciertos marcadores que también son necesarios de mantener durante el cifrado si pretendemos observar los datos de la imagen cifrada.

Nuestra idea era crear una función “cifrar_jpeg” que recibiera un jpg y lo cifrara manteniendo la cabecera y el resto de los marcadores esenciales. Sin embargo, a pesar de usar bibliotecas especializadas como “jpeglib.h” para detectar la cabecera y asegurarnos mediante xxd que esta y el marcador de fin de imagen estuvieran, no hemos sido capaces de hacer que el ordenador lo lea sin decir que la imagen está corrupta. Para obtener la biblioteca “jpeglib.h” ejecutar el comando “sudo apt-get install jpeginfo” y obtener la biblioteca.



Sin embargo, no es muy difícil entender por qué ECB funcionará mal para archivos estructurados como imágenes. ECB cifra cada bloque del archivo de forma independiente, lo que es ideal para poder paralelizar el algoritmo. Pero esta independencia tiene una gran desventaja. En una imagen, los píxeles (datos del archivo) guardan mucha relación con los que tienen a su alrededor. Por ejemplo, en una imagen se puede encontrar una zona con un espacio de píxeles del mismo color, Estos contendrán probablemente el mismo valor de datos y es muy probable que se repitan. Por ello, al cifrarse, aunque el resultado sea un número aleatorio muy distinto, no dejará de ser el mismo para todos ellos. Por ello, es muy probable que se pueda ver una silueta distinguible en la imagen cifrada en esa zona.

Esto, con el modo CBC no ocurre puesto que el cifrado de cada bloque depende del resultado anterior. Al añadir esa dependencia extra, se consigue un randomizado más cuando se va a cifrar un bloque y por ello los resultados, aunque el valor de los datos sea el mismo, serán distintos para cada bloque.

b. Programación del Triple DES

Implementación del algoritmo

Una vez programado el DES, no será muy complicado codificar el triple DES. El triple DES es un algoritmo de cifrado que no inventa nada, sino que simplemente ejecuta las funciones del DES tres veces. Lo único con lo que deberemos tener cuidado es con la clave inicial, que es tres veces más grande.

El triple DES cuenta con dos grandes ventajas con respecto al DES estándar. Lo primero es que mezcla aún más los bits dando una salida teóricamente más aleatoria y menos probable a ser predicha. También, la función cuenta con una clave tres veces más grande, lo que genera un espacio de claves mucho más grande que el de DES estándar (2^112 veces más grande). Además, es muy simple de implementar ya que está pensada para ser compatible con sistemas que ya cifraran y usaran el DES, por ello su programación nos ha resultado sumamente sencilla. Evidentemente, como se puede prever, cuenta con la gran desventaja de que es tres veces más lento que el DES estándar al cifrar o descifrar.

En este caso, el único modo que funciona correctamente con DES es el ECB, por lo que el vector de inicialización será siempre 0 cada vez que se llame a la función de encriptación de DES.

Se añaden al fichero “des.c” las funciones de cifrado y descifrado para el triple-des.

```c
/* Auxiliar function for triple_des_cypher */
uint64_t triple_des_cypher(uint64_t block, uint64_t cbc_block, uint64_t *keys1, uint64_t *keys2, uint64_t *keys3)
{
    uint64_t block_cyphered;

    block_cyphered = des_cypher(block, cbc_block, keys1);
    block_cyphered = des_decypher(block_cyphered, cbc_block, keys2);
    block_cyphered = des_cypher(block_cyphered, cbc_block, keys3);

    return block_cyphered;
}

/* Auxiliar function for triple_des_decypher */
uint64_t triple_des_decypher(uint64_t block, uint64_t cbc_block, uint64_t *keys1, uint64_t *keys2, uint64_t *keys3)
{
    uint64_t block_decyphered;

    block_decyphered = des_decypher(block, cbc_block, keys3);
    block_decyphered = des_cypher(block_decyphered, cbc_block, keys2);
    block_decyphered = des_decypher(block_decyphered, cbc_block, keys1);

    return block_decyphered;
}
```

Criptoanálisis y resultados

Para ejecutar el triple des, de nuevo podremos usar el comando “make run_triple_des_c” para cifrar y “run_triple_des_d” para descifrar. Observaremos que los archivos cifrados y descifrados tardan 3 veces más que con des.

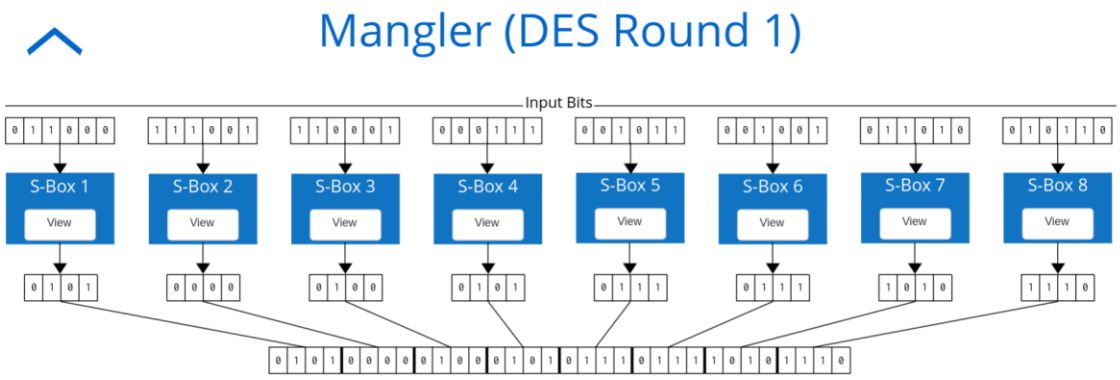
De igual manera que el DES, se tienen a disposición algunos archivos en la carpeta binarios para probar a cifrar y descifrar.

El triple DES es un algoritmo de cifrado más fuerte que el DES estándar por el hecho de contener una clave mucho más grande y difícil de cifrar por fuerza bruta. Pero por lo general, tampoco consigue un cifrado mucho más aleatorio que DES estándar, porque el DES estándar ya hacía un buen trabajo en ese aspecto.

3. Principios de diseño del DES

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

El ejercicio ahora nos pide estudiar la no linealidad de las s-boxes de DES. Entenderemos la no linealidad como la aleatoriedad que muestra la función de las s-boxes al devolver un valor. Es decir, que no exista ninguna función lineal que sea capaz de describirla. Se espera por ello, que al variar ligeramente la entrada de una ejecución esta devuelva un valor completamente distinto, o al menos lo más distinto posible. DES contiene 8 s-boxes que transforman 6 bits en 4. Todas ellas deben generar salidas muy distintas a pesar de que la entrada de 6 bits sea ligeramente distinta para cada una de ellas.



(Esquema que ilustra el funcionamiento de las s-boxes de DES hecho por BrantNielsen en DesVisualizer*).*

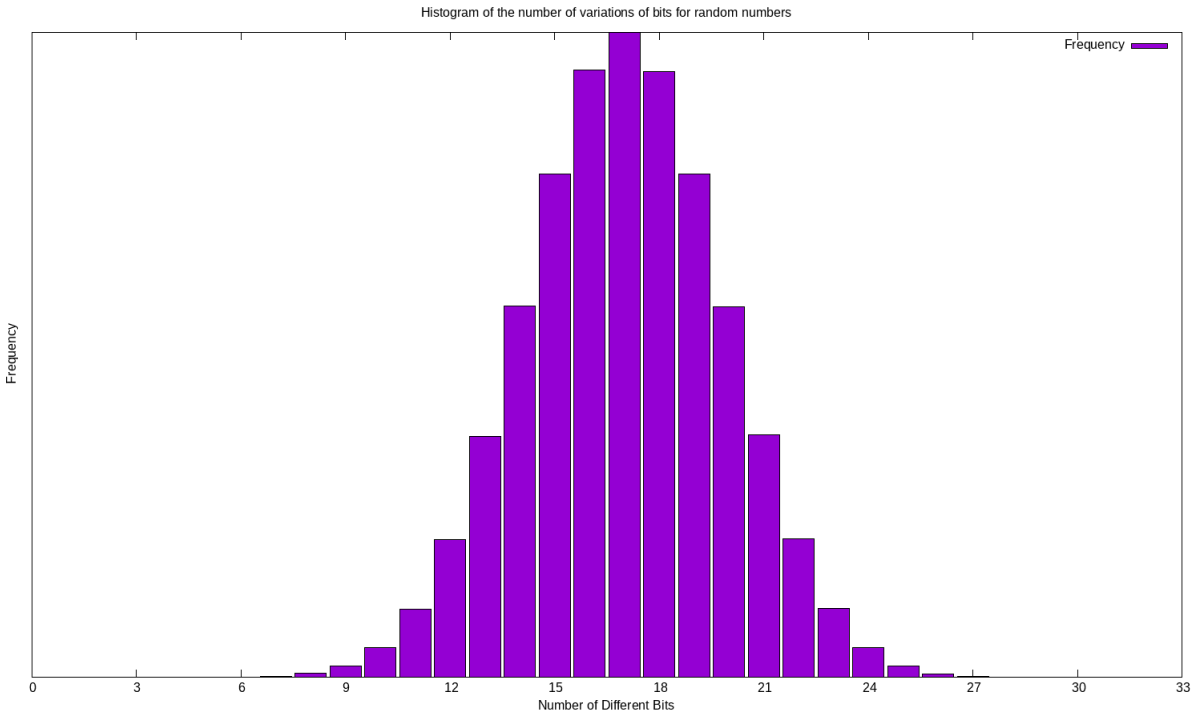
Para probar si se cumple este fenómeno, idearemos un programa que realice la sustitución de s-boxes con la función ya creada del DES con dos claves distintas y compararemos las salidas. Debemos tener en cuenta que necesitaremos crear claves aleatorias de 64 bits, por lo que crearemos una función para ello. Aunque generemos 64 bits aleatorios, s-boxes solo utiliza los 48 bits menos significativos, ignorando los del principio. Modificaremos la clave de tres formas distintas: incrementaremos o decrementaremos en uno cada bloque s-box, variaremos un bit cualquiera de cada bloque o generaremos otra clave completamente aleatoria. Estas tres formas nos permitirán observar el comportamiento de la sustitución de las s-boxes para todos los casos.

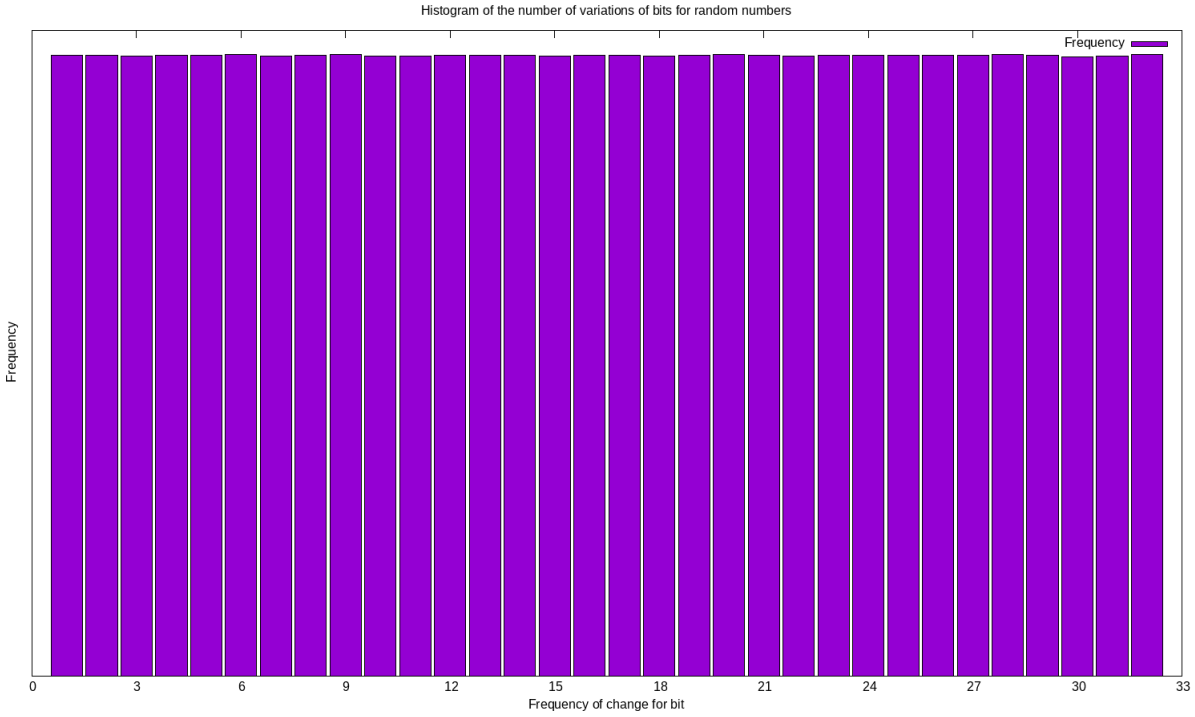
Luego, generaremos la sustitución y obtendremos las dos claves de 32 bits que compararemos y contaremos sus diferencias de dos formas distintas: compararemos cuáles de los 32 bits son distintos o contaremos el número de bits distintos entre una clave y otra. El modo “normal” hace referencia al número total de variaciones de ambas cadenas y el modo “bar” hace referencia a la variación de cada bit entre ambas cadenas. El usuario podrá ejecutar el programa con cualquiera de las seis combinaciones modificando los parámetros al ejecutar el programa. También tendrá a su disposición,

como siempre, el comando “make run_des_no_lineal” y podrá modificar los parámetros en el fichero del makefile.

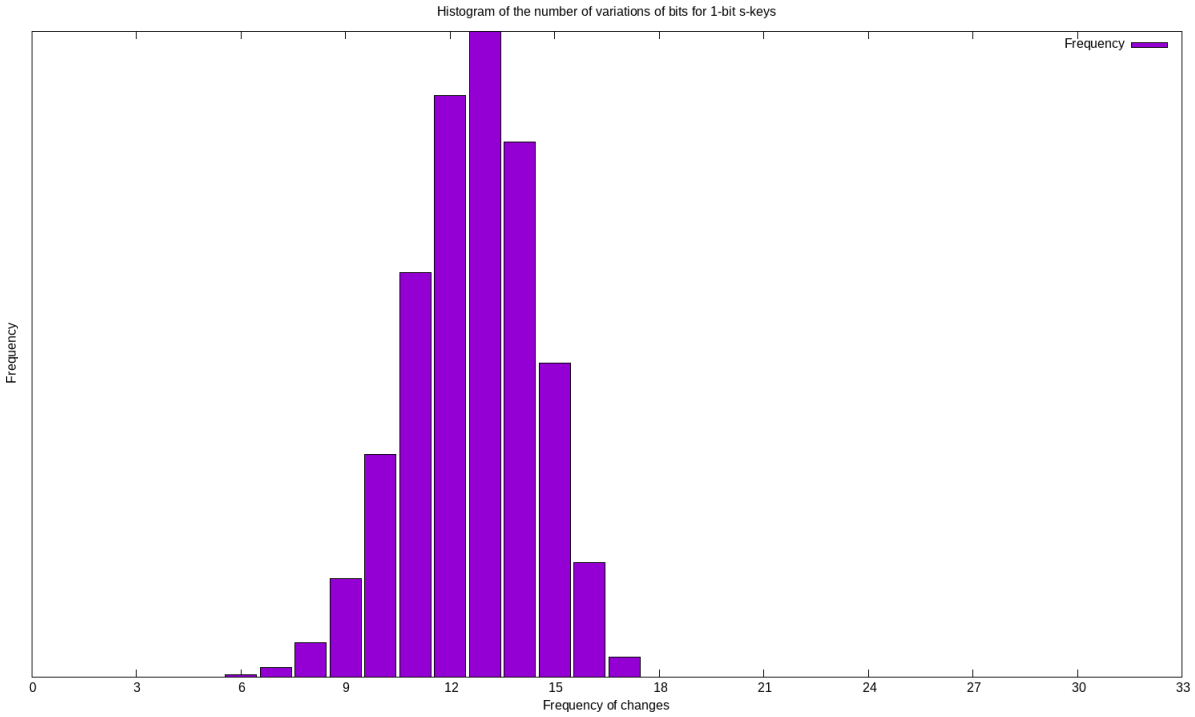
Ahora, mostraremos de forma gráfica los resultados obtenidos con para todos los modos del programa. Luego analizaremos los resultados.

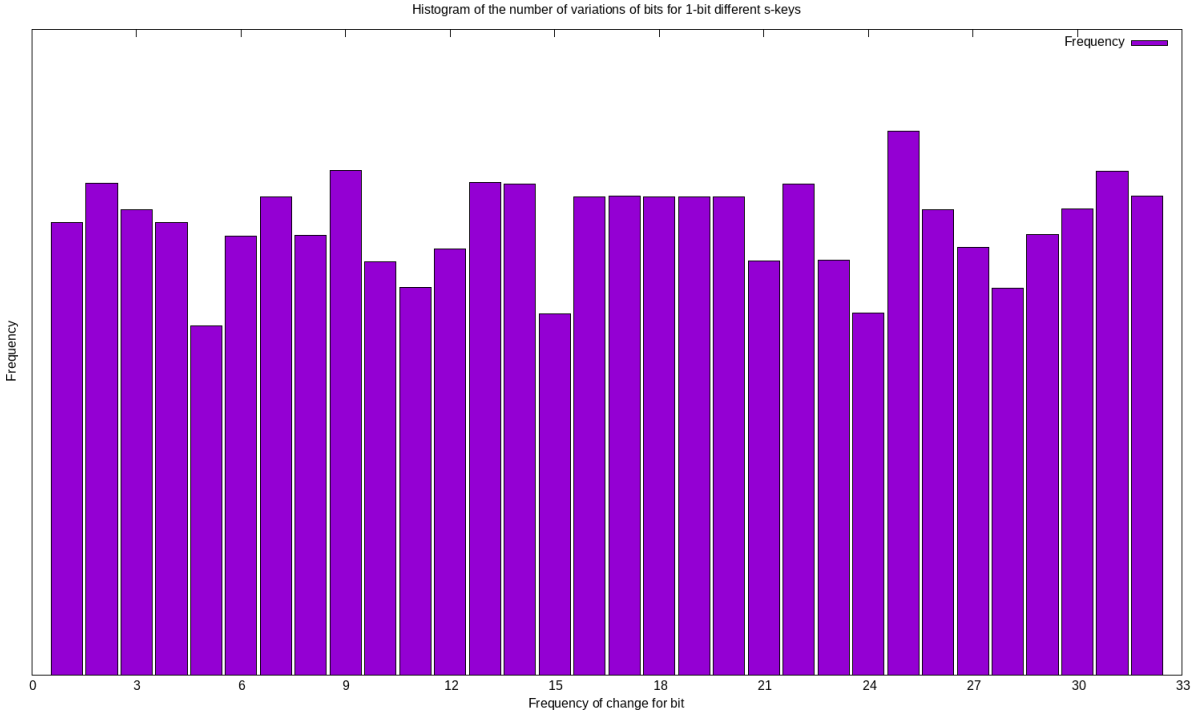
Para claves completamente distintas:



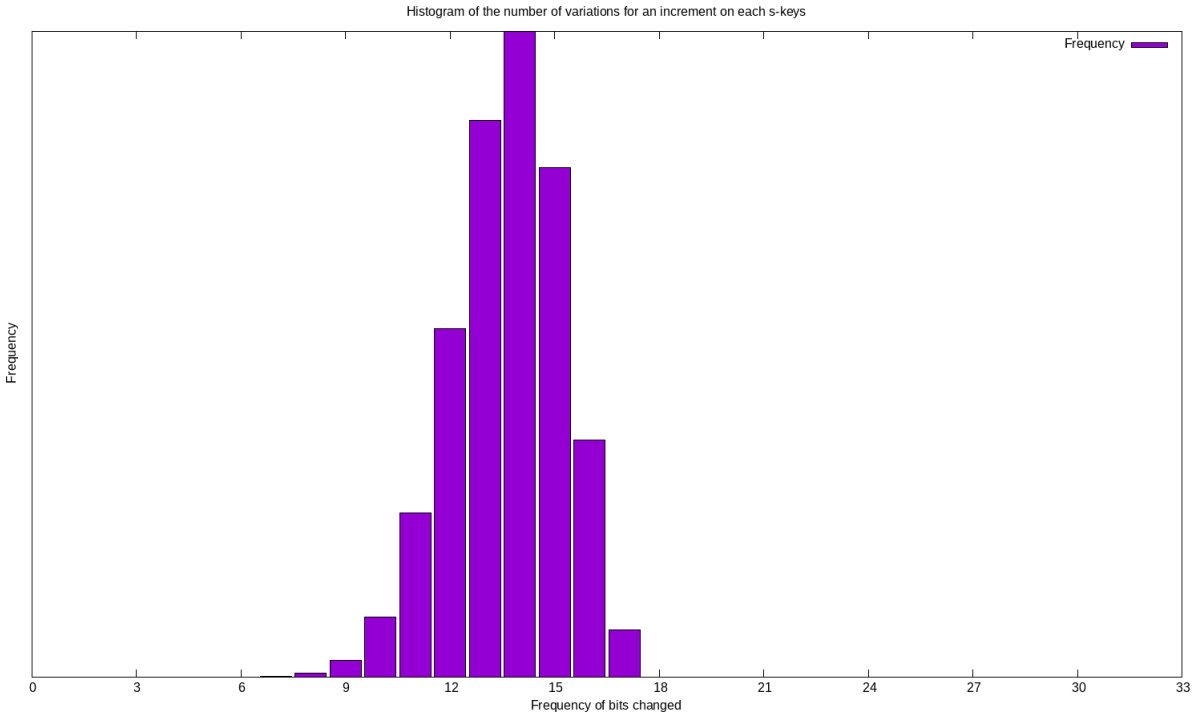


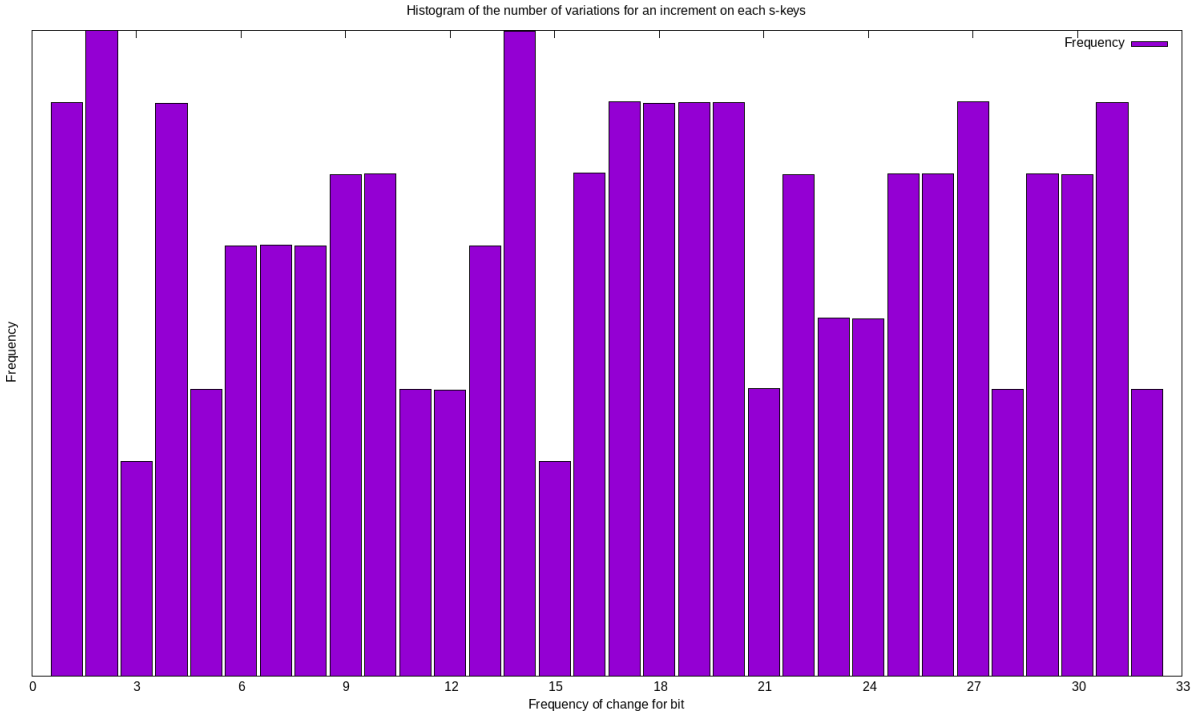
Para claves con un bit cualquiera de varianza en cada s-bloque.





Para claves con un incremento de varianza en cada s-bloque.





Como observamos, los resultados para las claves completamente distintas generan una distribución normal prácticamente perfecta y todos y cada uno de los bits varían con igual frecuencia. Esto, en realidad, es poco interesante, puesto que lo único que indica es que DES genera salidas distintas no repetitivas para todas las claves, ya que la

variación en la función en realidad lo estamos haciendo con el rand() al generar cada clave.

Los otros dos modos son más interesantes, pero como se aprecia no consiguen una variación tan perfecta como la de las dos claves aleatorias. Como observamos, para la variación de un bit cualquiera de cada s-clave, se genera una normal más o menos clara pero que no está centrada. Esto significa que existe una tendencia a que los bits varíen algo menos como podemos ver en la segunda imagen. Algunos bits como el 5º, 15º o el 24º tienden a variar bastante menos que el resto. Para el caso de la variación incremental en cada s-box, ocurre lo mismo, pero incluso con mayor diferencia. Se observan como algunos bits, como el 3º o el 15º que tienden a variar mucho menos.

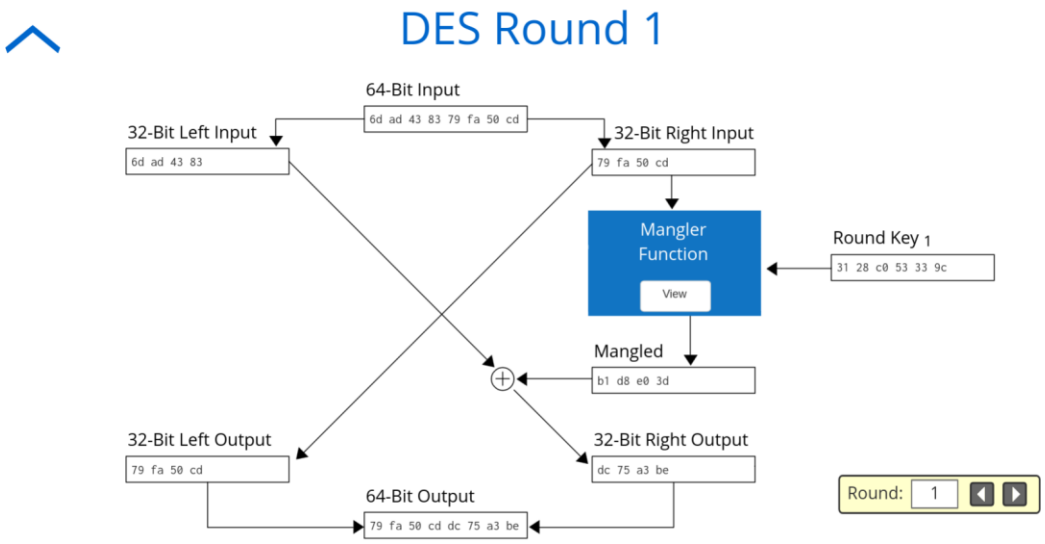
Los resultados indican que las S-boxes no están cumpliendo completamente con el criterio de no linealidad ideal, ya que se debería mostrar una variación más uniforme para todos los bits. Los motivos que podrían estar causando este resultado es una mala implementación de la manera en la que se calcula la no linealidad o en la codificación de la sustitución S-box de DES. Sin embargo, ambos motivos parecen improbables porque para las claves aleatorias sí se consigue una variación completamente uniforme y la implementación del DES muy seguramente sea correcta puesto que se ha probado en apartados anteriores. La sensación que da es que el diseño de las s-boxes de DES contiene limitaciones que no cumple los criterios estrictos de independencia lineal.

En cualquier caso, esta variación es mínima en comparación al DES completo. DES realiza muchas más variaciones y permutaciones a lo largo del algoritmo que mezclan y aleatorizan más la salida. Además, la sustitución de las s-boxes se hace 16 veces para cada cifrado. Si ejecutáramos esta prueba, pero a cada clave le realizamos una sustitución de s-boxes 16 veces muy probablemente los resultados finales serían mucho más uniformes, aunque no es una tarea sencilla porque las sustituciones reciben 48 bits y devuelven 32 bits. Si lo mezclásemos con las permutaciones y el resto de las operaciones del Feistel, sí se observaría el fenómeno con casi toda seguridad.

b. Estudio del Efecto de Avalancha

Ahora procederemos a estudiar el efecto avalancha que se genere para nuestro algoritmo de cifrado de DES. Debemos entender primero lo que es el efecto avalancha. El efecto avalancha es una propiedad deseable de los algoritmos de cifrado que explica la capacidad que tiene el algoritmo de con un mínimo cambio en su entrada, devolver una salida muy distinta. Es decir, una modificación mínima (como un bit) debería alterar aproximadamente el 50% de los bits del resultado.

DES es un algoritmo que mezcla y reordena los bits en varias ocasiones ocasionando un efecto avalancha progresivo. El ejercicio nos pide estudiar el efecto durante las rondas DES Feistel, en las que se cifra el mismo bloque con las subclaves durante 16 iteraciones. Deberíamos observar cómo a medida que se avanzan las rondas, la diferencia entre los bytes es cada vez mayor hasta estabilizarse.



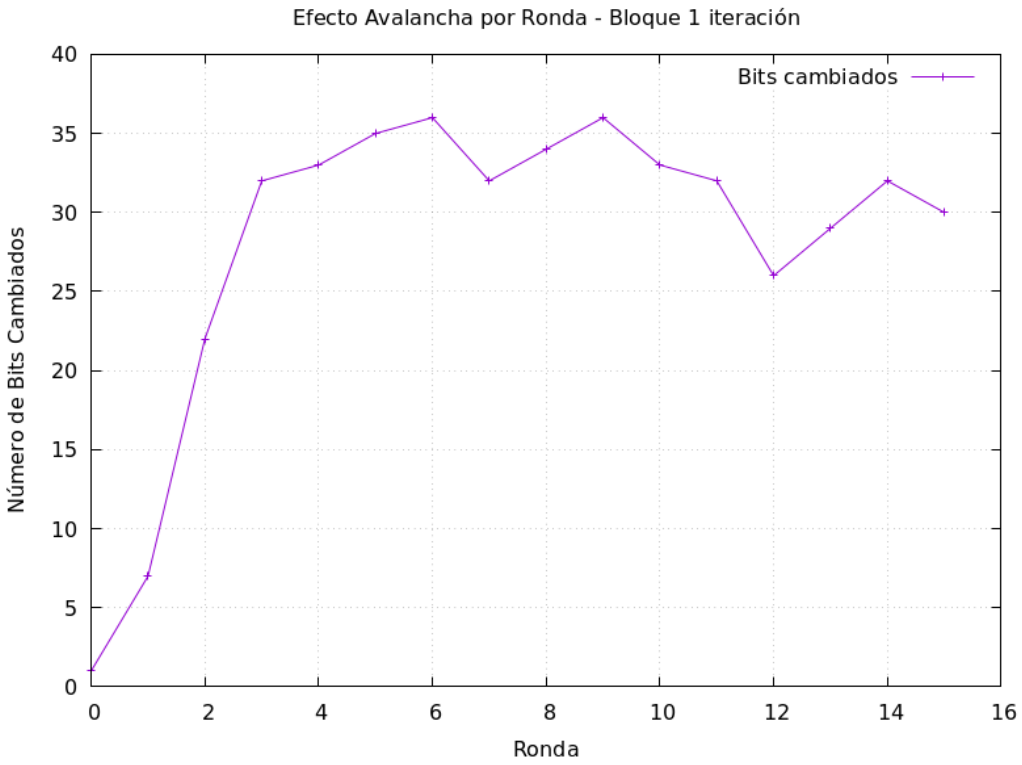
(Esquema que ilustra cada ronda Feistel en DES hecho por BrantNielsen en DesVisualizer*).*

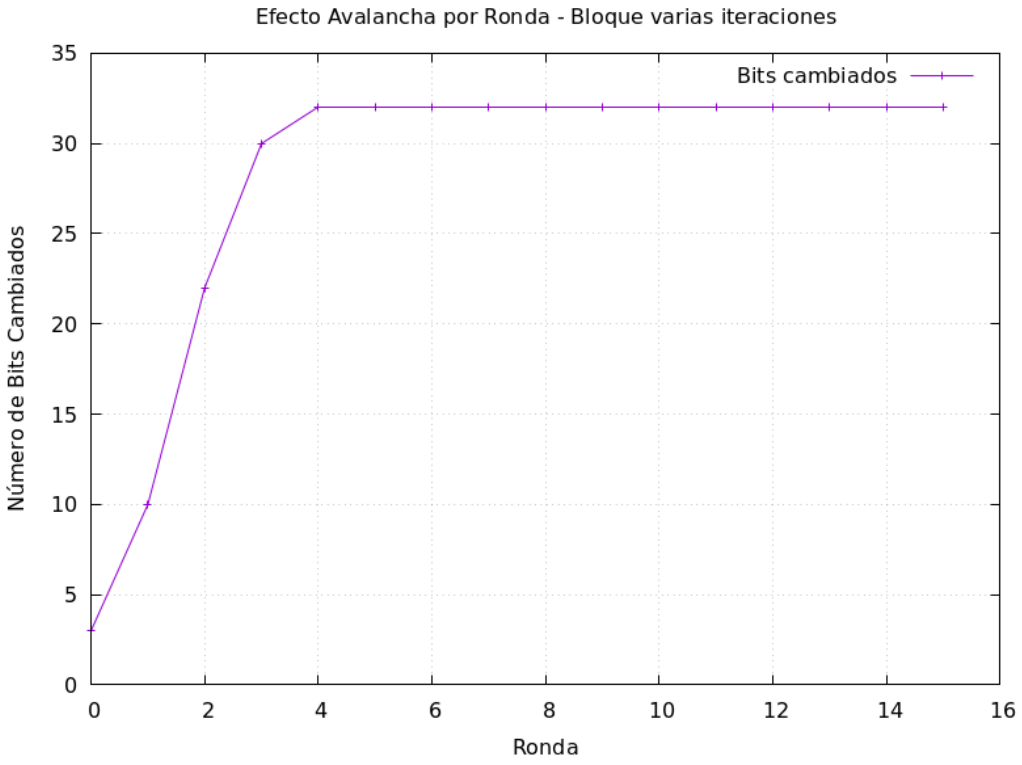
El plan para probar experimentalmente este fenómeno será el siguiente. Se va a crear un programa llamado "des_avalancha" al que le pasaremos: un bloque de 64 bits que será el bloque que se cifrará, una clave inicial para el cálculo de subclaves, un número de repeticiones para realizar varios intentos y obtener una media más representativa y si deseamos editar la clave o el bloque. El programa calculará cada ronda Feistel para dos bloques o dos claves distintos por un bit (elegido al azar). Tras cada ronda, calculará el número de bits distintos y lo guardará y proseguirá con la siguiente ronda. Luego, apuntaremos el número medio de bits distintos en cada ronda y crearemos un gráfico de líneas para observar la evolución a lo largo de cada ronda.

Para poder ejecutar el programa, una vez más, se puede hacer uso del comando "make run_des_avalancha" que ya contiene los parámetros para ejecutarse correctamente. Se podrán editar los parámetros a los deseados mediante la modificación del fichero

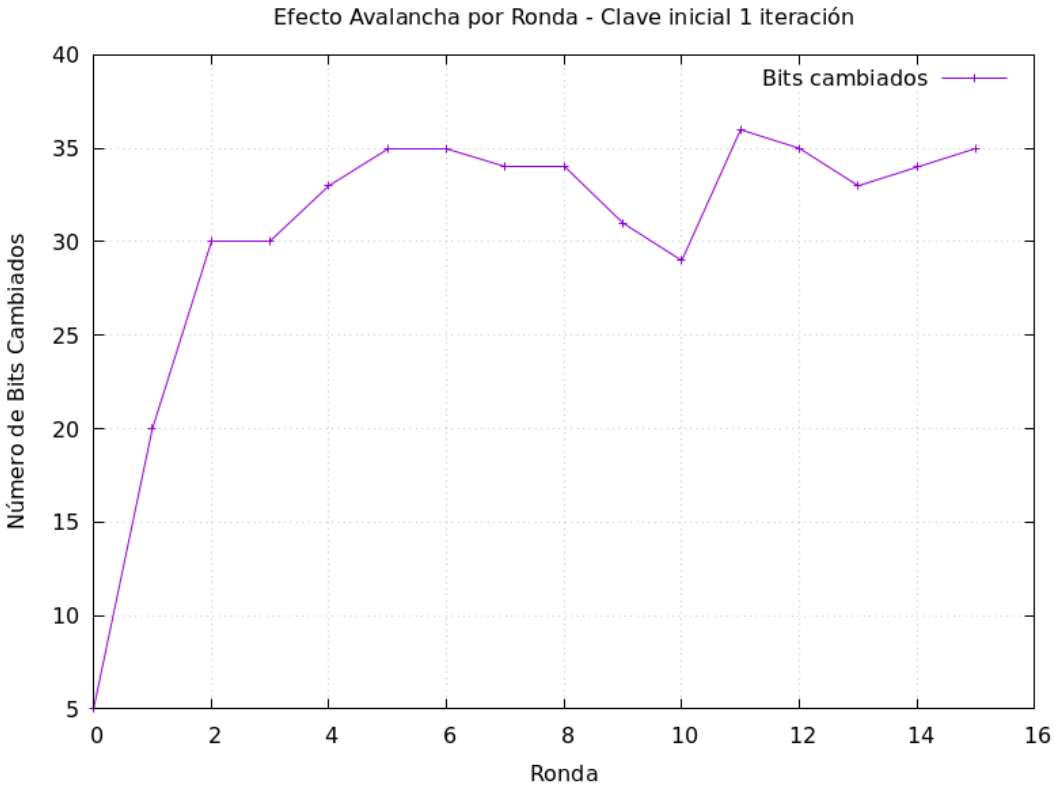
“makefile”. Para la ejecución del programa, se han implementado nuevas funciones para la ejecución de este, además de un nuevo script que generará un gráfico de líneas mediante gnuplot. Ejecutamos el programa alterando el bloque cifrante y la clave, en una y varias repeticiones y obtenemos los siguientes gráficos:

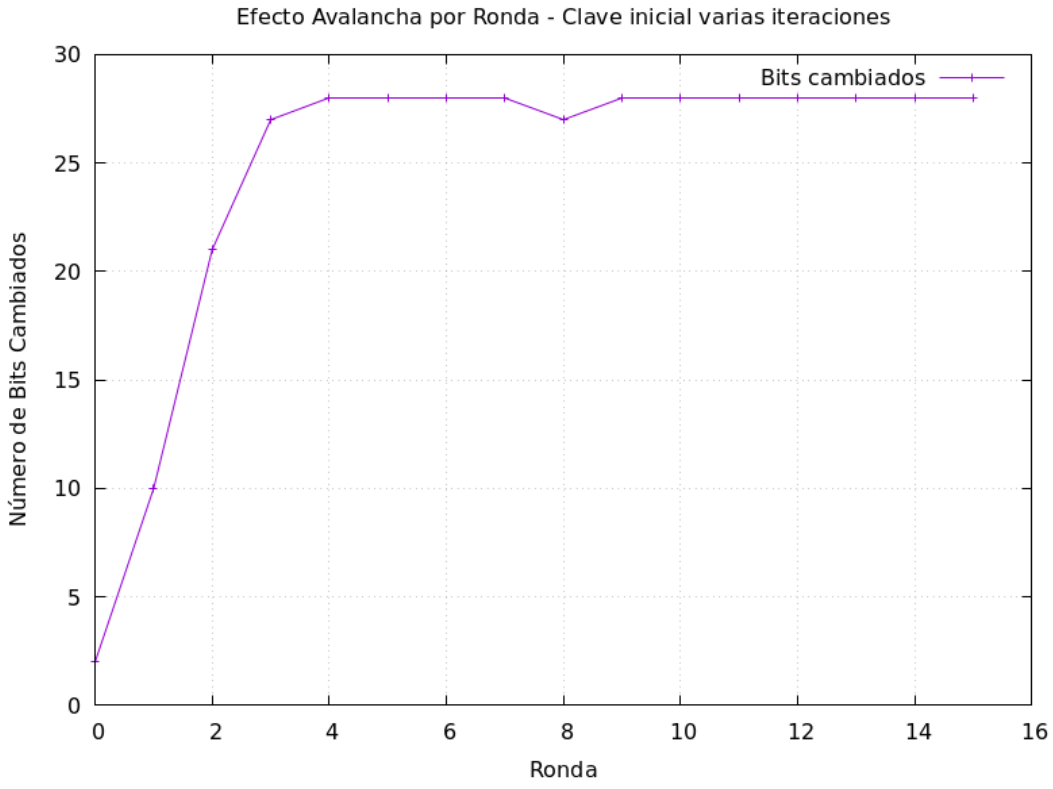
Efecto avalancha editando el bloque:





Efecto avalancha editando la clave inicial:





Como observamos en ambas imágenes, en este caso sí hemos obtenido unos resultados satisfactorios y acordes a los teóricos. El número de bits distintos es bajo durante las primeras rondas, pero en seguida se estabiliza en torno a los 32 bits. De hecho, realizando varias ejecuciones y calculando la media obtenemos un gráfico prácticamente perfecto.

4. Principios de diseño del AES

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

La no linealidad en el contexto de las S-boxes del AES se refiere a la resistencia contra ataques basados en el criptoanálisis, especialmente los ataques de correlación y de aproximación lineal. Esta característica es crucial para asegurar que las pequeñas modificaciones en la entrada resulten en cambios impredecibles en la salida.

Para realizar estas sustituciones haciendo uso de las S-boxes, hemos añadido las S-boxes del AES en el archivo *'utils.h'*, y hemos desarrollado la función *'direct_box_transformation',* que facilita la conversión de un array de 16 bytes (el elemento inicial de 128 bits descompuesto en bytes), a un elemento de 128 bits sustituido con las S-boxes.

Hemos desarrollado 2 métodos para comprobar la no linealidad de las S boxes del AES. Podemos encontrar ambos en el fichero *'aes_no_lineal',* en la carpeta *'aes'*. Para ejecutar este archivo podemos hacer uso del comando *'make run_aes_no_lineal',* el cual tiene 1 parámetro de entrada del que hablaremos más adelante.

El primer método que hemos desarrollado se basa en estudiar si existe alguna relación entre la entrada y la salida de una S-box con 2 entradas casi iguales a diferencia de 1 bit. Tras transformar ambas entradas, calculamos el número de bits diferentes entre las salidas, comprobando si este ligero cambio modifica la salida, y en qué medida lo hace.

Realizamos esta comprobación varias veces (las marcadas en la macro *'NUMBER_OF_TESTS'),* y generamos un histograma con las frecuencias de las diferencias obtenidas en cada test. Es aquí donde entra en acción el parámetro de entrada del programa comentado previamente. En caso de lanzar el programa con el parámetro '-t', el histograma se imprimirá por terminal. Sin embargo, si hacemos uso del parámetro '-o', se generará un histograma de mayor calidad con formato de imagen haciendo uso de un script de gnuplot. Por defecto el programa se ejecuta como si se hubiera introducido el parámetro '-o'.

El segundo método consiste en comprobar que no se cumple la propiedad aditividad, necesaria para que una función sea lineal. Esto es lo mismo que decir que las S-boxes del AES cumplen lo siguiente:

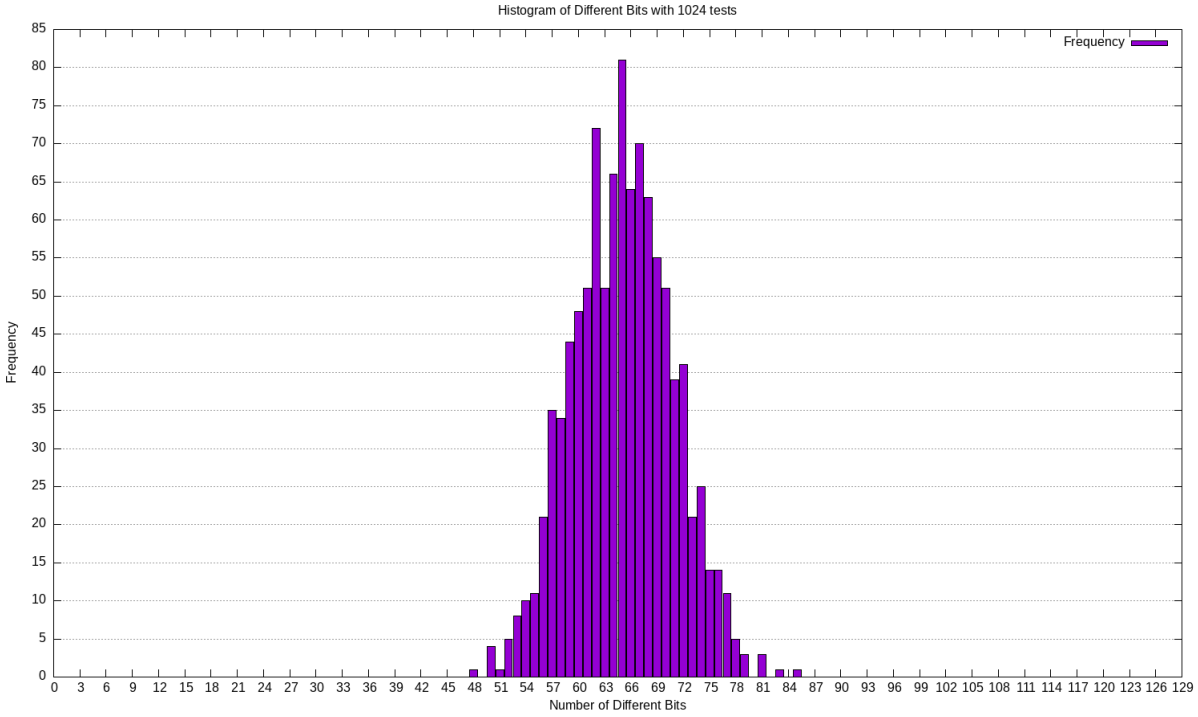
$$Sbox\left(a \bigoplus b\right) \neq Sbox(a) \bigoplus Sbox(b)$$

Puede darse que para casos aislados no se cumpla esta propiedad, pero la frecuencia de esto tiende a 0.

Para comprobarlo, generamos 2 elementos de 128 bits aleatorios, mediante la función *'random_128_change',* y les aplicamos la fórmula. Realizamos esta prueba varias veces

(las marcadas en la macro *'NUMBER_OF_TESTS'*), y guardamos el número de iteraciones que cumplen la condición. Finalmente imprimimos por terminal el resultado.

El resultado del primer método tras ejecutarlo es similar al siguiente:



El histograma representa la distribución de la cantidad de bits diferentes entre el valor de salida de una entrada base transformada por las S-boxes y el valor de salida de entradas alteradas aleatoriamente.

La forma del histograma se asemeja a una distribución normal centrada alrededor de un punto medio (en este caso, cerca de 64 bits diferentes, que es la mitad de 128 bits). Esta distribución indica que los cambios en los bits de entrada se dispersan de manera amplia y uniforme a lo largo de la salida, una característica esperada de una función no lineal.

Un sistema lineal podría mostrar picos en los extremos (muy pocos o muchos bits diferentes) si ciertas combinaciones de bits en la entrada resultan en diferencias predecibles en la salida. La falta de picos significativos en los extremos del histograma refuerza la idea de que las diferencias en los bits de entrada se propagan de manera impredecible, característica de la no linealidad.

La salida del segundo método es más fácil de interpretar, donde podemos observar que casi el 100% de las veces que probemos la fórmula mencionada previamente con entradas aleatorias, esta se cumplirá:

```
make run_aes_no_lineal
aes/aes_no_lineal -o
sbox(A xor B) has been different of sbox(A) xor sbox(B) 1024 out of 1024 times (100%)
```

b. Generación de las S-boxes del AES

Previo a la implementación

Para la codificación del Algoritmo de Euclides extendido para obtener los inversos multiplicativos en GF(2^8), hemos hecho uso de una versión especial simplificada para este campo.

Podemos encontrar las funciones en el fichero 'SBOX_AES.c', dentro de la carpeta 'aes'. Además de la función para calcular los inversos multiplicativos, nos encontraremos con los siguientes métodos:

- Función 'xtime': Aplica la transformación xtime a un elemento de 8 bits.

```c
uint8_t xtime(uint8_t x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1B : 0x00);
}
```

- Función 'affine_transformation': Aplica la transformación afín a los inversos multiplicativos calculados con la función extended_euclides_GF. Logramos esta transformación mediante desplazamientos y XORs:

```c
uint8_t affine_transformation(uint8_t q) {
    //printf("Initial: %x\n",q);
    uint8_t p[4] = {0,0, 0, 0};
    for (int i = 1; i <= 4; i ++) {
        p[i-1] = q << i | q >> (8 - i);
    }

    q = q^p[0]^p[1]^p[2]^p[3];

    return q;
}
```

- Función 'initialize_aes_sbox': Calcula los valores de la s-box directa del AES, haciendo uso del algoritmo de Euclides extendido y de la transformación afín:

```c
void initialize_aes_sbox(uint8_t sbox[256]) {
    extended_euclides_GF(sbox);
    for(int i = 0; i < 256; i++) {
        sbox[i] = affine_transformation(sbox[i]) ^ 0x63;
    }
}
```

- Función 'initialize_aes_inv_sbox': Tomando como base la salida de la función anterior, calcula la s-box inversa del AES:

```c
void initialize_aes_inv_sbox(uint8_t sbox[256], uint8_t inv_sbox[256]) {
    for (int i = 0; i < 256; i++) {
        inv_sbox[sbox[i]] = i;
    }
}
```

Implementación del algoritmo:

Para implementar el algoritmo de generación de las s-boxes del AES hemos seguido los siguientes pasos:

1. Parámetros necesarios para la ejecución del algoritmo:

Hemos decidido mantener los parámetros de entrada propuestos por el enunciado. Como se puede ver, contamos con 1 parámetros obligatorios (“*-C|-D”*) y 1 opcional *(“-o”)*.

2. El siguiente paso es generar la s-box directa del AES, y en caso de haber ejecutado el programa con el parámetro ‘*-D*’, también se calculan las s-boxes inversas:

```
/* Get the values of the direct sbox*/
initialize_aes_sbox(sbox);

if(modo == 'd') {
    initialize_aes_inv_sbox(sbox, inv_sbox);
}
```

3. A continuación, imprimimos estos valores en la variable *‘output’*, que será la encargada de contener los resultados para mostrarlos en el último paso:

```
/*Print the sboxes*/
if(modo == 'c') {
    for (int i = 1; i < 257; i++) {
        snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "%02x ", sbox[i-1]);
        if(i % 16 == 0 && i > 15) {
            snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "\n");
        }
    }
/*Print the inverse s boxes*/
} else {
    for (int i = 1; i < 257; i++) {
        snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "%02x ", inv_sbox[i-1]);
        if(i % 16 == 0 && i > 15) {
            snprintf(output + strlen(output), OUTPUT_BUFFER-strlen(output), "\n");
        }
    }
}
```

4. Por último, mostramos los resultados guardados en la variable ‘*output’,* mediante la función ‘*handleOutputText*’:

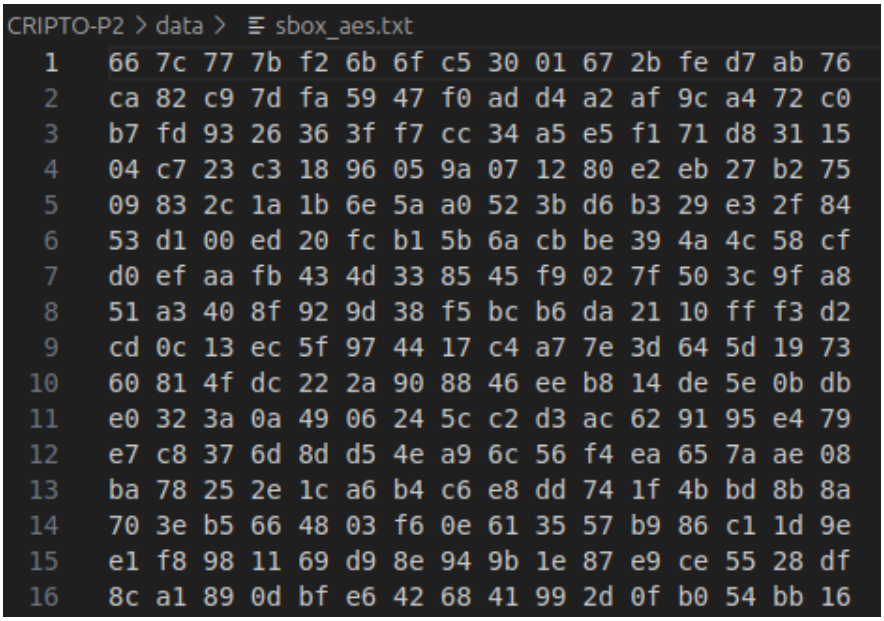
```
if (handleOutputText(fileout, output) == -1) {
    printf("Error al mostrar la salida\n");
    free(output);
    return 1;
}
```

Criptoanálisis y resultados:

Para probar este método, podemos hacer uso del makefile (‘*make run_SBOX_AES*’ , por defecto muestra la s-box directa, pero se puede modificar el parámetro de entrada a ‘*-D’* en el makefile). Por defecto, este comando utiliza el archivo de salida “data/sbox_aes.txt”, pero se puede modificar desde el makefile, o bien quitando el parámetro -o para mostrar la salida por terminal, o cambiando el fichero.

Como se puede observar, tanto la salida de la s-box directa como inversa coinciden con las tablas proporcionadas:

- s-box directa del AES:



- s-box inversa del AES: