CRIPTOGRAFÍA

PRÁCTICA 2: Seguridad Perfecta y Criptografía Simétrica

AUTORES

CARLOS GARCÍA SANTA

INGENIERÍA INFORMÁTICA

ESCUELA POLITÉCNICA SUPERIOR

UNIVERSIDAD AUTÓNOMA DE MADRID

17/04/2025

Índice

1. Seguridad Perfecta

a. Comprobación empírica de la Seguridad Perfecta del cifrado Afín

El script *seg-perf.py* implementa una comprobación empírica para verificar la Seguridad Perfecta del cifrado Afín mediante dos escenarios distintos:

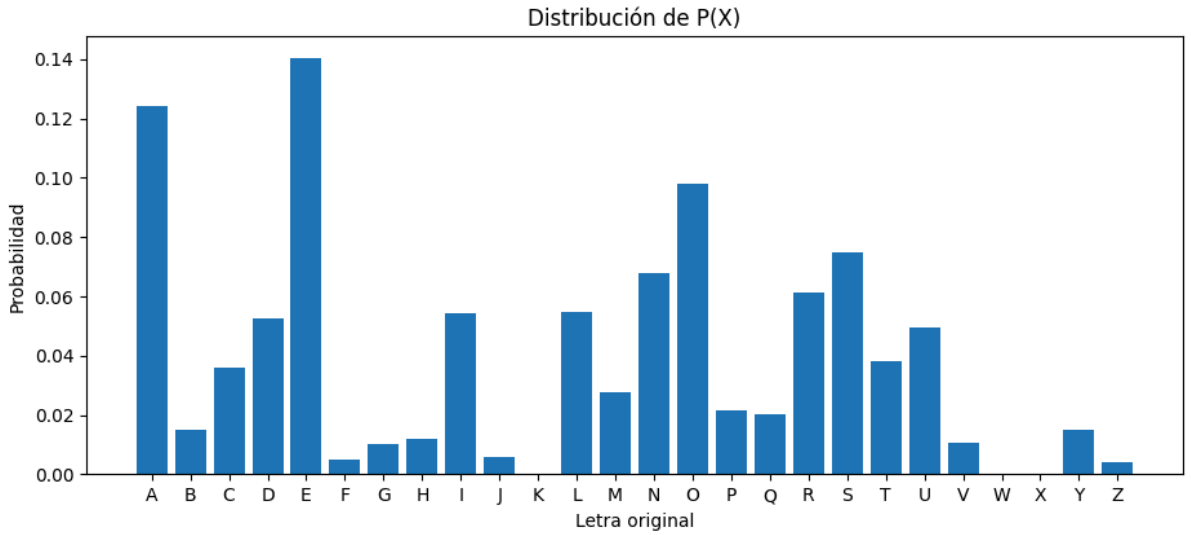
- Modo Equiprobable (-P):
  - Cada clave se selecciona uniformemente del conjunto de posibles claves, donde *a* es coprimo con *m*, y *b* varía en el rango [0, 25]. Para ello utilizamos la función *random choice* de *numpy* que nos permite elegir las claves según una distribución uniforme.
- Modo No Equiprobable (-I):
  - Las claves se seleccionan con una distribución no uniforme, asignando mayor probabilidad a valores específicos, esto se consigue mediante el uso de *random choice* con el parámetro *weights* inicializado a los valores indicados:
    - Coeficiente multiplicativo *a* (coprimos con m): [1, 11, 21] tienen peso 0.2, los demás 0.05.
    - Término constante *b* (rango [0, 25]): [0, 13, 25] tienen peso 0.1, los demás 0.03.

La seguridad perfecta se cumple cuando la probabilidad de un texto plano dada la probabilidad de un texto cifrado es igual a la probabilidad del texto plano, es decir, cuando no se puede extraer información del texto plano observando el texto cifrado. Esto se define como:

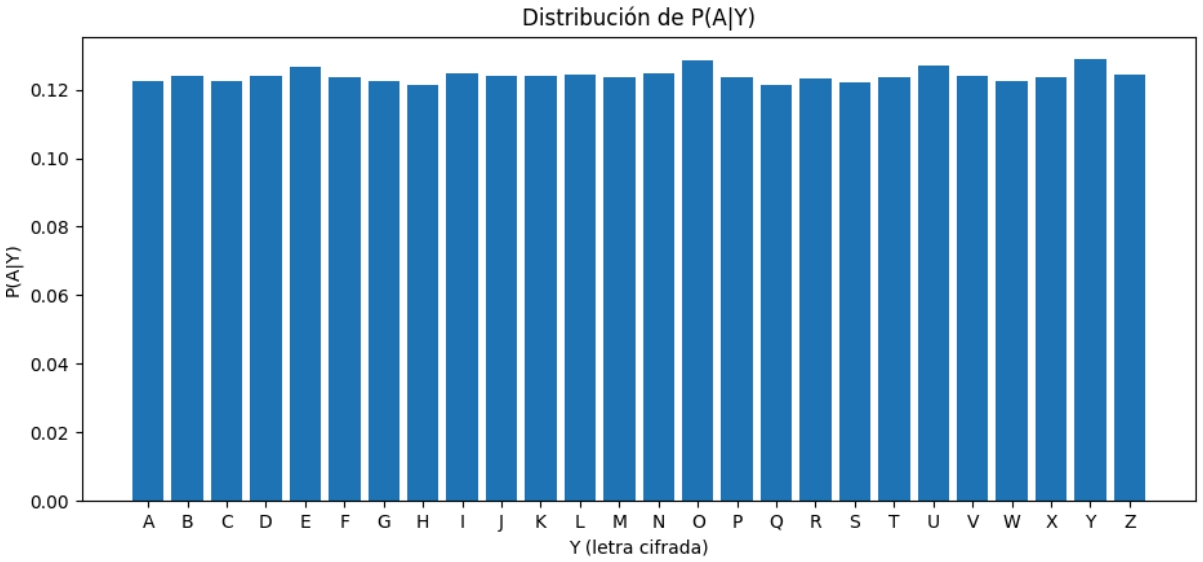
$$P_p(x \mid y) = P_p(x), \forall x \epsilon P, y \forall y \epsilon C$$

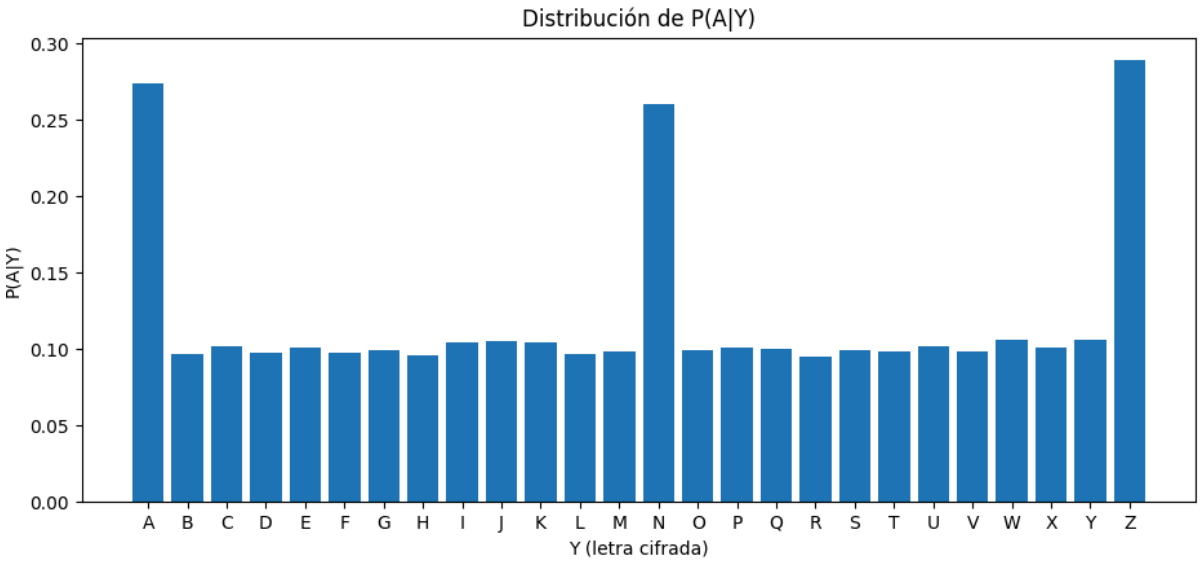
Con lo cual para comprobar si se satisface está fórmula para el método afín en ambos modos, hay que calcular $P_x(x)$ y $P_p(x \mid y)$ de un texto, que en este caso se ha elegido *El Quijote*.

Para calcular $P_x(x)$ basta con contar la frecuencia de caracteres en el texto sin cifrar:



Para calcular $P_p(x \mid y)$ utilizamos la siguiente expresión: $P(x, y) = P(x \mid y) * P(y) \Rightarrow P(x \mid y) = P(x, y) / P(y)$, calcular $P(x, y)$ es sencillo, puesto que es el la probabilidad de que aparezca cada combinación posible de caracteres del texto plano y cifrado, así mismo, calcular $P(y)$ también es muy sencillo, ya que consiste en simplemente obtener la probabilidad de cada carácter cifrado. Una vez tenemos estas dos probabilidades se calcula $P(x \mid y)$ como se muestra en la fórmula, y obtenemos lo siguiente para el modo equiprobable y no equiprobable respectivamente (se muestra únicamente para el carácter '*A*'):





Si analizamos la distribución $P_x(x)$ para la letra *'A'* observamos una frecuencia de aproximadamente un 12%, esta probabilidad (o frecuencia) coincide con las probabilidades $P(x \mid y)$ calculadas para el modo equiprobable, esto se repite para el resto de letras, con lo cual podemos afirmar que el método Afín con el modo equiprobable tiene seguridad perfecta. No obstante, para el modo no equiprobable se ve claramente como la distribución $P(x \mid y)$ no coincide con $P_x(x)$ con lo cual el texto cifrado nos aporta información sobre el texto plano y no se cumple la condición de seguridad perfecta.

2. Implementación del DES

a. Programación del DES

El script *des.py* implementa el algoritmo de cifrado DES de 16 rondas en modo CBC (Cipher Block Chaining), basado en el esquema clásico de Feistel. Para el desarrollo del mismo se ha seguido cuidadosamente la descripción detallada y paso a paso disponible en https://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm, se han utilizado las definiciones aportadas en Moodle para las cajas y permutaciones, y se ha empleado la biblioteca de *bitarray* que permite realizar manipulaciones a nivel de bit.

El cifrado DES se compone de varias fases, que han sido implementadas en funciones individuales dentro del código. A continuación, se detallan los bloques fundamentales:

2.1. Generación de subclaves

La función *generate_subkeys* genera un conjunto de 16 subclaves a partir de una clave inicial de 64 bits:

- La clave original (64 bits) pasa por la tabla *PC1*, eliminando bits de paridad y reduciendo la clave a 56 bits.
- Se divide la clave resultante en dos mitades *C* y *D* de 28 bits cada una.
- Ambas mitades se desplazan circularmente a la izquierda según la tabla *ROUND_SHIFTS*, que tiene un valor diferente para cada ronda.
- Tras cada rotación, las dos mitades se concatenan formando nuevamente 56 bits, y luego se aplica la permutación *PC2* para comprimir estos bits a una subclave de 48 bits.

Este proceso es repetido 16 veces generando un conjunto completo de subclaves para todas las rondas del cifrado.

2.2. Función Feistel

La función *f_function* implementa la operación no lineal $f(R_{i-1}, K_i)$ aplicada durante cada ronda del esquema Feistel. Dicha función realiza:

- El bloque de entrada de 32 bits es expandido a 48 bits con la tabla *E*, duplicando algunos bits específicos para asegurar la difusión.
- Se *realiza una operación XOR entre el bloque expandido y la subclave correspondiente a la ronda*.
- El resultado de la operación *XOR* se divide en 8 grupos de 6 bits, y cada uno de estos grupos pasa por una *caja S* diferente (*S_BOXES*), proporcionando un valor de salida de 4 bits por grupo. Este paso introduce la no linealidad en el cifrado.
- Los 32 bits resultantes de las cajas S se permutan con la caja *P* para garantizar confusión de los bits de salida.

2.3. Cifrado de un bloque

La función *encrypt_block* aplica la transformación DES completa a un bloque de 64 bits:

- El bloque es permutado según la tabla IP.
- El bloque permutado se divide en dos mitades de 32 bits, izquierda (L) y derecha (R).
- Ejecución de 16 rondas Feistel:
    - Se calcula la función Feistel sobre la mitad derecha.
    - El resultado se combina (mediante XOR) con la mitad izquierda.
    - Se intercambian ambas mitades (L y R) para la siguiente ronda.
- Tras las rondas, las mitades se intercambian una última vez, formando un bloque de 64 bits.
- Finalmente, el bloque pasa por la permutación inversa *IP INV* obteniendo así el bloque cifrado.

2.4. Cifrado de un archivo en modo CBC

La función *encrypt* realiza el cifrado de archivos utilizando DES en modo CBC:

1. Se convierte el contenido del archivo a una secuencia de bits usando bitarray. Si el tamaño no es múltiplo de 64, se hace padding pkcs7.
2. Se crea una clave aleatoria de 64 bits y un vector de inicialización (Se imprimen por pantalla para poder descifrar).
3. Cada bloque de texto plano se combina mediante XOR con el bloque cifrado anterior (o con el *IV* en el primer caso). El resultado es cifrado con la función *encrypt_block*.

Este procedimiento asegura que bloques idénticos de texto plano generen bloques cifrados distintos, siempre que se varíe el IV.

2.5. Descifrado de un archivo en modo CBC

La función *decrypt* realiza el descifrado del contenido previamente cifrado con el DES CBC:

1. La clave e IV utilizados durante el cifrado deben proporcionarse en formato hexadecimal. Las subclaves se generan nuevamente con la función *generate_subkeys* y se utilizan en orden inverso.

2. Cada bloque cifrado pasa por la función *encrypt_block*, pero usando las subclaves en orden inverso. El resultado se combina mediante XOR con el bloque cifrado anterior (o con el IV).

En el modo de cifrado ECB, se utiliza la misma clave para cada bloque, por lo que si dos bloques son iguales, su salida cifrada también lo será. En cambio, en el modo CBC, el cifrado de cada bloque depende del bloque anterior, lo que hace que incluso los bloques idénticos produzcan resultados diferentes.

Cuando se cifra un texto largo usando ECB, es sencillo detectar patrones repetitivos, lo cual facilita posibles ataques.

Esto mismo ocurre con las imágenes, que suelen contener áreas de color uniforme. Estas zonas se codifican de la misma forma, haciendo que la imagen sea vulnerable si se usa ECB.

b. Programación del Triple DES

El script *tdea-cbc.py* implementa la variante más robusta del DES, el Triple DES (TDEA) utilizando también el modo CBC. Este método incrementa la seguridad frente al DES normal mediante la aplicación sucesiva de cifrados y descifrados con tres claves diferentes, siguiendo un esquema que se define con la siguiente expresión: $E_{k3}(D_{k2}(E_{k1}(P))$. El código es muy similar a lo implementado en el *desCBC.py*, los únicos cambios relevantes son los indicados a continuación.

La función *encrypt_3des_cbc* genera aleatoriamente tres claves independientes, cada una de 64 bits, y luego aplica la función generate_subkeys para cada clave obteniendo así tres conjuntos distintos de subclaves (uno por clave). Para seguir el esquema $E_{k3}(D_{k2}(E_{k1}(P))$, las subclaves del segundo conjunto se invierten para hacer un descifrado, mientras que la primera y tercera se utilizan directamente para cifrar. Durante el cifrado, estas claves generadas se muestran en pantalla en formato hexadecimal, junto con un vector de inicialización (IV), al igual que en el *desCBC.py*.

En la función *decrypt_3des_cbc* las subclaves correspondientes se generan nuevamente y se invierten la primera y la última para reproducir correctamente el esquema de cifrado inverso, es decir, el esquema de descifrado, que sería: $D_{k1}(E_{k2}(D_{k3}(C))$.

3. Principios de diseño del DES

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

Para estudiar la no linealidad de las S-boxes utilizadas en el algoritmo DES, se ha implementado un experimento en el script *linealida_des.py* basado en la comprobación de la siguiente propiedad:

$$f(A \oplus B) \neq f(A) \oplus f(B)$$

Donde *f* es la función aplicada por las cajas S del DES a entradas de 48 bits.

La finalidad de este experimento es verificar si las S-boxes no cumplen dicha propiedad, lo que implicaría un no comportamiento lineal. Al ser un sistema criptográficamente seguro, se espera que la propiedad se cumpla y que las S-boxes sean no lineales, de modo que su salida no pueda predecirse mediante combinaciones lineales de las entradas.

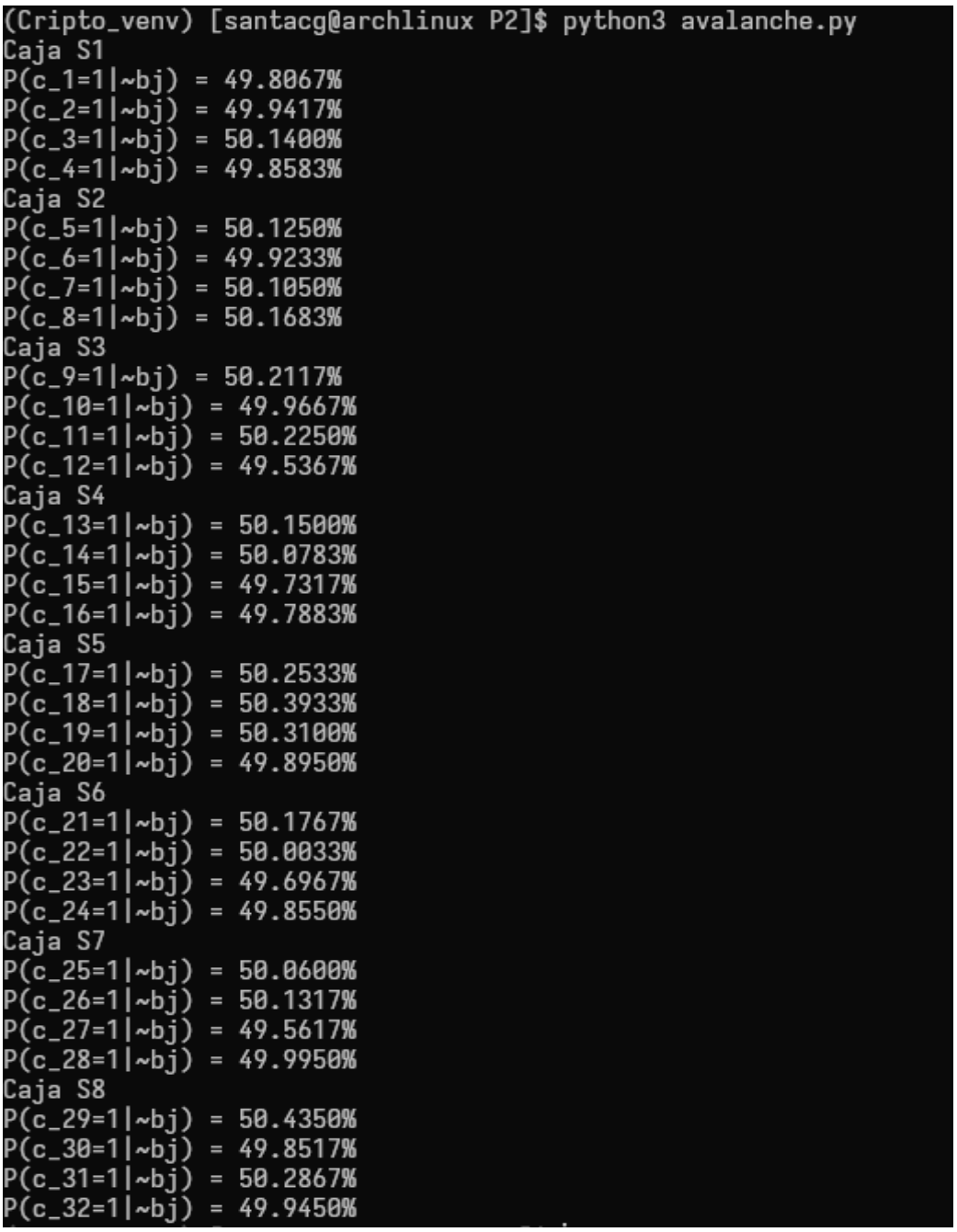
Para evaluar esto, se realizan 100.000 pruebas aleatorias en las que se compara el resultado de aplicar *f* a *A XOR B* con el resultado de aplicar *f* a *A* y a *B* por separado y luego calcular *f(A) XOR f(B)*. En todas las pruebas, los resultados fueron diferentes, indicando que la no linealidad se cumple siempre.

```
(Cripto_venv) [santacg@archlinux P2]$ python3 linealidad_des.py
Tests que NO cumplen la linealidad: 100000 de 100000
Porcentaje de no-linealidad: 100.00%
```

b. Estudio del Efecto de Avalancha

El efecto de avalancha es una propiedad fundamental del DES, esta propiedad establece que un pequeño cambio en la entrada debe provocar un cambio significativo en la salida, en principio, afectando al 50% de los bits resultantes. Este comportamiento asegura una mayor confusión en la relación entre entrada y salida, dificultando cualquier intento de predecir o reconstruir el mensaje original.

Para estudiar esta propiedad, se ha desarrollado un script *avalancha.py* que aplica un flip de un solo bit a cada grupo de 6 bits de entrada de las S-boxes del DES. En cada ejecución, se modifica uno de los 6 bits de entrada de cada S-box, y se observa cuántos de los 4 bits de salida se ven afectados por dicho cambio. Este proceso se repite para un total de 10.000 entradas aleatorias de 48 bits y se calculan las probabilidades medias de que cada uno de los 32 bits de salida (4 por cada una de las 8 S-boxes) tome el valor 1 al modificar un bit de entrada. Los resultados empíricos muestran que todos los bits de salida presentan una probabilidad muy cercana al 50% de activarse (tener el valor 1) cuando se altera uno de los bits de entrada, lo cual refleja un alto efecto de avalancha. A continuación, se muestran algunos de los porcentajes obtenidos:



Esto confirma que las S-boxes del algoritmo DES fueron diseñadas para cumplir eficazmente con el principio del efecto de avalancha, lo cual es crucial para garantizar la difusión y la seguridad del cifrado.

4. Principios de diseño del AES

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

Al igual que en el caso del DES, se ha realizado una comprobación empírica de la no linealidad de las S-boxes del AES mediante el script *linealidad_aes.py*. En este caso, se ha utilizado directamente la tabla de

sustitución estándar del AES (*S-box directa*), la cual opera sobre 1 byte y devuelve también un byte como salida.

El procedimiento seguido es análogo al del estudio realizado en el DES, es decir, se comprueba si se cumple $f(A \oplus B) \neq f(A) \oplus f(B)$. En este caso se generaran 1.000.000 de entradas aleatorias comprobando la no linealidad para cada par de entradas, y se obtiene lo siguiente:

```
(Cripto_venv) [santacg@archlinux P2]$ python3 linealidad_aes.py
Tests que NO cumplen la linealidad: 996162 de 1000000
Porcentaje de no-linealidad: 99.62%
```

Se observa que en el 99.62% de los casos la propiedad no se cumple, lo que indica que la S-box del AES es altamente no lineal, aunque no alcanza el 100% como ocurre con las S-boxes del DES. Esta pequeña desviación puede explicarse por la diferente naturaleza algebraica de ambas funciones, mientras que las S-boxes del DES están específicamente diseñadas con tablas construidas a mano para evitar cualquier comportamiento lineal, la S-box del AES se construye algebraicamente como la inversa multiplicativa en el campo finito de Galois GF(2^8), seguida de una transformación afín. Esta construcción permite un diseño más matemático, pero también introduce una mínima cantidad de casos en los que la propiedad de linealidad se cumple.

A pesar de ello, un 99.62% de no linealidad es más que suficiente para garantizar la resistencia del AES frente a ataques de tipo lineal, y confirma que su S-box cumple adecuadamente con dicho principio.

b. Generación de las S-boxes del AES

La generación de las cajas se ha implementado en el script *sbox_aes.py*, para realizar esta generación se ha utilizado la biblioteca *galois*, que permite operar de forma nativa sobre campos finitos de Galois. En concreto, se define el cuerpo GF(2^8) con el polinomio irreducible utilizado por el AES: $x^8 + x^4 + x^3 + x + 1$ representado en forma hexadecimal como *0x11B*.

El procedimiento seguido para construir ambas tablas es el siguiente:

- S-box directa:

    - Para cada uno de los 256 posibles valores de entrada (1 byte de entrada), se calcula su inverso multiplicativo en GF(2^8), salvo en el caso del valor 0, que por convenio se asigna a sí mismo.
    - Al resultado de esta inversión se le aplica una transformación afín bit a bit, definida como una combinación lineal de bits desplazados del byte, combinados mediante XOR con una constante fija c = 0x63.
- S-box inversa:
    - Para cada uno de los posibles valores de salida, se aplica primero la transformación afín inversa, con su correspondiente constante d = 0x05.
    - Al resultado se le aplica nuevamente la inversión multiplicativa en GF(2^8), reconstruyendo así la entrada original asociada a cada valor de la tabla directa.

Los resultados obtenidos coinciden con las tablas reales, primero se muestra la caja directa y luego la inversa:

```
0x63 0x7C 0x77 0x7B 0xF2 0x6B 0x6F 0xC5 0x30 0x01 0x67 0x2B 0xFE 0xD7 0xAB 0x76
0xCA 0x82 0xC9 0x7D 0xFA 0x59 0x47 0xF0 0xAD 0xD4 0xA2 0xAF 0x9C 0xA4 0x72 0xC0
0xB7 0xFD 0x93 0x26 0x36 0x3F 0xF7 0xCC 0x34 0xA5 0xE5 0xF1 0x71 0xD8 0x31 0x15
0x04 0xC7 0x23 0xC3 0x18 0x96 0x05 0x9A 0x07 0x12 0x80 0xE2 0xEB 0x27 0xB2 0x75
0x09 0x83 0x2C 0x1A 0x1B 0x6E 0x5A 0xA0 0x52 0x3B 0xD6 0xB3 0x29 0xE3 0x2F 0x84
0x53 0xD1 0x00 0xED 0x20 0xFC 0xB1 0x5B 0x6A 0xCB 0xBE 0x39 0x4A 0x4C 0x58 0xCF
0xD0 0xEF 0xAA 0xFB 0x43 0x4D 0x33 0x85 0x45 0xF9 0x02 0x7F 0x50 0x3C 0x9F 0xA8
0x51 0xA3 0x40 0x8F 0x92 0x9D 0x38 0xF5 0xBC 0xB6 0xDA 0x21 0x10 0xFF 0xF3 0xD2
0xCD 0x0C 0x13 0xEC 0x5F 0x97 0x44 0x17 0xC4 0xA7 0x7E 0x3D 0x64 0x5D 0x19 0x73
0x60 0x81 0x4F 0xDC 0x22 0x2A 0x90 0x88 0x46 0xEE 0xB8 0x14 0xDE 0x5E 0x0B 0xDB
0xE0 0x32 0x3A 0x0A 0x49 0x06 0x24 0x5C 0xC2 0xD3 0xAC 0x62 0x91 0x95 0xE4 0x79
0xE7 0xC8 0x37 0x6D 0x8D 0xD5 0x4E 0xA9 0x6C 0x56 0xF4 0xEA 0x65 0x7A 0xAE 0x08
0xBA 0x78 0x25 0x2E 0x1C 0xA6 0xB4 0xC6 0xE8 0xDD 0x74 0x1F 0x4B 0xBD 0x8B 0x8A
0x70 0x3E 0xB5 0x66 0x48 0x03 0xF6 0x0E 0x61 0x35 0x57 0xB9 0x86 0xC1 0x1D 0x9E
0xE1 0xF8 0x98 0x11 0x69 0xD9 0x8E 0x94 0x9B 0x1E 0x87 0xE9 0xCE 0x55 0x28 0xDF
0x8C 0xA1 0x89 0x0D 0xBF 0xE6 0x42 0x68 0x41 0x99 0x2D 0x0F 0xB0 0x54 0xBB 0x16
```

```
0x52 0x09 0x6A 0xD5 0x30 0x36 0xA5 0x38 0xBF 0x40 0xA3 0x9E 0x81 0xF3 0xD7 0xFB
0x7C 0xE3 0x39 0x82 0x9B 0x2F 0xFF 0x87 0x34 0x8E 0x43 0x44 0xC4 0xDE 0xE9 0xCB
0x54 0x7B 0x94 0x32 0xA6 0xC2 0x23 0x3D 0xEE 0x4C 0x95 0x0B 0x42 0xFA 0xC3 0x4E
0x08 0x2E 0xA1 0x66 0x28 0xD9 0x24 0xB2 0x76 0x5B 0xA2 0x49 0x6D 0x8B 0xD1 0x25
0x72 0xF8 0xF6 0x64 0x86 0x68 0x98 0x16 0xD4 0xA4 0x5C 0xCC 0x5D 0x65 0xB6 0x92
0x6C 0x70 0x48 0x50 0xFD 0xED 0xB9 0xDA 0x5E 0x15 0x46 0x57 0xA7 0x8D 0x9D 0x84
0x90 0xD8 0xAB 0x00 0x8C 0xBC 0xD3 0x0A 0xF7 0xE4 0x58 0x05 0xB8 0xB3 0x45 0x06
0xD0 0x2C 0x1E 0x8F 0xCA 0x3F 0x0F 0x02 0xC1 0xAF 0xBD 0x03 0x01 0x13 0x8A 0x6B
0x3A 0x91 0x11 0x41 0x4F 0x67 0xDC 0xEA 0x97 0xF2 0xCF 0xCE 0xF0 0xB4 0xE6 0x73
0x96 0xAC 0x74 0x22 0xE7 0xAD 0x35 0x85 0xE2 0xF9 0x37 0xE8 0x1C 0x75 0xDF 0x6E
0x47 0xF1 0x1A 0x71 0x1D 0x29 0xC5 0x89 0x6F 0xB7 0x62 0x0E 0xAA 0x18 0xBE 0x1B
0xFC 0x56 0x3E 0x4B 0xC6 0xD2 0x79 0x20 0x9A 0xDB 0xC0 0xFE 0x78 0xCD 0x5A 0xF4
0x1F 0xDD 0xA8 0x33 0x88 0x07 0xC7 0x31 0xB1 0x12 0x10 0x59 0x27 0x80 0xEC 0x5F
0x60 0x51 0x7F 0xA9 0x19 0xB5 0x4A 0x0D 0x2D 0xE5 0x7A 0x9F 0x93 0xC9 0x9C 0xEF
0xA0 0xE0 0x3B 0x4D 0xAE 0x2A 0xF5 0xB0 0xC8 0xEB 0xBB 0x3C 0x83 0x53 0x99 0x61
0x17 0x2B 0x04 0x7E 0xBA 0x77 0xD6 0x26 0xE1 0x69 0x14 0x63 0x55 0x21 0x0C 0x7D
```