Pontificia Universidad Católica de Chile
Departamento de Ciencia de la computación
IIC325 - Criptografia y Seguridad Computacional

Tarea 2

20 de junio de 2021

1º semestre 2021

José Manuel Domínguez - 17637449

Pregunta 1

a)

Recordemos que DES es una red de Feistel de 16 pasos. El key-schedule de DES funciona de la siguente manera. Primero entran 64 bits al bloque PC-1 (Permuted Choice 1), de estos 64 bits este se queda con 56, esto lo hace botando los bits 8, 16, 24, ... , 64 y estos se usan como bit de paridad. Luego, estos 56 bits son divididos en la mitad obteniendo 2 mitades de 28 bits (c_0 y d_0). En adelante estas dos mitades se tratan de forma separada.

Luego, cada una de las mitades se les hace un left-shift (más bien un left-rotate osea que el primer bit pasa a ser el último) le llamaremos a cada shift l_i con $i \in \{1, ..., 16\}$, i denota la ronda actual. Se rota 1 bit si es que $i = 1, 2, 9, 16$ y 2 en otro caso. Vale la pena mencionar que el total de shifts que se hace es 28 por lo que $c_0 = c_{16}$ y $d_0 = d_{16}$.

Para obtener la sub-llave de cada ronda se aplica un left-rotate a cada mitad de la llave y luego se pasa por el bloque bloque PC-2 (Permuted Choice 2), el cual también bota 8 bits dejándonos con una sub-llave de 48 bits. Se eligen 24 bits de la primera mitad c_i y 24 bits de la segunda mitad d_i y luego se concatenan ambas mitades.

b)

Buscamos romper DES de 3 pasos. Primero vamos a definir algunas notaciones útiles. Como sabemos el mensaje inicial se divide en 2 mitades estas les llamaremos l_0 y r_0, estas mitades pasan 3 veces por la red osea que en el paso i le llamaremos l_i y r_i. Al pasar el mensaje por la red DES de 3 pasos obviando la permutación final obtenemos l_3 y r_3. A las distintas redes de Feistel le llamaremos f_i, donde

$$f_1 = f(k_1, r_0) \tag{1}$$
$$f_2 = f(k_2, r_1) \tag{2}$$
$$f_2 = f(k_3, r_2) \tag{3}$$

Entonces dado lo anterior al pasar un mensaje conocido m y obtener de salida c, sabemos l_0, r_0 y l_3, r_3. Por principio de Kerckhoffs, asumimos que conocemos las funciones f_i con $i \in \{1, 2, 3\}$ y conocemos el key scheduler.

Como vimos en la parte anterior del key schedule, las llaves se pueden tratar como 2 mitades independientes entre si, la llave al pasar por el key scheduler se divide en 2 mitades independientes, las cuales se concatenan al final. Sabiendo esto y además conociendo f_1, f_2 y f_3 sabemos exactamente qué bits del output de los f_i afecta cada mitad.

Para romper la red haremos un ataque de "meet in the middle". Nos enfocaremos particularmente en r_1. Dado la estructura de la red $r_1 = l_0 \oplus f(k_1, r_0)$ (le diremos por arriba). Además, si vamos hacia el otro lado también $r_1 = r_3 \oplus f(k_3, r_2)$, también sabemos que $r_2 = l_3$ por lo que lo podemos escribirlo de la siguiente forma $r_1 = r_3 \oplus f(k_3, l_3)$ (le diremos por abajo)

$$r_1 = l_0 \oplus f(k_1, r_0) \tag{4}$$
$$r_1 = r_3 \oplus f(k_3, l_3) \tag{5}$$

Por lo tanto, podemos calcular r_1 por ambos lados variando solo la llave, ya que conocemos todos los elementos menos la llave. Como mencionamos previamente sabemos exactamente que bits del output de f_i afecta que mitad de la llave, entonces lo que debemos hacer es ir probando todas las llaves por ambos lados hasta que coincidan los bits de ambos r_1 (el de arriba con el de abajo).

Como la llave es de 56 bits, (inicialmente 64 pero 8 bits se usan de paridad). cada mitad tiene 28 bits, lo que nos deja con 2^{28} posibles llaves distintas para cada mitad. Nos vamos a concentrar en la 1era mitad primero. Lo que hacemos es obtener r_1 por arriba (4) y r_1 por abajo(5), luego, vemos si los bits de salida correspondientes a la 1era mitad calzan. Hacemos esto para cada una de las 2^{28} posibles llaves. De esta forma ejecutamos $2 * 2^{28}$ veces la red de sustitución/permutación. 2^{28} veces f_1 y 2^{28} veces f_3. De esta forma obtenemos la primera mitad de la llave en 2^{29} pasos. La segunda mitad sería lo mismo y nuevamente nos dejaría con 2^{29}. Esto nos deja un total de $2 * 2^{29} = 2^{30}$ pasos totales. Pero, por la naturaleza del key scheduler con 1 solo mensaje no podemos estar seguro si es efectivamente la llave correcta ya que tenemos 4 grados de libertad (pasamos de 28 bits a 24 bits), por lo que necesitamos más mensajes, lo que nos dejaría con un número un poco mayor a 2^{30} pasos, habría que probar las llaves candidatas en un segundo mensaje, en un tercero e incluso en un 4to para estar 100 % seguros. Si hay una llave que calza para todos los mensajes podemos estar seguros que esta es la llave.

Creo que esto se podría optimizar un poco ya que se puede ir variando ambas mitades de la llave al mismo tiempo. Esto se puede hacer ya que una mitad siempre afecta los mismos bits de salida de f_i, por lo que podría variarlos a la vez y revisar de forma independiente si cada mitad está correcta. Esto nos dejaría con 2^{29} ejecuciones totales de la red sustitución/permutación. 2^{28} ejecuciones por arriba, más 2^{28} ejecuciones por abajo.

Llamaremos a cada mitad de la llave kr_i y kl_i con $i \in \{1, ..., 2^{28}\}$. Entonces lo que haremos sería partir ejecutando la red por arriba con kr_1 y kl_1, lo mismo que por debajo, si los bits correspondientes a la 1era mitad coinciden por arriba y por abajo sabemos que esa podría ser una posible primera mitad y guardamos el valor, se aplica la misma lógica para la segunda mitad. Hacemos esto mismo para kr_2 y kl_2, ..., hasta $kr_{2^{28}}$ y $kl_{2^{28}}$. Esto nos deja con 2^{29} pasos totales (2^28 por arriba y 2^{28} por abajo). Pero como vimos anteriormente con 1 mensaje no basta para estar seguros que es la llave por lo que probamos todos los candidatos en 3 mensajes más para de esta forma estar 100 % seguros de que efectivamente tenemos la llave correcta. Esto nos deja con un total de ejecuciones entre 2^{29} y 2^{30} pasos.