PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IIC3253 - Criptografía y Seguridad Computacional
1er semestre del 2021
Vicente Merino

Tarea 2

Pregunta 1

Parte (a)

Podemos considerar el *key-schedule*, como una función de la forma:

$$\gamma : \{0,1\}^{64} \rightarrow (\{0,1\}^{48}, \dots \{0,1\}^{48}) \text{ (16 elementos)}$$

Ahora, debemos tomar en cuenta algo que se llaman los P-Boxes, que son esencialmente permutaciones que se realizan a los bits de la llave original (de 64 bits). Es importante notar que el octavo bit de cada byte de la llave corresponde a un bit de paridad. Definimos las P-Boxes **PC1** (Permuted Choice 1) y **PC2** (Permuted Choice 2) como las siguientes tablas, donde cada fila corresponde a un byte del número binario resultante:

PC1 =

Byte 1	57	49	41	33	25	17	9	63
Byte 2	55	47	39	31	23	15	1	58
Byte 3	50	42	34	26	18	7	62	54
Byte 4	46	38	30	22	10	2	59	51
Byte 5	43	35	27	14	6	61	53	45
Byte 6	37	29	19	11	3	60	52	44
Byte 7	36	21	13	5	28	20	12	4

Entonces si para **PC1** tomamos como input el número K = 11010000 00100111 11000100 11111000 10101000 10101101 11011110 10111001, el shuffle se puede ver de la siguiente forma:

Byte 1	K[57]=1	K[49]=1	K[41]=1	K[33]=1	K[25]=1	K[17]=1	K[9]=0	K[63]=0
Byte 2	K[55]=1	K[47]=0	K[39]=0	K[31]=0	K[23]=0	K[15]=1	K[1]=1	K[58]=0
Byte 3	K[50]=1	K[42]=0	K[34]=0	K[26]=1	K[18]=1	K[7]=0	K[62]=0	K[54]=1
Byte 4	K[46]=1	K[38]=0	K[30]=0	K[22]=1	K[10]=0	K[2]=1	K[59]=1	K[51]=0
Byte 5	K[43]=1	K[35]=1	K[27]=1	K[14]=1	K[6]=0	K[61]=1	K[53]=1	K[45]=1
Byte 6	K[37]=1	K[29]=1	K[19]=0	K[11]=1	K[3]=0	K[60]=1	K[52]=1	K[44]=0
Byte 7	K[36]=0	K[21]=0	K[13]=0	K[5]=0	K[28]=1	K[20]=0	K[12]=0	K[4]=1

Por lo que el resultado de la P-Box PC1, es 11111100 10000110 10011001 10010110 11110111 11010110 00001001. Podemos ver que aquí los bits de paridad desaparecen, lo cual es uno de los objetivos de esta

P-Box, además de permutar la llave. Podemos notar también que esta P-Box puede ser vista como una función de la forma:

$$\text{PC1} : \{0,1\}^{64} \to \{0,1\}^{56}$$

Ahora la P-Box PC2 está definida como:

PC2 =

Byte 1	14	17	11	24	1	5	3	28
Byte 2	15	6	21	10	23	19	12	4
Byte 3	26	8	16	7	27	20	13	2
Byte 4	41	52	31	37	47	55	30	40
Byte 5	51	45	33	48	44	49	39	56
Byte 6	34	53	46	42	50	36	29	32

Análogamente si tomamos como input K = 110111 111110 000100 000101 101010 110010 101111 110100, entonces el shuffle de la P-Box viene dado por:

Byte 1	K[14]=1	K[17]=1	K[11]=0	K[24]=1	K[1]=1	K[5]=1	K[3]=1	K[28]=1
Byte 2	K[15]=1	K[6]=1	K[21]=1	K[10]=0	K[23]=0	K[19]=0	K[12]=0	K[4]=1
Byte 3	K[26]=0	K[8]=0	K[16]=0	K[7]=0	K[27]=0	K[20]=1	K[13]=0	K[2]=1
Byte 4	K[41]=1	K[52]=0	K[31]=1	K[37]=0	K[47]=1	K[55]=0	K[30]=1	K[40]=1
Byte 5	K[51]=0	K[45]=0	K[33]=1	K[48]=0	K[44]=1	K[49]=0	K[39]=1	K[56]=1
Byte 6	K[34]=1	K[53]=1	K[46]=1	K[42]=1	K[50]=0	K[36]=1	K[29]=0	K[32]=0

Por lo que el output termina siendo 110111 111110 000100 000101 101010 110010 101111 110100. Está P-Box también se puede ver como una función de la forma:

$$\text{PC2} : \{0,1\}^{56} \to \{0,1\}^{48}$$

Otras dos P-Boxes de utilidad para el *key-schedule* son RHL(1) y RHL(2), que hacen un *shfit rotate* a la izquierda de cada mitad del número binario en una y dos posiciones respectivamente, esto, nuevamente se puede ilustrar por las siguientes tablas:

RHL(1) =

Byte 1	2	3	4	5	6	7	8	9
Byte 2	10	11	12	13	14	15	16	17
Byte 3	18	19	20	21	22	23	24	25
Byte 4	26	27	28	1	30	31	32	33
Byte 5	34	35	36	37	38	39	40	41
Byte 6	42	43	44	45	46	47	48	49
Byte 7	50	51	52	53	54	55	56	29

Entonces si tomamos como input K = 11111100 10000110 10011001 10010110 11110111 11010110 00001001, el output termina siendo 11111001 00001101 00110011 00111101 11101111 10101100 00010010. Esta *rotate left* puede ser vista como una función de la forma:

$$\text{RHL(1)} : \{0,1\}^{56} \to \{0,1\}^{56}$$

RHL(2) =

Byte 1	3	4	5	6	7	8	9	10
Byte 2	11	12	13	14	15	16	17	18
Byte 3	19	20	21	22	23	24	25	26
Byte 4	27	28	1	2	31	32	33	34
Byte 5	35	36	37	38	39	40	41	42
Byte 6	43	44	45	46	47	48	49	50
Byte 7	51	52	53	54	55	56	29	30

Entonces si tomamos como input K = 11111100 10000110 10011001 10010110 11110111 11010110 00001001, el output termina siendo 11110010 00011010 01100110 01111101 11101111 10101100 000100101 Esta *rotate left* puede ser vista como una función de la forma:

$$\text{RHL}(2) : \{0,1\}^{56} \to \{0,1\}^{56}$$

Esto puede ser visto también como que el input se divide en dos mitades y cada mitad hace un *left rotate* en uno o dos bits.

Ahora que tenemos todas estas definiciones podemos explicar correctamente como funciona el algoritmo de *key-schedule*:

1. Se genera la llave $k \in \{0,1\}^{64}$, con los bits de paridad incluidos.
2. Se utiliza esta llave como input de la P-Box PC1 y se obtiene $\hat{k} = \text{PC1}(k)$
3. Estamos en la ronda $i = 1$
4. Se hace un *left rotate* con alguna de las P-Box y se obtiene $\hat{k}_i = \text{RHL}(1)(\hat{k})$, o bien $\hat{k}_i = \text{RHL}(2)(\hat{k})$. El tipo de *left rotate* depende de cada ronda y se detalla más abajo.
5. Se genera la primera sub-llave, utilizando el P-Box PC2 con el resultado anterior $k_i = \text{PC2}(\hat{k}_i)$.
6. Hacemos $i = i + 1$ y volvemos al paso 4, pero ahora usamos como input $\hat{k}_{i-1}$, es decir desde el paso 2 en adelante, se calcula $\hat{k}_i = \text{RHL}(1)(\hat{k}_{i-1})$ o $\hat{k}_i = \text{RHL}(2)(\hat{k}_{i-1})$. Se sigue hasta que $i = 16$ y se hayan generado las 16 subllaves.
7. Se retorna $(k_1, k_2, \ldots, k_{16})$.

El tipo de *left rotate* que se aplica en cada ronda viene definido por la siguiente tabla:

Ronda	*left rotate*
1	RHL(1)
2	RHL(1)
3	RHL(2)
4	RHL(2)
5	RHL(2)
6	RHL(2)
7	RHL(2)
8	RHL(2)
9	RHL(1)
10	RHL(2)
11	RHL(2)
12	RHL(2)
13	RHL(2)
14	RHL(2)
15	RHL(2)
16	RHL(1)

El siguiente diagrama ilustra correctamente como se ve el algoritmo de *key-schedule*:

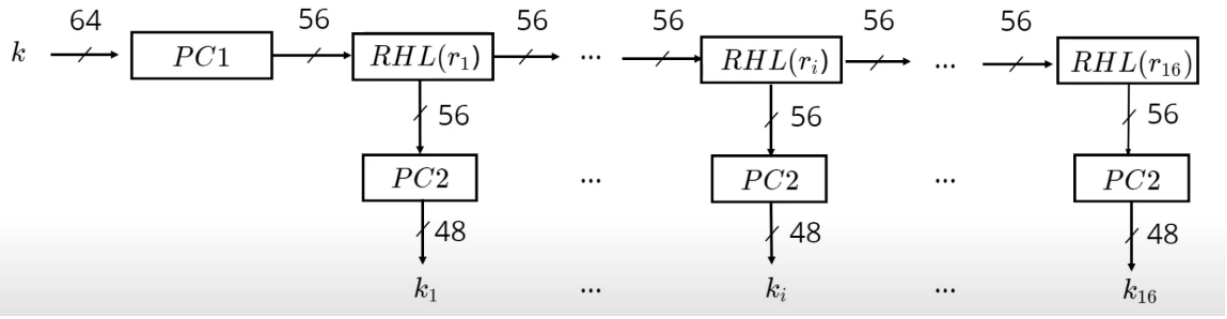

Figura 1: Diagrama del algoritmo de *key-schedule*

Esta parte fue elaborado en base al video DES - Part 1 - The Key schedule Algorithm y al artículo A Detailed Description of DES and 3DES Algorithms (Data Encryption Standard and Triple DES).

Parte (b)

En un DES de 3 rondas, tenemos que las ecuaciones vienen dadas por:

$$L_1 = R_0 \tag{1}$$

$$R_1 = L_0 \oplus f(k_1, R_0) \tag{2}$$

$$L_2 = R_1 \tag{3}$$

$$R_2 = L_1 \oplus f(k_2, R_1) \tag{4}$$

$$L_3 = R_2 \tag{5}$$

$$R_3 = L_2 \oplus f(k_3, R_2) \tag{6}$$

Si en (6) reemplazamos L_2 por R_1, y este a su vez por (1), obtenemos $R_3 = L_0 \oplus f(k_1, R_0) \oplus f(k_3, R_2)$. Si hacemos $\oplus L_0$ a ambos lados y reemplazamos R_2 por L_3, finalmente queda:

$$R_3 \oplus L_0 = f(k_1, R_0) \oplus f(k_3, L_3) \tag{7}$$

Si inspeccionamos (7), podemos ver que todas las variables son conocidas, ya que son parte del input o el output, a excepción de k_1 y k_3. (Que intentaremos adivinar). El procedimiento para romper el sistema es el siguiente:

0. Envío $m = L_0||R_0$ por el sistema y recibo $c = L_3||R_3$. Es decir, L_0, R_0, L_3, R_3 son conocidos.
1. Genero todas las combinaciones posibles de un número binario de 28 bits, que vendrán a representar los posibles valores de la mitad izquierda de $\hat{k}$ (la salida de PC1(k)). Este conjunto lo llamaremos $\hat{K}$.
2. Para cada instancia $\hat{k} \in \hat{K}$, genero la mitad izquierda de k_1 y k_3, según el algoritmo de *key-scheduling*, es decir $k_1 = PC2(RHL(1)(\hat{k}))$ y $k_3 = PC2(RHL(2)(RHL(1)(RHL(1)(\hat{k}))))$.
3. Calculo la mitad izquierda de $f(k_1, R_0)$ y la mitad izquierda de $f(k_3, L_3)$. (Más adelante veremos porqué esta función igual retorna las mitades izquierdas y derechas).
4. Calculo el $\oplus$ de los resultados del paso 3. y el $\oplus$ entre L_0 y R_3 (mitad izquierda). Si son distintos, vuelvo a 2. para las otras instancias. Si son iguales, quiere decir que acerté en la mitad izquierda de $\hat{k}$ y vuelvo a 1. ahora revisando para la mitad derecha (las operaciones de los pasos siguientes, también serían para la parte derecha).

5. Una vez obtenidas ambas mitades de $\hat{k}$, puedo recomponer k agregando los bits de paridad correspondientes y habré descifrado la llave.

Generar y hacer los cómputos correspondientes para cada mitad de la llave, toma 2^{28} (ya que son números de 28 bits). Por lo tanto descifrar la llave, puede tomar no más de $2 \cdot 2^{28} = 2^{29}$ pasos (más algún error bajo, en las distintas operaciones que hay que hacer, pero que sigue siendo menor a 2^{30}). Notar que en realidad no es necesario generar todos los números en el paso 1, lo que podría ser muy costoso, sino ir generando cada instancia a medida que se va revisando.

Ahora veamos, porqué funciona segmentar por mitades de llave. En primer lugar si vemos como se generan las distintas subllaves, vemos que se va propagando por medios de varios *left rotates*, sin embargo si vemos las tablas de RHL, podemos ver que esta rotación siempre se hace por segmentos (dos mitades), por lo que nunca se perderá información del $\hat{k}$ original si calculo una mitad y después la otra, solo se cambiará el orden de los bits al rotar cada mitad. Solamente se pierde información (se reduce el tamaño del output, frente al input) al hacer el PC2, pero con esto estamos listos porque hemos generado la subllave correspondiente y algo muy importante a notar es que el PC2 también mantiene el orden de ambas mitades, si vemos la tabla de PC2, podemos ver que entre los bytes 1 al 3, se mantienen en el rango de los bits 1 al 28, y de los bytes 4 al 6, se mantienen en el rango desde el 29 al 56, por lo que esta operación también mantiene la divisibilidad de la llave.

En segundo lugar, debemos ver como se computa la función f. Una vez generadas las llaves, estamos listos, ya que si bien se realiza una expansión, esta es de L_0, lo cual no afecta en nada a la divisibilidad de la llave, y al calcular las S-Boxes, esto se realiza byte a byte, por lo que también permite mantener la disivilidad. También es importante que como las operaciones $\oplus$ se hacen bit a bit, también permiten mantener la divisibilidad de la llave.

En conclusión, por todas estas propiedades (en particulas gracias a la forma de las P-Boxes de *key-scheduling*), es posible recuperar (atacar el protocolo), en no más de 2^{30} pasos.