

LABORATORIO 2: PERMUTATIONS

Torres Olivera Karla Paola

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
kpto997@gmail.com

Resumen: El presente trabajo consiste en la descripción e implementación en C++ de un algoritmo perteneciente a los cifrados por bloque, más específicamente DES simplificado. Para esto, primero se elaborará un programa capaz de realizar las permutaciones a nivel de caracter y a nivel de bit.

Palabras Clave: permutación, cifrado, descifrado.

1 Introducción

Históricamente, la seguridad ofrecida por un cifrado se basaba en gran medida en mantener en secreto su algoritmo de cifrado/descifrado. La criptografía moderna considera que tales cifrados no proporcionan un nivel adecuado de seguridad. Por ejemplo, no pueden ser utilizados por un grupo más grande de usuarios. Un problema surge cuando alguien quiere abandonar el grupo, esto provocaría que los demás deban cambiar el algoritmo. Dicho procedimiento se repetiría en caso de que alguien lo revele.

Un algoritmo de cifrado secreto tendría que estar diseñado de manera única para cada grupo de usuarios, lo que excluye la posibilidad de implementaciones de software o hardware listas para usar. De lo contrario, el adversario podría comprar un producto idéntico y ejecutar los mismos algoritmos de cifrado/descifrado.

La criptografía moderna resuelve los problemas de seguridad anteriores de tal manera que, por lo general, el cifrado utilizado es conocido públicamente, pero su ejecución de cifrado/descifrado utiliza una información privada adicional, llamada clave criptográfica, la cual es otro parámetro de entrada. Una clave generalmente es denotada por la letra K . Esta puede tomar un valor dentro de un amplio rango de posibles valores, generalmente números [1].

Tomando en cuenta lo anterior en esta práctica se mostrará un ejemplo de cifrado por bloque, los cuales pertenecen a la criptografía moderna. El algoritmo DES simplificado, nos servirá para entender de una manera más sencilla cómo funciona DES normal, debido a que como su nombre lo indica es una pequeña versión de este.

2 Conceptos Básicos

2.1 Cifradores por bloques

Como su nombre lo indica, los cifradores por bloque operan en “bloques” de datos. Tales bloques son típicamente de 64 o 128 bits de longitud, los cuales se transforman en bloques del mismo tamaño bajo la acción de una clave secreta.

Este tipo de cifrado, cifra un bloque de texto sin formato o mensaje m en otro bloque del mismo tamaño llamado c haciendo uso de una clave secreta k . Esto normalmente se denotará como $c = \text{ENC}_k(m)$. La forma exacta de la transformación de cifrado estará determinada por la elección del cifrado de bloque y el valor de la clave k . El proceso de cifrado se invierte mediante el descifrado, que utilizará la misma clave proporcionada por el usuario. Esto se denotará por $m = \text{DEC}_k(c)$.

Un cifrado de bloque tiene dos parámetros importantes:

1. El tamaño del bloque, que será denotado por b , y
2. El tamaño de la clave, que se indicará con k

El mapeo para este tipo de cifrados es una permutación del conjunto de entradas y, a medida que se varia la clave secreta, se obtienen diferentes permutaciones. Por lo tanto, un cifrado de bloque es una forma de generar una familia de permutaciones mediante una clave secreta k .

El tamaño de bloque b determina el espacio de todas las permutaciones posibles que pueda generar un cifrado de bloque. Por otro lado, el tamaño de la clave k determina el número de permutaciones que realmente se generan.

Las operaciones básicas de sustitución y permutación son particularmente importantes para este tipo de cifrado. La mayoría, si no todos, los cifradores por bloque contendrán alguna combinación de ambas, aunque la forma exacta de la sustitución y la permutación puede variar mucho [2].

2.1.1 Operación básica: Sustitución

La sustitución se usa a menudo como una forma de proporcionar confusión dentro de un cifrado. Tal sustitución podría diseñarse alrededor de una función aritmética como la suma de enteros o la multiplicación de enteros. Más típicamente, la sustitución se logra con una tabla de búsqueda, un cuadro de sustitución o lo que simplemente se denomina una *S-box*, las cuales deben diseñarse cuidadosamente para tener propiedades de seguridad específicas. Una desventaja es que las *S-box* generalmente necesitan ser almacenadas, incorporando así algunas restricciones de memoria. Siempre que las cajas no sean demasiado grandes, es poco probable que esto cause un problema significativo. Sin embargo, en las implementaciones de hardware exigentes, el espacio puede

ser escaso, mientras que en las implementaciones de software, las interacciones entre la *S-box* y la memoria caché en el procesador pueden conducir a variaciones de tiempo que pueden estar relacionadas con información clave [2].

2.1.2 Operación básica: Permutación

Las permutaciones a menudo se usan para contribuir a la buena difusión en un cifrado. Muy a menudo, la permutación se realiza a nivel de bit, DES es un ejemplo claro de este enfoque. Sin embargo, las permutaciones a nivel de bit tienen importantes implicaciones de rendimiento [2].

2.1.3 Redes SP

Como ya se mencionó anteriormente, la combinación de operaciones de sustitución y permutación es un componente importante para la mayoría de los diseños de cifrado de bloques. De hecho, una clase importante de cifradores por bloque (véase Figura 1) está diseñada utilizando solo estas operaciones. Llamadas redes de sustitución/permutación o simplemente redes SP, estas consisten en la aplicación repetida de una sustitución cuidadosamente elegida, una permutación y la adición de la clave. Una de las características importante de los cifrados por bloque es que dependen de varias rondas de cómputo. La función de la programación de claves es presentar una serie de claves para cada ronda de cifrado, las cuales se calculan a partir de de la ingresada por el usuario [2].

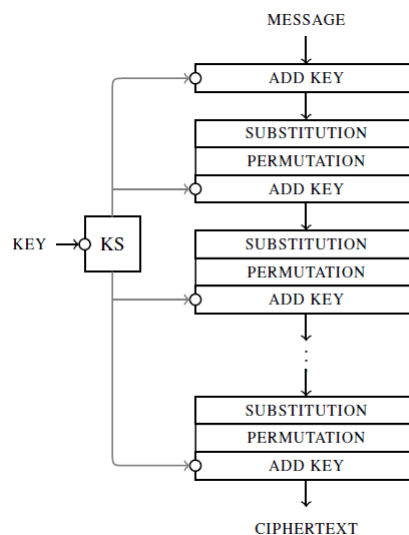


Figura 1: Red SP. La clave suministrada por el usuario se procesa utilizando un programa de claves (KS) para derivar un conjunto de claves redondas.

2.2 Modos de operación

Un cifrador por bloques como DES o AES se puede utilizar de varias maneras para cifrar una cadena de datos. Poco después de que se estandarizara DES, apareció otro estándar federal de EE.UUU., el cual ofrecía cuatro formas recomendadas de utilizarlo para el cifrado de datos. Desde entonces, estos modos de operación se han estandarizado internacionalmente y se pueden usar con cualquier cifrado de bloque. Los cuatro modos presentados en [3] son:

- Modo ECB: es fácil de usar, pero sufre posibles ataques de eliminación e inserción. Un error de bit en el texto cifrado genera un error de un bloque completo en el texto plano descifrado.
- Modo CBC: este es probablemente el mejor de los cuatro modos de operación originales, ya que ayuda a proteger contra ataques de eliminación e inserción. En este modo, un error de bit en el texto cifrado da no solo un error de un bloque en el texto plano correspondiente, sino también un error de bit en el siguiente bloque de texto plano descifrado.
- Modo OFB: este modo convierte un cifrado de bloque en un cifrado de flujo. Tiene la propiedad de que un error de un bit en el texto cifrado da un error de un bit en el texto plano descifrado.
- Modo CFB: este modo también convierte un cifrado de bloque en un cifrado de flujo. Un error de un solo bit en el texto cifrado afecta tanto a ese bloque como al siguiente, al igual que en el modo CBC.

A lo largo de los años se han presentado varios otros modos de operación. Probablemente el más popular presentado en [3] es:

- Modo CTR: este además de convertir el cifrado de bloque en un cifrado de flujo, también permite que los bloques se procesen en paralelo, lo que proporciona ventajas de rendimiento cuando el procesamiento en paralelo está disponible.

2.3 DES Simplificado

Des Simplificado es un algoritmo de cifrado por bloques. Utiliza un tamaño de bloque de 8 bits, y una clave secreta de 10 bits. Esto significa que si tenemos un mensaje de tamaño mayor, éste deberá partirse en bloques de 8 bits. Esto puede resultar muy conveniente, sobretodo si se esta trabajando con caracteres codificados en código ASII en el cual cada caracter se codifica con 8 bits, de esta manera se tomaría caracter por caracter y se cifraría usando la clave. El S-DES es un algoritmo de clave secreta, cuyo funcionamiento esta basado en el uso de permutaciones y sustituciones. Su estructura general se muestra en la Figura 2.

Como se puede observar en la Figura 2, para cifrar un bloque, inicialmente se aplica una *permutación inicial* (IP); una *función* f_k ; una permutación simple SW que lo único que hace es intercambiar las mitades derecha e izquierda del bloque; luego se vuelve a aplicar la función f_k y finalmente se aplica nuevamente una permutación que de hecho es la *permutación inversa* (IP^{-1}).

Para descifrar un bloque, se llevan a cabo las mismas operaciones, lo único que cambia es el orden en el que se utilizan las subclaves $k1$ y $k2$ que son generadas de la clave k .

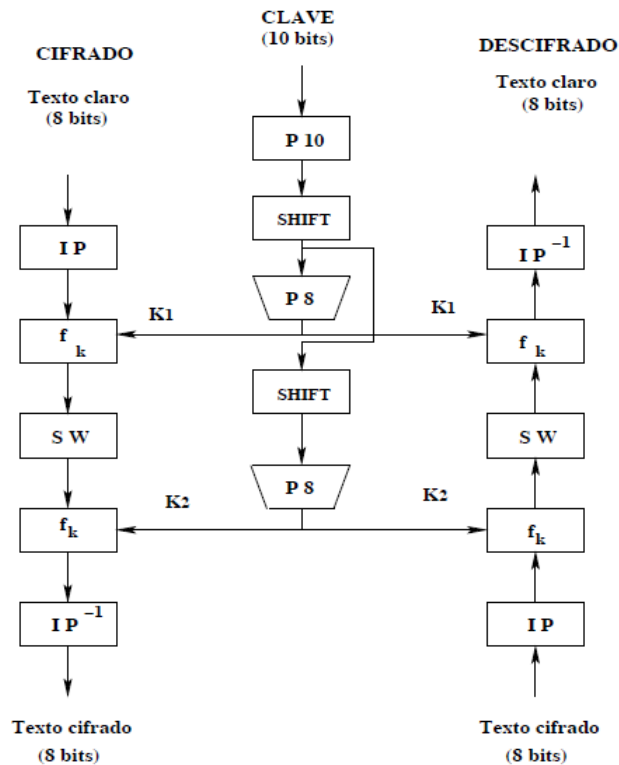


Figura 2: Diagrama general del DES

3 Experimentación y Resultados

3.1 Programa 1. Permutación de caracteres

3.1.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Se hizo uso de una variable global de tipo `map<int, int>`, con la finalidad de guardar la tabla de permutación encontrada en un `.txt` ingresado por el usuario. Al momento de leer dicho archivo su contenido se guarda como un `string`, por lo que, se tuvo que tokenizar usando como limitadores los espacios en blanco y posteriormente se hizo uso del comando `stoi` para convertir a entero.

Las permutaciones a nivel de bit funcionan de manera correcta para 8 bits. Sin embargo, dependiendo de la tabla de permutación puede funcionar para 16 bits. El programa que realiza dicha tarea se presentará en el apartado de DES Simplificado.

3.1.2 Permutación Inversa

La función encargada de calcular la permutación inversa (véase Listing 1) recibe como parámetro la cadena que tiene almacenado el renglón de los valores de Π_i . Posteriormente, como se mencionó en la sección 3.1 se divide dicha cadena consiguiendo los números pertenecientes a la tabla de permutación.

Es importante recordar que para calcular la permutación inversa primero se deben invertir los renglones y posteriormente acomodar los valores de Π_i^{-1} de manera ascendente. Sin embargo, haciendo uso de un `map` el procedimiento se reduce únicamente a la primera parte, debido a que, este tipo de dato acomoda los valores de las llaves de menor a mayor de manera automática.

Tomando en cuenta lo anterior, mientras se va separando la cadena se van a agregando los valores al `map` como llaves, por otro lado, su pareja se asigna con ayuda de un contador el cual simula los valores de i de la tabla de permutación. Finalmente, se escribe el `map` en un archivo de texto plano.

```

1 void inversePermutation( string *valueofPhi ){
2     string previous = "";
3     register int auxCont = 1;
4
5     for( register int i = 0; i < (*valueofPhi).size(); i++ ){
6         previous = previous + (*valueofPhi)[i];
7         if( (*valueofPhi)[i] == ' ' ){
8             tablePermutation.insert( pair<int, int>( stoi(
previous, nullptr, 10 ), auxCont ) );
9             auxCont++;
10            previous = "";
11        }
12    }

```

13 }

Listing 1: Función permutación inversa

El programa en ejecución así como los resultados obtenidos para este punto se muestran en las Figuras 3 y 4 respectivamente.

```

PROBLEMS  TERMINAL  ...  1: powershell v
Calculate inverse permutation
Permutation table file: PermutationTable.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY
\Laboratorio2\PermutationProgram>

```

Figura 3: Programa en ejecución

PermutationTa...	InversePermut...
Archivo Edición Formato Ver Ayuda	Archivo Edición Formato Ver Ayuda
1 2 3 4 5 6	1 2 3 4 5 6
6 4 5 2 1 3	5 4 6 2 3 1
10 Windows (CRLF) UTF-8	10 Windows (CRLF) UTF-8

a)
b)

Figura 4: a) Tabla de permutación ingresada b) Tabla permutación inversa

3.1.3 Generación permutación aleatoria

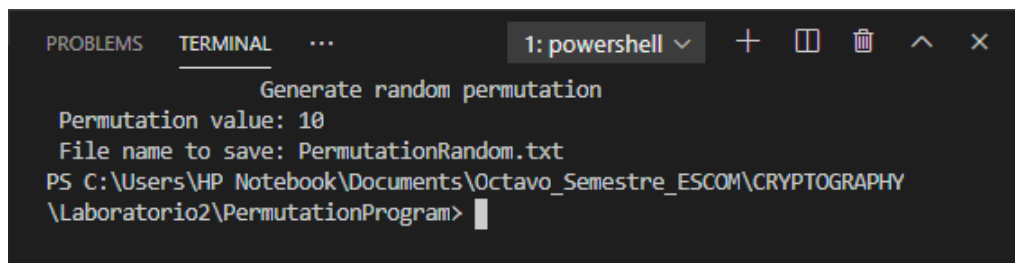
En caso de que el usuario desee generar una tabla de permutación aleatoria, debe ingresar el tamaño de esta (n) y el nombre del archivo con el que se guardará. Para la implementación de esta función se hizo uso de un `vector<int>` el cual se inicializó con números consecutivos en el rango de 1 to n . Posteriormente, con ayuda de la función `shuffle` se reacomodo el vector de manera “aleatoria”.

El único parámetro que recibe la función encargada de llevar a cabo esta tarea pertenece al tamaño de la tabla de permutación, mientras que su retorno corresponde al vector resultante de la función `shuffle`. El código fuente se observa en el Listing 2.

```
1 vector<int> randomPermutation( int n ){  
2  
3     vector<int> randomNumbers;  
4  
5     for( register int i = 0; i < n; i++ )  
6         randomNumbers.push_back( i+1 );  
7  
8     unsigned seed = chrono::system_clock::now().  
9     time_since_epoch().count();  
10  
11     shuffle ( randomNumbers.begin(), randomNumbers.end(),  
12     default_random_engine( seed ) );  
13  
14     return randomNumbers;  
15 }
```

Listing 2: Función generación de permutación aleatoria

El programa en ejecución, así como los resultados obtenidos para este punto se muestran en las Figuras 5 y 6 respectivamente.



```
PROBLEMS  TERMINAL  ...  1: powershell v + [ ] [X] ^ X  
Generate random permutation  
Permutation value: 10  
File name to save: PermutationRandom.txt  
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY  
\Laboratorio2\PermutationProgram>
```

Figura 5: Programa en ejecución

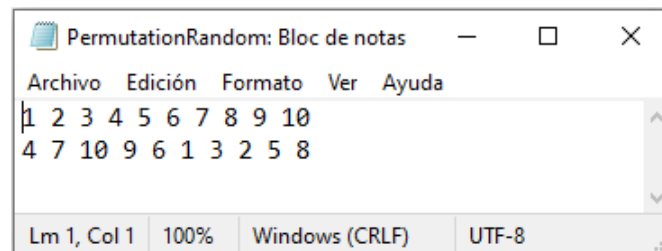


Figura 6: Archivo con la permutación generada

3.1.4 Permutation Cipher (cifrado y descifrado)

Para el cifrado y descifrado se hizo uso de la misma función. Sin embargo, se consideraron varios aspectos:

- Para realizar el cifrado primero se verificó que el tamaño del mensaje modulo el tamaño de la tabla de permutación fuera cero, en caso contrario, se agregaron caracteres especiales.
- Para cifrar se realizó una función encargada de mapear los valores de la tabla de permutación a la variable global `map<int, int>`.
- Para descifrar se le pide al usuario que ingrese la tabla de permutación original, posteriormente se manda llamar a la función encargada de encontrar el inverso (véase sección 3.2) y se escribe en un archivo *.txt*. Esta es la que se guarda en la variable `map` y con la que se trabaja en la función de cifrado/descifrado.
- Los resultados que se obtienen del cifrado y descifrado se guardan en archivos *.txt* separados, cuyos nombres empiezan con *Enc* y *Dec* respectivamente.

Una vez teniendo lo anterior procedemos a explicar el código fuente. Se hizo uso de memoria dinámica con la finalidad de ir modificando los caracteres de ciertas posiciones sin perder sus valores. En seguida, se hizo un ciclo `for` el cual recorre todo el contenido del archivo ingresado por el usuario y se busca dentro del `map` que contiene la tabla de permutación la posición a la cual se debe mover el caracter. Finalmente, realizamos el cambio correspondiente. Para determinar en qué bloque nos encontramos y realizar dicho cambio de manera correcta se hace uso de la operación modulo y de una variable bloque, la cual aumenta la cantidad del tamaño de la tabla de permutación cada vez que la posición del archivo en la que nos encontremos sea múltiplo de dicho tamaño.

El código fuente de la función encargada de realizar el cifrado y descifrado se visualiza en el Listing 3. Sus parámetros corresponden al texto a cifrar/descifrar y al texto cifrado/descifrado dependiendo de cual sea el caso.

```

1 void encryptAndDecryptPermutation( string *contenttoEncrypt,
  string *finalText ){
2
3     int position, bloque = 0;
4
5     char *messagePrevious = new char[ (*contenttoEncrypt).size
  () ];
6     for( register int i = 0; i < (*contenttoEncrypt).size(); i
  ++ ){
7
8         position = tablePermutation.find( ( i %
  tablePermutation.size() ) + 1 )->second;
9         messagePrevious[ position - 1 + bloque ] = (*
  contenttoEncrypt)[ i ];
10
11         if( ( i != 0 ) && ( ( i + 1 ) % tablePermutation.size()
  == 0 ) )
12             bloque = bloque + tablePermutation.size();
13
14     }
15
16     messagePrevious[ (*contenttoEncrypt).size() ] = '\0';
17     (*finalText) = messagePrevious;
18     cout<< " El mensaje cifrado es: " << *finalText << endl;
19
20     delete[] messagePrevious;
21
22 }

```

Listing 3: Función para cifrar y descifrar

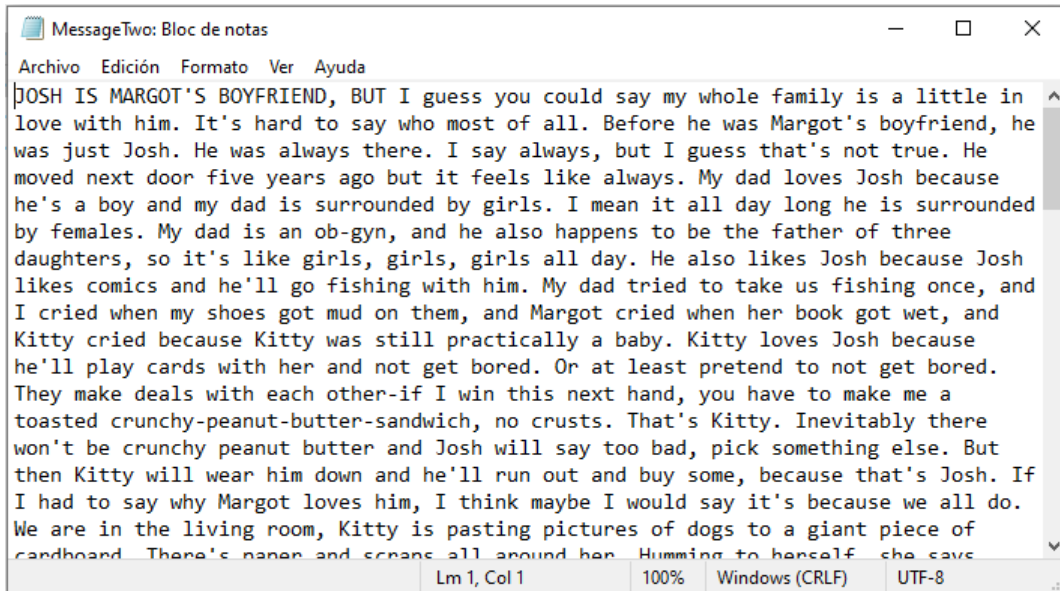
El programa en ejecución así como los resultados obtenidos, para el caso de cifrado se muestran en las Figuras 7 y 8 respectivamente.

```

PROBLEMS  TERMINAL  ...  2: powershell
Encrypt Permutation Cipher
File name to Encrypt: MessageTwo.txt
File name to Permutation: Permutation15.txt
Do you want encrypt/decrypt at the bit level? No
Special characters added: 11
El mensaje cifrado es: SHO SJA'TR OMIGBO YRSNUBDI EF,I ugeT c ys su
os.to aB lfm lor ehewaMaef soti' rynefbgosr hsewd tja,s ush .HJwl
ote eoe oyai b arvseg iltfulkisete al wye adMs .aylo vsdoebs Jehus
sshredouer uby. edlM em afsdaodiya-bns n,a nghsledy a ha peo ebtn
ayal s ekoJel ios boeasehs uhscJikse ma iclosc hfelngsiold ' ngm i
u nd emo emr,at ogMnhd acr idthehe wenboeo rt,t g okwndr i yei tatK
pl a lriwdc ays hoe td t ahrnt rbrg.a eedoOleeattrdne psto bnt er

```

Figura 7: Programa en ejecución



(a) Texto original

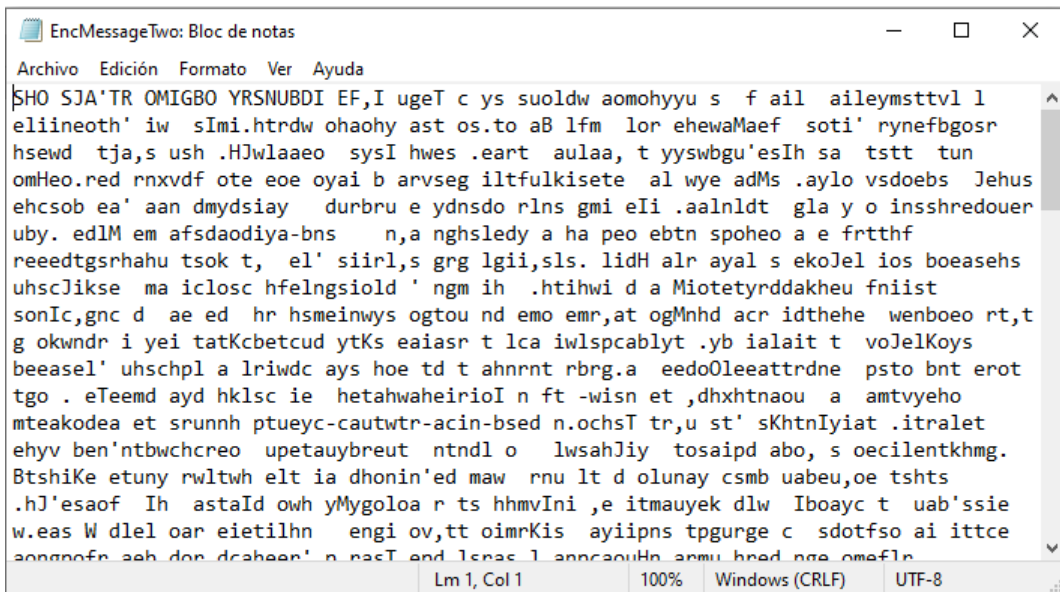
Permutation15: Bloc de n...

Archivo Edición Formato Ver Ayuda

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

Ln 100% Windows (CRLF) UTF-8

(b) Tabla de permutación a utilizar



(c) Texto cifrado


Figura 8: Resultados obtenidos para el cifrado

Por otro lado, el programa en ejecución, así como los resultados obtenidos para el descifrado se visualizan en las Figuras 9 y 10 respectivamente.

```

PROBLEMS  TERMINAL  ...  2: powershell
Decrypt Permutation Cipher
File name to Decrypt: EncMessageTwo.txt
File name to Permutation: Permutation15.txt
Do you want encrypt/decrypt at the bit level? No
El mensaje cifrado es: JOSH IS MARGOT'S BOYFRIEND, BUT I guess you
most of all. Before he was Margot's boyfriend, he was just Josh. He
  
```

Figura 9: Programa en ejecución

 InversePermutation15: Bloc d...
Archivo Edición Formato Ver Ayuda
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
3 4 13 5 7 1 10 15 14 11 8 2 9 6 12
Lm 1, 100% Windows (CRLF) UTF-8

(a) Tabla de permutación inversa

JOSH IS MARGOT'S BOYFRIEND, BUT I guess you could say my whole family is a little in love with him. It's hard to say who most of all. Before he was Margot's boyfriend, he was just Josh. He was always there. I say always, but I guess that's not true. He moved next door five years ago but it feels like always. My dad loves Josh because he's a boy and my dad is surrounded by girls. I mean it all day long he is surrounded by females. My dad is an ob-gyn, and he also happens to be the father of three daughters, so it's like girls, girls, girls all day. He also likes Josh because Josh likes comics and he'll go fishing with him. My dad tried to take us fishing once, and I cried when my shoes got mud on them, and Margot cried when her book got wet, and Kitty cried because Kitty was still practically a baby. Kitty loves Josh because he'll play cards with her and not get bored. Or at least pretend to not get bored. They make deals with each other-if I win this next hand, you have to make me a toasted crunchy-peanut-butter-sandwich, no crusts. That's Kitty. Inevitably there won't be crunchy peanut butter and Josh will say too bad, pick something else. But then Kitty will wear him down and he'll run out and buy some, because that's Josh. If I had to say why Margot loves him, I think maybe I would say it's because we all do. We are in the living room, Kitty is pasting pictures of dogs to a giant piece of cardboard. There's paper and scraps all around her. Humming to herself. she says.

Lm 1, Col 1 100% Windows (CRLF) UTF-8

(b) Texto descifrado

Figura 10: Resultados obtenidos para el descifrado

3.2 Programa 2. DES Simplificado

3.2.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Algunas de las consideraciones que se tomaron en cuenta para su realización fueron:

- Se diseñó una biblioteca la cual lleva a cabo la tarea de generación de llave para el caso del cifrado, lectura de la misma para el caso del descifrado, así como de las permutaciones y corrimientos necesarios para obtener las subllaves. Posteriormente, las funciones encontradas en esta biblioteca fueron llamadas en el programa encargado de cifrar y descifrar.
- Las tablas de permutación se declararon de manera local en la función encargada de realizar dicho intercambio de valores.
- Las matrices correspondientes a S_1 y S_2 se declararon de manera global en la implementación de la biblioteca encargada de realizar todo el proceso de cifrado y descifrado.
- Para permutar los bits del caracter se hizo uso de operaciones a nivel de bit, como son: corrimientos y mascarar. Una vez teniendo esto, se cambio al tipo de dato `bitset` para llevar a cabo todo el procedimiento restante.

3.2.2 Generación de la llave

Para la generación de la llave se hizo uso de `uniform_int_distribution` debido a que esta función permite generar valores aleatorios en el intervalo $[0, 1]$. Adicional a esto, hace uso de la fórmula de probabilidad discreta para generar dichos valores. Por otro lado, se hizo uso de `mt19937` para que cada vez que se corra el programa se genere una cadena distinta, esta función esta basada en el algoritmo *Mersenne Twister*, el cual tiene un mejor comportamiento estadístico que `rand`.

El código de la función encargada de generar la llave aleatoria se muestra en el Listing 4.

```

1 void generateKey( string *key ){
2     unsigned seed = std::chrono::system_clock::now().
      time_since_epoch().count();
3     std::mt19937 generator(seed);
4     std::uniform_int_distribution<int> distribution(0,1);
5
6     for ( size_t i = 0; i < 10; i++ )
7         *key = (*key) + to_string( distribution(generator));
8 }

```

Listing 4: Función para generar la llave

3.2.3 Permutación de bits

La función encargada de realizar la permutación a nivel de bit se visualiza en el Listing 5. Esta recibe como parámetro el caracter al cual le aplicará la permutación IP de S-DES.

Lo primero que se realizó fue consultar la tabla de permutación IP definida en el algoritmo para determinar la posición a la cual se moverá el bit i , posteriormente con ayuda de corrimientos y la compuerta AND se consultó el valor del bit que será cambiado a la posición Π_i . Por último, se prendió o apago el bit correspondiente haciendo uso de compuertas y de corrimientos.

```

1 void applyPermutationIP( char *messagetoEncrypt ){
2     long int position;
3     int tablePermutation[8] = {3, 1, 7, 2, 4, 6, 8, 5}, onOrOff
4     ;
5     char characterAux = *messagetoEncrypt;
6
7     for( register int i = 0; i < 8; i++ ){
8         position = tablePermutation[i] - 1;
9         onOrOff = ( characterAux & ( 1 << i ) );
10
11         if( onOrOff == 0 )
12             (*messagetoEncrypt) &= ~( 1 << position );
13
14         else
15             (*messagetoEncrypt) |= ( 1 << position );
16     }
17 }
```

Listing 5: Función para permutar bits

3.2.4 Cifrado S-DES

La función que realiza el cifrado se muestra en el Listing 6. Sin embargo, en la función `roundOne` se llaman a las demás, las cuales llevan a cabo todas las permutaciones necesarias, la búsqueda en las $S - Box$, la segunda ronda para el cifrado y finalmente la permutación IP^{-1} .

```

1 void EncryptSDES( char *messagetoEncrypt, bitset<8> subKey1,
2     bitset<8> subKey2, string nametoFile ){
3
4     applyPermutationIP( messagetoEncrypt );
5     roundOne( messagetoEncrypt, subKey1, subKey2, "Encrypt" );
6 }
```

Listing 6: Función para cifrar

Debido a que el procedimiento para la ronda 1 y 2 son muy similares, el código se implementó de tal manera que se pudiera reutilizar para ambos casos.

Eso se logró dividiendo cada una de las etapas en pequeñas funciones, además de que esto nos permitió tener un mejor manejo del código. Las funciones utilizadas para ambos casos se explicarán a continuación.

La primera función es la que realiza la permutación de expansión (véase Listing 7). Como se mencionó anteriormente, se hizo uso del tipo de dato `bitset` con la finalidad de tener un mejor manejo de los bits.

Un aspecto importante que se me había olvidado mencionar es que las tablas de permutación se modificaron, debido a que en la computadora la posición que el algoritmo maneja como la 8 es la 0. Sin embargo, el resultado es el mismo, ya que, solo se busca su equivalente.

Para realizar esta permutación, se pusieron en la tabla primero las dos posiciones a las cuales se moverá el bit 1, posteriormente las otras dos a las cuales se moverá el bit 2, y así sucesivamente. Con ayuda del `for` vamos modificando los valores correspondientes.

```

1 void expansionPermutationByte( bitset<8> *expansionPermutation,
   bitset<4> partRight ){
2
3     int onOrOff, tablePermutation[8] = { 2, 8, 3, 5, 4, 6, 1, 7
   }, aux = 0;
4
5     for( register int i = 0; i < 8; i+=2 ){
6         (*expansionPermutation)[ tablePermutation[i] -1 ] =
   partRight[ aux ];
7         (*expansionPermutation)[ tablePermutation[i+1] -1 ] =
   partRight[ aux ];
8         aux++;
9     }
10
11 }

```

Listing 7: Función para realizar la permutación de expansión

La segunda función pertenece a la encargada de realizar la operación *XOR* del resultado obtenido anteriormente con la subllave, así como la búsqueda en las *S – Box*. Su implementación se visualiza en el Listing 8.

Uno de sus parámetros de esta función pertenece a un entero llamado *type*, esta variable fue utilizada como bandera para determinar si se debía buscar en S_1 o S_2 , eso conlleva a que también sirve para determinar con qué parte de la permutación estamos trabajando y para realizar la operación *XOR* únicamente una vez. Para poder obtener el valor entero del `bitset` y hacer la búsqueda en la *S – Box* correspondiente se hizo de su método `.to_ulong()`, así como un cast a entero.

Las operaciones que no son explicadas a detalle, es debido a que corresponden únicamente a la división del `bitset` que contiene la permutación extendida en su parte izquierda y derecha con ayuda de ciclos `for`.

```

1 void xorOperationAndFindTable( bitset<8> *operatorOne, bitset
  <8> operatorTwoKey, bitset<2> *resultTable, int type ){
2
3     bitset<2> column, row;
4     bitset<16> resultBinaryTable;
5     int numCol, numRows, result;
6
7     if( type == 1 ){
8         (*operatorOne) ^= operatorTwoKey;
9         for( register int i = 0; i < 2; i++ )
10             column[i] = (*operatorOne)[i + 5];
11
12         row[0] = (*operatorOne)[4];      row[1] = (*operatorOne)
[7];
13     }
14     else{
15         for( register int i = 0; i < 2; i++ )
16             column[i] = (*operatorOne)[i + 1];
17
18         row[0] = (*operatorOne)[0];      row[1] = (*operatorOne)
[3];
19     }
20
21     numCol = (int)(column.to_ulong());
22     numRows = (int)(row.to_ulong());
23
24     if( type == 1 )
25         result = S0[numRows][numCol];
26     else
27         result = S1[numRows][numCol];
28
29     resultBinaryTable = bitset<16>(result);
30     (*resultTable)[0] = resultBinaryTable[0];
31     (*resultTable)[1] = resultBinaryTable[1];
32
33 }

```

Listing 8: Función para realizar la XOR y la búsqueda en las $S - Box$

Un paso intermedio para el cual no se realizó ninguna función fue la concatenación de los valores encontrados en las $S - BOX$, sin embargo, esa línea de código (véase Listing 9) también es común tanto para la ronda 1 y 2.

```

1 S0andS1 = bitset<4>( tableLeft.to_string() + tableRight.
  to_string() );

```

Listing 9: Concatenación de los números pertenecientes a S_1 y S_2

La última función pertenece a la encargada de realizar la permutación $P4$ y la operación XOR entre el resultado de esta y la parte izquierda obtenida en una de las etapas anteriores. El código fuente se muestra en el Listing 10.

Para llevar a cabo estas acciones, se definió de manera local la tabla de permutación y al igual que en los casos anteriores con ayuda de un ciclo `for` se realizaron los cambios correspondientes. Posteriormente, se realizó la operación *XOR*, es por esto que uno de los parámetros de la función corresponde a la parte izquierda obtenida en una de las etapas anteriores. Por último, se concatena el resultado con la parte derecha original generando el resultado final.

```

1 void P4andXor( bitset<4> *S0andS1, bitset<4> partLeft, bitset
  <4> partRight, bitset<8> *resultRound1 ){
2
3     int P4[] = {4, 2, 1, 3};
4     bitset<4> resultP4;
5
6     for( register int i = 0; i < 4; i++ )
7         resultP4[i] = (*S0andS1)[ P4[i] - 1 ];
8
9     resultP4 ^= partLeft;
10
11     (*resultRound1) = bitset<8>( resultP4.to_string() +
12     partRight.to_string() );
13 }

```

Listing 10: Función para la permutación P_4 y operación *XOR*

La ronda 2 a diferencia de la 1 hace un paso de más correspondiente a la permutación inversa (P^{-1}). En el Listing 11 se muestra la función diseñada para llevar a cabo dicha tarea. Para esto se hizo uso de un ciclo `for` el cual realiza la permutación P^{-1} y con ayuda de corrimientos vamos sumando las potencias en las cuales la cadena binaria tiene un 1. Esto se realizó con la finalidad de obtener su valor en decimal, para posteriormente en la función `roundOne` transformarlo a su correspondiente ASCII con ayuda de la instrucción `static_cast<char>`

```

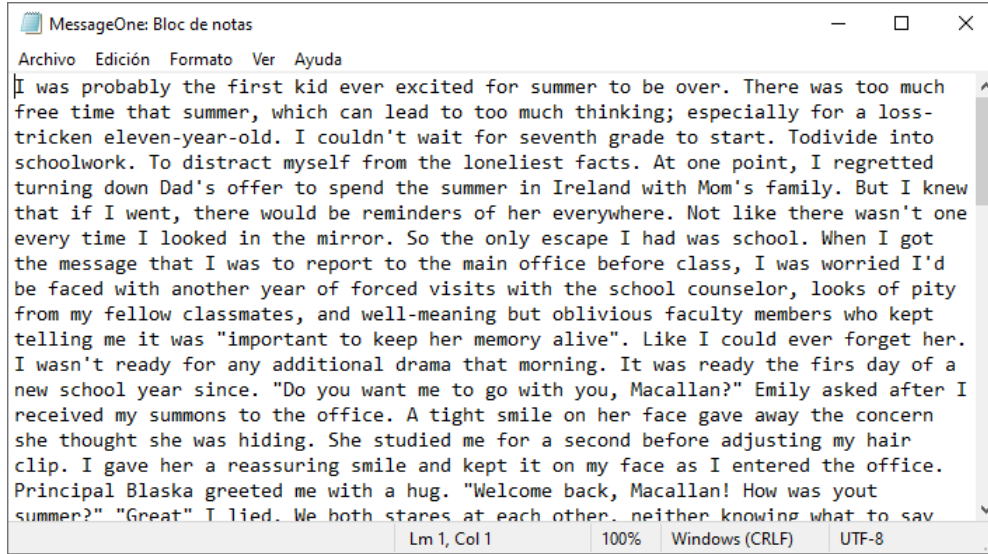
1 void permutationInverse( bitset<8> resultRound2, long int *
  potencia ){
2
3     int permutationTable[] = {3, 1, 7, 2, 4, 6, 8, 5};
4     bitset<8> resultInverse;
5
6     for( register int i = 0; i < 8; i++ )
7         if( ( resultRound2[ permutationTable[i] - 1 ] ) == 1 )
8             *potencia += ( 1 << i );
9 }

```

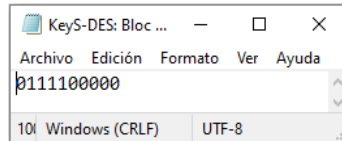
Listing 11: Función para obtener la permutación P^{-1}

Para verificar que el programa funciona de manera adecuada, se realizó una prueba con un archivo de texto de 5KB mandando cada uno de los caracteres

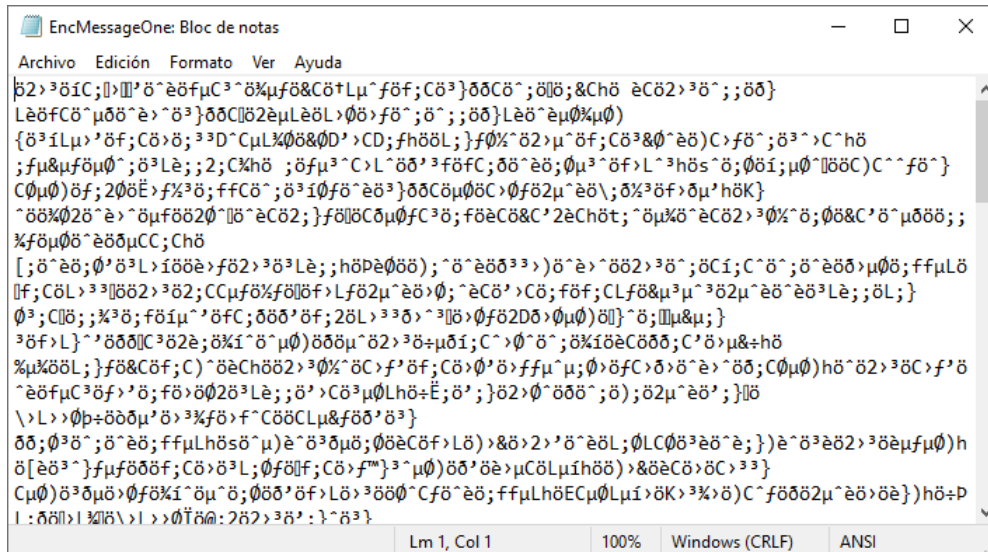
de dicho mensaje a la función de cifrado, los resultados obtenidos se muestran en las Figuras 12 y 11.



(a) Texto original



(b) Llave generada



(c) Texto cifrado

Figura 11: Resultados obtenidos para el cifrado

```

PROBLEMS  TERMINAL  ...  1: powershell  +  [ ]  [X]  ^  X

          Encrypt S-DES
File to encrypt: MessageOne.txt
Subllave 1: 00010100
Subllave 2: 01010100
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY
\Laboratorio2\S-DESProgram>

```

Figura 12: Programa en ejecución

3.2.5 Descifrado S-DES

El procedimiento de cifrado y descifrado de S-DES es el mismo con excepción de que ahora en la ronda 1 se trabaja con la subllave 2 y en la ronda 2 con la 1. Tomando esto en cuenta, el código se implementó de tal manera que con ayuda de banderas se pudieran realizar estos cambios.

Las funciones correspondientes al cifrado (véase Listing 6) y descifrado (véase Listing 12) al momento de llamar a la función `roundOne` mandan como un parámetro la acción a realizar. Esto se debe a que en dicha función se hizo uso de una condicional, la cual dependiendo del valor de este parámetro manda la subllave correspondiente a la función `xorOperationAndFindTable`. Este procedimiento se repite en la función `roundTwo`. El pequeño fragmento de código que realiza esto se muestra en el Listing 13.

```

1 void DecryptSDES( char *messagetoEncrypt, bitset<8> subKey1,
   bitset<8> subKey2, string nametoFile ){
2
3     applyPermutationIP( messagetoEncrypt );
4     roundOne( messagetoEncrypt, subKey1, subKey2, "Decrypt" );
5
6 }

```

Listing 12: Función para descifrar

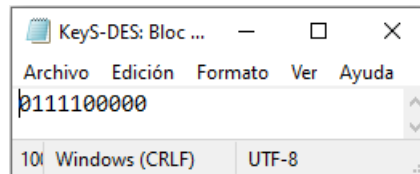
```

1     if( actiontoDo.compare("Decrypt") ){
2         xorOperationAndFindTable( &expansionPermutationRound2,
   subKey2, &tableLeft, 1 );
3         xorOperationAndFindTable( &expansionPermutationRound2,
   subKey2, &tableRight, 2 );
4     }
5     else{
6         xorOperationAndFindTable( &expansionPermutationRound2,
   subKey1, &tableLeft, 1 );
7         xorOperationAndFindTable( &expansionPermutationRound2,
   subKey1, &tableRight, 2 );
8     }

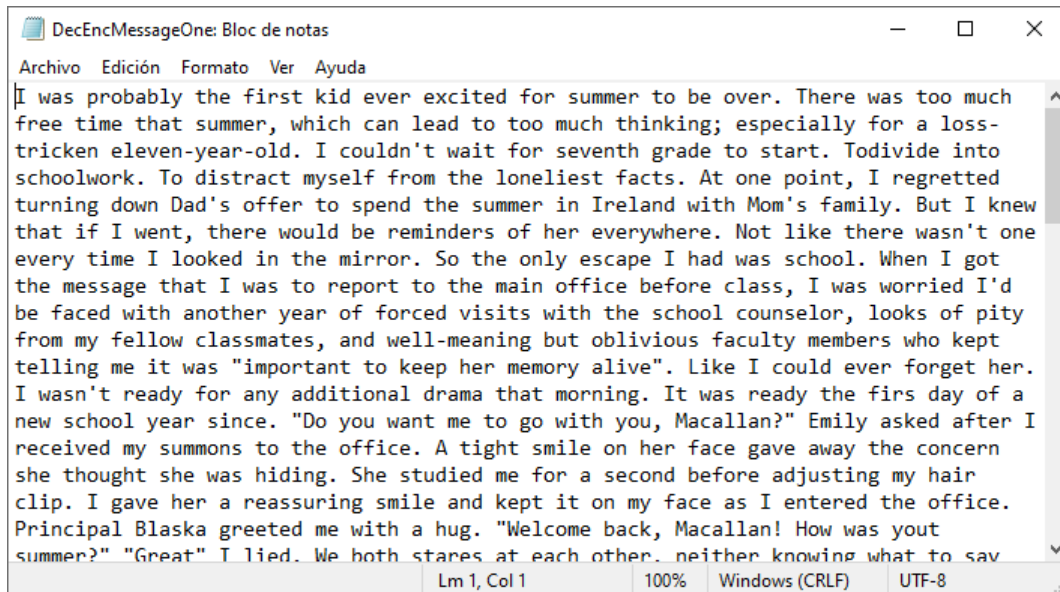
```

Listing 13: Validación para determinar qué subclave mandar

El código restante que se utilizó para el descifrado es el mismo que se explicó en la sección 3.2.4. Los resultados obtenidos para el ejemplo de prueba se muestran en las Figuras 13 y 14.



(a) Llave utilizada para el cifrado



(b) Texto descifrado

Figura 13: Resultados obtenidos para el descifrado

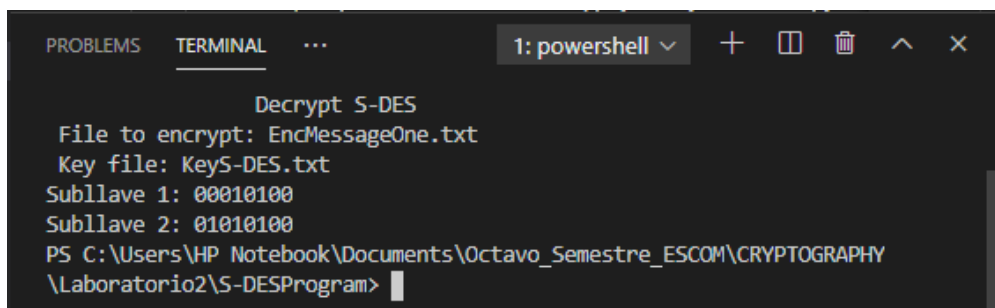


Figura 14: Programa en ejecución

3.3 Modo de operación OFB

3.3.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Algunas de las consideraciones que se tomaron en cuenta fueron:

- Se diseñaron dos bibliotecas, la primera realiza todos los procesos necesarios para generar y almacenar el vector de inicialización, así como para obtenerlo en el caso del descifrado. El segundo realiza la tarea de mandar a llamar a las funciones de S-DES, se podría decir que es el programa principal.
- Para la generación del vector de inicialización se hizo uso de las instrucciones explicadas en la sección 3.2.2.

3.3.2 Generación del vector de inicialización

La implementación de la función encargada de generar el vector de inicialización se muestra en el Listing 14. Como se mencionó anteriormente, las funciones utilizadas fueron las mismas que para la generación de llave en S-DES. Sin embargo, ahora el ciclo `for` es de 0-7.

```

1 void vectorIVGeneration( string *vectorIV ){
2
3     unsigned seed = std::chrono::system_clock::now().
time_since_epoch().count();
4     std::mt19937 generator(seed);
5     std::uniform_int_distribution<int> distribution(0,1);
6
7     for ( size_t i = 0; i < 8; i++ )
8         *vectorIV = (*vectorIV) + to_string( distribution(
generator));
9
10    /*vectorIV = "10111101";
11 }
```

Listing 14: Función para generar el vector IV

Una vez teniendo el vector de inicialización se hace uso de otra función la cual almacena dicho valor en un *.txt* con el nombre *IV – OFBMode.txt*.

3.3.3 Modo de operación OFB: Cifrado

La implementación de la función que realiza el cifrado del mensaje haciendo uso del modo de operación OFB se muestra en el Listing 15. Esta función recibe como parámetros el contenido del texto plano ingresado por el usuario así como el nombre del mismo. Posteriormente, con ayuda de la función presentada en

la sección 3.4.2 se genera el vector de inicialización, el cual es almacenado en un archivo de texto.

```

1 void modeOfOperationOFBCipher( string *contentText, string
   nameFile ){
2
3     string vectorIV, messageEncrypt = "", nametoSave;
4     char inputSDES, outputOFB;
5
6     bitset<10> binaryKey;
7     bitset<8> subKey1, subKey2, valueBinary;
8
9     vectorIVGeneration( &vectorIV );
10    saveVectorIV( vectorIV );
11    KeysGenerateSDES( &binaryKey, &subKey1, &subKey2, 1 );
12
13    inputSDES = static_cast<char>( bitset<8>( vectorIV ).
to_ulong() );
14
15    for( register int i = 0; i < (*contentText).size(); i++ ){
16        EncryptSDES( &inputSDES, subKey1, subKey2 );
17        outputOFB = inputSDES ^ (*contentText)[i];
18        messageEncrypt = messageEncrypt + outputOFB;
19    }
20
21    nametoSave = "../Archivos/Enc" + nameFile;
22    freopen( nametoSave.c_str(), "w", stdout );
23    cout<< messageEncrypt;
24    fclose(stdout);
25
26 }

```

Listing 15: Función para cifrar haciendo uso de OFB

Para la generación de las llaves y subllaves de S-DES se hace uso de la función `KeysGenerateSDES` encontrada en la librería `SDES.h`, la cual recibe como un parámetro una bandera que nos ayuda a determinar si es que se debe generar de manera automática (caso del cifrado) o si debe de pedir el nombre del *.txt* que la contiene y posteriormente almacenarla (caso del descifrado). Para la primera opción se debe mandar un 1 y para la segunda un 0.

Debido a que la función encargada de cifrar con S-DES recibe un `char`, se necesitó castear el `bitset` que almacena el vector de inicialización en este tipo de dato con ayuda de la función `static_cast<char>`.

El modo de operación OFB se muestra en la Figura 15. Ya que en este caso se esta trabajando con 8 bits, no fue necesario llevar a cabo la fase de *truncate*. La primera entrada a nuestro cifrador de bloque (S-DES) corresponde al vector de inicialización, por lo que, en el código se declaró una variable de tipo `char` a la cual antes de entrar al `for` se le asignó dicho valor.

Posteriormente, la función `EncryptSDES` retorna el caracter cifrado en la

misma variable `inputSDES`, por lo que, no fue necesario modificarla, debido a que la entrada en la siguiente ronda pertenece a lo obtenido en la anterior. Lo único que se realiza es la operación *XOR* entre este y el caracter de texto claro correspondiente. Finalmente, vamos concatenando cada uno de los caracteres hasta obtener el mensaje final cifrado, el cual es almacenado en un archivo de texto cuyo nombre inicia con *Enc*.

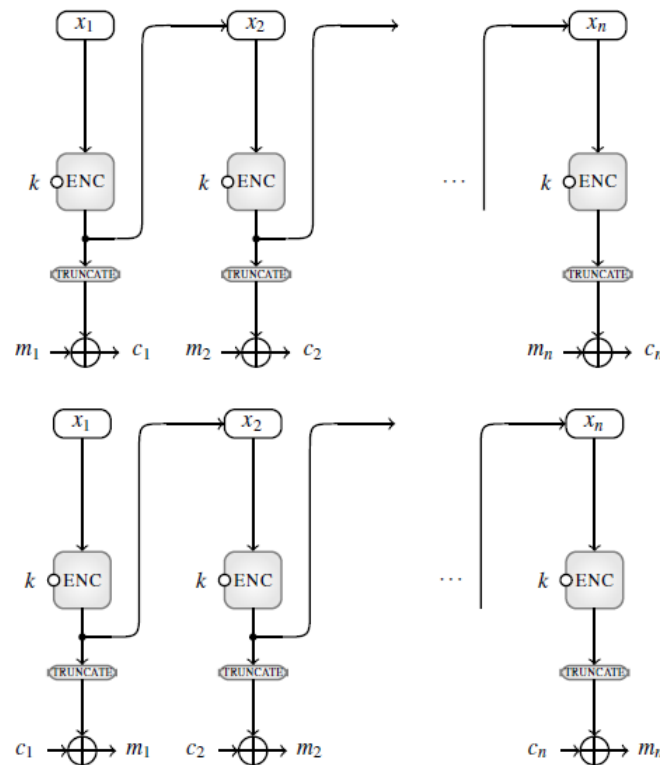


Figura 15: Cifrado y descifrado con el modo de operación OFB

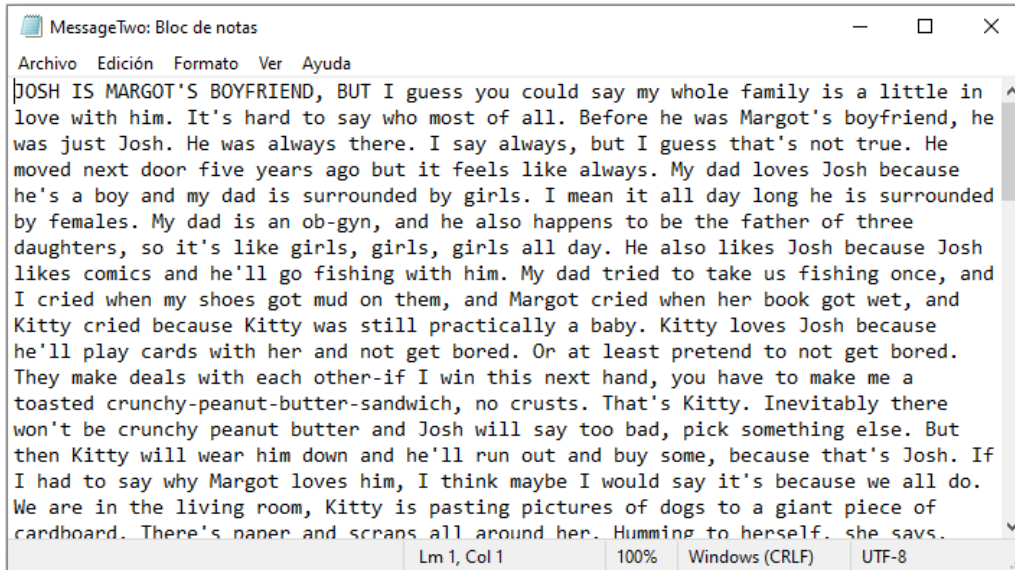
El programa en ejecución, así como los resultados obtenidos se muestran en las Figuras 16 y 17 respectivamente.

```

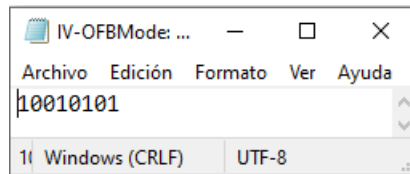
PROBLEMS  TERMINAL  ...  1: powershell  +  [ ]  [X]  ^  X
Encrypt Mode of Operation: OFB
File to encrypt: MessageTwo.txt
Subllave 1: 00000011
Subllave 2: 00111100
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY
\Laboratorio2\ModeOfOperationOFB>

```

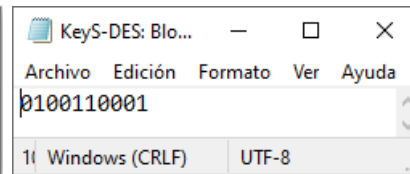
Figura 16: Programa en ejecución



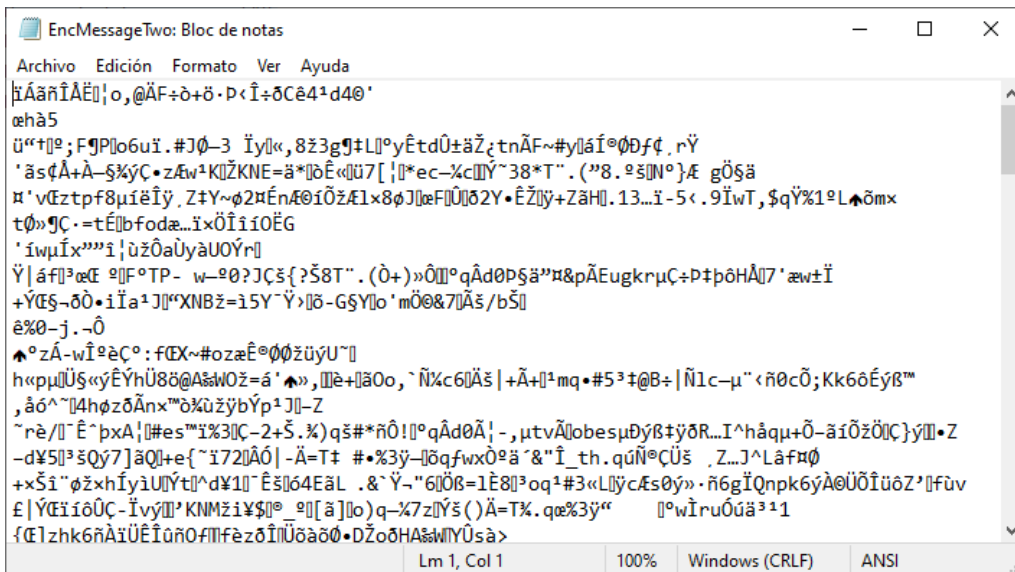
(a) Texto original



(b) Vector IV generado



(c) Llave generada S-DES



(d) Texto cifrado

Figura 17: Resultados obtenidos para el cifrado OFB

3.3.4 Modo de operación OFB: Descifrado

Una de las características del modo de operación OFB es que el procedimiento del cifrado y descifrado es prácticamente el mismo. Debido a esto, únicamente se modificaron dos cosas para la implementación del descifrado (véase Listing 16).

- El parámetro utilizado para indicar si se debe generar o no la llave de S-DES cambio su valor a 0, es decir, ahora pedirá el nombre del archivo y almacenará su contenido.
- No se manda llamar a la función encargada de generar el vector de inicialización, en su lugar se llama a la función encargada de obtener los datos del archivo que lo contienen.

El procedimiento es el mismo que el cifrado, se guarda el valor del vector de inicialización en un `char`, el cual será la primera entrada de nuestro cifrador por bloques (S-DES), el resultado obtenido no se modifica debido a que es usado en la siguiente ronda, y únicamente se realiza la operación *XOR* con el carácter del archivo cifrado correspondiente. Finalmente, se concatenan todos los resultados y se escribe en un archivo de texto.

```

1 void modeOfOperationOFBDecrypt( string *contentText, string
   nameFile ){
2
3     string vectorIV, messageEncrypt = "", nametoSave, nameIV;
4     char inputSDES, plaintext, outputOFB;
5
6     bitset<10> binaryKey;
7     bitset<8> subKey1, subKey2, valueBinary;
8
9     cout<<" File IV: ";
10    cin>> nameIV;
11    getIV( &vectorIV, "../Archivos/" + nameIV );
12    KeysGenerateSDES( &binaryKey, &subKey1, &subKey2, 0 );
13
14    inputSDES = static_cast<char>( bitset<8>( vectorIV ).
   to_ulong() );
15
16    for( register int i = 0; i < (*contentText).size(); i++ ){
17        EncryptSDES( &inputSDES, subKey1, subKey2 );
18        outputOFB = inputSDES ^ (*contentText)[i];
19        messageEncrypt = messageEncrypt + outputOFB;
20    }
21
22    nametoSave = "../Archivos/Dec" + nameFile;
23    freopen( nametoSave.c_str(), "wb", stdout );
24    cout<< messageEncrypt;
25    fclose(stdout);

```

26
27 }

Listing 16: Función para descifrar haciendo uso de OFB

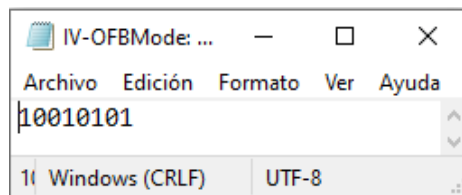
El programa en ejecución, así como los resultados obtenidos se muestran en las Figuras 18 y 19 respectivamente.

```

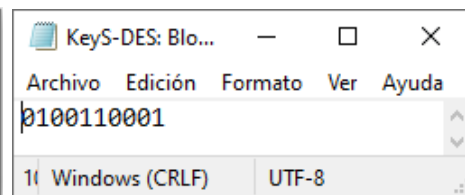
1: powershell
Decrypt Mode of Operation: OFB
File to encrypt: EndMessageTwo.txt
File IV: IV-OFBMode.txt
Key file: KeyS-DES.txt
Subllave 1: 00000011
Subllave 2: 00111100

```

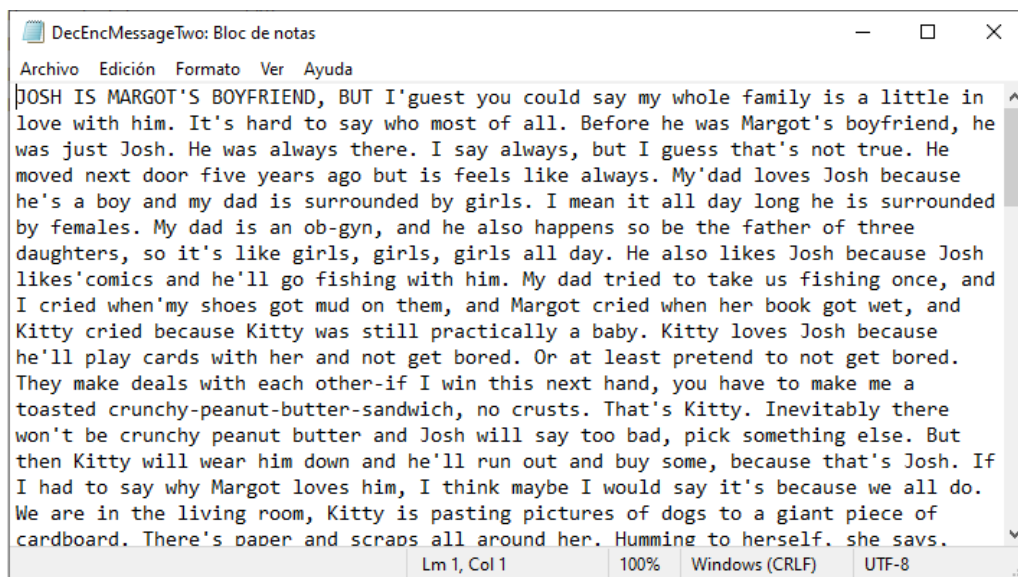
Figura 18: Programa en ejecución



(a) Vector IV utilizado



(b) Llave S-DES utilizada



(c) Texto descifrado

Figura 19: Resultados obtenidos para el descifrado OFB

Otra característica de este modo de operación es que no hace uso del algoritmo implementado para el descifrado S-DES. A pesar de esto, en la sección 3.2.5 se muestra un ejemplo de su correcto funcionamiento.

3.3.5 Modificación función permutationInverse

Con anterioridad en la sección 3.2.4 se presentaron todas las funciones utilizadas tanto para la ronda uno como para la ronda dos de S-DES, una de estas pertenece a la encargada de realizar la permutación inversa. Sin embargo, durante la implementación del modo de operación OFB se encontró una manera más sencilla de realizar dicha tarea. El código modificado se muestra en el Listing 17.

Como se puede observar ahora ya no se hace uso de corrimientos, si no que, primero se realiza la permutación y posteriormente se hace uso del método `.to_ulong()` para obtener el valor entero de ese binario.

```

1 void permutationInverse( bitset<8> resultRound2, long int *
   potencia ){
2
3     int permutationTable[] = {2, 4, 1, 5, 8, 6, 3, 7};
4     bitset<8> resultInverse;
5
6     for( register int i = 0; i < 8; i++ )
7         resultInverse[ permutationTable[i] - 1 ] = resultRound2
   [i];
8
9     *potencia = (int)resultInverse.to_ulong();
10 }

```

Listing 17: Función permutationInverse modificada

4 Manual de usuario

4.1 Permutacion de caracteres

Para correr el programa correspondiente al cifrado y descifrado haciendo uso de permutaciones de caracteres es necesario encontrarse dentro de la carpeta *PermutationProgram*. Dentro de esta, se encuentra otra carpeta con el nombre *Archivos* en la cual deben estar los archivos a utilizar (tabla de permutación, texto a cifrar, texto a descifrar, etc.) y en la cual se guardan los resultados generados por el programa.

Las características que debe tener la matriz de permutación son:

- El primer renglón corresponde a los valores de i ordenados de manera ascendente, empezando por el 1.
- El segundo renglón pertenece a los valores de Π_i .
- Cada uno de los valores de ambos renglones deben estar separados únicamente por un espacio en blanco y se debe dejar uno adicional al final de cada renglón.

Para generar el ejecutador del programa es necesario escribir en consola el siguiente comando:

```
g++ -std=c++11 main.cpp Files.cpp Process.cpp
    PermutationCipher.cpp -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

4.2 S-DES

Para correr el programa correspondiente al cifrado y descifrado haciendo uso de S-DES es necesario encontrarse dentro de la carpeta *S – DESProgram*. Dentro de esta, se encuentra otra carpeta con el nombre *Archivos* en la cual debe estar el mensaje a cifrar y en la que se guardan los resultados generados por el programa.

Para generar el ejecutador del programa es necesario escribir en consola el siguiente comando:

```
g++ -std=c++11 main.cpp Files.cpp ProcessSDES.cpp SDES.cpp
    TransformKey.cpp -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

4.3 Modo de operación: OFB

Para correr el programa correspondiente al cifrado y descifrado haciendo uso del modo de operación OFB es necesario encontrarse dentro de la carpeta *ModeOfOperationOFB*. Dentro de esta, se encuentra otra carpeta con el nombre *Archivos* en la cual debe estar el mensaje a cifrar y en la que se guardan los resultados generados por el programa.

Para generar el ejecutador del programa es necesario escribir en consola el siguiente comando:

```
g++ -std=c++11 main.cpp Files.cpp ProcessSDES.cpp SDES.cpp
    TransformKey.cpp ModeOFB.cpp ProcessOFB.cpp -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

References

- [1] C. Kościelny, M. Kurkowski, and M. Srebrny, *Modern Cryptography Primer: Theoretical Foundations and Practical Applications*. Berlin Heidelberg: Springer-Verlag, 2013.
- [2] L. R. Knudsen and M. J. B. Robshaw, *The Block Cipher Companion*. New York: Springer-Verlag, 2011.
- [3] N. P. Smart, *Cryptography Made Simple*. New York: Springer-Verlag, 2016.