Cryptography, Sem: 2020-2, 3CM9, Laboratorio 3, 30/Marzo/2020

Laboratorio 3: Finite Fields

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 campos finitos para $3 \leq n \leq 8$, así como del algoritmo utilizado para generar las subllaves en AES (Advanced Encryption Standard), considerando una llave de tamaño de 128 bits y de 192 bits. Para cada uno de los programas se presentarán las consideraciones tomadas en cuenta para su implementación.
Palabras Clave: AES, campos finitos, key.

1 Introducción

El cifrado simétrico, también conocido como cifrado convencional, de clave secreta o de clave única, era el único que se usaba antes del desarrollo del cifrado de clave pública a finales de los 70. Aún hoy continúa siendo el más usado de los dos tipos de cifrado.

Este tipo de cifrado depende de la privacidad de la clave, no de la privacidad del algoritmo. Esta característica es la causa de su uso tan extendido. El hecho de que el algoritmo no tenga que ser secreto significa que los fabricantes pueden desarrollar implementaciones con chips de bajo coste de los algoritmos de cifrado de datos [1].

El estándar de cifrado avanzado (AES) es el cifrado simétrico más utilizado en la actualidad. Aunque el término "estándar" en su nombre solo se refiere a aplicaciones del gobierno en los Estados Unidos, el cifrador por bloques AES también es obligatorio en varios estándares de la industria y se usa en muchos sistemas comerciales. En AES, la aritmética de campo de Galois se usa en la mayoría de las fases, especialmente en la fase $S - BOX$ y $MixColumn$ [2].

Tomando en cuenta lo anterior es importante conocer cómo trabaja AES, y más acerca de los campos finitos, por lo que, en esta práctica además de presentarse una introducci ón teórica se implementarán programas que nos ayuden a un mejor entendimiento.

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 [3].

2.2 Advanced Encryption Standard (AES)

En 1999, el Instituto Nacional de Estándares y Tecnología de EE.UU. (NIST) indicó que DES solo debía usarse para sistemas heredados y en su lugar debería utilizarse el triple DES (3DES). Aunque 3DES resiste los ataques de fuerza bruta con la tecnología actual, hay varios problemas con este. Primero, no es muy eficiente con respecto a las implementaciones de software. Otra desventaja es el tamaño de bloque relativamente pequeño de 64 bits, que es un inconveniente en ciertas aplicaciones, por ejemplo, si se quiere construir una función hash a partir de un cifrado por bloque. Finalmente, si uno está preocupado por los ataques con computadoras cuánticas, que podrían convertirse en realidad en unas pocas décadas, son deseables longitudes de clave del orden

de 256 bits. Lo anterior llevó a el NIST a la conclusión de que se necesitaba un cifrador por bloque completamente nuevo.

En 1997, el NIST solicitó propuestas para un nuevo estándar de cifrado avanzado (AES). A diferencia del desarrollo DES, la selección del algoritmo para AES fue un proceso abierto administrado por este Instituto. En 2001, declaró el cifrador por bloque Rijndael como el nuevo AES y lo publicó como un estándar final. Rijndael fue diseñado por dos jóvenes criptógrafos belgas.

El cifrado AES es casi idéntico al cifrado de bloque Rijndael. El tamaño del bloque y la clave de Rijndael varían entre 128, 192 y 256 bits. Sin embargo, el estándar AES solo requiere un tamaño de bloque se 128 bits. El número de rondas internas del cifrado depende de la longitud de la clave de acuerdo con la Tabla 1.

Tamaño de la llave	# rounds
128 bit	10
192 bit	12
256 bit	14

Tabla 1: Longitudes de clave y número de rondas para AES

A diferencia de DES, AES no tiene una estructura Feistel. Este tipo de redes no cifra un bloque completo por iteración. Sin embargo, AES cifra los 128 bits en una. Esta es una razón por la cual tiene un número comparativamente pequeño de rondas.

AES consta de las llamadas capas. Cada capa manipula los 128 bits de la ruta de datos. A esta también se le conoce como el estado del algoritmo. Solo hay tres tipos diferentes de capas, las cuales se repiten en cada una de las rondas a excepción de la primera. En AES, la aritmética de campo de Galois se usa en la mayoría de las capas, especialmente durante las $S-BOX$ y $MixColumn$ [2].

2.2.1 Campos finitos

Un campo finito, a veces también llamado campo de Galois, en términos generales, es un conjunto finito de elementos en los que podemos sumar, restar, multiplicar e invertir. El número de elementos en el campo se denomina *orden* o *cardinalidad* del campo.

Se establece un grupo donde cada una de las operaciones tiene un inverso. Si la operación se llama suma, la operación inversa es resta; si la operación es multiplicación, la inversa es división (o multiplicación con el elemento inverso) [2].

2.2.2 Campos de extensión GF(2^m)

En AES, el campo finito contiene 256 elementos y se denota como $GF(2^8)$. Este campo se eligió porque cada uno de los elementos se puede representar por un byte. Para las transformaciones $S-BOX$ y $MixColumn$, AES trata cada byte de la ruta de datos interna como un elemento de dicho campo y los manipula realizando aritmética en ese campo finito.

Sin embargo, si el orden de un campo finito no es primo, y 2^8 claramente no lo es, la operación de suma y multiplicación no puede representarse mediante números enteros módulo 2^8. Dichos campos con $m > 1$ se denominan campos de extensión. Para tratar con estos, se necesita una notación diferente para cada elemento y diferentes reglas para realizar aritmética.

En los campos de extensión, los elementos $GF(2^m)$ no se expresan como enteros, sino como polinomios con coeficientes en $GF(2)$. Los polinomios tienen un grado máximo de $m-1$, de modo que hay m coeficientes en total para cada elemento. En el campo $GF(2^8)$, que se usa en AES, cada elemento $A \in GF(2^8)$ se representa como:

$$A(x) = a_7x^7 + ... + a_1x + a_0, \qquad a_i \in GF(2) = 0, 1$$

Es importante observar que el conjunto de los 256 polinomios es el campo finito 2^8. Cada uno de estos simplemente puede almacenarse de forma digital como un vector de 8 bits.

$$A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$$

En particular, no se tienen que almacenar los factores x_7, x_6, etc. Es claro que las potencias hacen referencia al bit al cual pertenece cada coeficiente [2].

2.2.3 Key Schedule

El *key schedule* toma la clave de entrada original (de longitud 128, 192 o 256 bits) y deriva las suclaves utilizadas en AES. El número de estas subclaves es igual al número de rondas más uno. Por lo tanto, para la longitud de la clave de 128 bits, el número de rondas es $n_r = 10$, y hay 11 subclaves, esto se repite para los demás tamaños. Estas se calculan de forma recursiva, es decir, para derivar la subclave k_i, se debe conocer el valor de k_{i-1}, etc.

El *key schedule* en AES está orientada a palabras, donde 1 palabra = 32 bits. Las subclaves se almacenan en una matriz de expansión de claves W, la cual, consta de palabras. Existen diferentes programas de claves para los tres tamaños de 128, 192 y 256 bits, que son bastantes similares. En las Figuras 1 y 2 se muestran los diagramas para encontrar las subclaves para 128 y 192 bits respectivamente [2].

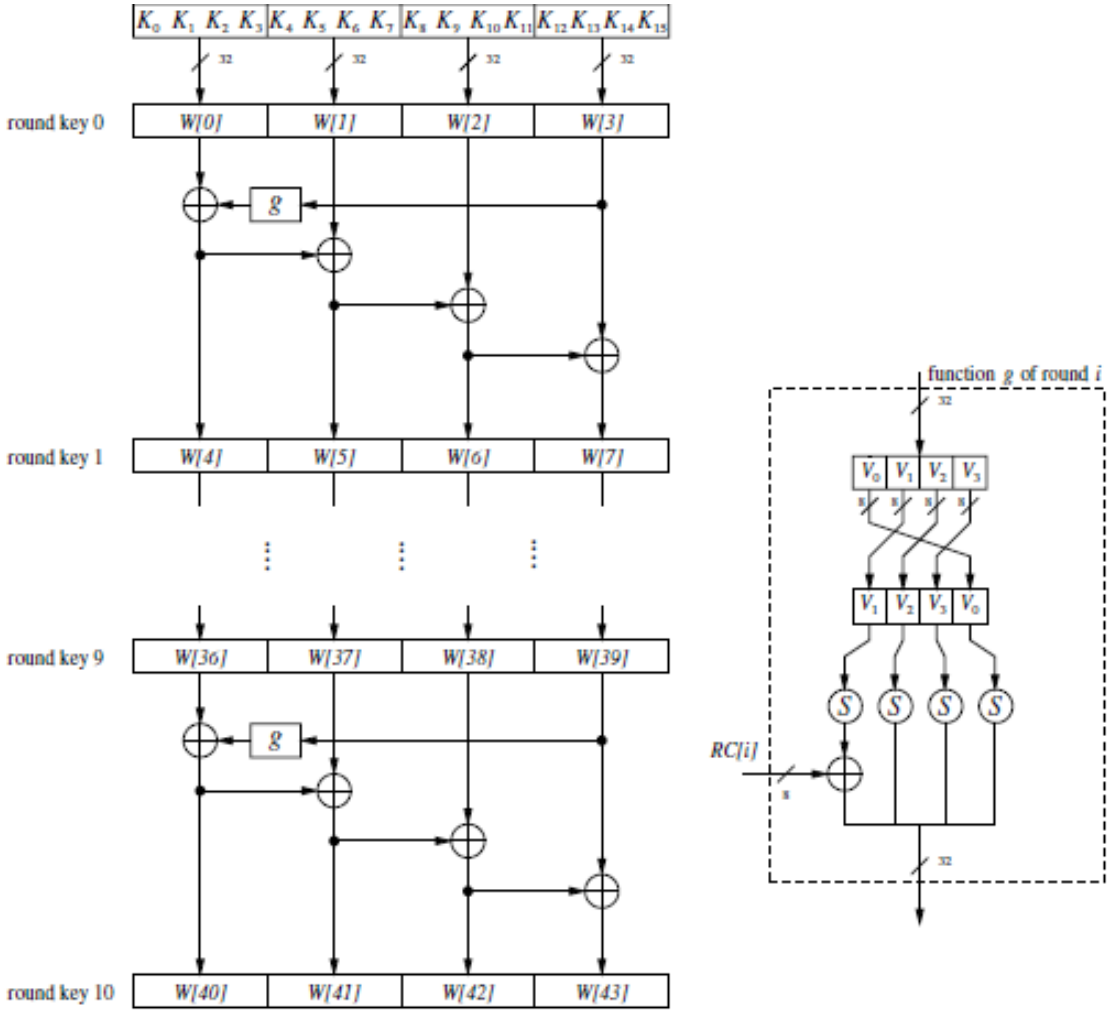


Figura 1: *Key schedule* AES para tamaño de clave de 128 bits

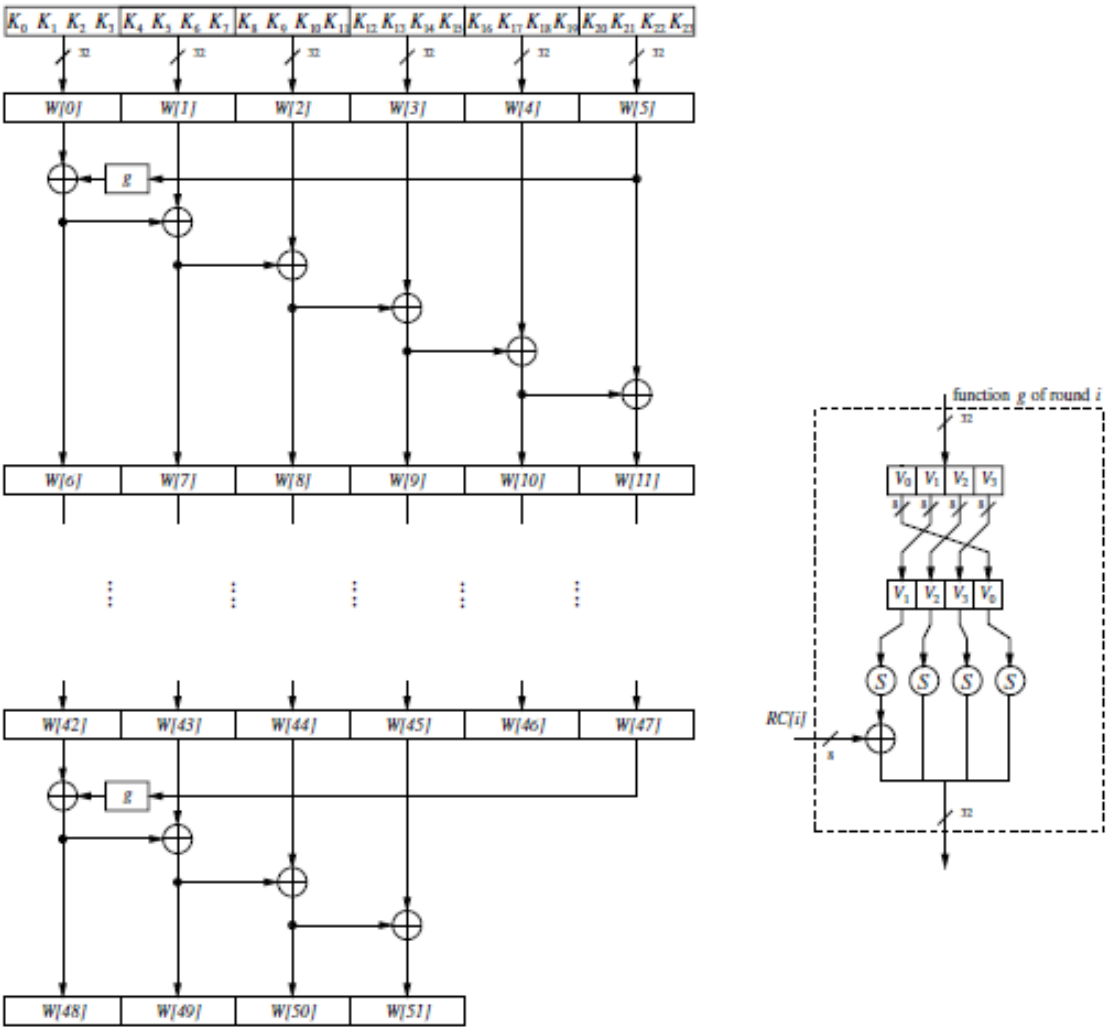


Figura 2: *Key schedule* AES para tamaño de clave de 192 bits

3 Experimentación y Resultados

3.1 Finite Fields $GF(2^n)$

3.1.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Para su elaboración se tomaron en cuenta los siguientes aspectos:

- Se hizo uso del tipo de dato `bitset` con la finalidad de tener un mejor manejo de los bits. Sin embargo, para determinar su tamaño no se puede hacer uso de un `int`, por lo que, se puso el mayor número de bits que pudiese contener, que en este caso serían 9 (para $GF(2^8)$), trabajando en cada uno de los casos con los necesarios.

- El archivo llamado defs.h contiene las rutas donde se guardarán los archivos generados por el programa, así como la primera parte del nombre de los mismos, debido a que durante la ejecución se les concatena el valor de n. Por otro lado, contiene la ruta donde se encuentra la tabla de los polinomios a utilizar (véase Listing 1).

```
1 #ifndef DEFS_H
2 #define DEFS_H
3
4 #define FILELIST "./Archivos/PrimitivePolynomials.txt"
5 #define SAVEDFILEHEX "./Archivos/HexTableGF"
6 #define SAVEDFILEPOL "./Archivos/PolTableGF"
7
8 #endif
```

Listing 1: Librería defs.h

- Se hicieron dos librerías cuyas implementaciones se encargan de llevar acabo todos los procedimientos necesarios para calcular la tabla de multiplicación. Estos se presentarán de manera más detallada en las siguientes secciones.

3.1.2 Implementación librería ProcessGF2.h

La implementación de la librería ProcessGF2.h consta de tres funciones:

1. La primera obtiene el polinomio irreducible correspondiente a la n ingresada por el usuario del archivo *.txt* donde se encuentran y lo almacena en un `string`.

   Recibe como parámetros el valor de n, el cual ayuda a determinar qué renglón del archivo de texto plano es la que se debe de leer, así como un apuntador a `string` en la cual se almacenará el polinomio.

2. Una vez teniendo lo anterior, la segunda función es la encargada de pasar el polinomio irreducible a nivel de bit. Para esto, se hizo uso del `string` que lo contiene y posteriormente, se separó la cadena usando como delimitador la ",". Sin embargo, fue necesario hacer uso de la función `stoi` para poder convertir parte del string a número obteniendo así la potencia, es decir, la posición del bit a prender. La implementación de dicha función se muestra en el Listing 2.

   El método *set* fue el que se utilizó para prender los bits de nuestro `bitset`, el cual como se mencionó anteriormente es de tamaño 9.

```
void getPowers( string polynomialString, bitset<9> *
    polynomialBits ){

     string auxString = "";

     for( register int i = 1; i < polynomialString.size() -
     1; i++ )
         if( polynomialString[i] != ',' ){
             auxString = polynomialString[i];
             (*polynomialBits).set( stoi( auxString,
    nullptr, 10 ) );
             auxString = "";
         }
}
```

Listing 2: Función getPowers

3. Finalmente la última función es la encargada de realizar el algoritmo proporcionado por la profesora, el cual calcula la multiplicación de dos polinomios modulo el polinomio irreducible haciendo uso de corrimientos y operaciones XOR.

   Esta función recibe 4 parámetros: el número de la fila y columna de la tabla de multiplicación, los cuales jugarán el papel de los polinomios a multiplicar; el valor de n y el `bitset` que almacena la representación binaria del polinomio irreducible. Por otro lado, retorna otro `bitset` que contiene el resultado final.

   Lo primero que se realizó fue obtener el valor binario de la fila y columna. Es importante que todos los `bitset` sean del mismo tamaño para poder realizar las operaciones. Por lo que, dichos valores binarios se almacenaron en este tipo de dato de tamaño 9 (véase Listing 3).

```
    bitset<9> rowBits(row), columnBits(column);
```

Listing 3: Obtención de los valores binarios de la fila y columna

   El algoritmo nos indica que uno de los polinomios se debe "desglosar", es decir, se puede representar como m polinomios dependiendo de los

bits encendidos que tenga. Por lo que, la solución final dependerá de soluciones parciales de cada uno de los m polinomios. Para un mayor entendimiento se nombrará dicho polinomio como $g(x)$ y al segundo operando como $f(x)$.

Lo primero que se realizó fue verificar si $g(x)$ contaba con el término 1, en caso de que se cumpla se almacena en el `bitset` que contendrá el resultado final, la primera solución parcial la cual corresponde al valor del polinomio $f(x)$ original. Posteriormente, con ayuda de un ciclo `for` recorremos a $f(x)$, realizándole un corrimiento a la izquierda en cada iteración, verificamos si la posición n está encendida y en caso de que se cumpla realizamos el XOR con el polinomio irreducible. Finalmente verificamos si el polinomio que "desglosamos" tiene la potencia indicada por el índice del `for` para determinar si forma parte de las soluciones parciales o no y así realizar un XOR de las soluciones anteriores con el valor del polinomio $f(x)$ acutal.

El código fuente se muestra en el Listing 4.

```
1  bitset<9> calMultiplication( int row, int column, int n,
       bitset<9> polynomialBits ){
2
3      bitset<9> rowBits(row), columnBits(column);
4      bitset<9> result;
5
6      if( columnBits[0] == 1 )
7          result = rowBits;
8
9      for( register int i = 1; i < n; i++ ){
10         rowBits = rowBits << 1;
11         if( rowBits[n] == 1 )
12             rowBits ^= polynomialBits;
13         if( columnBits[i] == 1 )
14             result ^= rowBits;
15     }
16
17     return result;
18 }
```

Listing 4: Algoritmo de multiplicación usando corrimientos y XOR

3.1.3 Implementación librería GF2.h

La implementación de la librería GF2.h hace uso de las funciones explicadas en la sección anterior. Adicional a esto, realiza la tarea de guardar los resultados en el archivo *.txt*, redirigiendo el buffer de salida a este.

- Para guardar los valores en hexadecimal lo único que se realizó fue mandar al buffer de salida los `bitset` que retornaba la función `calMultiplication`.

Una ventaja de haber usado el tipo de dato `bitset` es que permite convertir su valor de manera muy sencilla a hexadecimal, sin necesidad de quitar los bits restantes para $n = 1, 2, 3, ..., etc$, ya que podemos recordar que el tamaño estándar, por decirse así, fue de 9.

En el Listing 5 se muestra esta pequeña parte del código.

```
1                   cout<< hex << uppercase << result.to_ulong
    ();
```

Listing 5: Guardado de datos en hexadecimal

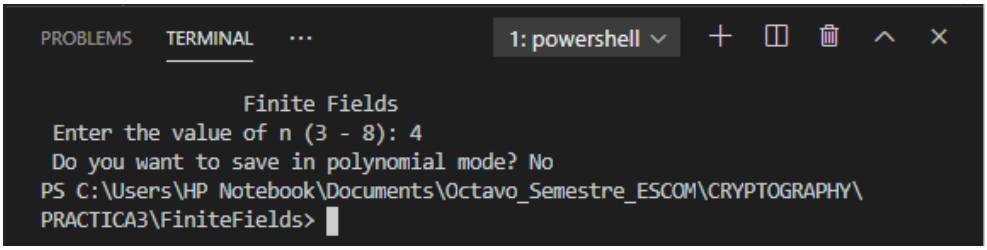
- Por otro lado, para almacenar los valores en modo de polinomio se hizo un ciclo `for`, el cuál va recorriendo cada uno de los `bitset` retornados por la función que realiza la multiplicación y cada vez que encuentra un 1 manda al buffer de salida su posición, indicando la potencia del polinomio. Lo demás se realizó con la finalidad de darle formato al archivo (veáse Listing 6).

```
1                   for( register int k = 0; k < n; k++ ){
2                       if( result[k] == 1 ){
3                           cout<< k;
4                           cont++;
5                       }
6                       if( cont == bitSet )
7                           break;
8                       if( result[k] == 1 ){
9                           cout<< ",";
10                      }
11                  }
```

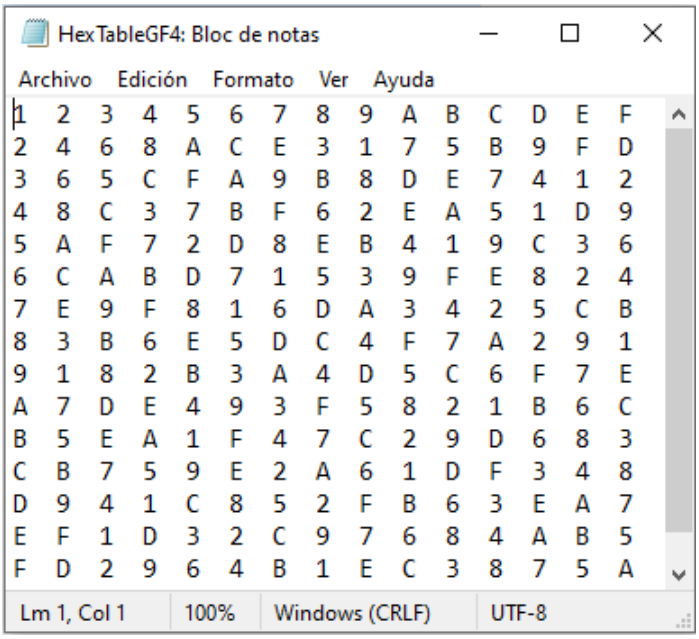
Listing 6: Guardado de datos en polinomio

Finalmente, para mandar cada uno de los polinomios a multiplicar a la función correspondiente se hicieron dos `for` anidados, el primero indicando el número del renglón de la matriz y el segundo perteneciente a la columna.

El programa en ejecución así como los resultados obtenidos para la representación en hexadecimal se muestran en la Figura 3. Por otro lado, en caso de que el usuario desee almacenar la tabla en forma de polinomios el resultado se muestra en la Figura 4.

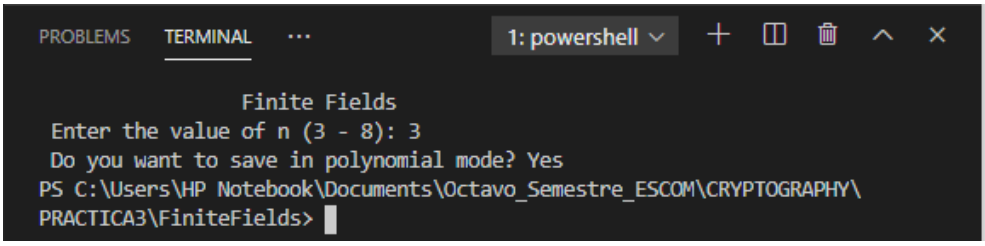


(a) Programa en ejecución

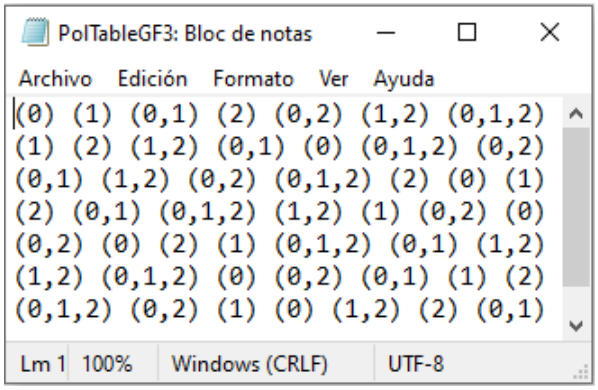


(b) Archivo *.txt* que contiene la tabla para $GF(2^4)$

Figura 3: Resultados obtenidos en modo Hexadecimal



(a) Programa en ejecución



(b) Archivo *.txt* que contiene la tabla para $GF(2^3)$

Figura 4: Resultados obtenidos en modo Polinomio

3.2 Key Schedule de AES con llave de 128 bits

3.2.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Para su elaboración se tomaron en cuenta los siguientes aspectos:

- La matriz correspondiente a la $S-Box$ de AES se declaró de manera global en el programa encargado de llevar a cabo las funciones intermedias del algoritmo. Para esto, se hizo una matriz bidimensional de tipo `string` con la finalidad de que al momento de querer convertir algún valor perteneciente a esta en binario fuera más sencillo.

- Las constantes C_i al igual que la $S-Box$ fueron declaradas en un arreglo de tipo `string`.

- Se implementaron dos librerías: en una se encuentran todas las implementaciones de las funciones encargadas de realizar las tareas intermedias y otra que hace uso de esta y lleva a cabo el procedimiento para calcular las 10 subllaves.

3.2.2 Pseudocódigo utilizado

Para llevar a cabo la implementación del Key Schedule se hizo uso del pseudocódigo encontrado en [4] y el cual se muestra Algorithm 1.

La función RotBytes es la encargada de realizar el corrimiento circular hacia arriba de la útlima columna de nuestra matriz conformada por $W_0,\ W_1,\ W_2,\ W_3$. Por otro lado, la parte de buscar los valores en la $S-Box$ y posteriormente la sustitución de los mismos se lleva a cabo en la función `SubBytes`.

Al momento de realizar la implementación, no se declaró una matriz de 4×43, en su lugar se hizo uso de una de tamaño 4×4 modificando sus valores según correspondía, esto se realizó con la finalidad de ahorrar espacio en memoria.

En las secciones posteriores se explicará la implementación del pseudocódigo.

Algorithm 1 AES key schedule

$W_0 \leftarrow K_0,\ W_1 \leftarrow K_1,\ W_2 \leftarrow K_2,\ W_3 \leftarrow K_3$
for $i \leftarrow 1$ *to* 10 **do**
  $T \leftarrow$ `RotBytes`(W_{4i-1})
  $T \leftarrow$ `SubBytes`(T)
  $T \leftarrow \mathrm{T} \oplus RC_i$
  $W_{4i} \leftarrow W_{4i-4} \oplus\ T$
  $W_{4i+1} \leftarrow W_{4i-3} \oplus\ W_{4i}$
  $W_{4i+2} \leftarrow W_{4i-2} \oplus\ W_{4i+1}$
  $W_{4i+3} \leftarrow W_{4i-1} \oplus\ W_{4i+2}$
end

3.2.3 Implementación de funciones intermedias

La implementación de las funciones intermedias, es decir, de las que son ocupadas para realizar el algoritmo presentado en Algorithm 1 se encuentran en el archivo KeyScheduleProcess.cpp.

Para obtener la llave del *.txt* proporcionado por el usuario, se diseñó una función la cual abre el archivo y guarda su contenido en un `string`. Posteriormente, con ayuda de otra función se convierte cada uno de los caracteres hexadecimales en su correspondiente valor binario y se almacena en nuestra matriz de `bitset`, la cual como se mencionó anteriormente es de 4×4. Para esto, fue necesario generar subcadenas y posteriormente con ayuda de `stoi` convertirlas a número indicando que se encuentran en base 16, es decir, hexadecimal. Es importante recordar que cada celda esta formada por 8 bits, lo cual indica que necesitamos convertir dos letras, por lo que, el valor de cada una se guarda en un `bitset` de tamaño 4 para posteriormente concatenarlas y obtener el valor final que será almacenado (véase Listing 7).

```
    for( register int column = 0; column < 4; column++ )
        for( register int row = 0; row < 4; row++ ){
            firstPart = stoi( keyString.substr(aux, 1), nullptr
, 16);
            secondPart = stoi( keyString.substr(aux+1, 1),
nullptr, 16);
            W[row][column] = bitset<4+4>(firstPart.to_string()
+ secondPart.to_string());
            aux += 2;
        }
```

Listing 7: Cambio de dato a binario y almacenamiento en la matriz

La variable T se declaró como una matriz de `bitset` y consta de 4 filas y 1 columna.

La función `RotBytes` se muestra en el Listing 8. Como se puede observar, lo primero que se realizó fue almacenar el valor correspondiente a la primera posición de la última columna de nuestra matriz W en la última fila de nuestra matriz T. Posteriormente, con ayuda de un ciclo `for` se fueron rellenando las demás posiciones.

```
void RotBytes( bitset<8> W[4][4], bitset<8> T[4][1] ){
    T[3][0] = W[0][3];

    for( register int i = 0; i < 3; i++ )
        T[i][0] = W[i+1][3];
}
```

Listing 8: Función encargada de hacer el corrimiento de la última columna

Para realizar la búsqueda en la matriz de sustitución ($S - Box$) es importante recordar que los primeros 4 bits corresponden a la fila, mientras que los últimos indican la columna. La función encargada de realizar dicha búsqueda, así como las modificaciones necesarias se muestra en el Listing 9. Esta función, recibe como parámetros la tabla T, la cual hasta el momento contiene los valores resultantes de realizar el corrimiento anterior.

Debido a que cada celda del `bitset` consta de 8 bits fue necesario hacer cuatro corrimientos a la derecha para obtener el valor de la fila a buscar, posteriormente se hizo cast a `int` y se almacenó en una variable. Por otro lado, para obtener el valor correspondiente a la columna se hizo uso de una mascará con la finalidad de eliminar los cuatro bits más significativos, se le realizó un cast a `int` y se almacenó en otra variable. Posteriormente, se buscaron estas posiciones en la matriz de sustitución, almacenando el resultado en un `string`, debido a que como se mencionó anteriormente la $S - Box$ fue declarada como un arreglo bidimensional de este tipo de dato. Como paso final, se hizo el cast de la cadena obtenida a entero con ayuda de la función `stoi` indicando que su valor esta en base 16, es decir, hexadecimal, guardando el resultado nuevamente en la matriz T.

El procedimiento anterior se debe repetir para las cuatro filas, por lo que, se hizo uso de un ciclo `for` (véase Listing 9).

```
void SubBytes( bitset<8> T[4][1] ){
    int rowSBox, columnSBox;
    bitset<8> mask("00001111");
    string SBoxresult;

    for( register int i = 0; i < 4; i++){
        rowSBox =  (int)(T[i][0] >> 4).to_ulong();
        columnSBox = (int)(T[i][0] & mask).to_ulong();

        SBoxresult = SBox[rowSBox][columnSBox];
        T[i][0] = stoi( SBoxresult, nullptr, 16 );
    }
}
```

Listing 9: Búsqueda en la matriz $S - Box$ y sustitución de valores

El siguiente paso corresponde a realizar la operación lógica XOR de la primera posición de T con la variable RC_i, el valor de esta última depende de qué subclave es la que queremos calcular, sin embargo, ya tiene valores definidos, los cuales como se mencionó en la sección 3.2.1 fueron almacenados en un arreglo de tipo `string`.

Se diseñó una función que realiza esta tarea y la cual se puede observar en el Listing 10. Esta recibe como parámetros la matriz T, y el número de ronda en la que nos encontramos. Debido a que las rondas empiezan en 1, es necesario restarle esta cantidad antes de realizar la consulta en nuestro arreglo que almacena los valores de RC_i, posteriormente se hace el cast de `string` a entero con ayuda de la función `stoi`, se obtiene su valor en binario y se realiza la operación XOR con la primera posición de nuestra matriz T.

```
void XorRC( bitset<8> T[4][1], int i ){

    T[0][0] ^= bitset<8>( stoi( ConstCi[i-1], nullptr, 16 ) );

}
```

Listing 10: Operación $T \oplus RC_i$

El paso final del algoritmo consta en modificar las columnas y filas de nuestra matriz W. La función encargada de llevar a cabo esta acción se muestra en el Listing 11.

Lo primero que se realizó fue la operación $W_0 \oplus T$ con ayuda del primer ciclo `for`, almacenando el resultado en la primera columna de la misma matriz W. Posteriormente, se hizo uso de dos ciclos `for` anidados los cuales recorren las columnas de la 2 a la 4 y todas las filas de cada una, dentro de estos se realiza la misma operación lógica pero ahora con el resultado obtenido en la columna anterior, es por esto que, el índice de column inicia en 1. Los valores que se van obteniendo en cada iteración se van almacenando en la misma matriz W.

```
void modifyValues( bitset<8> T[4][1], bitset<8> W[4][4] ){

    for( register int i = 0; i < 4; i++ )
        W[i][0] ^= T[i][0];

    for( register int column = 1; column < 4; column++ )
        for( register int row = 0; row < 4; row++ ){
            W[row][column] ^= W[row][column-1];
        }

}
```

Listing 11: Modificación matriz W

La última función intermedia corresponde a la que imprime la matriz W en el archivo de texto plano. Para esto, se hicieron nuevamente dos ciclos `for` anidados y en el cuerpo de estos se realiza la conversión del contenido del `bitset` a hexadecimal, con ayuda del comando mostrado en la sección 3.1.3.

3.2.4 Implementación de Algorithm 1

El programa que contiene la llamada de las funciones explicadas anteriormente se llama KeySchedule.cpp. En este podemos ver la implementación del pseudocódigo utilizado como base para la elaboración del Key Schedule de AES para una llave de tamaño de 128 bits.

Lo primero que se realiza es llamar a las funciones encargadas de obtener el contenido del *.txt* ingresado por el usuario, así como llenar la matriz W con dichos valores. Posteriormente, se cambia el buffer de salida al archivo de texto que contendrá las 10 subclaves, es decir, cada vez que mandemos a imprimir algo con la instrucción `cout` el valor se escribirá en el *.txt*.

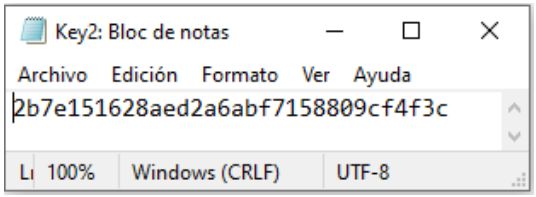
Una vez teniendo lo anterior, se realiza el ciclo `for` mostrado en el pseudocódigo, el cual nos ayudará a obtener las subclaves, dentro de este mandamos llamar a cada una de las funciones explicadas anteriormente en el orden indicado por el mismo, incluyendo al final la función encargada de imprimir la matriz, la cual escribirá en el archivo los valores de W antes de modificarla en la siguiente ronda. La implementación se muestra en el Listing 12.

```
void KeyScheduleAES( string nameFile ){
    string keyString, nametoSave;
    bitset<8> W[4][4], T[4][1];

    nametoSave = FINALFILE + nameFile;

    getKey( nameFile, &keyString );
    keyBit( keyString, W );

    freopen( nametoSave.c_str() , "w", stdout );

```

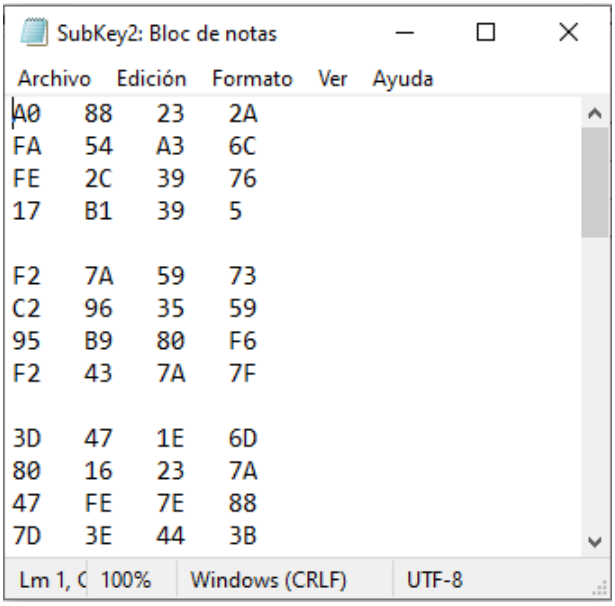
```
12      for( register int i = 1; i <= 10; i++ ){
13          RotBytes( W, T );
14          SubBytes( T );
15          XorRC( T, i );
16          modifyValues( T, W );
17          imprimirMatriz( W );
18          cout<< "\n";
19      }
20
21      fclose(stdout);
22  }
```

Listing 12: Implementación de Algorithm 1

Para verificar que el programa calcula las 10 subclaves de manera correcta se tomó como referencia el ejemplo mostrado en la norma, la cual se encontró en internet. En el Appendix A - Key Expansion Examples se incluirá la parte del PDF que lo contiene. El programa en ejecución, así como los resultados obtenidos se muestran en las Figuras 6 y 5 respectivamente.



(a) Cihper Key



(b) Primeras 3 subllaves generadas

Figura 5: Resultados obtenidos de Key Schedule de AES 128 bits

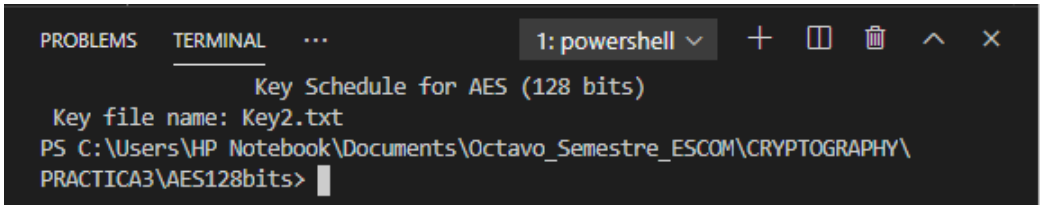


Figura 6: Programa en ejecución

3.3 Key Schedule de AES con llave de 192 bits

3.3.1 Consideraciones del programa

El programa se implementó en el lenguaje de programación C++. Para su elaboración se tomaron en cuenta los siguientes aspectos:

- Se hicieron uso de las funciones elaboradas para el Key Schedule de AES con un tamaño de llave de 128 bits, algunas necesitaron modificaciones en cuanto los índices de los ciclos `for`, estas se mostrarán más adelante.
- Se utilizaron dos arreglos auxiliares de tipo `bitset` para almacenar las últimas dos columnas de ciertas rondas y posteriormente poder concatenarlas con las primeras dos de la ronda siguiente.
- La matriz de sustitución y las constantes C_i se declararon de manera global y de tipo `string` al igual que en el caso de 128 bits.
- El pseudocódigo utilizado para la implementación fue el mismo presentado en la sección 3.2.2 con la excepción de que ahora la matriz W estará conformada por seis elementos, de los cuales, los últimos dos formarán parte de la primera subllave.

3.3.2 Implementación de funciones intermedias

En esta sección se explicarán únicamente las funciones que se modificaron del caso de 128 bits para adaptarlas a 192 bits. La primera hace referencia a la función encargada de almacenar en la matriz W los valores leídos del *.txt* que contiene la llave, debido que como se mencionó anteriormente dicha matriz estará formada por seis columnas.

Ahora el ciclo `for` que hace referencia a las columnas, irá desde 0 hasta 5, dejando todo lo demás igual (véase Listing 13).

```
1     for( register int column = 0; column < 6; column++ )
2         for( register int row = 0; row < 4; row++ ){
```

Listing 13: Cambio de dato a binario y almacenamiento en la matriz

La segunda función que se modificó es la encargada de realizar el corrimiento de la última columna, debido a que ahora no se utilizará la columna 4, si no, la 6. Por lo que, al momento de acceder a las posiciones de la matriz W se usa el índice 5. Este cambió se puede ver en el Listing 14.

```
void RotBytes( bitset<8> W[4][6], bitset<8> T[4][1] ){
    T[3][0] = W[0][5];

    for( register int i = 0; i < 3; i++ )
        T[i][0] = W[i+1][5];
}
```

Listing 14: Función para hacer el corrimiento de la última columna

Las funciones encargadas de realizar la búsqueda en la matriz de sustitución y de realizar las modificaciones correspondientes, así como la que lleva a cabo la operación XOR de la primera fila de la matriz T con la variable C_i se quedaron igual. Por otro lado, la encargada de modificar los valores de la matriz W cambió únicamente en el índice del primer ciclo `for`, el cual hace referencia a la columna de la matriz tomando ahora valores de 0 a 5, este pequeño fragmento de código se muestra en el Listing 15.

```
    for( register int column = 1; column < 6; column++ )
        for( register int row = 0; row < 4; row++ ){
            W[row][column] ^= W[row][column-1];
        }
```

Listing 15: Modificación de la función modifyValues

Cada una de las subllaves esta conformada por una matriz de 4×4, por lo que, para poder calcular las 12 es necesario en algunos casos tomar únicamente las primeras cuatro columnas de W sin perder el valor de las otras dos, debido a que estas formarán parte de la siguiente subllave.

Analizando un poco este proceso, me di cuenta que en las rondas pares siempre se deben almacenar las últimas dos columnas de W. Mientras que en las impares, se obtendrán dos subllaves: la primera estará conformada por la concatenación de las últimas columnas de la ronda anterior y las primeras dos de la actual, y la segunda serán las cuatro columnas restantes.

Tomando en cuenta lo anterior, la función que sufrió más cambios fue la encargada de almacenar las subllaves en el archivo *.txt*, debido a que, aquí se hizo la verificación mencionada anteriormente y dependiendo de la ronda en la que nos encontramos imprime únicamente las primeras cuatro columnas de W ó realiza la concatenación y posteriormente la escritura de las últimas cuatro columnas.

Un error que tenía es que no había tomado en consideración que una parte la primera subllave esta conformada por los valores W_5 y W_6 originales. El código fuente se muestra en el Listing 16.

```
    if( ( i % 2 ) == 0)
        for( register int row = 0; row < 4; row++ ){
            for( register int column = 0; column < 4; column++ )
                cout<< hex << uppercase << (W[row][column]).to_ulong() << "   ";
            cout<< "\n";
        }

    else{
        for( register int row = 0; row < 4; row++ ){
            for( register int column = 0; column < 2; column++ )
                cout<< hex << uppercase << (leftOver[row][column]).to_ulong() << "   ";
            for( register int column = 0; column < 2; column++ )
                cout<< hex << uppercase << (W[row][column]).to_ulong() << "   ";
            cout<< "\n";
        }
        cout<< "\n";
        for( register int row = 0; row < 4; row++ ){
            for( register int column = 2; column < 6; column++ )
                cout<< hex << uppercase << (W[row][column]).to_ulong() << "   ";
            cout<< "\n";
        }
```

Listing 16: Escribir subllaves al archivo de texto

Como se puede observar, para realizar la concatenación de las columnas se realizan dos ciclos `for`: el primero recorre el auxiliar que almacena los valores de las columnas de la ronda anterior y el segundo recorre el `bitset` que contiene los valores de la ronda actual. Finalmente, se almacena la otra subllave perteneciente a las 4 columnas restantes.

3.3.3 Función adicional para almacenar las últimas dos columnas

Se diseñó una función la cual realiza el almacenamiento de las últimas dos columnas de la matriz W en la matriz auxiliar. Para esto, la función recibe ambas matrices y con ayuda de dos ciclos `for` anidados se va haciendo una copia de los valores requeridos (véase Listing 17).

```
void saveValues( bitset<8> W[4][6], bitset<8> leftOver[4][2] ){
    for( register int column = 0; column < 2; column++ )
        for( register int row = 0; row < 4; row++ )
            leftOver[row][column] = W[row][column + 4];
```

Listing 17: Función que almacena las últimas dos columas de W

3.3.4 Implementación de Algortithm 1

El programa presentado en la sección 3.2.4 fue utilizado como base para la implementación de Key Schedule de AES con una llave de tamaño de 192 bits. De hecho, solo se le realizaron dos modificaciones:

- La primera consiste en que se realizó una condicional, la cual determina en qué casos es necesario almacenar las últimas dos columnas de la matriz W en nuestro arreglo auxiliar, llamando a la función saveValues presentada anteriormente.
- La segunda modificación consiste en el número de rondas, debido a que, como se recordará para el caso de 128 bits el ciclo `for` iba de 1 a 10, sin embargo, para 192 bits va de 1 a 8. Esto implica que las variables C_i utilizadas fueron menos.

En el Listing 18 se muestra el pequeño fragmento de código donde se pueden ver los cambios mencionados anteriormente.

```
1     for( register int i = 1; i <= 8; i++ ){
2         if( ( i % 2 ) != 0 )
3             saveValues( W, leftOver );
4         RotBytes( W, T );
5         SubBytes( T );
6         XorRC( T, i );
7         modifyValues( T, W );
8         imprimirMatriz( W, i, leftOver );
9         cout<< "\n";
10    }
```

Listing 18: Modificación de KeySchedule.cpp

Para verificar que el programa calcula las 12 subclaves de manera correcta se tomó como referencia el ejemplo mostrado en la norma, la cual se encontró en internet. En el Appendix A - Key Expansion Examples se incluirá la parte del PDF que lo contiene. El programa en ejecución, así como los resultados obtenidos se muestran en las Figuras 7 y 8 respectivamente.

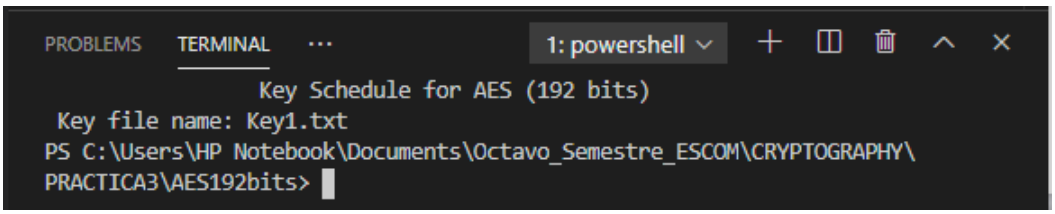
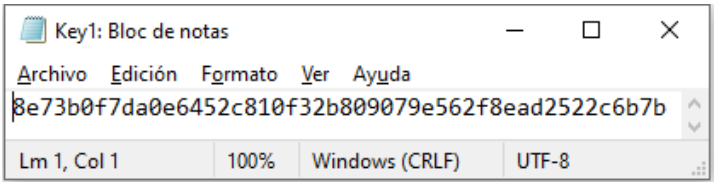
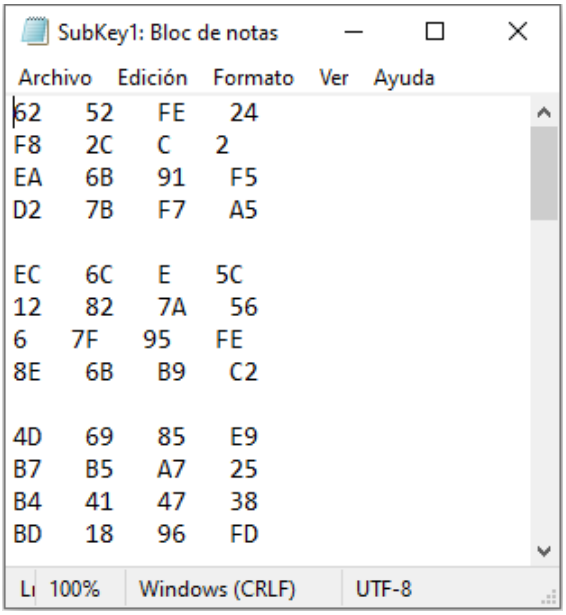


Figura 7: Programa en ejecución



(a) Cihper Key



(b) Primeras 3 subllaves generadas

Figura 8: Resultados obtenidos de Key Schedule de AES 192 bits

4 Manual de usuario

4.1 Finite Fields

Para correr el programa que calcula la tabla de multiplicación de $GF(2^n)$ es necesario encontrarse dentro de la carpeta $FiniteFields$. Dentro de esta, se encuentra otra con el nombre *Archivos* en la cual se guardará la tabla generada por el programa.

Durante la ejecución se le pide al usuario que ingrese el valor de n. Por otro lado, en caso de que desee guardar el archivo en representación polinomial debe contestar la pregunta con *Yes*, en caso contrario con *No*.

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

```
g++ -std=c++11 main.cpp GF2.cpp ProcessGF2.cpp -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

4.2 Key Schedule de AES con llave de 128 bits

Para correr el programa de Key Schedule de AES considerando un tamaño de llave de 128 bits es necesario encontrarse dentro de la carpeta $AES128bits$. Dentro de esta, se encuentra otra con el nombre *Archivos* en la cual se guardará el archivo que contiene las 10 subllaves.

Durante la ejecución se le pide al usuario que ingrese el nombre del *.txt* que contiene la llave de 128 bits en hexadecimal, el cual no debe de tener espacios en blanco.

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

```
g++ -std=c++11 main.cpp KeySchedule.cpp KeyScheduleProcess.cpp
                                -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

4.3 Key Schedule de AES con llave de 192 bits

Para correr el programa de Key Schedule de AES considerando un tamaño de llave de 192 bits es necesario encontrarse dentro de la carpeta $AES192bits$. Dentro de esta, se encuentra otra con el nombre *Archivos* en la cual se guardará el archivo que contiene las 12 subllaves.

Durante la ejecución se le pide al usuario que ingrese el nombre del *.txt* que contiene la llave de 192 bits en hexadecimal, el cual no debe de tener espacios en blanco.

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

```
g++ -std=c++11 main.cpp KeySchedule.cpp KeyScheduleProcess.cpp
                              -o main
```

Posteriormente para su ejecución ponemos:

```
.\main
```

Appendix A - Key Expansion Examples

This appendix shows the development of the key schedule for various key sizes. Note that multi-byte values are presented using the notation described in Sec. 3. The intermediate values produced during the development of the key schedule (see Sec. 5.2) are given in the following table (all values are in hexadecimal format, with the exception of the index column (i)).

A.1 Expansion of a 128-bit Cipher Key

This section contains the key expansion of the following cipher key:

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for ***Nk*** = 4, which results in

w_0 = **2b7e1516** w_1 = **28aed2a6** w_2 = **abf71588** w_3 = **09cf4f3c**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

A.2 Expansion of a 192-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      8e 73 b0 f7 da 0e 64 52 c8 10 f3 2b
                  80 90 79 e5 62 f8 ea d2 52 2c 6b 7b
```

for ***Nk*** = 6, which results in

w_0 = **8e73b0f7** w_1 = **da0e6452** w_2 = **c810f32b** w_3 = **809079e5**

w_4 = **62f8ead2** w_5 = **522c6b7b**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e

9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb095386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
19	485af057					69b54118	21efb14f
20	21efb14f					85a74796	a448f6d9
21	a448f6d9					e92538fd	4d6dce24
22	4d6dce24					e75fad44	aa326360
23	aa326360					bb095386	113b30e6
24	113b30e6	3b30e611	e2048e82	08000000	ea048e82	485af057	a25e7ed5
25	a25e7ed5					21efb14f	83b1cf9a
26	83b1cf9a					a448f6d9	27f93943
27	27f93943					4d6dce24	6a94f767
28	6a94f767					aa326360	c0a69407
29	c0a69407					113b30e6	d19da4e1
30	d19da4e1	9da4e1d1	5e49f83e	10000000	4e49f83e	a25e7ed5	ec1786eb
31	ec1786eb					83b1cf9a	6fa64971
32	6fa64971					27f93943	485f7032
33	485f7032					6a94f767	22cb8755
34	22cb8755					c0a69407	e26d1352
35	e26d1352					d19da4e1	33f0b7b3
36	33f0b7b3	f0b7b333	8ca96dc3	20000000	aca96dc3	ec1786eb	40beeb28
37	40beeb28					6fa64971	2f18a259
38	2f18a259					485f7032	6747d26b
39	6747d26b					22cb8755	458c553e
40	458c553e					e26d1352	a7e1466c
41	a7e1466c					33f0b7b3	9411f1df
42	9411f1df	11f1df94	82a19e22	40000000	c2a19e22	40beeb28	821f750a
43	821f750a					2f18a259	ad07d753

44	ad07d753					6747d26b	ca400538
45	ca400538					458c553e	8fcc5006
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

A.3 Expansion of a 256-bit Cipher Key

This section contains the key expansion of the following cipher key:

```
Cipher Key =      60 3d eb 10 15 ca 71 be 2b 73 ae f0 85 7d 77 81
                  1f 35 2c 07 3b 61 08 d7 2d 98 10 a3 09 14 df f4
```

for ***Nk*** = 8, which results in

w_0 = **603deb10** w_1 = **15ca71be** w_2 = **2b73aef0** w_3 = **857d7781**

w_4 = **1f352c07** w_5 = **3b6108d7** w_6 = **2d9810a3** w_7 = **0914dff4**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i–Nk]	w[i]= temp XOR w[i-Nk]
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229

References

[1] W. Stallings, *Fundamentos de seguridad en redes: aplicaciones y estándares*, 2nd ed. Madrid: Pearson Educación, 2004.

[2] C. P. J. Pelzl, *Understanding Cryptography*, 2nd ed. Berlin Heidelberg: Springer-Verlag, 2010.

[3] L. R. Knudsen and M. J. B. Robshaw, *The Block Cipher Companion*. New York: Springer-Verlag, 2011.

[4] N. P. Smart, *Cryptography Made Simple*. New York: Springer-Verlag, 2016.