



INSTITUTO POLITÉCNICO
NACIONAL



ESCUELA SUPERIOR DE CÓMPUTO

CRYPTOGRAPHY

Operations in binary fields

Autores:

González Núñez Daniel Adrián
Hernández Castellanos César Uriel

Docente:

Dra. Sandra Díaz Santiago

Ingeniería en Sistemas Computacionales

8 de marzo de 2020

1. Introducción.

1.1. Álgebra abstracta

El álgebra abstracta es la parte de la matemática que estudia las estructuras algebraicas como las de grupo, anillo, cuerpo o espacio vectorial

1.2. Campos de Galois

En álgebra abstracta, un cuerpo finito, campo finito o campo de Galois es un cuerpo definido sobre un conjunto finito de elementos. Los cuerpos finitos son importantes en criptografía.

1.3. S-box

En criptografía, una S-Box (substitution box) es un componente básico de los algoritmos de cifrado de clave simétrica. En los cifradores por bloques son usadas a menudo para oscurecer la relación existente entre texto plano y texto cifrado

1.4. Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) es uno de los algoritmos de cifrado más utilizados y seguros actualmente disponibles. Es de acceso público, y es el cifrado que la Agencia de Seguridad Nacional utiliza, es un esquema de cifrado por bloques adoptado como un estándar.

1.5. Construcción del campo finito de AES

Una de las cosas más importantes de AES es conocer muy bien el campo, así como sus operaciones y su representación. Sabemos del anterior punto que es necesario encontrar un polinomio irreducible con coeficientes en $A = A_2$ de grado 8.

- $(10001111) \quad x^7+x^3+x^2+x+1$
- $(11001100) \quad x^7+x^6+x^3+x^2$
- $(10101010) \quad x^7+x^5+x^3+x$

1.6. Operaciones básicas en AES

1.7. Matriz de estado

El AES utiliza una clave de cifrado que puede ser 128, 192 o 256 bits de largo, y se aplica en unidades de datos, llamados bloques, cada uno de los cuales es de 128 bits de largo. El algoritmo AES comienza copiando cada bloque de 16 bits en una matriz bidimensional llamada el Estado, para crear una matriz de bytes de 4×4 . El algoritmo realiza una operación exclusiva \oplus que devuelve "verdadero" si uno u otro de sus operandos es verdadero. Esto se conoce como \oplus RoundKey", y está entre las primeras cuatro filas del programa clave y la matriz de Estado.

1.8. Operaciones matemáticas

Tras la operación inicial exclusivo \oplus , el algoritmo de cifrado AES entra en su bucle principal, en el que realiza repetidamente cuatro operaciones matemáticas diferentes en la matriz de Estado: "SubBytes", "ShiftRows", "MixColumns" y \oplus RoundKey". Estas operaciones emplean una combinación de suma, multiplicación, rotación y sustitución para cifrar cada byte en la matriz de Estado. El bucle principal se ejecuta 10, 12 o 14 veces dependiendo del tamaño de la clave de cifrado. Una vez que se completa la ejecución, el algoritmo copia la matriz de estado a su salida en forma de texto cifrado.

1.9. SubBytes

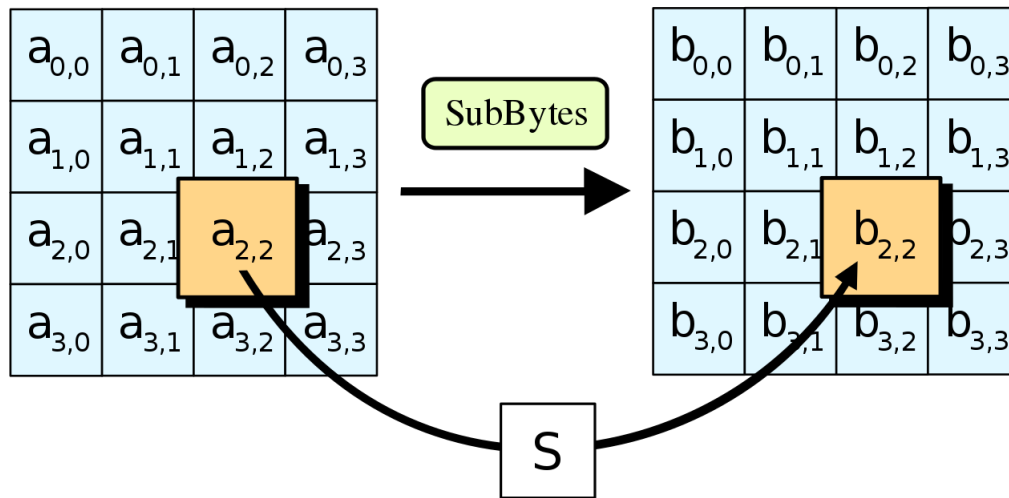


Figura 1: SubBytes

En la fase de SubBytes, cada byte en el state es reemplazado con su entrada en una tabla de búsqueda fija de 8 bits, S ; $b_{ij} = S(a_{ij})$.

1.10. ShiftRows

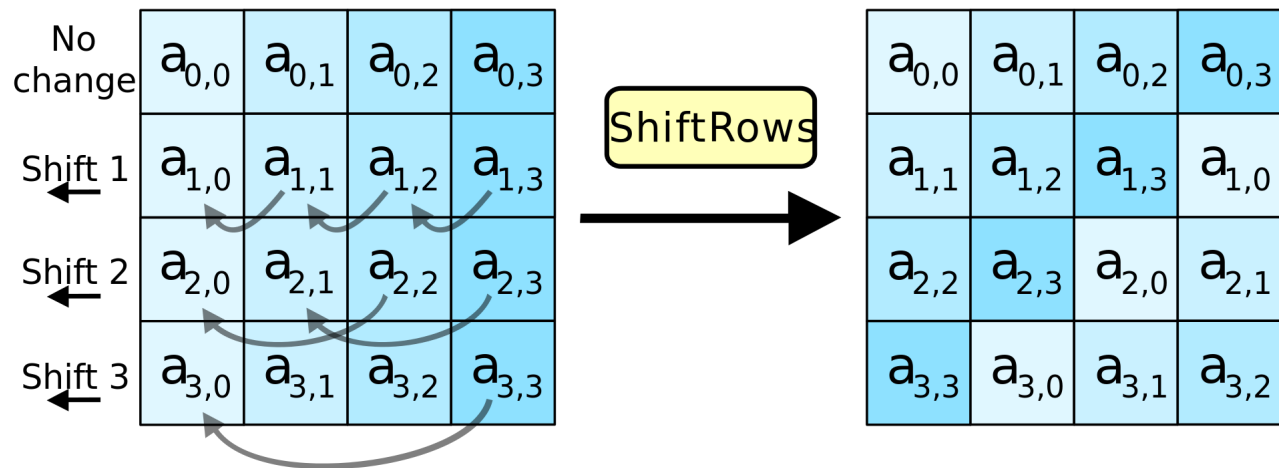


Figura 2: ShiftRows

En el paso ShiftRows, los bytes en cada fila del state son rotados de manera cíclica hacia la izquierda. El número de lugares que cada byte es rotado difiere para cada fila.

1.11. MixColumns

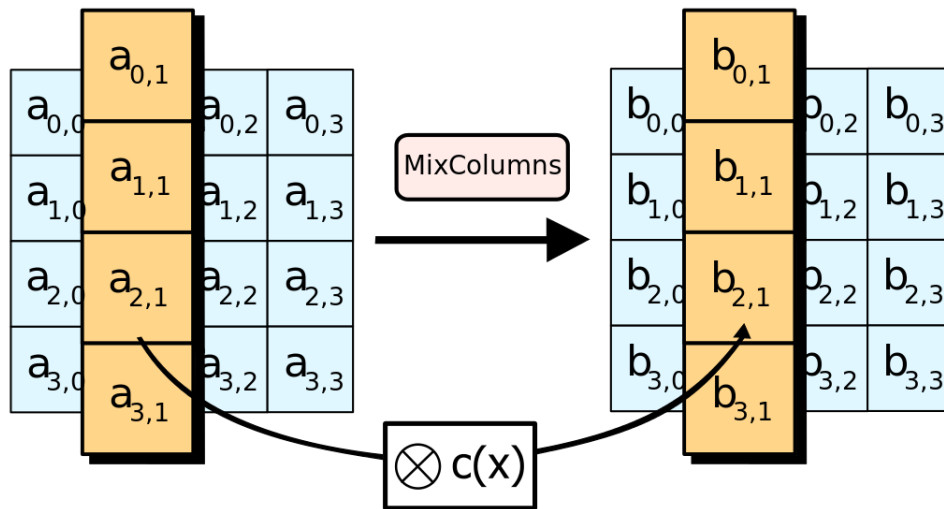


Figura 3: MixColumns

En el paso MixColumns, cada columna del state es multiplicada por un polinomio constante $c(x)$.

1.12. AddRoundKey

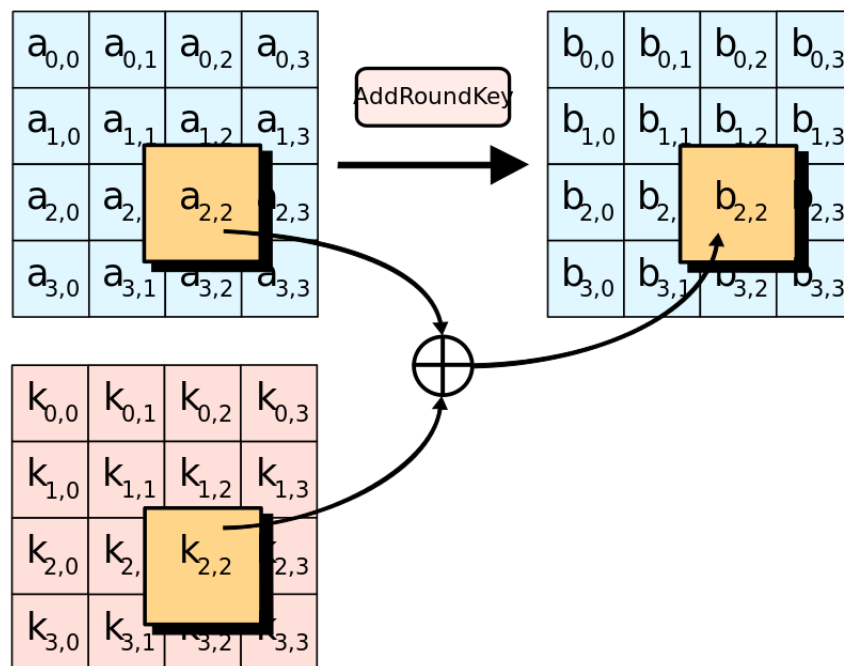


Figura 4: AddRoundKey

En el paso AddRoundKey, cada byte del state se combina con un byte de la subclave usando la operación XOR (\oplus).

1.13. Operaciones matemáticas

2. Descripción del problema

1. Diseñe una función que haga una multiplicación en un campo binario. Esta función debe recibir como parametro un polinomio irreducible de grado n , m , y dos elementos, a y b , pertenecientes a $GF(2^n)$. La función deberá retornar $a * b \bmod m$. Por favor use la representación binaria para a , b y m . El resultado deberá ser impreso en representación binaria. **NO USAR** arreglos para guardar los valores de a , b y m .
2. Diseñe una función que tome una cadena binaria a e imprima su representación en polinomio.
3. Diseñe una función que implemente la operación descrita en la sección anterior y genere la S-box.

3. Descripción de la solución

3.1. Main

En la función principal de inicio mostramos un menú al cual el usuario ingresará y se le mostrarán 3 opciones, una por cada inciso de la práctica. Debemos aclarar que los inputs para cada uno de los incisos están claramente especificados: El input deberá ser una **CADENA BINARIA** que represente al polinomio irreducible o a los números correspondientes.

```

1 def main():
2     clearTerminal()
3     opc = int(input("1.- Multiplication in a binary field\n2.- Get the polynomial
4         ↪ representation \n3.- Get the S-box value \nInput:"))
5     clearTerminal()
6     if(opc == 1):
7         print("All the values should be introduced using a binary representation")
8         mod = input("Introduce the irreducible polynomial:")
9         mod = int(getNumber(mod))
10        a = input("Introduce number A:")
11        a = int(getNumber(a))
12        b = input("Introduce number B:")
13        b = int(getNumber(b))
14        ans = mul(a,b,mod)
15        ans = getString(ans)
16        print("Answer: " + str(ans))
17    elif(opc == 2):
18        print("All the values should be introduced using a binary representation")
19        a = input("Introduce number A:")
20        ans = getPoly(a)
21        print("Answer: "+str(ans))
22    else:
23        generateSbox()

```

Figura 5: Función principal de inicio

3.2. Inciso 1

Para esta función hicimos uso del pseudocódigo realizado en clase para la multiplicación de dos polinomios en AES , solo que generalizamos el grado del polinomio y el módulo para que sea aceptable con cualquier grado. Primero mostraremos la función usada para la multiplicación de un polinomio de grado n por sólo una x , o sea un polinomio de simple de grado 1, tal y como definimos en clase a la multiplicación básica.

```

1 def basic(b,mod):
2     ult = int(math.log2(mod))
3     flag = False
4     b = b << 1
5     if (b&(1<<ult)):
6         b = b ^ (1<<ult)
7         b = b^(mod^(1<<ult))
8     return b

```

Figura 6: Multiplicación básica

Lo que se hace es hacer un corrimiento, hacia la izquierda, de bits al polinomio a multiplicar. Si es que el bit más significativo de este polinomio, al hacerse el corrimiento, empata con el grado del polinomio irreducible que usamos como módulo entonces se hará un *XOR* de las dos variables para obtener el resultado final. Si es que no se logra obtener el empate, al polinomio solo se le efectúa el corrimiento.

Para la segunda parte de este inciso desarrollamos la función principal que obtiene los dos polinomios a multiplicar y el polinomio irreducible a usar.

Lo que se hace es agarrar uno de los dos polinomios e ir checando, bit por bit, si uno de estos se encuentra prendido en su representación binaria o no. Si un bit se encuentra prendido lo que se hace es obtener la posición del bit, para así saber la cantidad de veces que se tendrá que llamar a la función de la multiplicación básica. Es decir, si tenemos a la cadena 100, sabemos que tenemos que hacer la multiplicación 2 veces por que el bit prendido representa a x^2 .

Cada vez que se realiza alguna de estas operaciones tendremos que sobrescribir el resultado final con el cual iniciamos, por lo que tendremos que hacer un *XOR* con el resultado de cada multiplicación.

```

1 def mul(a,b,mod):
2     tam = int(math.log2(a))+1
3     i = 0
4     res = 0
5     while (i < tam):
6         if(a &(1<<i)):
7             aux = b
8             for j in range(i):
9                 aux = basic(aux,mod)
10            res = res ^ aux
11        i= i+1
12    return res

```

Figura 7: Multiplicación de dos polinomios modulo m

Al final se retornará un número que representa el resultado del polinomio. En el main este número se transformará a cadena binaria para ser mostrado como resultado final.

3.3. Inciso 2

Para este inciso recibimos como parámetro una cadena binaria que simboliza a un número del cual debemos obtener su representación como un polinomio.

```

1 def getPoly(s):
2     tam = int(len(s))
3     res = ""
4     for i in range(tam):
5         if(s[i] == '1'):
6             pos = tam-i-1
7             if(pos == 1):
8                 res+="x+"
9             elif(pos == 0):
10                res+="1+"
11            else:
12                res += ("x^" + str(tam-i-1) + "+")
13    return res[0:len(res)-1]

```

Figura 8: Función encargada de obtener el polinomio

Lo que hace esta función es obtener la cadena binaria e ir chequeando, carácter a carácter, si es que este número es un 1 o un 0, es decir, si ese elemento estaría considerado en la representación polinómica o no.

Si la función encuentra un 1, lo que hace es calcular el desplazamiento desde el inicio del recorrido para así asignarle el exponente correspondiente. Si el bit prendido se encuentra hasta el final de la cadena en lugar de imprimir un x elevada a un exponente, lo que hará será imprimir un $+1$. En caso de que se encuentre un 0 no hará nada y seguirá con su recorrido normal.

3.4. Inciso 3

La función principal en donde se genera la S-box consta de una lista que contiene todos los elementos de nuestro alfabeto hexadecimal, los caracteres del 0 al 9 y de la A a la F, y trabaja con dos ciclos que funcionan a manera de concatenación para formar todos los valores posibles de caracteres hexadecimales, es decir: 21, AC, BE, 0F, etc.

```

1 def generateSbox():
2     alpha = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']
3     for i in range(len(alpha)):
4         for j in range(len(alpha)):
5             if(i == 0 and j == 0):
6                 print(str(hex(99)).split('x')[1], end = " ")
7             else:
8                 print(individualValue(alpha[i]+alpha[j]), end = " ")
9     print("")

```

Figura 9: Función encargada de generar la S-box

Una vez que generemos algún valor hexadecimal, lo que haremos será llamar a nuestra función que nos obtiene el valor de la S-box que deberá encontrarse en la intersección de esos dos caracteres. Pero, para poder explicar a fondo esa función, primero debemos presentar otras dos funciones que nos servirán para hacer el cálculo de una manera eficiente.

La primera de estas funciones la usaremos para poder calcular el inverso multiplicativo de nuestro valor hexadecimal que corresponde a la fila y la columna de donde queremos obtener nuestro valor de la S-box. Es común que para conseguir estos valores, en especial para *AES* ya que es un campo muy usado, se usen tablas en las cuales solo se tendrá que buscar el valor deseado. Nosotros, en cambio, aplicamos un algoritmo de **exponenciación binaria** para poder hacer 254 multiplicaciones (254 es el valor definido para un campo con $n = 8$) del valor introducido consigo mismo, todo eso modulo nuestro polinomio irreducible. Al terminar este proceso, el valor en hexadecimal que nos presentará será el inverso multiplicativo de nuestro valor inicial.

```

1  def binPow(a,b,mod):
2      res = 1
3      while(b > 0):
4          if(b & 1):
5              res = mul(res,a,mod)
6              a = mul(a,a,mod)
7              b >>= 1
8      return res
9
10 def getMultiplicativeInverse(a):
11     a = int("0x"+a,0)
12     mod = int(pow(2,8)+pow(2,4)+pow(2,3)+pow(2,1)+pow(2,0))
13     res = binPow(a,254,mod)
14     return hex(res).split('x')[1]

```

Figura 10: Funciones usadas para computar el inverso multiplicativo

Otra función que debemos definir será en la que, después de haber calculado el inverso multiplicativo, haremos la multiplicación de las matrices correspondientes a la sección teórica. Para hacer este cálculo de manera eficiente, usamos un ciclo que va hasta el grado de nuestro campo (8), en el cual iremos haciendo la operación de *XOR* entre los valores de cada celda siempre cuidando el modulo que debemos de aplicar, para al final solo sumar otra vez al vector *c*. Al final de todas estas operaciones lo que obtendremos será una cadena binaria que represente nuestro resultado.

```

1  def getSboxValue(b,c):
2      res = ""
3      for i in range(8):
4          res += str(int(b[i]) ^ int(b[(i+4)%8]) ^ int(b[(i+5)%8]) ^ int(b[(i+6)%8]) ^
5                  ↪ int(b[(i+7)%8]) ^ int(c[i]))
6      return res

```

Figura 11: Funciones usada para conseguir un valor de la S-box

En la función principal para generar un valor individual de la S-box, lo que se hace es calcular el inverso multiplicativo de la casilla deseada (con uso de la función ya explicada anteriormente) para después usar la otra función para así conseguir el valor en específico que ira insertado en esa casilla. Como obtenemos una cadena binaria lo que haremos será convertir esta misma a un número decimal, para después pasarlo a hexadecimal y por último desplegarlo en la pantalla junto a los demás valores que se irán generando.

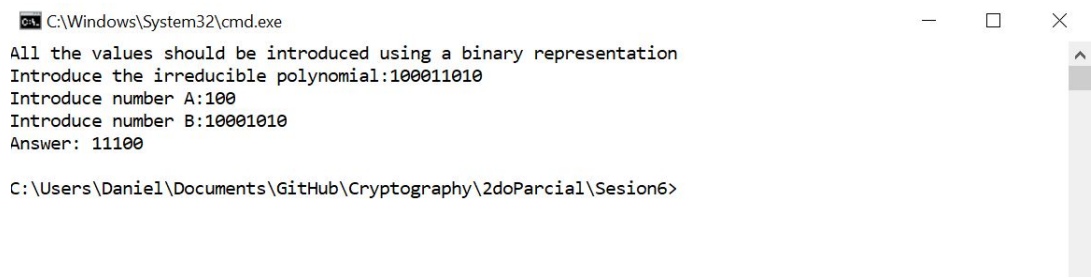

```

1 def individualValue(a):
2     b = getMultiplicativeInverse(a)
3     b = int("0x"+b,0) #Ya es entero
4     b = getString(b)# Sacamos la cadena binaria
5     b = b[::-1]# Volteamos esa cadena
6     # Llenamos de ceros si hacen falta
7     while(len(b) < 8):
8         b+='0'
9     c = "11000110"# Definida por default
10    ans = getSboxValue(b,c)
11    ans = ans[::-1]# Volteamos lo obtenido
12    ans = getNumber(ans)# Obtenemos el entero de la cadena binaria
13    ans = hex(ans)# Obtenemos el hexa del entero

```

Figura 12: Funciones usada generar un valor en específico

4. Capturas de pantalla



```

C:\Windows\System32\cmd.exe
All the values should be introduced using a binary representation
Introduce the irreducible polynomial:100011010
Introduce number A:100
Introduce number B:10001010
Answer: 11100

C:\Users\Daniel\Documents\GitHub\Cryptography\2doParcial\Sesion6>

```

Figura 13: Multiplicación en campo binario



```

C:\Windows\System32\cmd.exe
All the values should be introduced using a binary representation
Introduce number A:100011010
Answer: x^8+x^4+x^3+x

C:\Users\Daniel\Documents\GitHub\Cryptography\2doParcial\Sesion6>

```

Figura 14: Representación como polinomio

```

C:\Windows\System32\cmd.exe

63 7c 77 7b f2 6b 6f c5 30 1 67 2b fe d7 ab 76
ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
4 c7 23 c3 18 96 5 9a 7 12 80 e2 eb 27 b2 75
9 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
53 d1 0 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
d0 ef aa fb 43 4d 33 85 45 f9 2 7f 50 3c 9f a8
51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
cd c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e b db
e0 32 3a a 49 6 24 5c c2 d3 ac 62 91 95 e4 79
e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 8
ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
70 3e b5 66 48 3 f6 e 61 35 57 b9 86 c1 1d 9e
e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
8c a1 89 d bf e6 42 68 41 99 2d f b0 54 bb 16

C:\Users\Daniel\Documents\GitHub\Cryptography\2doParcial\Sesion6>

```

Figura 15: S-box

5. Conclusión

El estudio de funciones sobre campos finitos ha llevado a determinar propiedades que son adecuadas en criptografía, como lo son las permutaciones sobre los campos, particularmente aquellas inducidas por polinomios. Este tipo de permutaciones tiene aplicación en la criptografía como lo son el caso de sistemas de cifrado como DES y AES.

Además de la aplicación mencionado anteriormente, existen otras áreas de estudio en las cuales los campos de Galois tienen influencia en Cohomología de Galois, Teoría de Galois, geometría de Galois algunas áreas de sistemas dinámicos sobre campos de Galois, matrices y grupos clásicos sobre campos de Galois, biología, bioinformática, entre otros.

En tiempos modernos, sobre todo en el manejo de información digital, desarrollo de comunicaciones los campos de Galois tienen un papel importante en su desarrollo y aplicación.