OPERATIONS IN BINARY FIELDS

STUDENTS: BASTIDA PRADO JAIME ARMANDO

SOLORIO PAREDES DANIEL

PROFESSOR: DÍAZ SANTIAGO SANDRA

SUBJECT: CRYPTOGRAPHY

GROUP: 3CM6

April 3rd 2019

1. THEORY

1.1. Obtaining the entry in the S-Box for a = 25

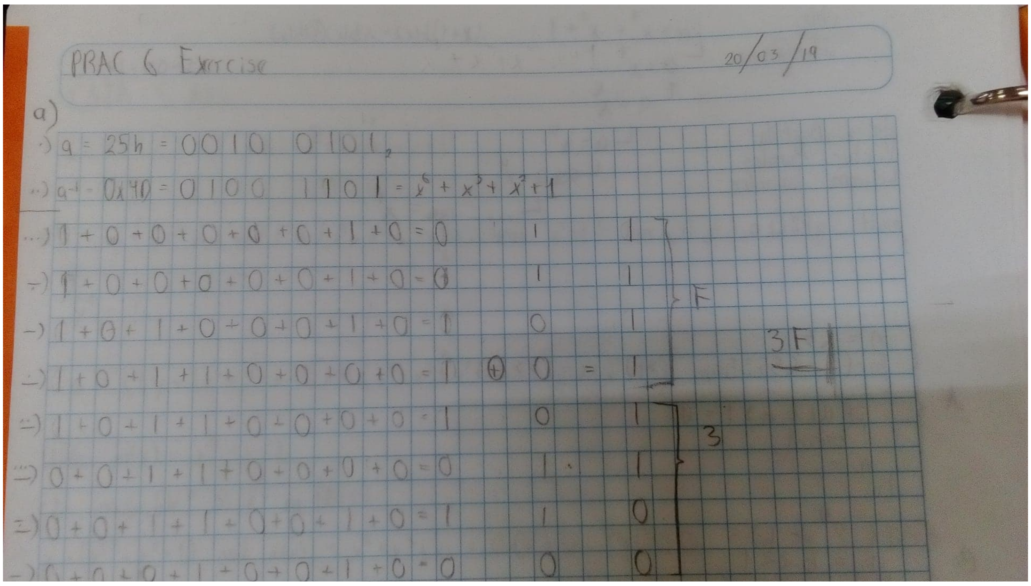
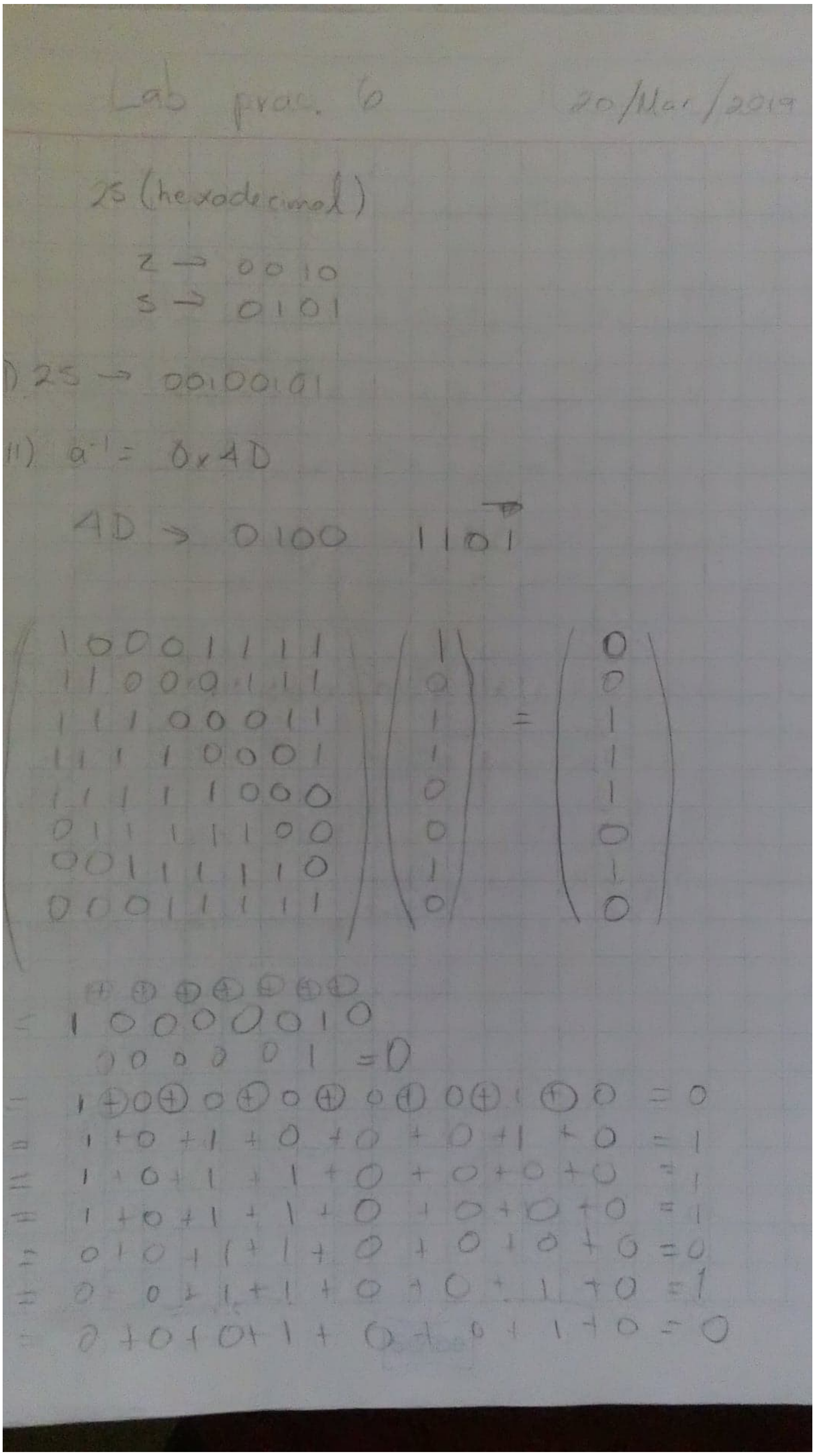


Figura 1: Exercise a) Jaime



Lab prac. 6   20/Mar/2019

25 (hexadecimal)

2 → 0010
5 → 0101

i) 25 → 00100101

ii) $a^{-1} = 0x4D$

4D → 0100 1101

$$\begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix}1\\0\\1\\1\\0\\0\\1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\1\\1\\1\\0\\1\\0\end{pmatrix}$$

$= 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0$
$0\ 0\ 0\ 0\ 0\ 1 = 0$
$= 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$
$= 1 + 0 + 1 + 0 + 0 + 0 + 1 + 0 = 1$
$= 1 + 0 + 1 + 1 + 0 + 0 + 0 + 0 = 1$
$= 1 + 0 + 1 + 1 + 0 + 0 + 0 + 0 = 1$
$= 0 + 0 + 1 + 1 + 0 + 0 + 0 + 0 = 0$
$= 0 + 0 + 1 + 1 + 0 + 0 + 1 + 0 = 1$
$= 0 + 0 + 0 + 1 + 0 + 0 + 1 + 0 = 0$

Figura 2: Exercise a) Daniel

1.2. Explaining point b)

To prove that the operations we saw in class are equal to the ones that we saw in the lab we must prove that doing $[a(x) * (x^4 + x^3 + x^2 + x^1 + 1)]\, mod\, m(x)$ is the same as multiplying $a(x)$ by the matrix given in the lab. We refer to this one:

$$\begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix}$$

Figura 3: Matrix

To prove this we take a general polynomial $a(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_01$ and multiply it by $b(x) = b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_01$. Resulting in the following matrix:

```
x^{11}  x^{10}  x^{9}  x^{8}  x^{7}
x^{10}  x^{9}   x^{8}  x^{7}  x^{6}
x^{9}   x^{8}   x^{7}  x^{6}  x^{5}
x^{8}   x^{7}   x^{6}  x^{5}  x^{4}
x^{7}   x^{6}   x^{5}  x^{4}  x^{3}
x^{6}   x^{5}   x^{4}  x^{3}  x^{2}
x^{5}   x^{4}   x^{3}  x^{2}  x^{1}
x^{4}   x^{3}   x^{2}  x^{1}  x^{0}
```

Figura 4: Matrix result of $a(x) * b(x)$

Then if we apply $mod\, m(x)$, where $m(x) = x^8 + 1$, to each entry in that matrix, the result is:

```
x^{7}   x^{3}   x^{2}  x^{1}  x^{0}
x^{7}   x^{6}   x^{2}  x^{1}  x^{0}
x^{7}   x^{6}   x^{5}  x^{1}  x^{0}
x^{7}   x^{6}   x^{5}  x^{4}  x^{0}
x^{7}   x^{6}   x^{5}  x^{4}  x^{3}
x^{6}   x^{5}   x^{4}  x^{3}  x^{2}
x^{5}   x^{4}   x^{3}  x^{2}  x^{1}
x^{4}   x^{3}   x^{2}  x^{1}  x^{0}
```

Figura 5: Matrix result of $(a(x) * b(x))\, mod\, m(x)$

Which, now quite resembles this:

```
1 0 0 0 1 1 1 1
1 1 0 0 0 1 1 1
1 1 1 0 0 0 1 1
1 1 1 1 0 0 0 1
1 1 1 1 1 0 0 0
0 1 1 1 1 1 0 0
0 0 1 1 1 1 1 0
0 0 0 1 1 1 1 1
```

Figura 6: Matrix with just coefficients

Results in turn, that the matrix is just a representation of the operation $((a(x) * b(x))\, mod\, m(x)$ in a vector bitwise form.The rest of the operations to do are the same that the ones we saw in class, therefore no need to prove them.

2. IMPLEMENTATION

This program implements 6 main functions that help the program produce the S-Box.

2.1. find_m_degree()

This function helps finding the degreee of the m(x) polynomial in hex form (or any other polynomial passed to it). To do so it goes over each of the bits in the polynomial, from the MSB to LSB, until it finds a bit set to 1. Then it takes the degree from doing the following operation $log_2(i) = ln(i)/ln(2)$ over the "i" variable.

```
unsigned short find_m_degree(unsigned int m)
{
    for(unsigned int i = MAX_DEGREE_15; i > 0; i = i >> 1)
        if((i & m) != 0)
            return (unsigned short) (log((double) i) / log(2));
}
```

Figura 7: find_m_degree() function

2.2. find_inverse_in_table()

This function simply finds the inverse of any polynomial (in hex form) passed to it by searching in the text file, containing the multiplicative inverse in $GF(2^8)$ (the second argument).

```
unsigned int find_inverse_in_table(unsigned int a, const char *mult_inv_table_fn)
{
    FILE *read_fp;
    unsigned int i, a_inverse;
    char ch;

    //Reading Multiplicative Inverse Table in GF(2^8)
    if((read_fp = fopen(mult_inv_table_fn, "rb")) == NULL)
    {
        printf("|+|ERROR: Can't open: %s. Try again.\n", mult_inv_table_fn);
        exit(EXIT_FAILURE);
    }
    //Skipping lines i.e. moving over X
    for(i = 0; i < (a >> 4); i++)
    {
        while((ch = getc(read_fp)) != '\n')
            ;
    }
    //Moving over Y
    for(i = 0; i < (a & 0x0F); i++)
        fscanf(read_fp, "%x", &a_inverse);

    fscanf(read_fp, "%x", &a_inverse);
    fclose(read_fp);

    return a_inverse;
}
```

Figura 8: find_inverse_in_table() function

2.3. mult_gf2_n_monomial()

This one is the most important function in the program. It is a recursive function which finds the result of multiply any monomial; e.g. x^6, x^2, x, 1; by any polynomial e.g. $b(x) = x^4+x^3+x^2+x^1+1$ or $b(x) = x^7 + x^4$. Similar to the distribution property when having: $x^n(x^n + x^{n-1} ... x^1 + 1) = x^n * x^n + x^n * x^{n-1} + x^n * x^1 + x^n * 1$.

There are three cases when entering the function, the first one is the basic case i.e. the multiplication is between $1 * b(x)$, the return value is just $b(x)$.

The second case is when we have $x * b(x)$, in this case we do only a left shift of $b(x)$ when $b_{n-1} = 0$ where n is the degree of $b(x)$, but when $b_{n-1} = 1$ we left shift $b(x)$ and do xor with $m(x)$.

The third case is when we have $x^j * b(x)$ where j is an integer greater than 1. In this case we call the function recursively with x^{j-1} and multiply the result of that by x^1.

```
unsigned int mult_gf2_n_monomial(unsigned int a, unsigned int b, unsigr
{
    unsigned int i, aux;

    for(i = MAX_DEGREE_15; i > 0; i = i >> 1)
    {
        if((i & a) == 1) // 1 * b(x) = b(x)
            return b;
        else if((i & a) == 2) // x * b(x) = b(x) << 1
        {
            aux = 1;
            if(((aux << (m_degree - 1)) & b) != 0)
            {
                b = b << 1;
                b = b ^ m;
            }
            else
                b = b << 1;
            return b;
        }
        else if((i & a) != 0)//x^2 or greater, call recursive
        {
            b = mult_gf2_n_monomial(a >> 1, b, m, m_degree);
            aux = 1;
            if(((aux << (m_degree - 1)) & b) != 0)
            {
                b = b << 1;
                b = b ^ m;
            }
            else
                b = b << 1;
            return b;
```

Figura 9: mult_gf2_n_monomial() function

2.4. mult_gf2_n()

This function does the multiplication of any polynomial $a(x)$ by any polynomial $b(x)$ i.e. $(a(x) * b(x))\, mod\, m(x)$. This is done by taking every monomial in $a(x)$ and passing it to the function mult_gf2_n_monomial() thus multiplying by $b(x)$ every monomial in $a(x)$. The result is xored with the previous result until we have multiplied every monomial in $a(x)$ with $b(x)$.

```c
unsigned int mult_gf2_n(unsigned int a, unsigned int b, unsigned int m, unsigned short m_degree)
{
    unsigned int aux_b = 0, result = 0, i;
    //DOING a * b mod m(x)
    //Through this loop we go over every monomial in "a" e.g. a(x) = x^6 + x^3 + x^2 + 1 and b(x
    //In the first iteration we are going to pass to the function a = x^6 multiplied by b(x)
    //The result of that is stored into aux_b and xored with the resul, and so on.
    for(i = MAX_DEGREE_15; i > 0; i = i >> 1)
    {
        if((i & a) != 0)
        {
            aux_b = mult_gf2_n_monomial(i, b, m, m_degree);
            result = result ^ aux_b;
        }
    }

    return result;
}
```

Figura 10: mult_gf2_n() function

2.5. parse_to_polynomial()

The way this function works is quite simple, it just checks every bit set in the polynomial passed to it and adds the corresponding text into a string using sprintf(), the returns the string.

```c
char * parse_to_polynomial(unsigned int number)
{
    char *polynomial;
    polynomial = calloc(FILENAME_MAX, sizeof(char));
    unsigned int i;

    for(i = MAX_DEGREE_15; i > 0; i = i >> 1)
    {
        if((i & number) == 128)
            sprintf(polynomial + strlen(polynomial), "x^7 + ");
        else if((i & number) == 64)
            sprintf(polynomial + strlen(polynomial), "x^6 + ");
        else if((i & number) == 32)
            sprintf(polynomial + strlen(polynomial), "x^5 + ");
        else if((i & number) == 16)
            sprintf(polynomial + strlen(polynomial), "x^4 + ");
        else if((i & number) == 8)
            sprintf(polynomial + strlen(polynomial), "x^3 + ");
        else if((i & number) == 4)
            sprintf(polynomial + strlen(polynomial), "x^2 + ");
        else if((i & number) == 2)
            sprintf(polynomial + strlen(polynomial), "x^1 + ");
        else if((i & number) == 1)
            sprintf(polynomial + strlen(polynomial), "1");
    }
```

Figura 11: parse_to_polynomial() function

2.6. construct_sbox()

The last function takes a polynomial $b(x)$, $m(x)$ and $c(x)$ and the file name of the file where we want to store the S-Box. The it makes use of every of the other five functions to construct the S-Box and place it in the file desired. It also saves the S-Box in polynomial form in another file.

```c
void construct_sbox(unsigned int b, unsigned int m, unsigned int c, const char *sbox_table_fn)
{
    unsigned int a, a_inverse, result;
    unsigned short m_degree;
    const char *mult_inv_table_fn = "MultInverseTable.txt";
    const char *sbox_polynomial_table_fn = "SBoxPolynomial.txt";
    char *polynomial;
    FILE *write_fp1, *write_fp2;

    write_fp1 = fopen(sbox_table_fn, "w");
    write_fp2 = fopen(sbox_polynomial_table_fn, "w");
    m_degree = find_m_degree(m);
    short jump = 0;
    for(a = 0; a < 256; a++)
    {
        if(jump == 16)
        {
            fprintf(write_fp1, "\n");
            fprintf(write_fp2, "\n");
            jump = 0;
        }
        a_inverse = find_inverse_in_table(a, mult_inv_table_fn);
        result = mult_gf2_n(a_inverse, b, m, m_degree);         //Result of multiply a(x) * b(
        result = result ^ c;
        fprintf(write_fp1 ,"%.2X ", result);
        polynomial = parse_to_polynomial(result);
        fprintf(write_fp2, "%s\n", polynomial);
        jump++;
```

Figura 12: construct_sbox() function