A practical way to generalize propagation of linear masks through linear transformations

Jean-Philippe.Bossuat@unil.ch

02.08.2018

1 Introduction

1.1 Paper content

This short paper aims to provide a practical way to understand and implement ways to calculate input and output mask propagation through linear transformations. The diffusion part of modern ciphers often uses complex linear transformations (as an example take the AES cipher), and it is not always obvious how to calculate input and output masks.

The basics of linear cryptanalysis will not be recalled, but a good introduction can be found in *Linaer cryptanalysis Method for DES Cipher* from Mitsuru Matsui, and *A tutorial on Linear and Differential Cryptanalysis* from Howard M. Heys. Also, an example will be based on the Rijdael MixColumns linear transformation (https://en.wikipedia.org/wiki/Rijndael_MixColumns).

1.2 About me and my motivations

I am law student, doing a master in information law, security and criminality. I took some cryptography courses, and since then this has become a subject of interest.

In one of the cryptography course I took, we were introduced to linear cryptanalysis, but we did not spent much time on it, and we especially did not see how linear masks propagate through complex linear functions. So I tried to find an answer on the web, but unfortunately I did not find a paper dedicated to it, or only solutions for a specific cipher that assumed the reader was already advanced in cryptanalysis.

So I took time to investigate the subject, clarify it and make this paper, aimed at people who are not experts, but still interested in the subject. This is the result of my research.

2 The basics about the propagation of linear masks

To begin with, everything that will follow will be based on the two simple rules needed to describe the propagation of linear masks through xor and duplicate gates (Figure 1). For non linear functions, the only way (to my knowledge) is to do an exhaustive search (usually with a complexity of $\approx n^3$).

Figure 1: xor and duplicate gates

For a xor gate, the output mask is duplicated to the two input masks, and for a duplicate gate, the input mask is the xor of the two output masks (Figure 2).

Figure 2: mask propagation through xor and duplicate gates

So, given a function that can be described with only xor and duplicate gates, the propagation of an output to its input mask, can simply be computed by swapping the xor and duplicate gates and running the linear function backward.

Here is a more concrete example with a more complicated but still simple linear function F, with A as input and B as output. To compute the input mask a from an output mask b, we rewrite the function as F', swap xor and duplicate gates, and run it backward with b as input (Figure 3). Of course, A, B, a, b can have any bit length.

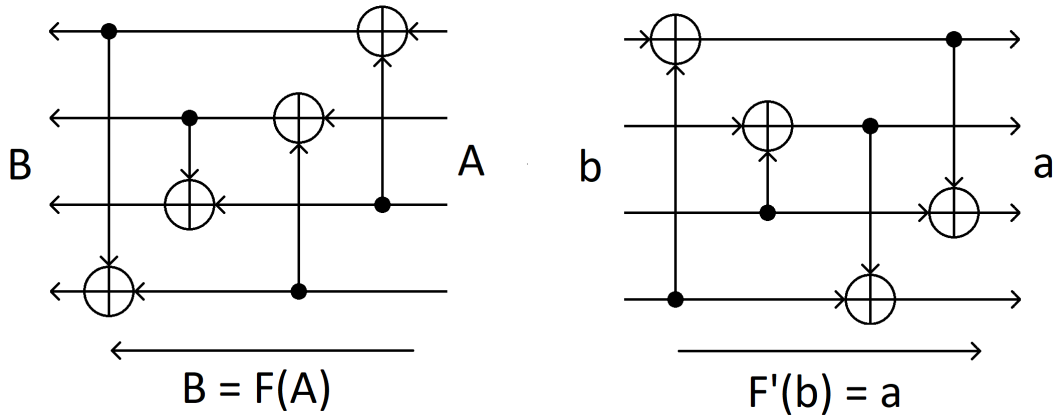

Figure 3: F and F'

To compute the output mask b from the input mask a we need the inverse of F'.

3 Generalizing the propagation

The next step is to understand that any function that can be described only with xor and duplicate gates can also be described with a binary matrix.

Let's say that $A = (A_0, A_1, A_2, A_3)$ and $B = (B_0, B_1, B_2, B_3)$ are words of 4 bytes. Then $F(A) = B$ (from Figure 3) can be written as :

$$\begin{aligned} B_0 &= A_0 \oplus A_2 \\ B_1 &= A_1 \oplus A_3 \\ B_2 &= A_1 \oplus A_2 \oplus A_3 \\ B_3 &= A_0 \oplus A_2 \oplus A_3 \end{aligned}$$

Which is equivalent to dot product of the vector (A_0, A_1, A_2, A_3) by the following binary matrix (later the vectors will be omitted) :

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix}.$$

And $F'(b) = a$ (also from Figure 3) can be written as :

$$\begin{aligned} a_0 &= b_0 \oplus b_3 \\ a_1 &= b_1 \oplus b_2 \\ a_2 &= b_0 \oplus b_2 \oplus b_3 \\ a_3 &= b_1 \oplus b_2 \oplus b_3 \end{aligned}$$

Which is equivalent to vector multiplication by the binary matrix :

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Looking at those two matrices, we observe that they are just the transpose of each other. So, if we are able to describe a linear function with a binary matrix, then we only need to transpose it to get the function to apply to the output mask to compute the input mask. To be able to compute the output mask from the input mask, the matrix needs to be inverted.

3.1 Composed linear functions

But what about linear function that is not only composed of basic xor and duplicate gates? (ex : bit rotation, GF(2^8) arithmetic...) ? Well, this can be solved with what was previously described.

First, we must find a way to describe all the different linear function as binary matrices, treat them as element of a bigger binary matrix (we will have matrices into matrices), and then transpose everything.

Lets take the previous linear functions F, F', and add some bit rotation to it :

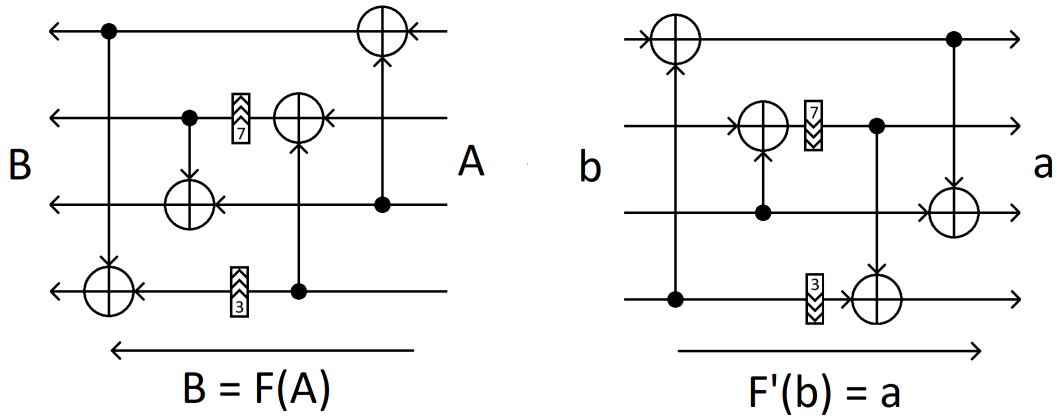

Figure 4: F and F'

The new $F()$:

$$\begin{aligned} B_0 &= A_0 \oplus A_2 \\ B_1 &= ((A_1 \oplus A_3) <<< 7) \\ B_2 &= A_2 \oplus ((A_1 \oplus A_3) <<< 7) \\ B_3 &= A_0 \oplus A_2 \oplus (A_3 <<< 3) \end{aligned}$$

And the new $F'()$:

$$\begin{aligned} a_0 &= b_0 \oplus b_3 \\ a_1 &= ((b_1 \oplus b_2) >>> 7) \\ a_2 &= b_0 \oplus b_2 \oplus b_3 \\ a_3 &= ((b_1 \oplus b_2) >>> 7) \oplus (b_3 >>> 3) \end{aligned}$$

Assuming we work with bytes, bit rotation are easily expressed with binary matrices. Here are the matrices, respectively for a rotation of 3 and 7 to the left :

$$M_{rot3} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad M_{rot7} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

And for a rotation of 3 and 7 to the right (which are simply transposes of the rotation to the left, a special case where the inverse is also the transpose) :

$$M^t_{rot3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \; M^t_{rot7} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Let's rewrite the matrices of $F()$ and $F'()$ (from Figure 3) with M_3, M_7, M^t_3, M^t_7:

$$F = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & M_{rot7} & 0 & M_{rot7} \\ 0 & M_{rot7} & 1 & M_{rot7} \\ 1 & 0 & 1 & M_{rot3} \end{bmatrix}, \; F' = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & M^t_{rot7} & M^t_{rot7} & 0 \\ 1 & 0 & 1 & 1 \\ 0 & M^t_{rot7} & M^t_{rot7} & M^t_{rot3} \end{bmatrix}$$

And voilà. Those two matrices are the binary representation of new F and F' and we can compute the input mask from the output mask from F with F'. Of course, to be technically exhaustive, do not forget that the 1 and 0 coefficients should also be seen as binary 8x8 matrices (respectively the identity matrix and the zero matrix), because F and F', with 4 bytes as input and output, are actually 32x32 binary matrices.

4 Practical example : AES MixColumns

Given everything that was described in the previous sections, we will now find the binary matrix that fully describe the AES Mix Column, and its transpose to be able to compute the masks.

Given an input word (a, b, c, d) and an ouput word (x_0, x_1, x_2, x_3) of 4 bytes, the linear transformation of AES can be described as a matrix, whose coefficients are elements of GF(2^8):

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

This is a good start, but this is not a binary matrix, we still have coefficients that are greater than 1. So we need to describe those multiplication by 2 and 3 in GF(2^8) with binary matrices.

The irreducible polynomial used for the modular reduction in the GF(2^8) arithmetic of AES is $x^8 + x^4 + x^3 + x + 1$, which is 100011011 in binary.

Concretely, given an element $x \in$ GF(2^8), to multiply it by 2, we simply do a left shift and xor with 0b100011011 if the result of the shift gets above 0b11111111 (255). To multiply by 3, we simply multiply by 2 and add the input.

Those two multiplications can be described with 8x8 binary matrices. The easiest way to find the matrices is to think how those operations would be implemented in hardware. Well, there is an efficient way to do it. Given a 8 bit inputs, $y_0y_1y_2y_3y_4y_5y_6y_7$, first store y_0, do a left shift, and xor the result with $(0b00011011 \cdot y_0)$. And the multiplication by 3 is simply a multiplication by two xored with the input.

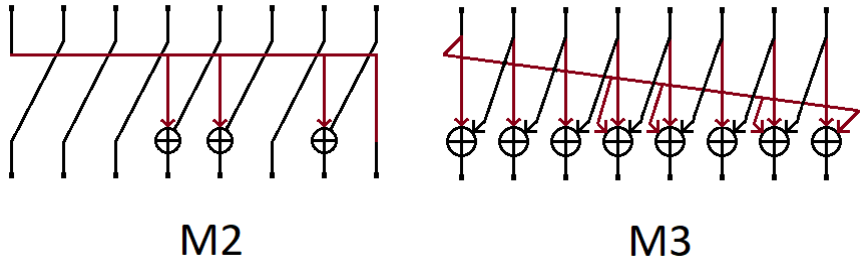

Figure 5: possible hardware implementation of the multiplication by 2 and 3 in GF(2^8) mod $x^8 + x^4 + x^3 + x + 1$.

Those multiplications by 2 and 3 are described by the following binary matrices :

$$M_{mul2} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad M_{mul3} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

We now replace the 2 and 3 coefficients by M_{mul2} and M_{mul3} :

$$\begin{bmatrix} M_{mul2} & M_{mul3} & 1 & 1 \\ 1 & M_{mul2} & M_{mul3} & 1 \\ 1 & 1 & M_{mul2} & M_{mul3} \\ M_{mul3} & 1 & 1 & M_{mul2} \end{bmatrix},$$

which gives us a complete description of the AES MixColumns with a 32x32 binary matrix. It's transpose is :

$$\begin{bmatrix} M^t_{mul2} & 1 & 1 & M^t_{mul3} \\ M^t_{mul3} & M^t_{mul2} & 1 & 1 \\ 1 & M^t_{mul3} & M^t_{mul2} & 1 \\ 1 & 1 & M^t_{mul3} & M^t_{mul2} \end{bmatrix}.$$

We now have a working description of the AES Mix Column and its transpose to compute the masks. The inverse of the AES MixColumns can be computed in several ways. Either by finding the inverse of the polynomial $3x^3 + x^2 + x + 2 \bmod x^4 + 1$ using the extended euclidean algorithm, which is $11x^3 + 13x^2 + 9x + 14$, and finding the binary matrices for the multiplication by 9,11,13 and 14 (which can be done with a double and add algorithm using the M_{mul2} and $M_{identity}$ matrices, ex: $M_{mul14} = (((M_{mul2} \oplus M_{identity}) \cdot M_{mul2}) \oplus M_{identity}) \cdot M_{mul2})$ or by inverting the 32x32 binary matrix.

What's following is some python code illustrating this example.

```
from random import randrange

# Precomputed parity S-BOX from 0 to 255
P = [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]

# Computes the parity of a 64 bit unsigned integer
def parity(x):
    #x ^= x >> 32
    x ^= x >> 16
    x ^= x >> 8
    return P[x&0xFF]



def M(x,m,s):
    """ INPUT : x = vector (as integer), m = matrix (as list of vectors),
        OUTPUT : matrix multiplication of x*m
    """

    z = 0
    for i in range(s):
        z<<=1
        z |= parity(x & m[i])

    return z

def transpose(x,s):
    """ INPUT :  x = list of masks, s = mask bit length
        OUTPUT : transposed masks
    """
    #Converts the masks into binary lists
    m = []
    for mask in x:
        tmp = []
        for i in range(s):
            tmp += [(mask>>(s-i-1))&1]
        m += [tmp]

    #Transposes
    t = [[m[j][i] for j in range(len(m))] for i in range(len(m[0]))]

    #Converts the binary lists into masks
    m = []
    for x in t:
        mask = 0
        for bit in x:
            mask<<=1
            mask |= bit
```

```
            m += [mask]

    return m

s = 8 # bit length of the vectors


m2 = [0b01000000,
      0b00100000,
      0b00010000,
      0b10001000,
      0b10000100,
      0b00000010,
      0b10000001,
      0b10000000,]

m3 = [0b11000000,
      0b01100000,
      0b00110000,
      0b10011000,
      0b10001100,
      0b00000110,
      0b10000011,
      0b10000001,]

m14 = [0b01110000,
       0b10111000,
       0b01011100,
       0b00101110,
       0b01100111,
       0b01000011,
       0b00100001,
       0b11100000,]

m11 = [0b11010000,
       0b11101000,
       0b11110100,
       0b11111010,
       0b00101101,
       0b11000110,
       0b11100011,
       0b10100001,]

m13 = [0b10110000,
       0b11011000,
       0b01101100,
       0b10110110,
       0b11101011,
       0b01000101,
       0b10100010,
       0b01100001,]

m9 = [0b10010000,
      0b11001000,
      0b11100100,
      0b01110010,
```

```
      0b10101001,
      0b11000100,
      0b01100010,
      0b00100001,]

m2t = transpose(m2,s)
m3t = transpose(m3,s)

m14t = transpose(m14,s)
m11t = transpose(m11,s)
m13t = transpose(m13,s)
m9t = transpose(m9,s)

def mix_column(x):
    a = x>>24 & 0xFF
    b = x>>16 & 0xFF
    c = x>> 8 & 0xFF
    d = x     & 0xFF

    x0 = M(a,m2,s) ^ M(b,m3,s) ^   c        ^   d
    x1 =   a       ^ M(b,m2,s) ^ M(c,m3,s) ^   d
    x2 =   a       ^   b       ^ M(c,m2,s) ^ M(d,m3,s)
    x3 = M(a,m3,s) ^   b       ^   c       ^ M(d,m2,s)

    return x0<<24 | x1<<16 | x2<<8 | x3

def mix_column_t(x):
    a = x>>24 & 0xFF
    b = x>>16 & 0xFF
    c = x>> 8 & 0xFF
    d = x     & 0xFF

    x0 = M(a,m2t,s) ^   b         ^   c         ^ M(d,m3t,s)
    x1 = M(a,m3t,s) ^ M(b,m2t,s) ^   c         ^   d
    x2 =   a        ^ M(b,m3t,s) ^ M(c,m2t,s) ^   d
    x3 =   a        ^   b        ^ M(c,m3t,s) ^ M(d,m2t,s)

    return x0<<24 | x1<<16 | x2<<8 | x3


def mix_column_INV(x):
    a = x>>24 & 0xFF
    b = x>>16 & 0xFF
    c = x>> 8 & 0xFF
    d = x     & 0xFF

    x0 = M(a,m14,s) ^ M(b,m11,s) ^ M(c,m13,s) ^ M(d,m9,s)
    x1 = M(a,m9,s)  ^ M(b,m14,s) ^ M(c,m11,s) ^ M(d,m13,s)
    x2 = M(a,m13,s) ^ M(b,m9,s)  ^ M(c,m14,s) ^ M(d,m11,s)
    x3 = M(a,m11,s) ^ M(b,m13,s) ^ M(c,m9,s)  ^ M(d,m14,s)

    return x0<<24 | x1<<16 | x2<<8 | x3


def mix_column_INV_t(x):
    a = x>>24 & 0xFF
```

```
    b = x>>16 & 0xFF
    c = x>> 8 & 0xFF
    d = x     & 0xFF

    x0 = M(a,m14t,s) ^ M(b,m9t,s)  ^ M(c,m13t,s) ^ M(d,m11t,s)
    x1 = M(a,m11t,s) ^ M(b,m14t,s) ^ M(c,m9t,s)  ^ M(d,m13t,s)
    x2 = M(a,m13t,s) ^ M(b,m11t,s) ^ M(c,m14t,s) ^ M(d,m9t,s)
    x3 = M(a,m9t,s)  ^ M(b,m13t,s) ^ M(c,m11t,s) ^ M(d,m14t,s)

    return x0<<24 | x1<<16 | x2<<8 | x3

assert mix_column(0xdb135345)       == 0x8e4da1bc
assert mix_column_t(0xdb135345)     == 0xca29ad90
assert mix_column_INV(0x8e4da1bc)   == 0xdb135345
assert mix_column_INV_t(0xca29ad90) == 0xdb135345
```