



北京邮电大学

Beijing University of Posts and Telecommunications

# 分析报 告

题	目	AES 逆向分析
姓	名	████████████████████
学	号	████████████████████
班	级	2021211805
指	导	付俊松
教	师	

## 目录

AES 算法 .....	2
AES 算法 .....	2
AES 算法流程图 .....	4
密钥加法层 .....	5
字节代换层 .....	6
行位移——ShiftRows .....	6
列混淆——MixColumn .....	7
AES 密钥生成 .....	8

AES 逆向分析 .....	9
Main 函数 .....	9
AesEncrypt 函数 .....	11
KeyExpansion .....	13
LoadStateArray .....	20
AddRoundKey .....	23
SubBytes .....	25
Shift_Rows .....	26
MixColumns .....	29
GMul .....	33
StoreStateArray .....	38
总结 .....	40

## AES 算法

### AES 算法

主要有四种操作处理，分别是密钥加法层(也叫轮密钥加，英文 Add Round Key)、字节代换层(SubByte)、行位移层(Shift Rows)、列混淆层(Mix Column)。而明文 x 和密钥 k 都是由 16 个字节组成的数据(当然密钥还支持 192 位和 256 位的长度，暂时不考虑)，它是按照字节的先后顺序从上到下、从左到右进行排列的。而加密出的密文读取顺序也是按照这个顺序读取的，相当于将数组还原成字符串的模

样了，然后再解密的时候又是按照  $4 \times 4$  数组处理的。AES 算法在处理的轮数上只有最后一轮操作与前面的轮处理上有些许不同(最后一轮只是少了列混淆处理)，在轮处理开始前还单独进行了一次轮密钥加的处理。在处理轮数上，我们只考虑 128 位密钥的 10 轮处理。接下来，就开始一步步的介绍 AES 算法的处理流程了。

输入的字节顺序:

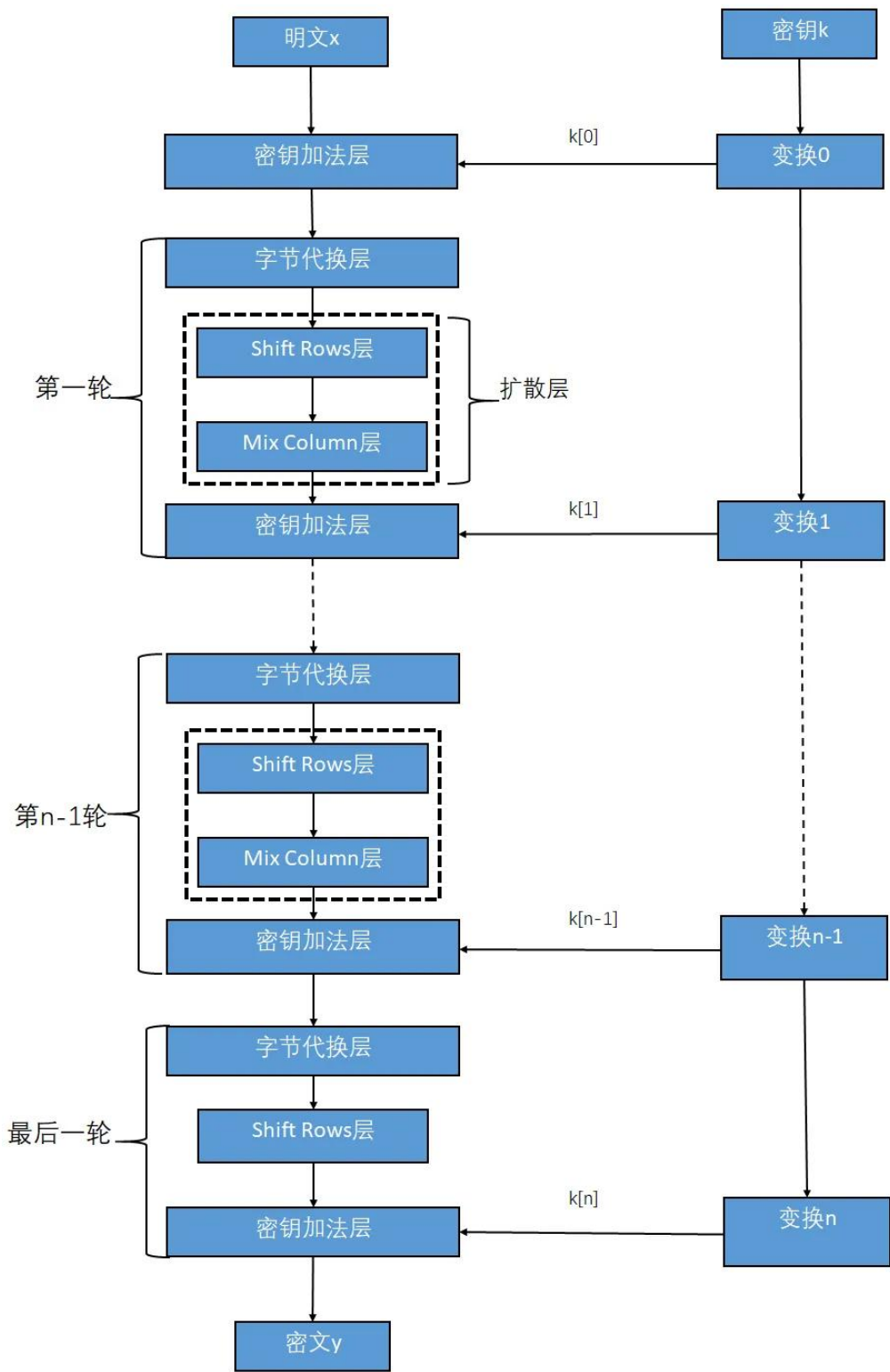
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----



规定的字节排列方式:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

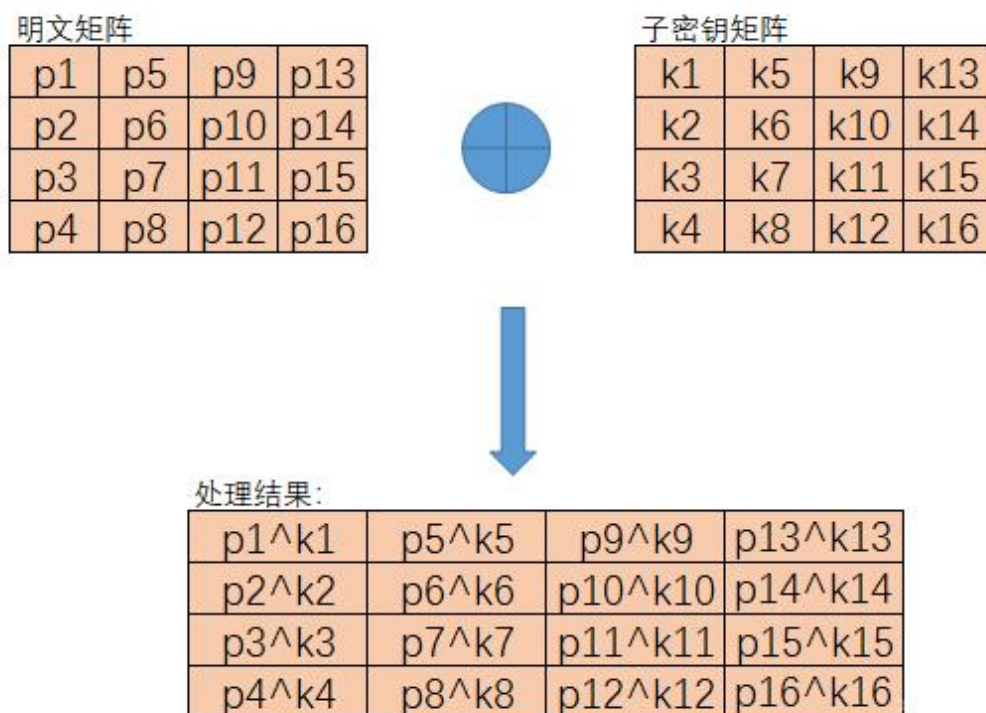
AES 算法流程图



## 密钥加法层

在密钥加法层中有两个输入的参数，分别是明文和子密钥  $k[0]$ ，而且这两个输入都是 128 位的。 $k[0]$  实际上就等同于密钥  $k$ ，具体原因在密钥生成中进行介绍。我们前面在介绍扩展域加减法中提到过，在扩展域中加减法操作和异或运算等价，所以这里的处理也就异常的简单了，只需要将两个输入的数据进行按字节异或操作就会得到运算的结果。

图示：



## 字节代换层

字节代换层的主要功能就是让输入的数据通过 S\_box 表完成从一个字节到另一个字节的映射，这里的 S\_box 表是通过某种方法计算出来的，具体的计算方法将在进阶部分进行介绍，我们基础部分就只给出计算好的 S\_box 结果。S\_box 表是一个拥有 256 个字节元素的数组，可以将其定义为一维数组，也可以将其定义为 16•16 的二维数组，如果将其定义为二维数组，读取 S\_box 数据的方法就是要将输入数据的每个字节的高四位作为第一个下标，第四位作为第二个下标，略有点麻烦。这里建议将其视作一维数组即可。逆 S 盒与 S 盒对应，用于解密时对数据处理，我们对解密时的程序处理称作逆字节代换，只是使用的代换表盒加密时不同而已。

列号 行号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38	0xBF	0x40	0xA3	0x9E	0x81	0xF3	0xD7	0xFB
1	0x7C	0xE3	0x39	0x82	0x9B	0x2F	0xFF	0x87	0x34	0x8E	0x43	0x44	0xC4	0xDE	0xE9	0xCB
2	0x54	0x7B	0x94	0x32	0xA6	0xC2	0x23	0x3D	0xEE	0x4C	0x95	0x0B	0x42	0xFA	0xC3	0x4E
3	0x08	0x2E	0xA1	0x66	0x28	0xD9	0x24	0xB2	0x76	0x5B	0xA2	0x49	0x6D	0x8B	0xD1	0x25
4	0x72	0xF8	0xF6	0x64	0x86	0x68	0x98	0x16	0xD4	0xA4	0x5C	0xCC	0x5D	0x65	0xB6	0x92
5	0x6C	0x70	0x48	0x50	0xFD	0xED	0xB9	0xDA	0x5E	0x15	0x46	0x57	0xA7	0x8D	0x9D	0x84
6	0x90	0xD8	0xAB	0x00	0x8C	0xBC	0xD3	0x0A	0xF7	0xE4	0x58	0x05	0xB8	0xB3	0x45	0x06
7	0xD0	0x2C	0x1E	0x8F	0xCA	0x3F	0x0F	0x02	0xC1	0xAF	0xBD	0x03	0x01	0x13	0x8A	0x6B
8	0x3A	0x91	0x11	0x41	0x4F	0x67	0xDC	0xEA	0x97	0xF2	0xCF	0xCE	0xF0	0xB4	0xE6	0x73
9	0x96	0xAC	0x74	0x22	0xE7	0xAD	0x35	0x85	0xE2	0xF9	0x37	0xE8	0x1C	0x75	0xDF	0x6E
10	0x47	0xF1	0x1A	0x71	0x1D	0x29	0xC5	0x89	0x6F	0xB7	0x62	0x0E	0xAA	0x18	0xBE	0x1B
11	0xFC	0x56	0x3E	0x4B	0xC6	0xD2	0x79	0x20	0x9A	0xDB	0xC0	0xFE	0x78	0xCD	0x5A	0xF4
12	0x1F	0xDD	0xA8	0x33	0x88	0x07	0xC7	0x31	0xB1	0x12	0x10	0x59	0x27	0x80	0xEC	0x5F
13	0x60	0x51	0x7F	0xA9	0x19	0xB5	0x4A	0x0D	0x2D	0xE5	0x7A	0x9F	0x93	0xC9	0x9C	0xEF
14	0xA0	0xE0	0x3B	0x4D	0xAE	0x2A	0xF5	0xB0	0xC8	0xEB	0xBB	0x3C	0x83	0x53	0x99	0x61
15	0x17	0x2B	0x04	0x7E	0xBA	0x77	0xD6	0x26	0xE1	0x69	0x14	0x63	0x55	0x21	0x0C	0x7D

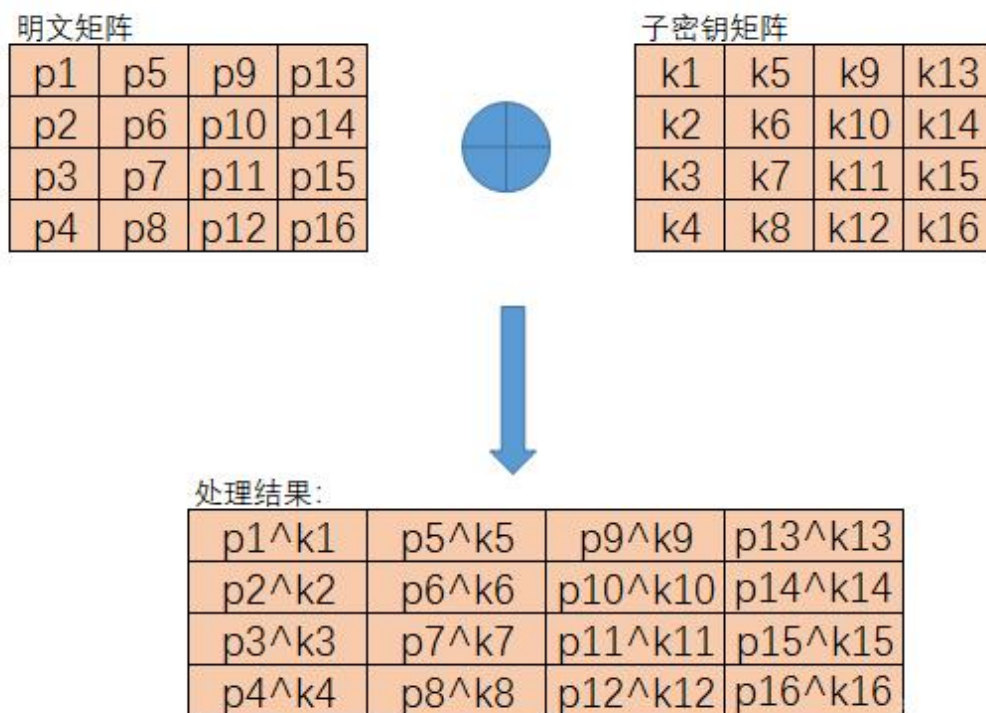
## 行位移——ShiftRows

行位移操作最为简单，它是用来将输入数据作为一个 4•4 的字节矩阵进行处理的，然后将这个矩阵的字节进行位置上的置换。ShiftRows 子层属于 AES 手动的

扩散层，目的是将单个位上的变换扩散到影响整个状态当，从而达到雪崩效应。在加密时行位移处理与解密时的处理相反，我们这里将解密时的处理称作逆行位移。它之所以称作行位移，是因为它只在  $4 \cdot 4$  矩阵的行间进行操作，每行 4 字节的数据。在加密时，保持矩阵的第一行不变，第二行向左移动 8Bit(一个字节)、第三行向左移动 2 个字节、第四行向左移动 3 个字节。而在解密时恰恰相反，依然保持第一行不变，将第二行向右移动一个字节、第三行右移 2 个字节、第四行右移 3 个字节。操作结束！

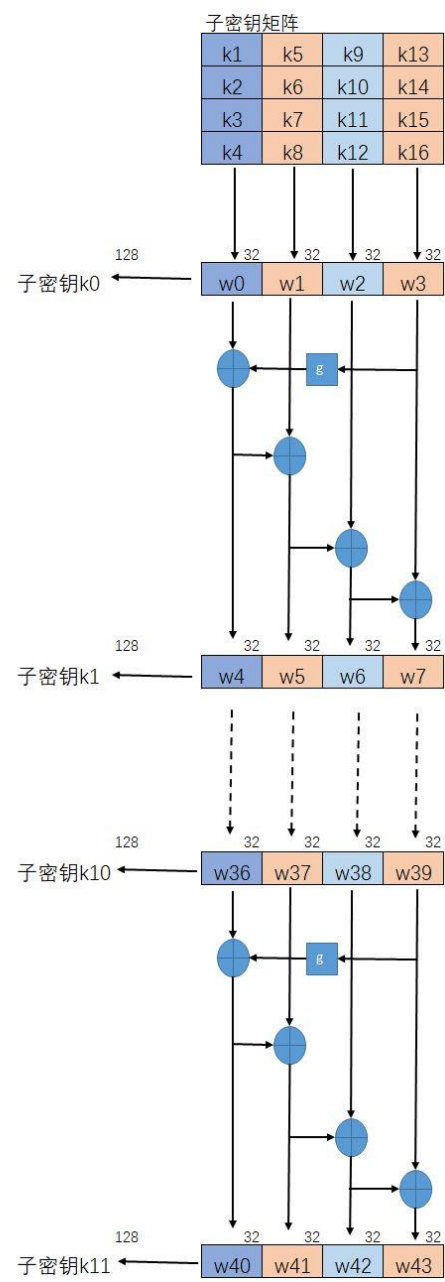
## 列混淆——MixColumn

混淆子层是 AES 算法中最为复杂的部分，属于扩散层，列混淆操作是 AES 算法中主要的扩散元素，它混淆了输入矩阵的每一列，使输入的每个字节都会影响到 4 个输出字节。行位移子层和列混淆子层的组合使得经过三轮处理以后，矩阵的每个字节都依赖于 16 个明文字节成可能

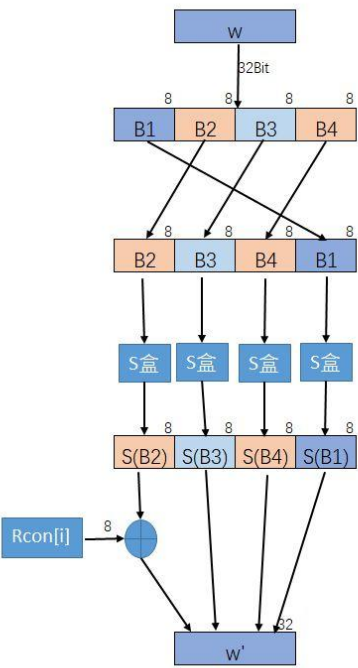




# AES 密钥生成



密钥生成大致流程







```

.text:00401E6E      push     offset aPleaseProvideT ; "Please provide the Flag:"
.text:00401E73      call     _printf
.text:00401E78      add     esp, 4
.text:00401E7B      lea     eax, [ebp+var_44]
.text:00401E7E      push     eax
.text:00401E7F      push     offset aS ; "%5"
.text:00401E84      call     _scanf
.text:00401E89      add     esp, 8
.text:00401E8C      push     20h ; int
.text:00401E8E      lea     ecx, [ebp+var_84]
.text:00401E94      push     ecx ; int
.text:00401E95      lea     edx, [ebp+var_44]
.text:00401E98      push     edx ; int
.text:00401E99      push     10h ; count
.text:00401E9B      lea     eax, [ebp+src]
.text:00401E9E      push     eax ; src
.text:00401E9F      call     j__aesEncrypt
.text:00401EA4      add     esp, 14h
.text:00401EA7      mov     [ebp+var_4], 0
.text:00401EAE      jmp     short loc_401EB9

```

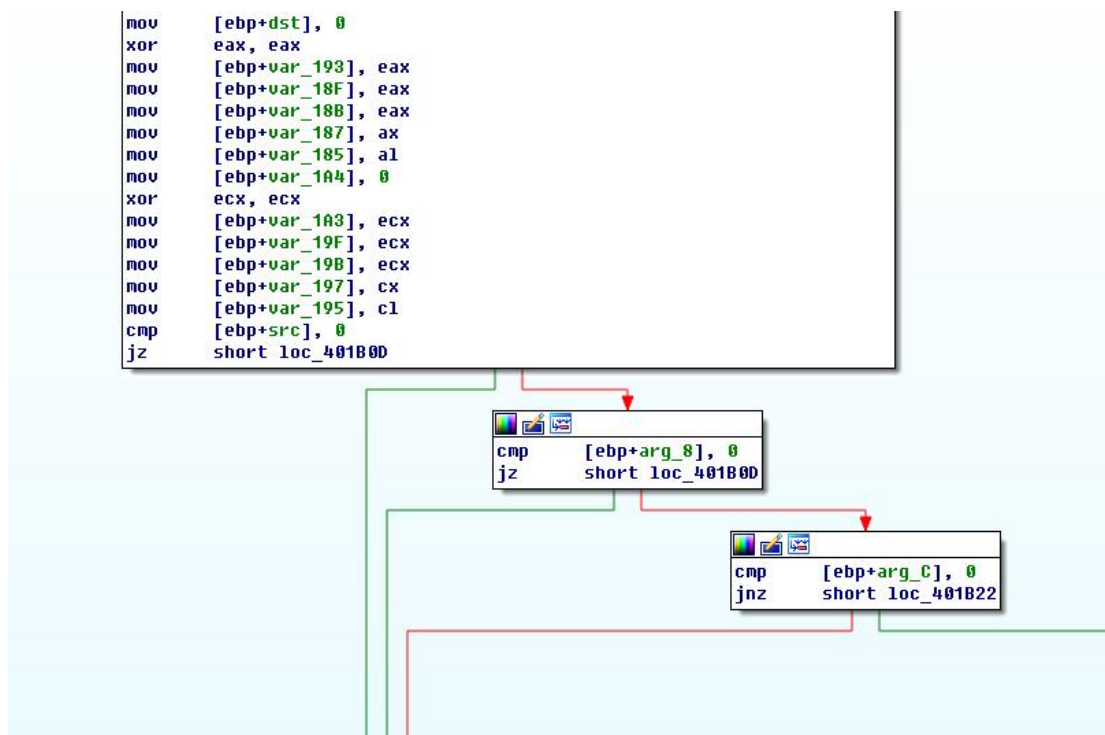
进入main函数，首先就是push字符串” Please provide the flag”，调用Printf打印，然后ESP（栈顶）+4，push [ebp+var\_44]，并调用scanf函数并把字符串放入[ebp+var\_44]，lea然后将[ebp+var\_44]中的值赋给edx，lea然后将[ebp+var\_84]中的值赋给ecx，其中ebp+src存放的是“1234567890123456”字符串，然后调用aesEncrypt函数对输入的字符串加密。

## AesEncrypt 函数

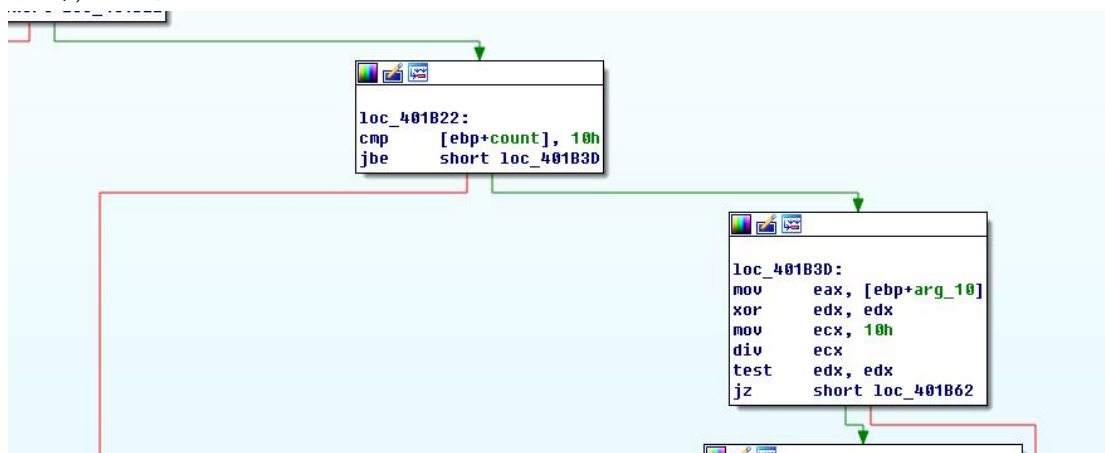
```
var_17B= dword ptr -17Bh
var_177= word ptr -177h
var_175= byte ptr -175h
var_174= dword ptr -174h
var_170= dword ptr -170h
var_16C= byte ptr -16Ch
var_8= dword ptr -8
var_4= dword ptr -4
src= dword ptr 8
count= dword ptr 0Ch
arg_8= dword ptr 10h
arg_C= dword ptr 14h
arg_10= dword ptr 18h

push    ebp
mov     ebp, esp
sub     esp, 1E4h
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_1E4]
mov     ecx, 79h
mov     eax, 0CCCCCCCCh
rep stosd
mov     eax, [ebp+arg_C]
mov     [ebp+var_170], eax
lea     ecx, [ebp+var_16C]
mov     [ebp+var_174], ecx
mov     [ebp+var_184], 0
xor     edx, edx
mov     [ebp+var_183], edx
mov     [ebp+var_17F], edx
mov     [ebp+var_17B], edx
mov     [ebp+var_177], dx
mov     [ebp+var_175], dl
mov     [ebp+dst], 0
xor     eax, eax
mov     [ebp+var_193], eax
mov     [ebp+var_18F], eax
mov     [ebp+var_18B], eax
mov     [ebp+var_187], ax
mov     [ebp+var_185], al
mov     [ebp+var_1A4], 0
xor     ecx, ecx
mov     [ebp+var_1A3], ecx
mov     [ebp+var_19F], ecx
mov     [ebp+var_19B], ecx
mov     [ebp+var_197], cx
mov     [ebp+var_195], cl
cmp     [ebp+src], 0
jz      short loc_401B0D
```

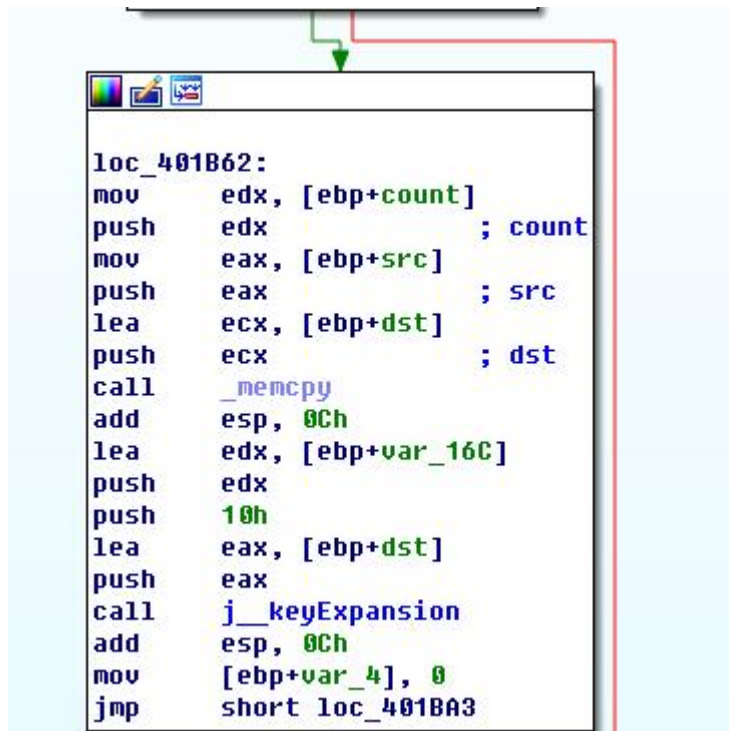
首先将参数[ebp+arg\_c], 即[ebp+var\_84]放入[ebp+var\_170], 然后将ebp+var\_16C 的值传给[ebp+var\_174], 然后进行一系列置零操作, 给[ebp+dst]赋值为 0, 再进行一系列赋 0 操作, 然后将[ebp+src]与 0 比较, 明显不同, 跳转只 loc\_401b0d。



将 0 与 [ebp+arg\_8] 比较，不相同，不跳转，接着比较 [ebp+arg\_c] 与 0，不相同，跳转

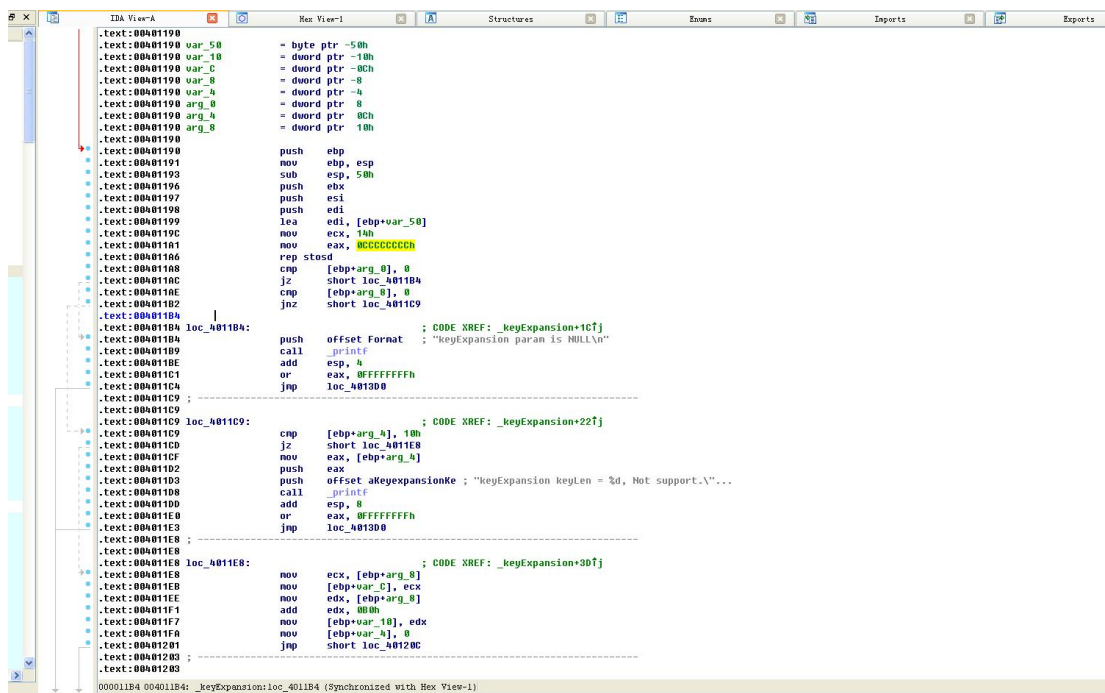


将 count 与 10h 比较，（count=16，jbe 大于等于跳转），则跳转，然后为 eax 赋值 20h（因为 argc\_10=32），edx 置零，ecx 赋值 10h，对 eax 进行除法，余数为 0，edx=0，test 检查 edx 是否为 0，为 0 跳转

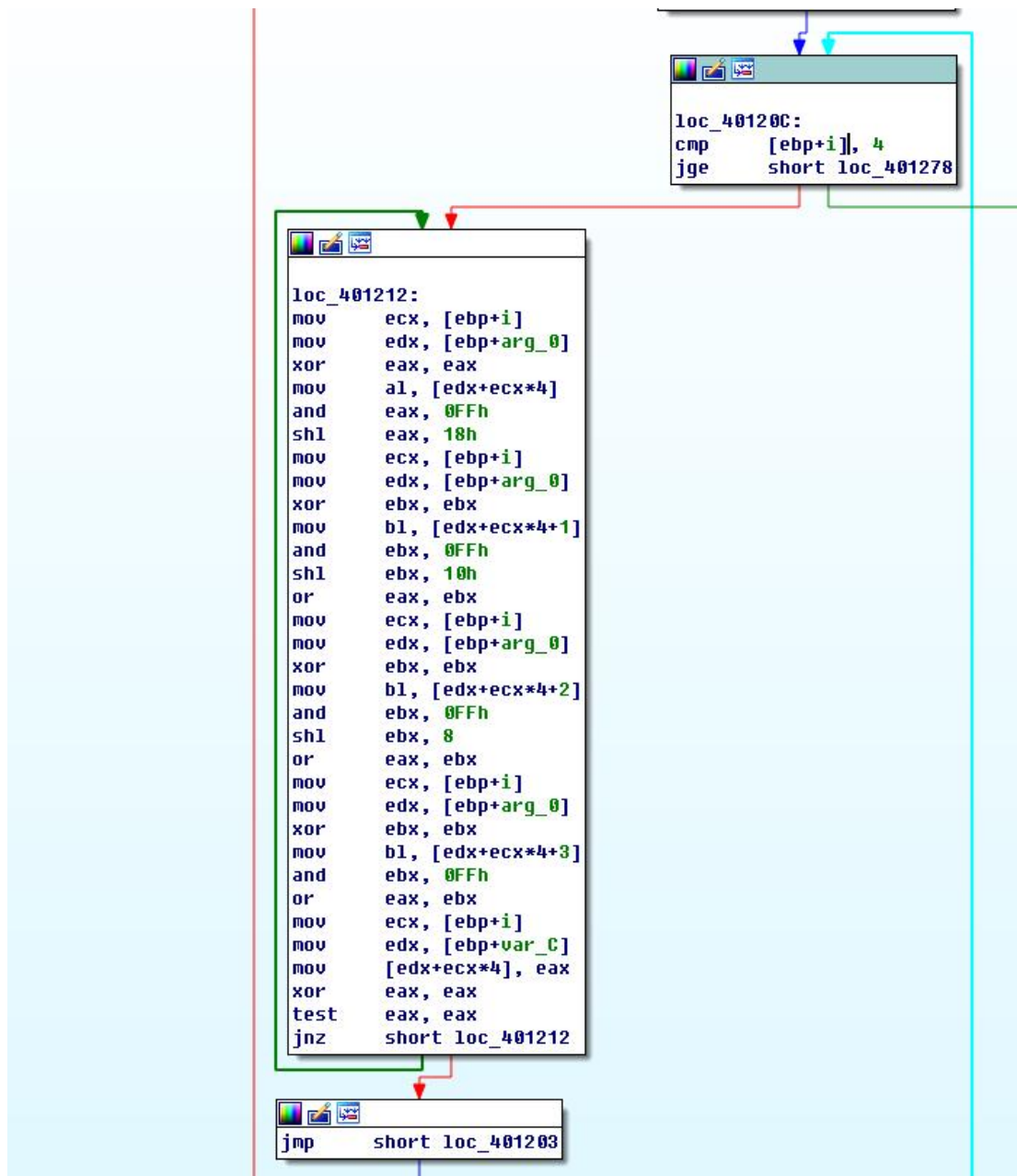


依次压入 count, src, dst, 调用 memcpy 函数, 将 src 复制到 [ebp+dst] 中。然后依次 push [ebp+var\_16c], 10h, [ebp+dst]。然后调用 keyExpansion 函数, 输出到 [ebp+var\_16c], 因为为密码编排, 所以参数应当有密钥, 所以猜测密钥为 dst, 即为 src。

## KeyExpansion



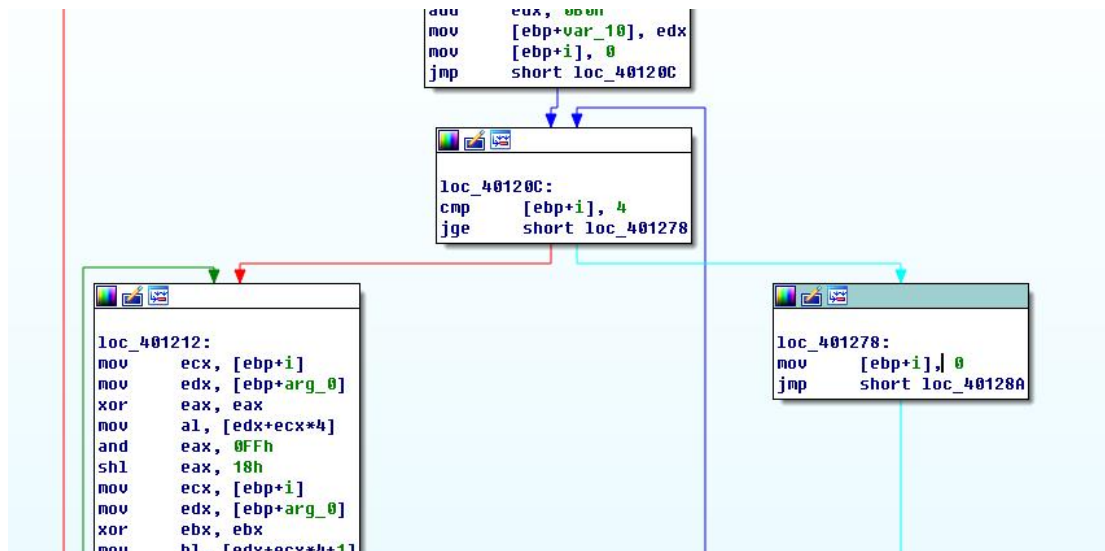
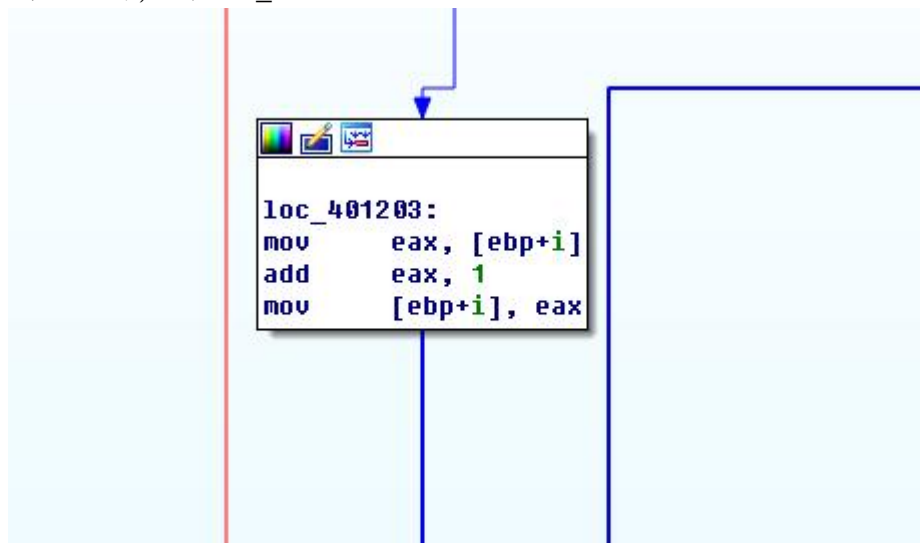
进入密钥生成函数，arg\_0 = dst, arg\_4 = 10h, arg\_8 = \_c16, 将 0 与 [ebp+arg\_0] 比较，不相同所以不跳转，再让 0 与 [ebp+arg\_8]，不相同，则跳转。然后让 10h 与 [ebp+arg\_4] 比较，相同，跳转。然后将 [ebp+arg\_8] 传入 [ebp+vac\_C]。接着把 [ebp+arg\_8]+176 后传入 [ebp+vac\_10]。猜测：176 等于 11\*16\*8bit，所以应该是放拓展后的密钥。然后给 [ebp+vac\_4] 赋值为 0，要进入一个循环。命名为 i。



首先将 [ebp+i] 的值传给 ecx，密钥传给 edx，eax 置零。然后将 [edx+i\*4] 的值传给 al，并将 eax 左移 24 位，然后将密钥的第 4\*i 字节放在 eax 的高位（处理 w4 的 g 函数），然后密钥拓展，因为密钥拓展为每 4 位字节，所以 ecx\*4。同理后边的将将密钥的第 4\*i+1 字节传给 bl，然后左移 16 位，然后 ebx 与 eax 或操作，得到了对应的子密钥第 i 列的前十六位，然后剩下的 16 位进行类似操作。最后第 i 列的 32 位密钥保存在了 eax。然后通过将把内存地址为 ebx+i 处的数据赋给 ecx，将 [ebp+var\_c] 的值给 edx，再把 eax 就是第 i 列的 32 位密钥，



放在 $[edx+ecx*4]$ . 即将密钥放进 $[ebp+var\_c]$ 中。然后将  $eax$  置零。Test 为 0, 则不跳转, 到  $loc\_401203$ 。



循环四次得到密钥  $W$ ,  $W[0]-W[3]$ , 存放在  $[ebp+var\_c]$ 。然后跳出循环。然后进入  $loc\_401278$ 。



```

.text:00401278 loc_401278:                ; CODE XREF: _keyExpansion+80↑j
.text:00401278                mov     [ebp+i], 0
.text:0040127F                jmp     short loc_40128A
; -----
.text:00401281 loc_401281:                ; CODE XREF: _keyExpansion+106↑j
.text:00401281                mov     ecx, [ebp+i]
.text:00401284                add     ecx, 1
.text:00401287                mov     [ebp+i], ecx
.text:0040128A loc_40128A:                ; CODE XREF: _keyExpansion+EF↑j
.text:0040128A                cmp     [ebp+i], 0Ah
.text:0040128E                jge     loc_40136B
.text:00401294                mov     edx, [ebp+var_C]
.text:00401297                mov     eax, [edx+0Ch]
.text:0040129A                shr     eax, 10h
.text:0040129D                and     eax, 0FFh
.text:004012A2                xor     ecx, ecx
.text:004012A4                mov     cl, _S[eax]
.text:004012A8                shl     ecx, 18h
.text:004012AD                and     ecx, 0FF000000h
.text:004012B3                mov     edx, [ebp+var_C]
.text:004012B6                mov     eax, [edx+0Ch]

```

将[ebp+i]置0，进入 loc\_10128a，继续进入循环 10 次。

```

mov     edx, [ebp+var_C]
mov     eax, [edx+0Ch]
shr     eax, 10h
and     eax, 0FFh
xor     ecx, ecx
mov     cl, _S[eax]
shl     ecx, 18h
and     ecx, 0FF000000h
mov     edx, [ebp+var_C]
mov     eax, [edx+0Ch]
shr     eax, 8
and     eax, 0FFh
xor     edx, edx
mov     dl, _S[eax]
shl     edx, 10h
and     edx, 0FF0000h
xor     ecx, edx
mov     eax, [ebp+var_C]
mov     edx, [eax+0Ch]
and     edx, 0FFh
xor     eax, eax
mov     al, _S[edx]
shl     eax, 8
and     eax, 0FF00h
xor     ecx, eax
mov     edx, [ebp+var_C]
mov     eax, [edx+0Ch]
shr     eax, 18h
and     eax, 0FFh
xor     edx, edx
mov     dl, _S[eax]
and     edx, 0FFh
xor     ecx, edx
mov     eax, [ebp+var_C]
mov     edx, [eax]
xor     edx, ecx
mov     eax, [ebp+i]

```

（容易知道其实就是生成 10 个子密钥），首先我们得处理  $w[3+4k]$  的密钥。先将密钥  $w$  传给  $edx$ 。然后将  $[edx+0Ch]$  穿给  $eax$ ，其实就是把  $w[3]$ （4 个字节）传给  $eax$ 。然后向右移 16 位得到 00B1B2。在 S 盒中寻址，然后放入置零后的  $ecx$  的低八位，然后左移 24 位，将其让在  $ecx$  的最高八位。（即对  $W$  中的 B2 进行处理）。类似地后边对即对  $W$  中的 B3, B4, B1 进行处理），最后到  $xor\ ecx\ edx$ ，将  $g$  函数后的  $w$  值给  $ecx$ 。然后将  $[ebp+var\_c]$  传给  $edx$ 。并与  $ecx$  异或。其实就是  $K0$  与  $w$ （没有进行  $rcon$  的）进行异或。然后将  $[ebp+i]$  的值给  $eax$ 。

```

mov     eax, [ebp+i]
xor     edx, ds:rcon[eax*4]
mov     ecx, [ebp+var_c]
mov     [ecx+10h], edx
mov     edx, [ebp+var_c]
mov     eax, [ebp+var_c]
mov     ecx, [edx+4]
xor     ecx, [eax+10h]
mov     edx, [ebp+var_c]
mov     [edx+14h], ecx
mov     eax, [ebp+var_c]
mov     ecx, [ebp+var_c]
mov     edx, [eax+8]
xor     edx, [ecx+14h]
mov     eax, [ebp+var_c]
mov     [eax+18h], edx
mov     ecx, [ebp+var_c]
mov     edx, [ebp+var_c]
mov     eax, [ecx+0Ch]
xor     eax, [edx+18h]
mov     ecx, [ebp+var_c]
mov     [ecx+1Ch], eax
mov     edx, [ebp+var_c]
add     edx, 10h
mov     [ebp+var_c], edx
jmp     loc_401281

```

然后将  $edx$  与  $rcon(eax*4)$  异或。实际上就是实现了补充完成了  $rcon$  的异或，即：先  $W[j-1] \oplus W[j-4] \oplus Rcon[j/4]$  而不是  $W[j-1] \oplus Rcon[j/4] \oplus W[j-4]$ 。

然而结果是一样的。然后得到了  $w[j]$ ，或者是  $w[4*i]$ 。然后将  $edx$  传入

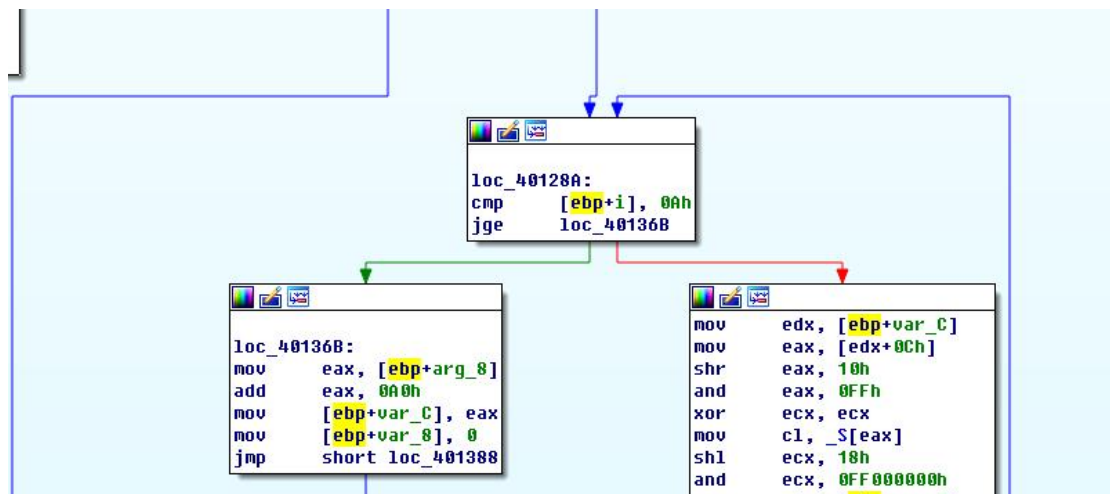
$[edx+var\_c+16]$ （开始存放  $k1$  的位置）。然后开始处理  $w[1+4*i]$ ， $w[2+4*i]$ ，

$w[3+4*i]$ 。与  $w[4i]$  进行类似的操作，但不进行异或  $rcon[]$ 。依次将结果放入

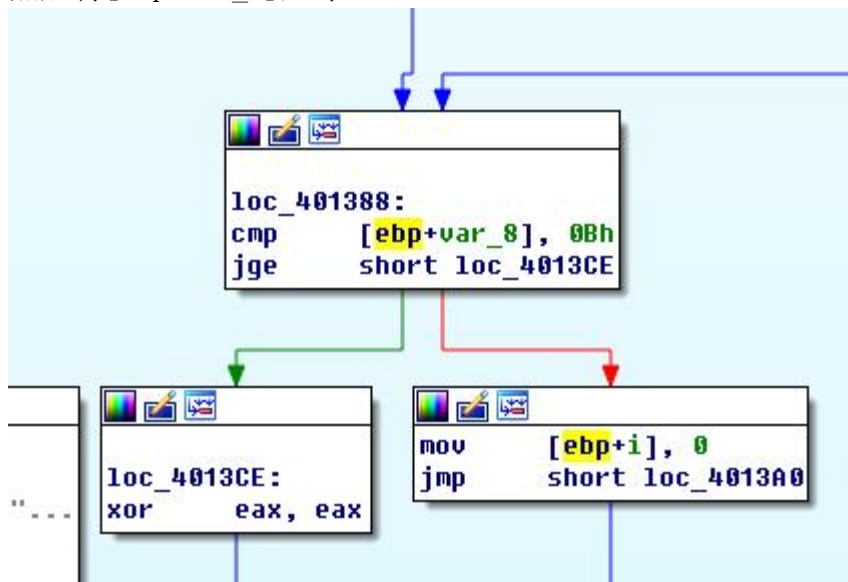
$[edx+var\_c+20]$ ， $[edx+var\_c+24]$ ， $[edx+var\_c+28]$ ，接着  $[edx+var\_c]+16$ ，进

入下一轮循环生成下一个  $ki$ 。循环执行 10 次。最后得到所有密钥

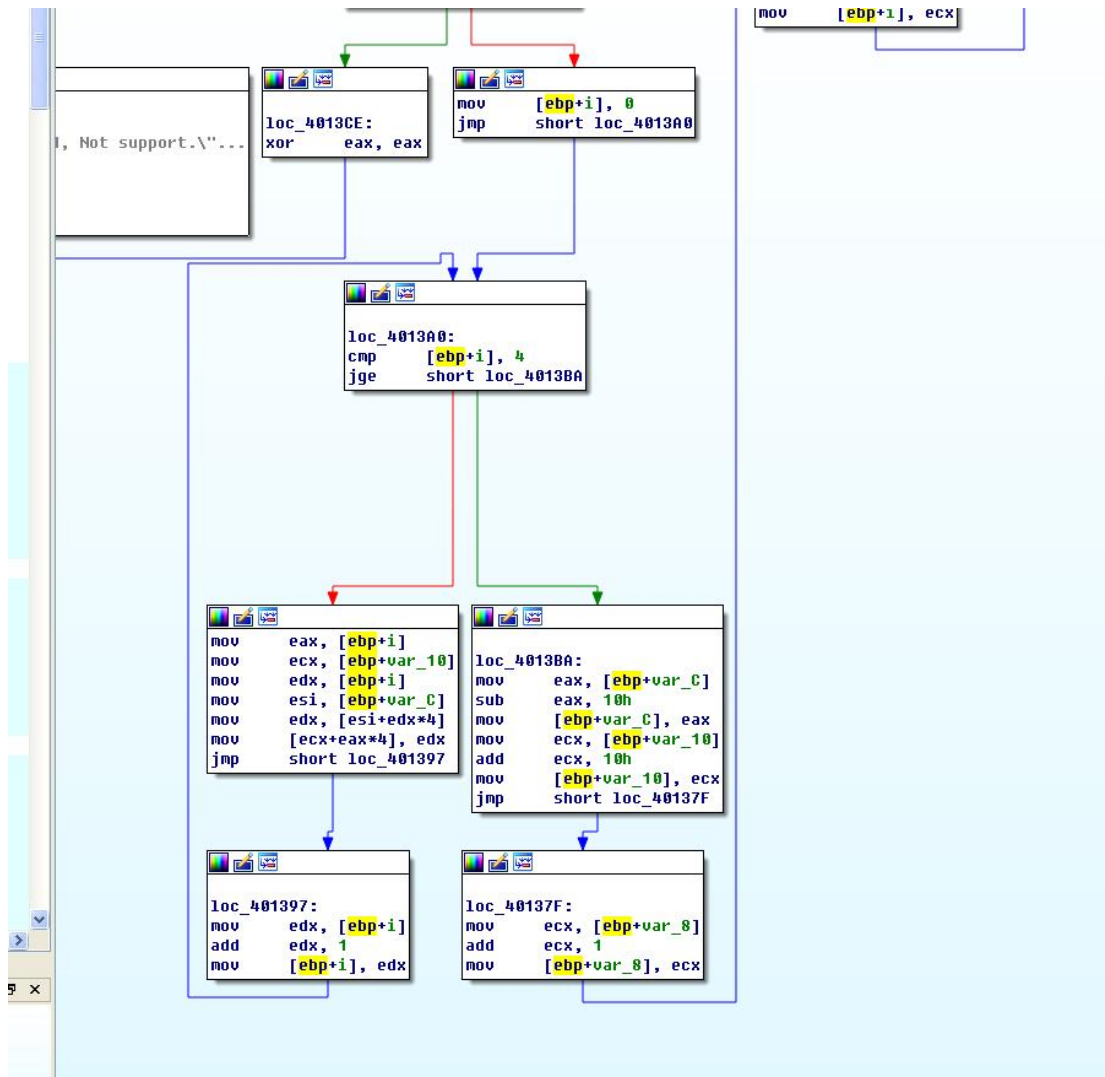
$k0-k11[w[0]-w[43]]$ 。



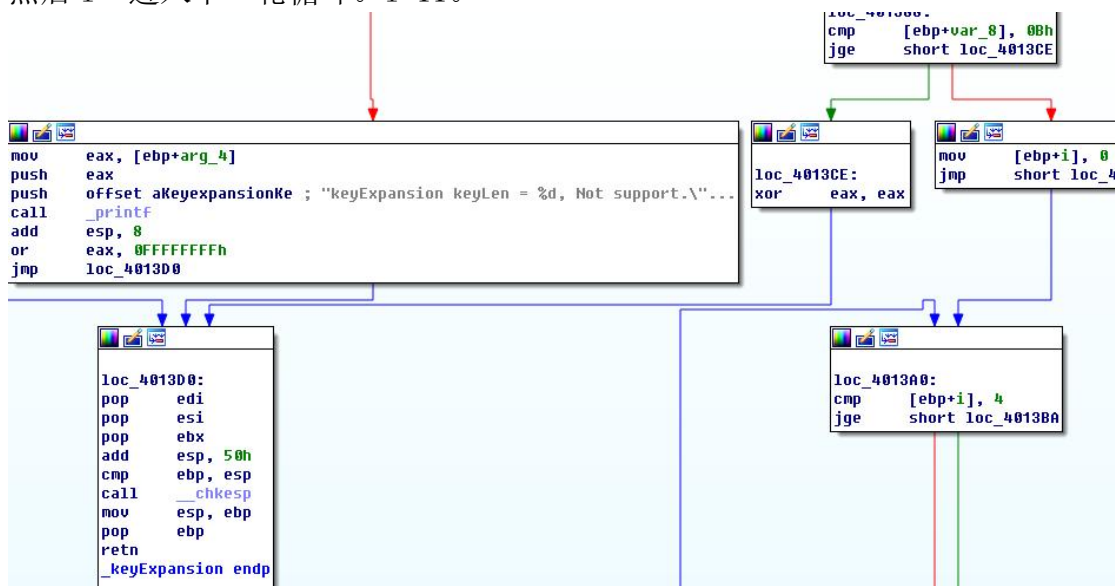
然后跳出循环, 到达 loc\_40136B, 然后将 [ebp+arg\_8+160] 传给 [ebp+var\_C], 然后将 [ebp+var\_8] 置零,



很明显 [ebp+var\_8]=0, 不跳转, 然后将 [ebp+i] 的值赋 0,



又进入循环。从 K10 存放的位置开始，通过循环把子密钥 4 个字节流复制到 [ebp+var\_10] 中，循环四次后到 loc\_4013BA 中，然后分别会 [ebp+var\_10]+16 与 [ebp+var\_C]-16, 实现，在下一轮的外层循环将 Ki-1 复制到 [ebp+var\_10] 中。然后 i++ 进入下一轮循环。i=11。



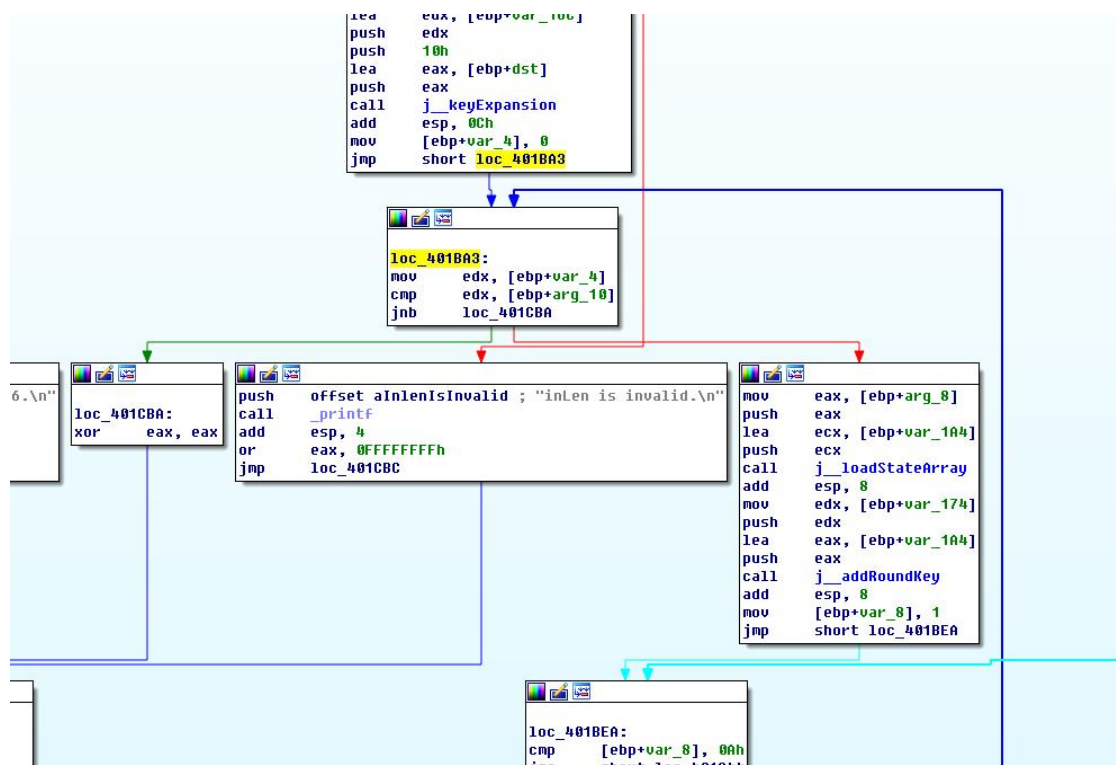
```

loc_401B62:
mov     edx, [ebp+count]
push    edx                ; count
mov     eax, [ebp+src]
push    eax                ; src
lea     ecx, [ebp+dst]
push    ecx                ; dst
call    _memcpy
add     esp, 0Ch
lea     edx, [ebp+var_16C]
push    edx
push    10h
lea     eax, [ebp+dst]
push    eax
call    j__keyExpansion
add     esp, 0Ch
mov     [ebp+var_4], 0
jmp     short loc_401BA3

```

然后进入到 loc\_4013C 中，将 eax 置零。然后进入 loc\_013D0。结束 keyExansion。返回到原函数。所以子密钥 K0-K11 存放在 [ebp+var\_10]，而 [ebp+var\_10] = [ebp+arg\_8]，即为调用函数前的 [ebp+var\_16]。

## LoadStateArray



The screenshot displays the HxD Hex Editor interface with the following components:

- Top Panel:** Shows the file path `12A View-A` and the current address `000010A0`.
- Left Panel:** Displays the assembly code for the current address:
 

```

      mov     edi, [ebp+var_A8]
      mov     ecx, 12h
      mov     eax, 00000000h
      rep stqsd
      mov     [ebp+1], 0
      jmp     short loc_A0109A
      
```
- Right Panel:** Shows the control flow graph (CFG) with the following blocks:
  - loc\_A0109A:**

```

      cmp     [ebp+1], 4
      jge     short loc_A01099
      
```
  - loc\_A01099:**

```

      mov     [ebp+1], 0
      jmp     short loc_A010B2
      
```
  - loc\_A010B2:**

```

      xor     eax, eax
      pop     edi
      pop     esi
      pop     ebx
      mov     esp, ebp
      pop     ebp
      retn
      _loadStateArray endp
      
```
  - loc\_A010B7:**

```

      cmp     [ebp+1], 4
      jge     short loc_A010B07
      
```
  - loc\_A010B07:**

```

      jmp     short loc_A01091
      
```
  - loc\_A01091:**

```

      mov     eax, [ebp+1]
      add     eax, 1
      mov     [ebp+1], eax
      
```
  - loc\_A010B9:**

```

      mov     ecx, [ebp+1]
      add     ecx, 1
      mov     [ebp+1], ecx
      
```
  - loc\_A010B5:**

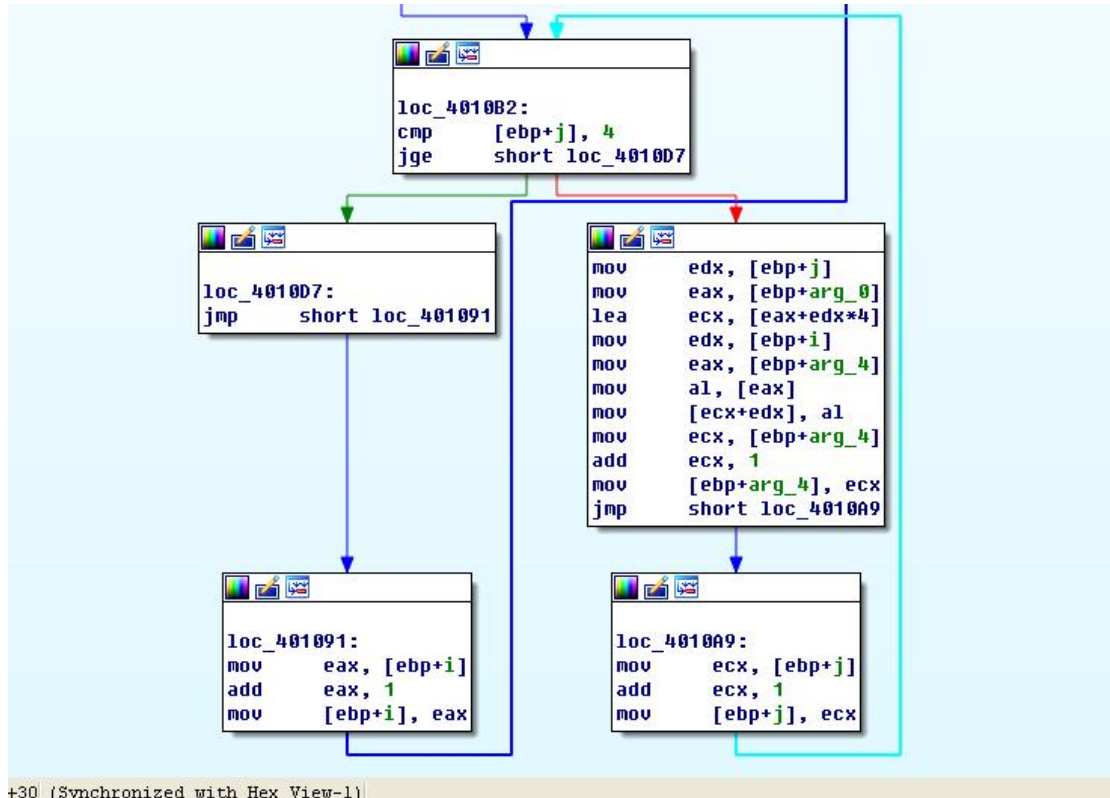
```

      mov     edx, [ebp+1]
      mov     eax, [ebp+arg_0]
      lea     ecx, [eax+edx*4]
      mov     edx, [ebp+1]
      mov     eax, [ebp+arg_4]
      mov     al, [eax]
      mov     [ecx*edx], al
      mov     ecx, [ebp+arg_4]
      add     ecx, 1
      mov     [ebp+arg_4], ecx
      jmp     short loc_A010B9
      
```

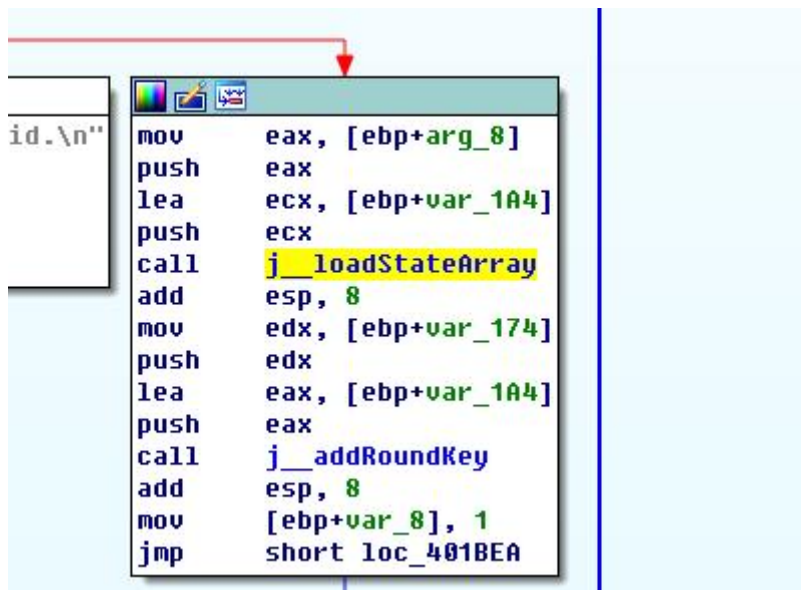
The status bar at the bottom indicates the current address: `00.00% (-572,303) (891,272) 000010A0 00A010A0: _loadStateArray+30 (Synchronized with Hex View-1)`.

i, j 被置零。主要循环内容在以下图中





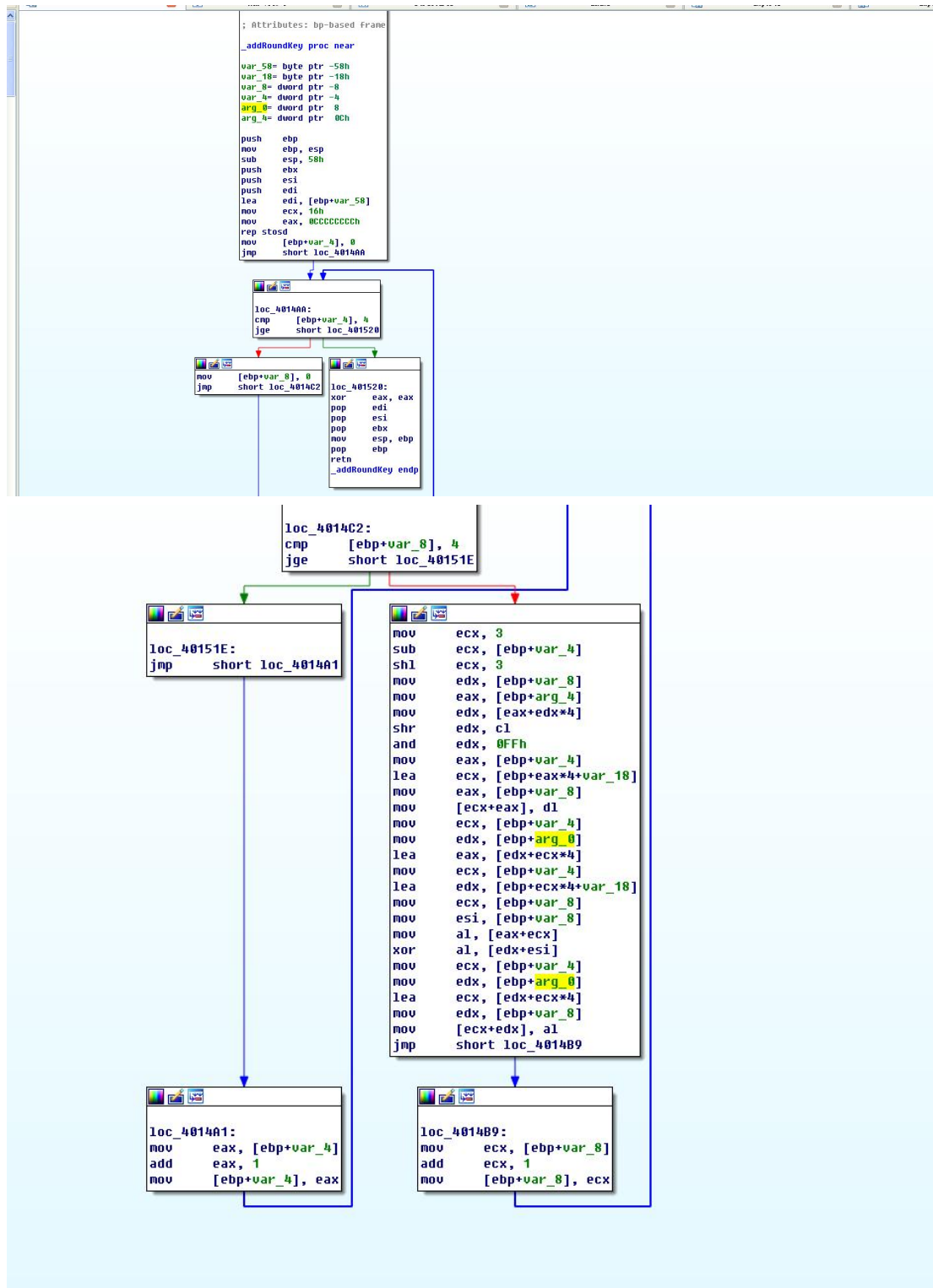
将 $[eax+edx*4]$ 的地址传给 ecx, 即为 $[ebp+arg\_0+j*4]$ 的地址传给 ecx, i 的值赋给 edx, 对 $[ebp+arg\_4]$ 自加 1, 然后, ecx 指向下一个字节。然后将 ecx 的值给 $[ebp+arg\_4]$ 然后依次进行内循环, 外循环。类似的。可以看出没有对明文进行多少操作。只是将明文分组一个个字节的分别放入 $[ebp+arg\_0+j*4+i]$ 中, 即: 将明文改成矩阵的形式存储。方便后面的运算。最后返回 $[ebp+arg\_4]$ , 即将分组后的明文存放在 $[ebp+var\_1A4]$ 中。循环完成后, 回到 main 函数。

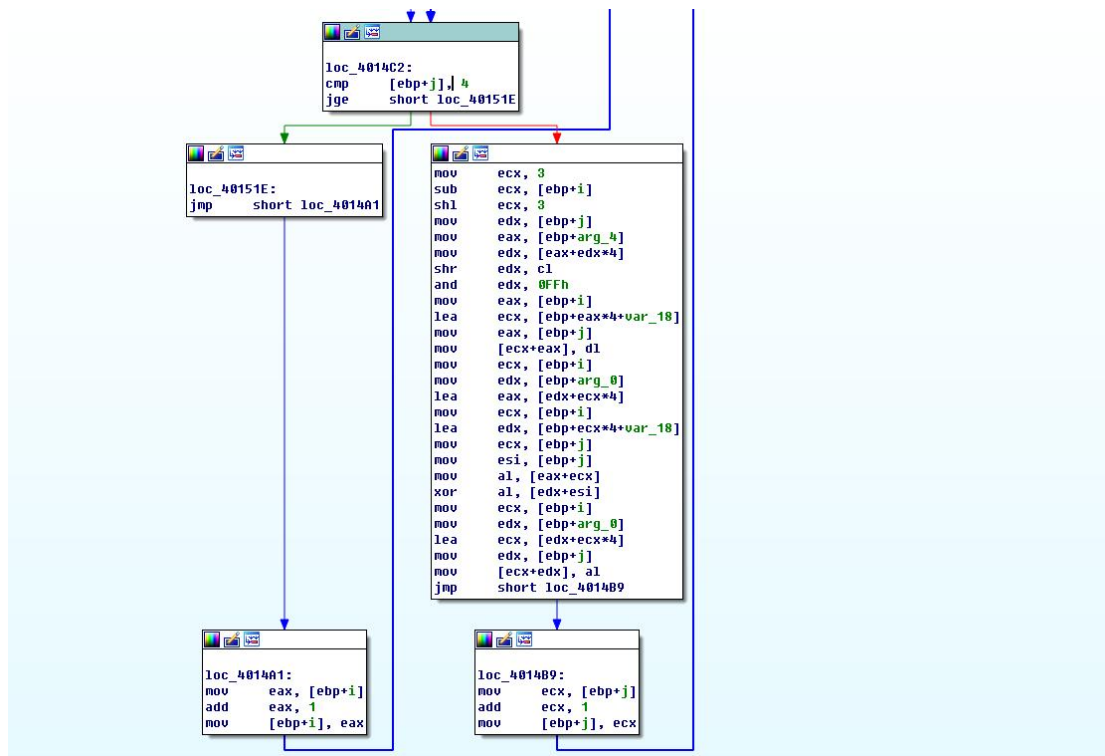


压入分组后明文的地址 $[ebp+var\_1A4]$ , 还有 $[ebp+var\_174]$ , 因为 $[ebp+var\_174]$ 存放的是 $[ebp+var\_16]$ 的地址, 并且 $[ebp+var\_16]$ 存放密钥所以, push 的还有密钥。进入 addRoundKey 函数。

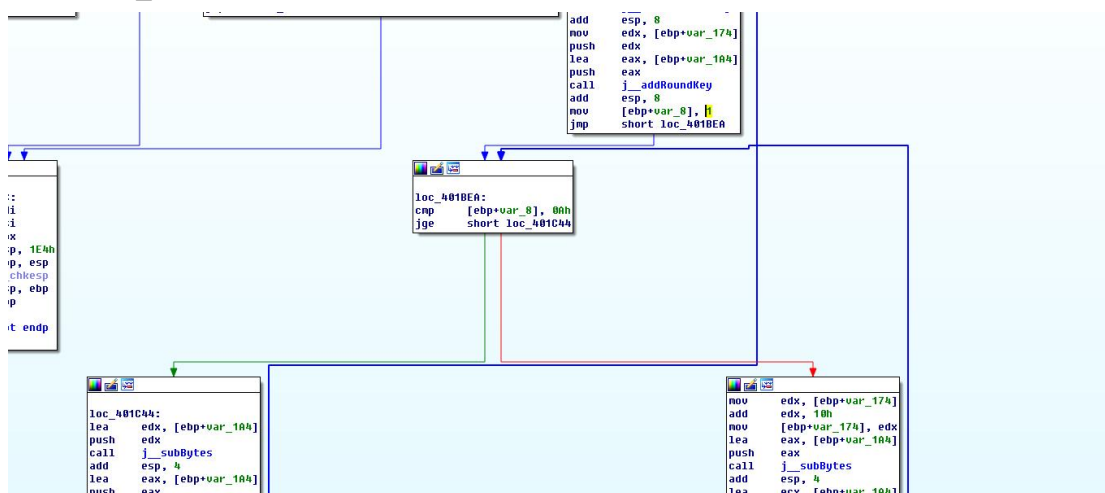


# AddRoundKey





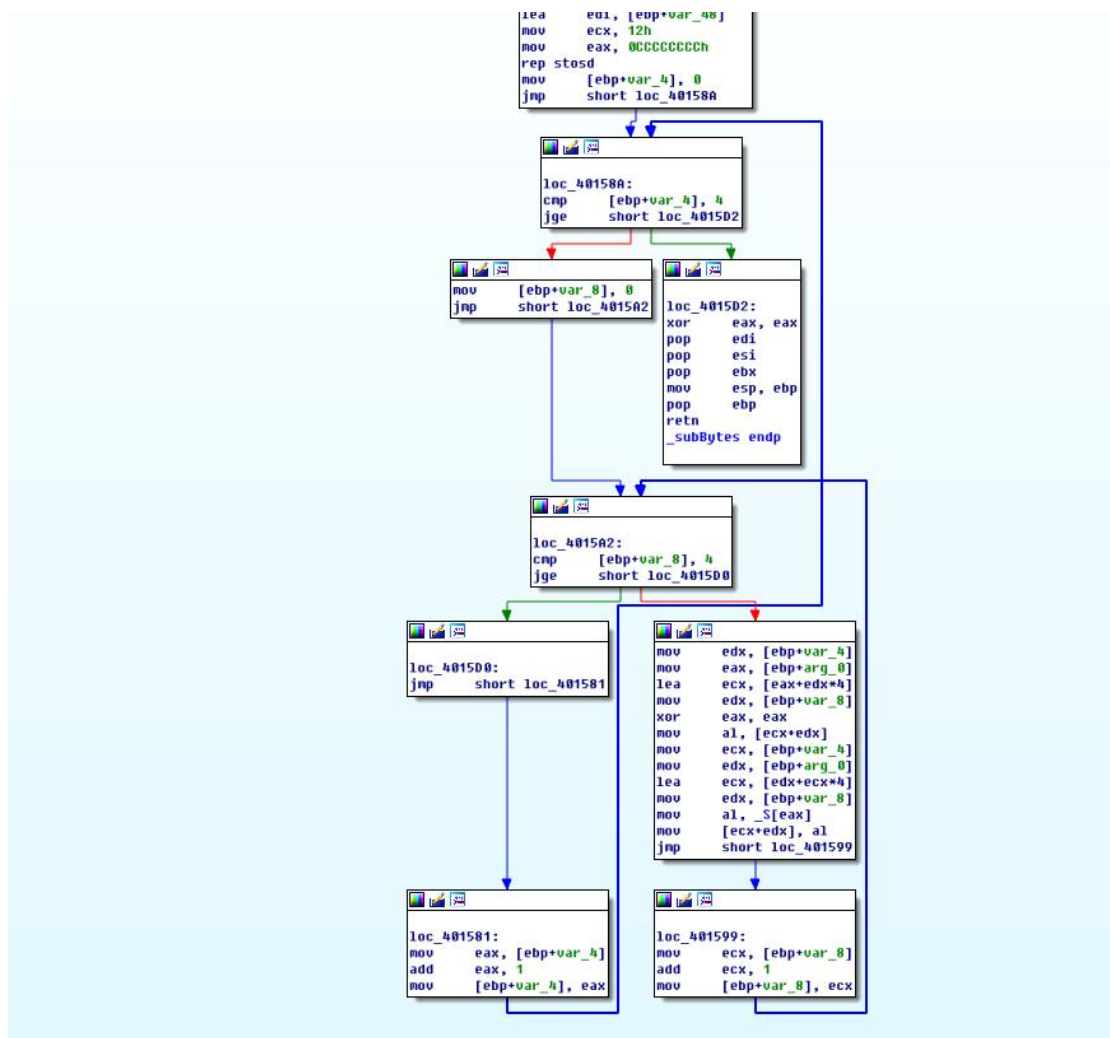
可以看到结构相同于上一个函数 `loadStateArray`，只是循环中的内容不一样。类似令 `var_4 = i, var_8 = j`。首先将密钥的第 `4*j` 的字节传入 `[ebp+var_18+j*4]` 中，完成对密钥的矩阵分组。然后取对应的明文中的第 `j+4*i` 字节放入 `eax` 与 `[ebp+var_18+j*4]` 进行异或，完成第 `i` 行第 `j` 个字节与子密钥 `K0` 的异或，将结果存放到 `[ebp+arg_0+j*4]`。然后依次进行剩下的异或操作，知道 `j = 4, i+1`，进行下一行的异或，直到 `i = 4, j = 4`。循环结束。完成了对明文的第一轮加密。然后加密后的第一轮的结果存放在 `[ebp+arg_0+4*4]` 中，即为 `[ebp+var_1A4]` 上。



返回 `main` 函数后，令 `[ebp+var_8] = 1`，从下一个框来看又是一个循环，容易看出，右边是中间的十次加密，左边是最后的一次加密。首先 `cmp [ebp+var_8], 0Ah` 不跳转。然后将 `[ebp+var_174] + 16` 传到 `[ebp+var_174]`（用

下一个子密钥)随后 push, 然后将[ebp+var\_1A4]的地址也 push。调用 subBytes。参数为 K1, 和加密一次后的密文。

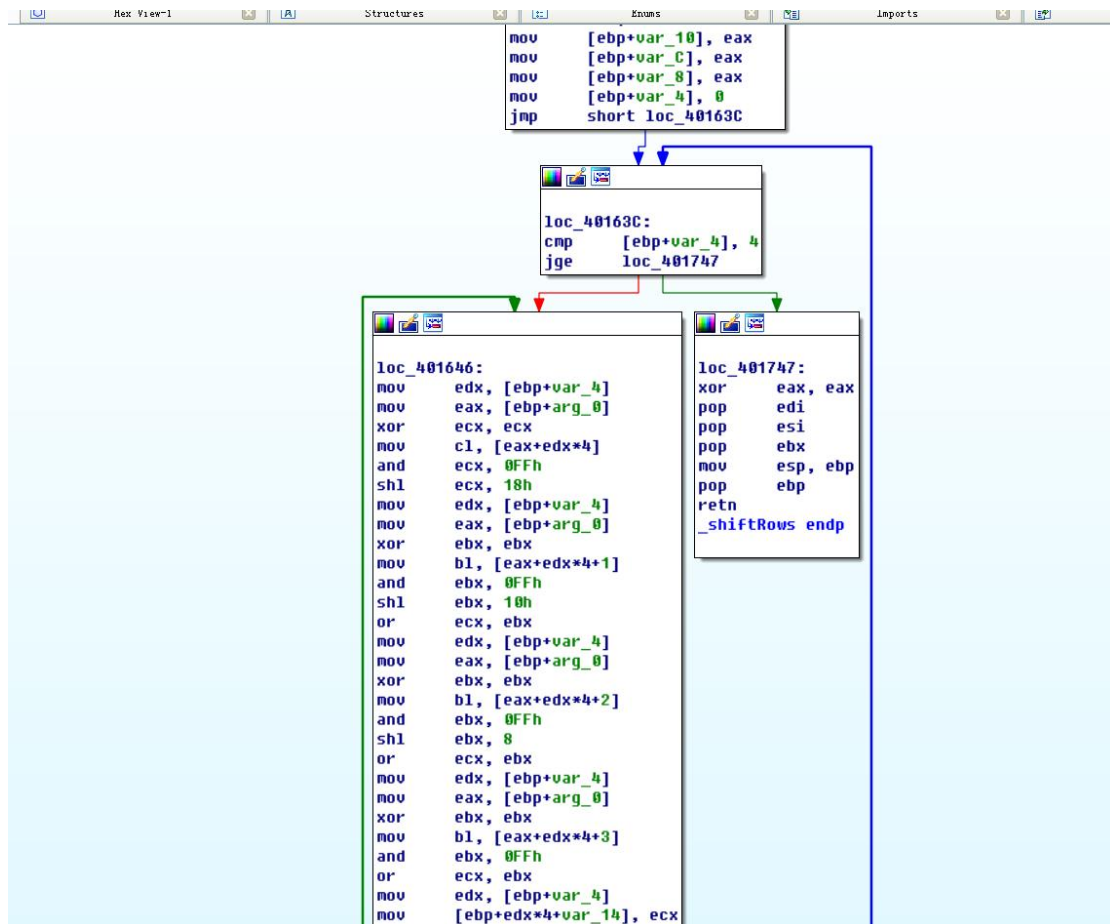
## SubBytes



从 subbyte 中可以看出将[ebp+var\_4]与[ebp+var\_8]作为循环, 一个内循环, 一个外循环。4\*4 共 16 次循环。再看到 S 盒, 应该就是字节代换层, 将 16 个字节 s 盒代换。

首先将[ebp+var\_4\*4+arg\_0]传给 ecx, 然后 j 传给 edx, 并将[ecx+edx]传给置零后的 eax 的低八位。其实就是明文数组下标 4\*i+j 的字节传给 eax。然后在 S 盒进行代换。并将它的值传回 eax 的低八位。然后再把 eax 的低八位传回明文数组下标 4\*i+j 的字节。然后内循环 var\_8 自加到 4, 然后外循环 r\_4 自加, var\_8 置零重新开始。直到 var\_4=4。然后跳出循环。结束 sub\_byte 函数。然后继续 push[ebp+var\_1A4]的地址, call shift\_rows 函数。

## Shift\_Rows



将[ebp+var\_4]置零，令 var\_4 为 i 以此为循环 4 次。看得出来应该是行移位对应的每行。

```

loc_401646:
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
xor     ecx, ecx
mov     cl, [eax+edx*4]
and     ecx, 0FFh
shl     ecx, 18h
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
xor     ebx, ebx
mov     bl, [eax+edx*4+1]
and     ebx, 0FFh
shl     ebx, 10h
or      ecx, ebx
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
xor     ebx, ebx
mov     bl, [eax+edx*4+2]
and     ebx, 0FFh
shl     ebx, 8
or      ecx, ebx
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
xor     ebx, ebx
mov     bl, [eax+edx*4+3]
and     ebx, 0FFh
or      ecx, ebx
mov     edx, [ebp+i]
mov     [ebp+edx*4+var_14], ecx
xor     eax, eax
test    eax, eax
jnz     short loc_401646

```

```

loc_401747:
xor     eax, eax
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
_shiftRows endp

```

```

mov     ecx, [ebp+i]
shl     ecx, 3
mov     edx, [ebp+i]
mov     eax, [ebp+edx*4+var_14]
shl     eax, cl
mov     ecx, [ebp+i]
shl     ecx, 3
mov     edx, 20h
sub     edx, ecx
mov     ecx, [ebp+i]
mov     esi, [ebp+ecx*4+var_14]
mov     ecx, edx
shr     esi, cl
or      eax, esi
mov     edx, [ebp+i]
mov     [ebp+edx*4+var_14], eax

```

首先就是一次取出 $[ebp+4*i+arg\_0+0]$ ,  $[ebp+4*i+arg\_0+1]$ ,  $[ebp+4*i+arg\_0+2]$ ,  $[ebp+4*i+arg\_0+3]$ 令他们为 abcd, 以第一行的移位为例。通过左移, 以及异或的操作, 把这 4 个字节放入 ecx (abcd) 中。然后将 ecx 放回 $[ebp+4*i+var\_14]$ 中存为 (dcba) 的形式。分别将 $[ebp+4*i+var\_14]$ 即为 dcba, 分别左移  $8*i$  位, 右移  $8*(4-i)$ 。然后异或得到对应的式子。 $i=0$ , 为 abcd,  $i=1$ , 为 bcda 等等类似。

```

loc_4016D9:
mov     eax, [ebp+i]
mov     ecx, [ebp+eax*4+var_14]
shr     ecx, 18h
and     ecx, 0FFh
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
mov     [eax+edx*4], cl
mov     ecx, [ebp+i]
mov     edx, [ebp+ecx*4+var_14]
shr     edx, 10h
and     edx, 0FFh
mov     eax, [ebp+i]
mov     ecx, [ebp+arg_0]
mov     [ecx+eax*4+1], dl
mov     edx, [ebp+i]
mov     eax, [ebp+edx*4+var_14]
shr     eax, 8
and     eax, 0FFh
mov     ecx, [ebp+i]
mov     edx, [ebp+arg_0]
mov     [edx+ecx*4+2], al
mov     eax, [ebp+i]
mov     ecx, [ebp+eax*4+var_14]
and     ecx, 0FFh
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
mov     [eax+edx*4+3], cl
xor     ecx, ecx
test    ecx, ecx
jnz     short loc_4016D9

```

然后分别右移 24, 16, 8, 0 位分别放入对应的  $[ebp+arg\_0+i*4+0]$ 、 $[ebp+arg\_0+i*4+1]$ 、 $[ebp+arg\_0+i*4+2]$ 、 $[ebp+arg\_0+i*4+3]$  的位置中。然后 test ecx 是否置零。至此字节代换层完成字节代换，实现对密文的对应字节代换。返回函数。继续 Push[ebp+var\_1A4]。Call mixColumns 函数。

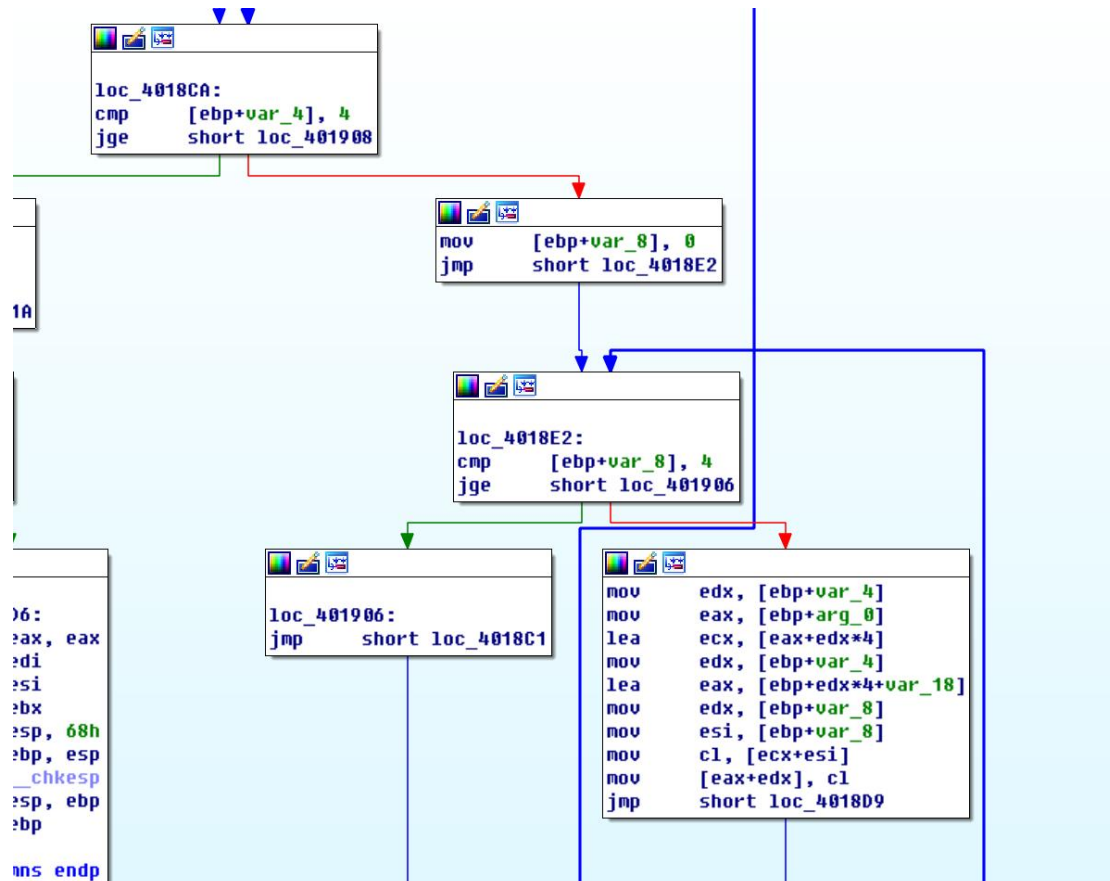


## MixColumns

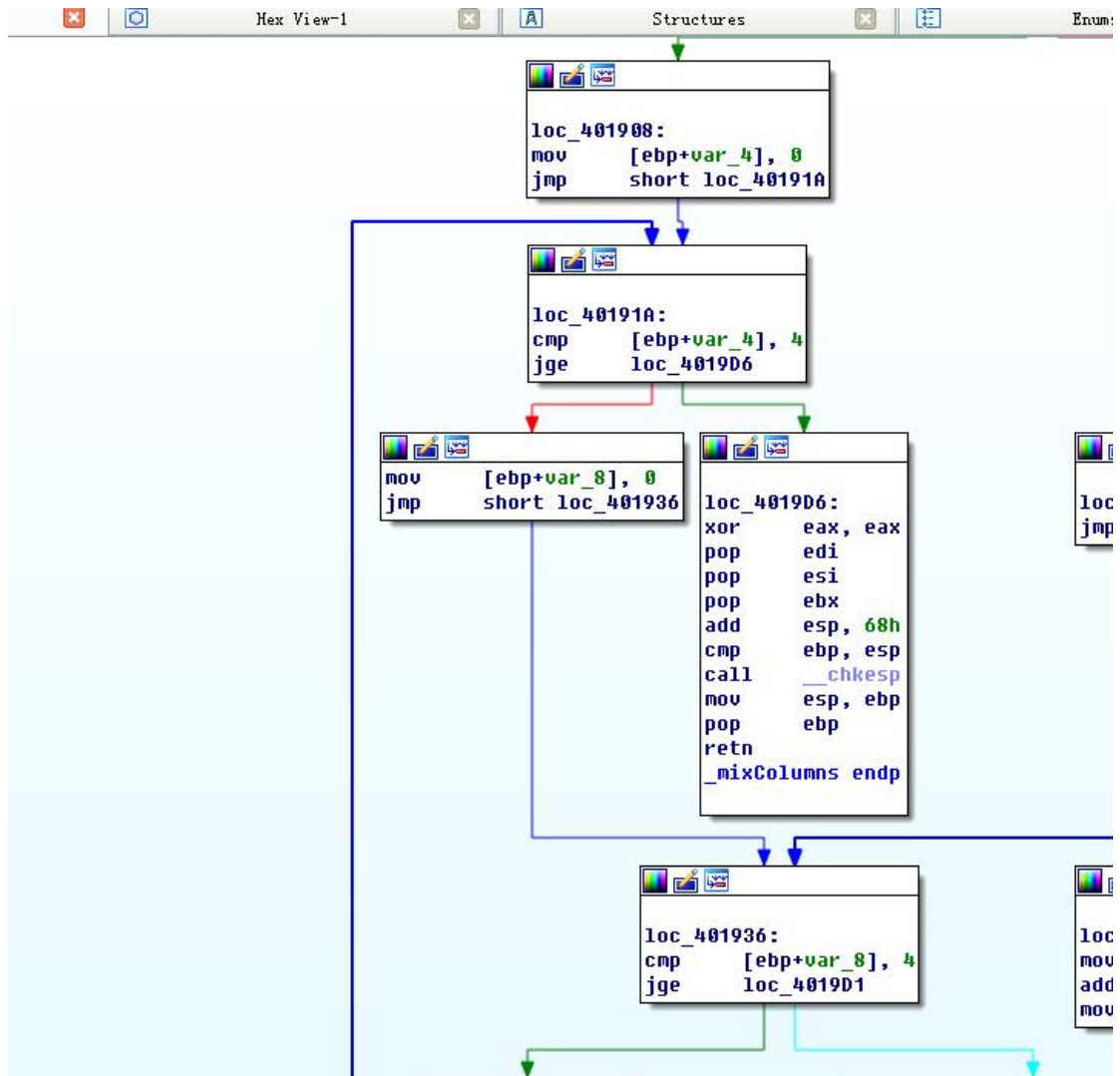
```
mov     eax, 00000000h
rep stosd
mov     [ebp+var_28], 2
mov     [ebp+var_27], 3
mov     [ebp+var_26], 1
mov     [ebp+var_25], 1
mov     [ebp+var_24], 1
mov     [ebp+var_23], 2
mov     [ebp+var_22], 3
mov     [ebp+var_21], 1
mov     [ebp+var_20], 1
mov     [ebp+var_1F], 1
mov     [ebp+var_1E], 2
mov     [ebp+var_1D], 3
mov     [ebp+var_1C], 3
mov     [ebp+var_1B], 1
mov     [ebp+var_1A], 1
mov     [ebp+var_19], 2
mov     [ebp+var_4], 0
jmp     short loc_4018CA
```

看到 0、1、2、3。就能猜到是列混淆矩阵。





Cmp, jge 不符合条件，不跳转，然后进入到  $\text{var}_4 = i$ ,  $\text{var}_8 = j$  的一个双循环。外为  $i$ , 内为  $j$ 。将  $[\text{ebp}+4i+j]$  字节依次通过循环传入  $[\text{ebp}+4i+j+\text{var}_{18}]$  中。然后循环结束。Jge 跳转到  $\text{loc}_{4011908}$  中。



看的出来，又是以  $\text{var}_4 = i$ ， $\text{var}_8 = j$  的一个双循环。外为  $i$ ，内为  $j$ 。为  $4*4$  次循环。

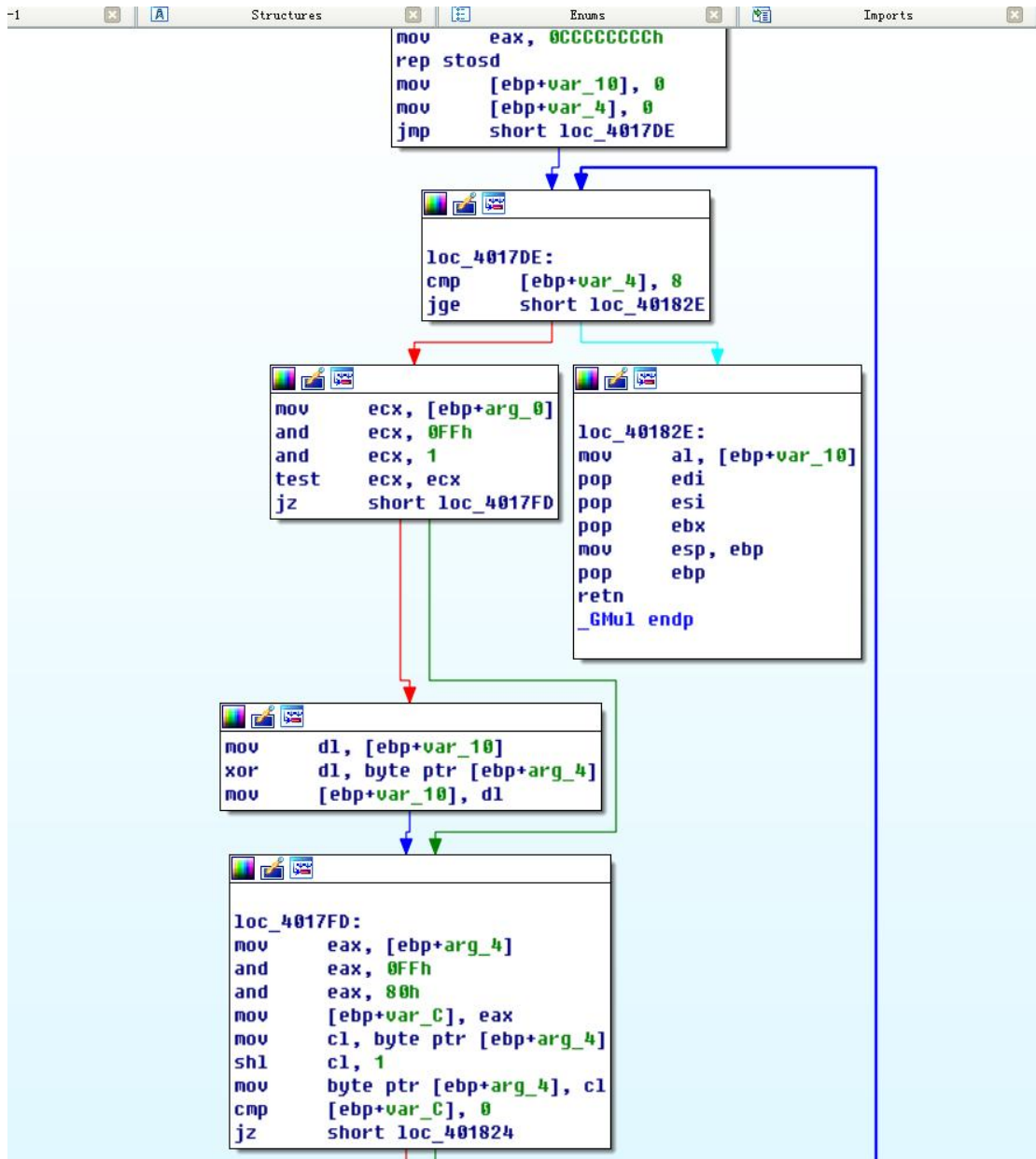
Structures

Enums

```
mov     ecx, [ebp+j]
mov     dl, [ebp+ecx+var_18]
push    edx
mov     eax, [ebp+i]
mov     cl, [ebp+eax*4+var_28]
push    ecx
call    j__GMul
add     esp, 8
mov     bl, al
and     ebx, 0FFh
mov     edx, [ebp+j]
mov     al, [ebp+edx+var_14]
push    eax
mov     ecx, [ebp+i]
mov     dl, [ebp+ecx*4+var_27]
push    edx
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     eax, [ebp+j]
mov     cl, [ebp+eax+var_10]
push    ecx
mov     edx, [ebp+i]
mov     al, [ebp+edx*4+var_26]
push    eax
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     ecx, [ebp+j]
mov     dl, [ebp+ecx+var_C]
push    edx
mov     eax, [ebp+i]
mov     cl, [ebp+eax*4+var_25]
push    ecx
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
lea     ecx, [eax+edx*4]
mov     edx, [ebp+j]
mov     [ecx+edx], bl
jmp     loc_40192D
```

然后就是 `push[ebp+j+var_18]` 就是 `push[ebp+j+arg_0]` 就是待列混淆的数的第 `j` 的字节。还有 `push` 矩阵的第 `i` 行。然后调用 `GMul` 函数。

## GMul



可知存在一个循环为 `var_4=0`，以此为 8 次循环。`var_10 = 0` 猜测出，应该是存放矩阵乘法的结果。`Cmp` 不跳转。验证矩阵的明文的第 `j` 列是否为全 0。若为 0，则跳转到 `loc_4017FD` 中。不为 0 则不跳转。将传入的字节与 `[ebp+var_10]` 异或并把结果存入 `[ebp+var_10]` 中。然后进入 `loc_4017FD`。






×

Enums

×

Imports

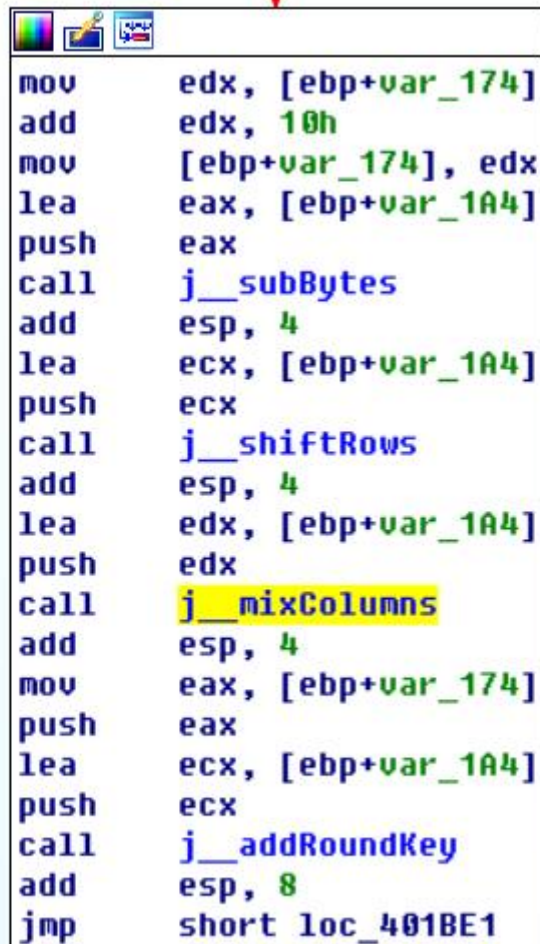


```
mov     ecx, [ebp+j]
mov     dl, [ebp+ecx+var_18]
push    edx
mov     eax, [ebp+i]
mov     cl, [ebp+eax*4+var_28]
push    ecx
call    j__GMul
add     esp, 8
mov     bl, al
and     ebx, 0FFh
mov     edx, [ebp+j]
mov     al, [ebp+edx+var_14]
push    eax
mov     ecx, [ebp+i]
mov     dl, [ebp+ecx*4+var_27]
push    edx
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     eax, [ebp+j]
mov     cl, [ebp+eax+var_10]
push    ecx
mov     edx, [ebp+i]
mov     al, [ebp+edx*4+var_26]
push    eax
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     ecx, [ebp+j]
mov     dl, [ebp+ecx+var_C]
push    edx
mov     eax, [ebp+i]
mov     cl, [ebp+eax*4+var_25]
push    ecx
call    j__GMul
add     esp, 8
and     eax, 0FFh
xor     ebx, eax
mov     edx, [ebp+i]
mov     eax, [ebp+arg_0]
lea     ecx, [eax+edx*4]
mov     edx, [ebp+j]
mov     [ecx+edx], bl
jmp     loc_40192D
```



返回 mixColumns, 后续继续对明文的第 i 列的第 j 行字节与矩阵的第 i 行第 j 列元素分别调用 GMul 函数三次。并将所有的结果异或。就得到了列混淆的第 Aji 的元素。然后存放在 [ebp+arg\_0+4\*i+j] 的。共循环 16 次。然后得到了所有的 A 元素。

至此 mixColumns 函数结束。



```
mov     edx, [ebp+var_174]
add     edx, 10h
mov     [ebp+var_174], edx
lea     eax, [ebp+var_1A4]
push    eax
call    j__subBytes
add     esp, 4
lea     ecx, [ebp+var_1A4]
push    ecx
call    j__shiftRows
add     esp, 4
lea     edx, [ebp+var_1A4]
push    edx
call    j__mixColumns
add     esp, 4
mov     eax, [ebp+var_174]
push    eax
lea     ecx, [ebp+var_1A4]
push    ecx
call    j__addRoundKey
add     esp, 8
jmp     short loc_401BE1
```

然后继续调用 addRoundkey 函数加密。  
共计完成 10 轮 subByte-shiftRow-mixColumns-addRoundkey。



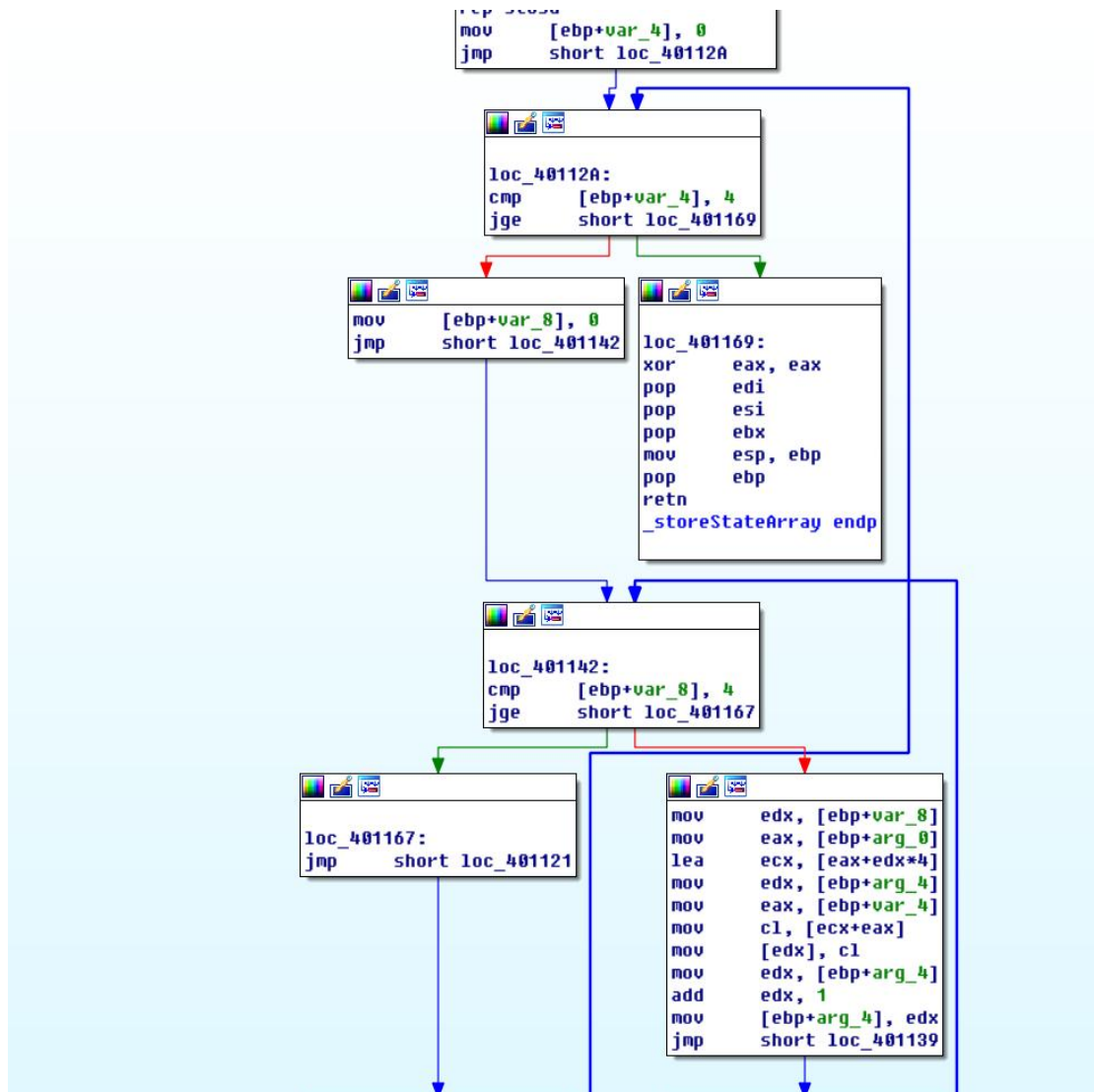
ot endp

```
loc_401C44:
lea     edx, [ebp+var_1A4]
push    edx
call    j__subBytes
add     esp, 4
lea     eax, [ebp+var_1A4]
push    eax
call    j__shiftRows
add     esp, 4
mov     ecx, [ebp+var_174]
add     ecx, 10h
push    ecx
lea     edx, [ebp+var_1A4]
push    edx
call    j__addRoundKey
add     esp, 8
mov     eax, [ebp+var_170]
push    eax
lea     ecx, [ebp+var_1A4]
push    ecx
call    j__storeStateArray
add     esp, 8
mov     edx, [ebp+var_170]
add     edx, 10h
mov     [ebp+var_170], edx
mov     eax, [ebp+arg_8]
add     eax, 10h
mov     [ebp+arg_8], eax
lea     ecx, [ebp+var_16C]
mov     [ebp+var_174], ecx
jmp     loc_401B9A
```

```
loc_401B9A:
mov     ecx, [ebp+var_4]
```

然后第十一次只有 subByte-shiftRow- addRoundkey。然后 push 加密后的密文 [ebp+var\_1A4] 和 [ebp+var\_170] 为函数返回的结果，call storeStateArray 函数。

## StoreStateArray



一看又是一个双层循环。`[ebp+var_4]`与`[ebp+var_8]`的双循环。令 `var_4 = i`, `var_8 = j`。将`[ebp+arg_0+4*j+i]`放入`[ebp+arg_4]`中。然后对`[ebp+arg_4]`自加。即将`[ebp+arg_4]`变为下一个待写入的地址。最后循环结束。完成了对密文的按字节存放。

```
lea    ecx, [ebp+var_1A4]
push   ecx
call   j__storeStateArray
add    esp, 8
mov    edx, [ebp+var_170]
add    edx, 10h
mov    [ebp+var_170], edx
mov    eax, [ebp+arg_8]
add    eax, 10h
mov    [ebp+arg_8], eax
lea    ecx, [ebp+var_16C]
mov    [ebp+var_174], ecx
jmp    loc_401B9A
```

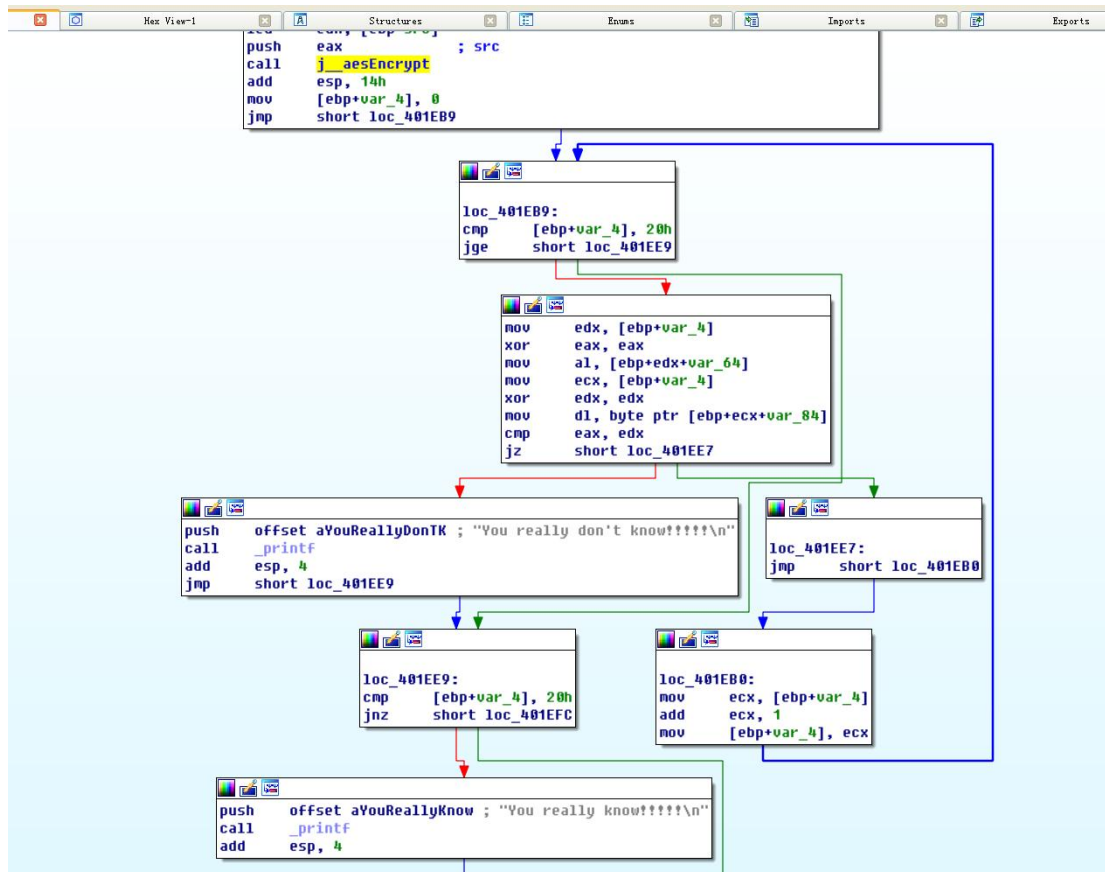


```
loc_401B9A:
mov    ecx, [ebp+var_4]
add    ecx, 10h
mov    [ebp+var_4], ecx
```

然后对密文数组与未经加密的明文数组自加 16，进入明文的后 16 个字节的加密。并将之前拓展的密钥传给[ebp+var\_174]。

然后对 i+16, 进入下一轮循环。对明文的后 16 字节进行加密。当加密两轮后，i=32。退出 aesEncrypt 函数。

返回 main 函数。



加密结果存储在[ebp+var\_84],然后将[ebp+var\_4]赋值为 0,循环 32 次,对加密后的密文与指定的密文是否相同。即 将[ebp+var\_4+var\_64]的一个个字节与[ebp+var\_84+ var\_4]一个个字节比较。Cmp eax edx 一旦有一位不相同则直接输出 “you really don’ t know!!!!” 直到比完 32 个字节。输出正确答案: “you really know!!!!”。至此程序结束。

## 总结

在实验过程中遇到了各种各样的困难,什么参数,指令都不太会。经过一步步的学习,一步步深入,逐渐慢慢熟悉了汇编指令,对大部分的操作都能基本的了解,但阅读得还是比较慢,对汇编指令还是不够熟悉,还是需要进行大量得阅读,理解。路途还很长还要不断努力!

最后分析汇编可知,明文为: “abcdefghijklmnopqrstuvwxyz123456”