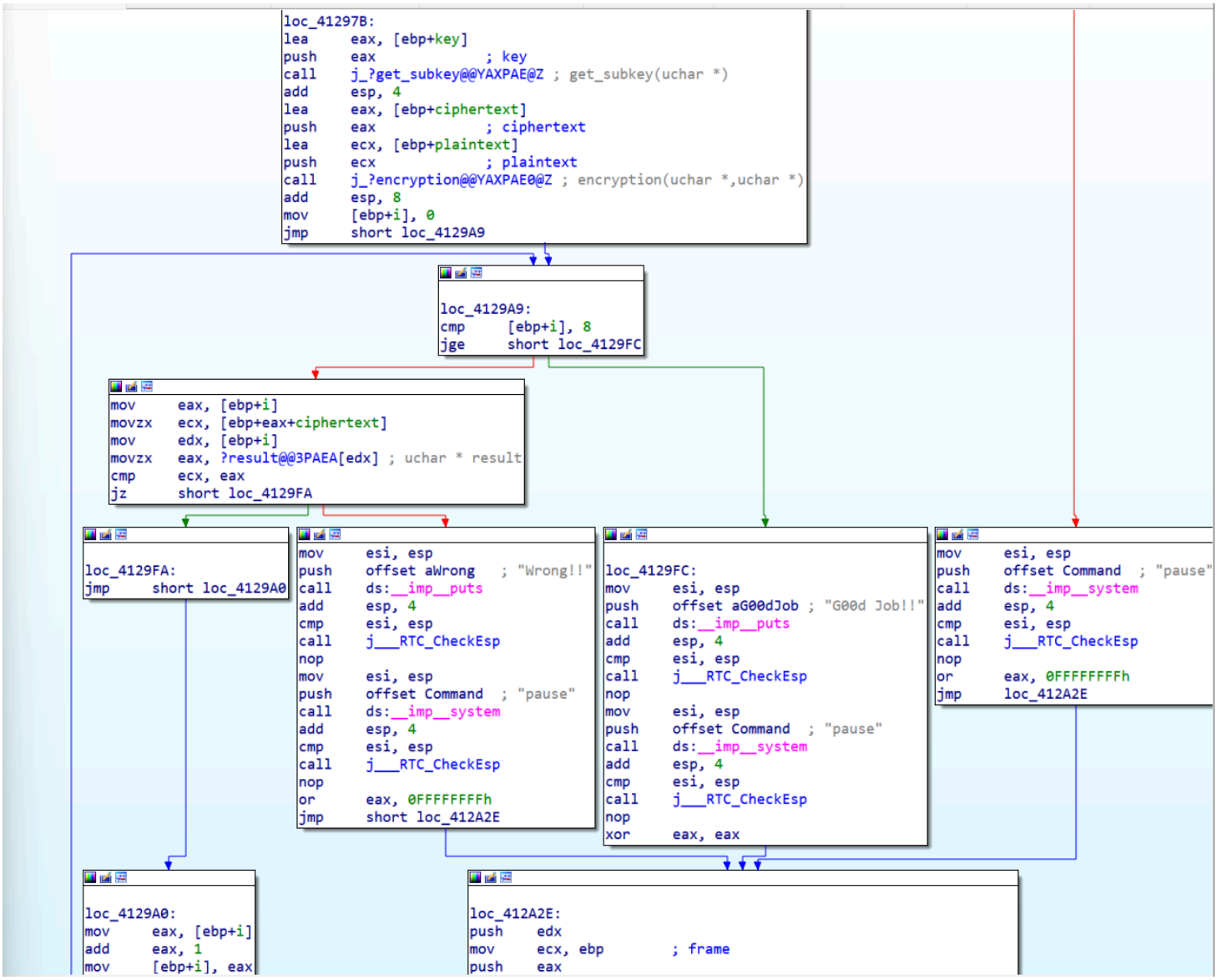
DES算法逆向分析

主逻辑分析



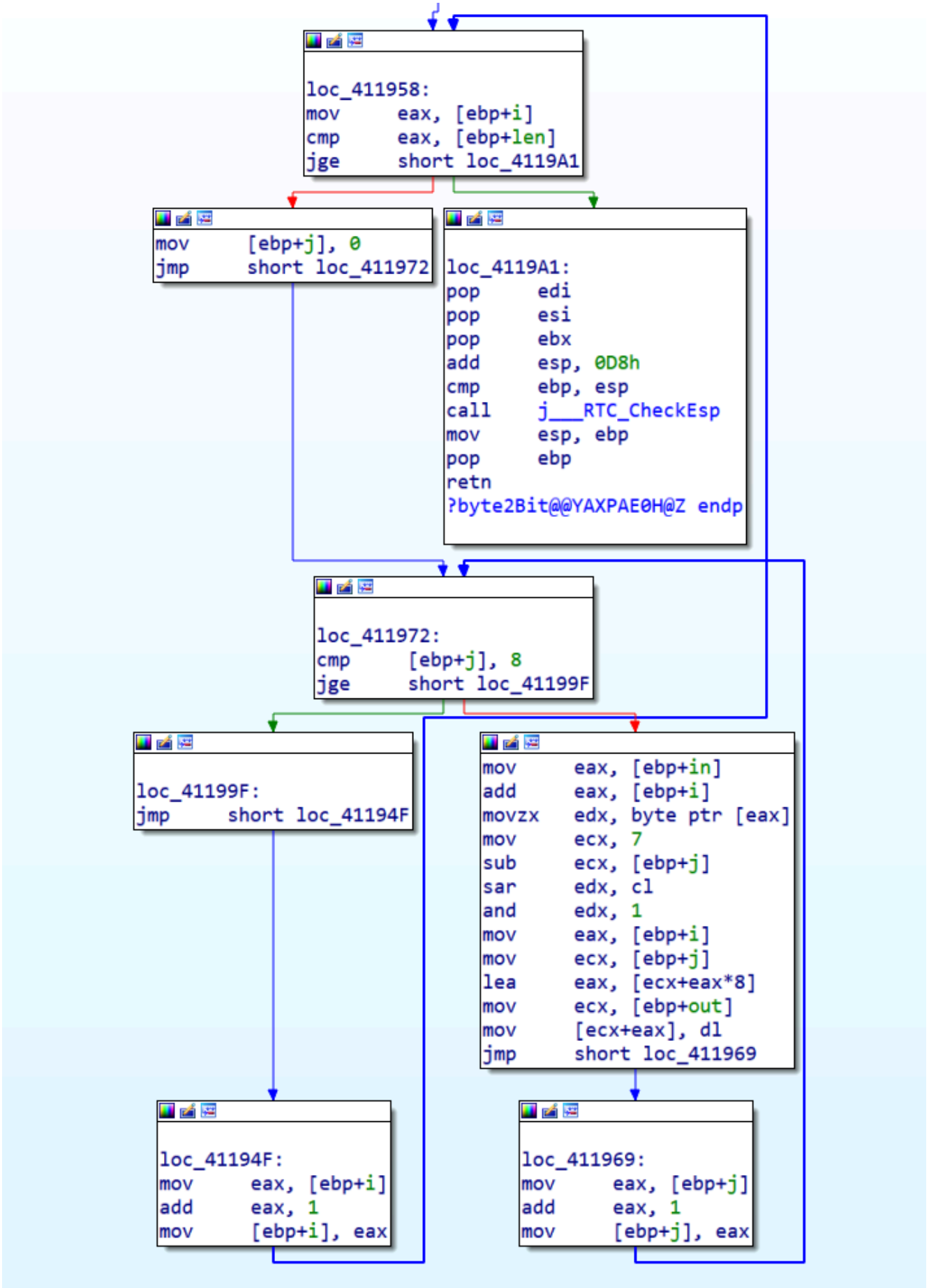
1. 首先通过以 `key` 为参数，调用 `get_subkey()`，即密钥扩展过程，然后调用 `encryption()` 加密，结果存入 `ciphertext`；
2. 后续部分即循环比较 `ciphertext` 和 `result`，相同则通过（8个字节）。

密钥扩展流程 `get_subkey()` 分析

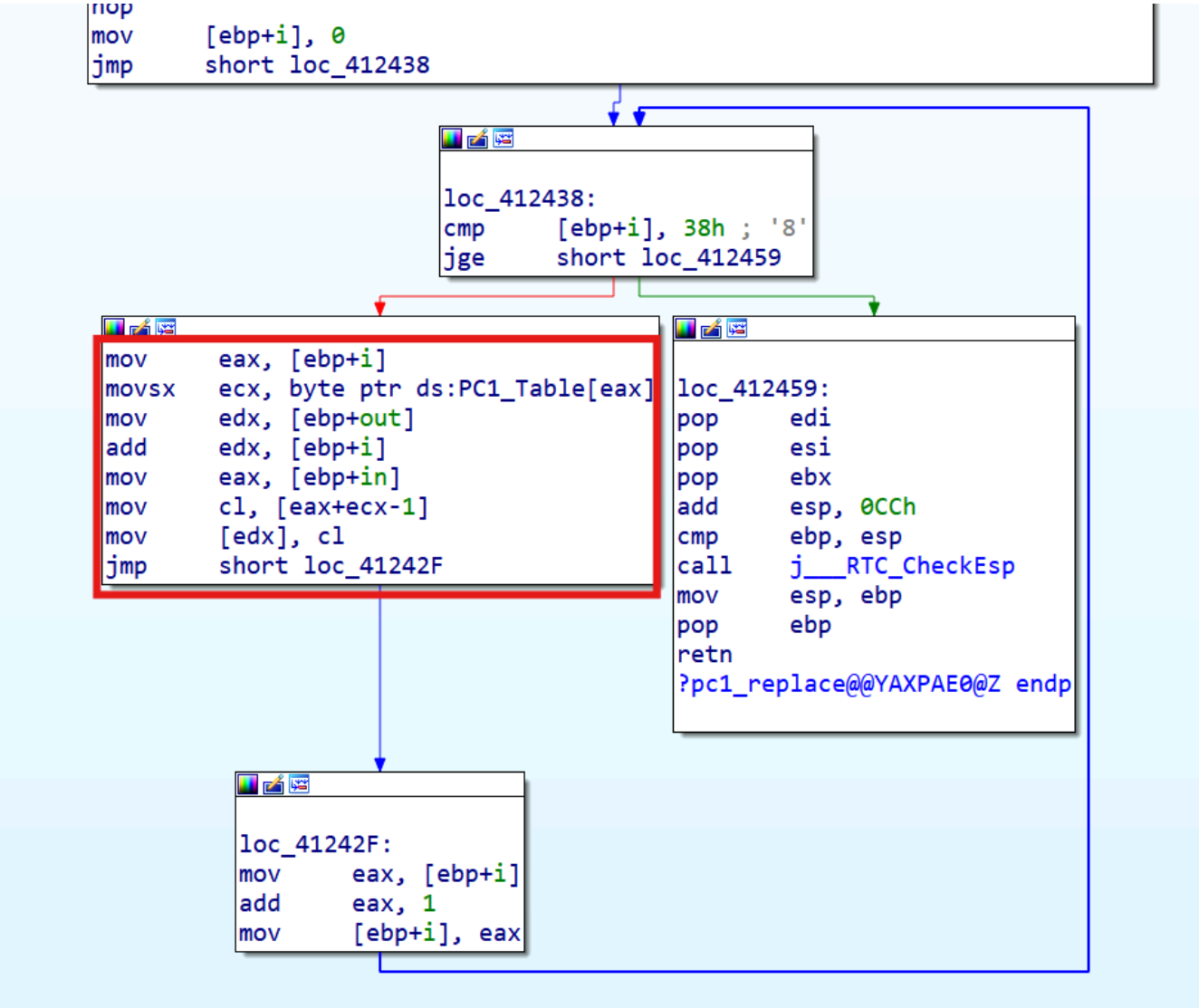
1. 首先调用 `byte2Bit()`，将输入密钥的字节转换成bit流数组，然后调用 `pc1_replace()`，进行PC1置换：

```
push    8               ; len
lea     eax, [ebp+key_temp]
push    eax             ; out
mov     ecx, [ebp+key]
push    ecx             ; in
call    j_?byte2Bit@@YAXPAE0H@Z ; byte2Bit(uchar *,uchar *,int)
add     esp, 0Ch
lea     eax, [ebp+pc1_result]
push    eax             ; out
lea     ecx, [ebp+key_temp]
push    ecx             ; in
call    j_?pc1_replace@@YAXPAE0@Z ; pc1_replace(uchar *,uchar *)
add     esp, 8
mov     [ebp+i], 0
jmp     short loc_4120B3
```

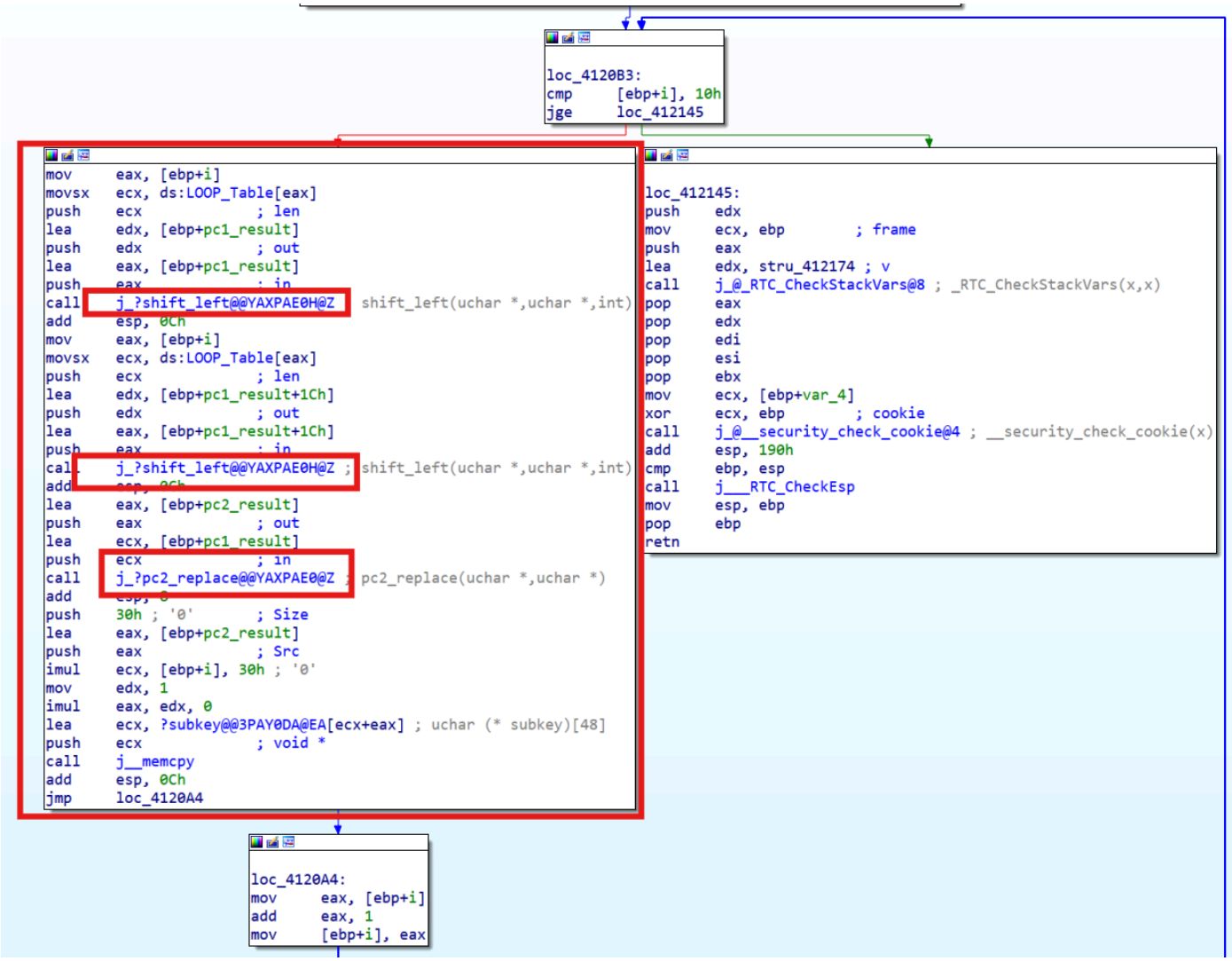
- `byte2Bit()`：定义两层循环，第一层0到 `key` 的长度，第二层0到8（1字节对应8比特）。右下角第二个方框为处理过程，对输入依次右移 `7-j` 位，并和 `1` 做与操作，结果放入 `out` 数组（ `i*8` ）的 `i` 行 `j` 列，即将字节输入的每一位依次放到 `out` 的每一行，主要是用于后续置换操作，可以直接取数；



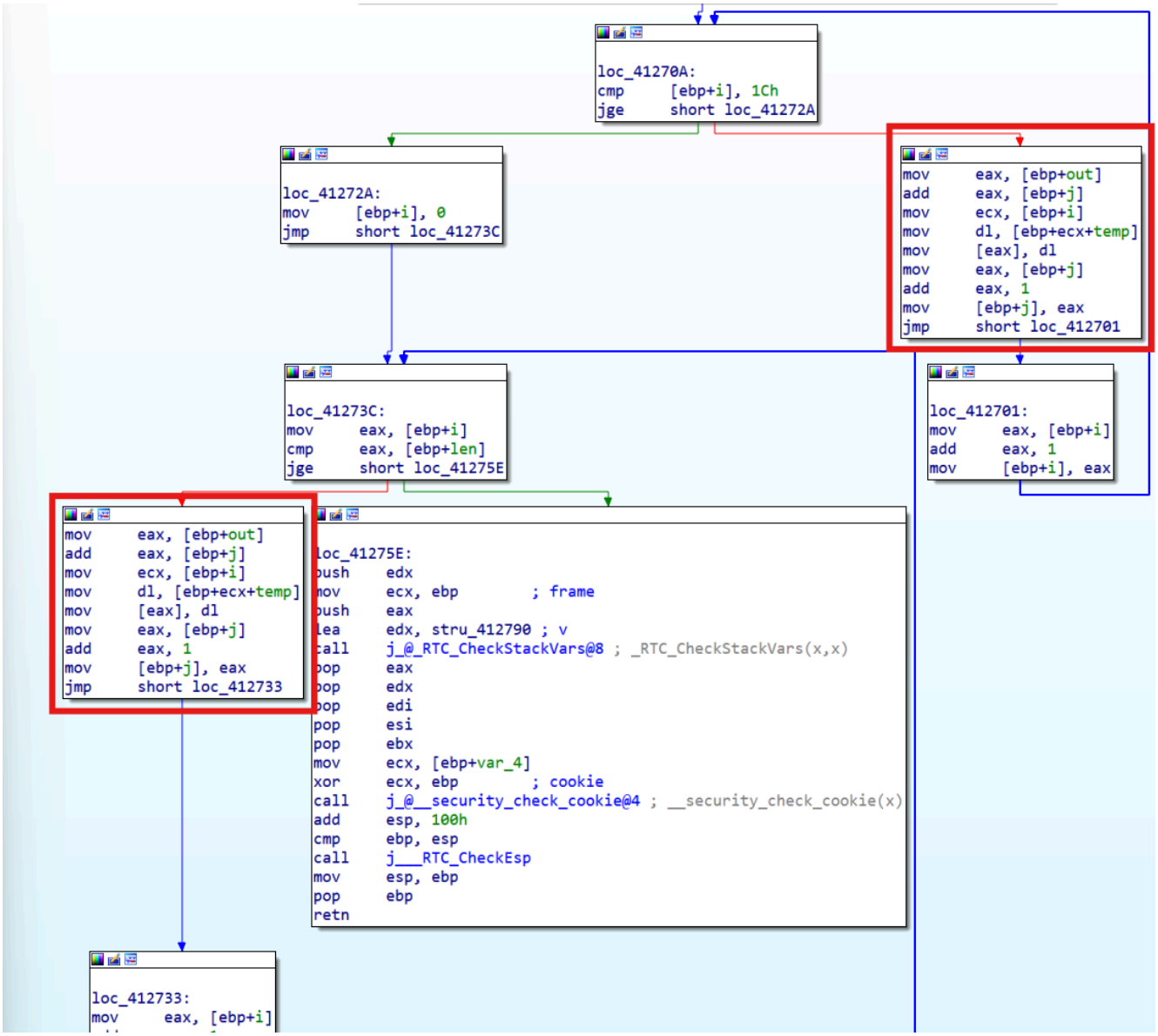
- `pc1_replace()`：循环56次，遍历 `PC1` 数组，将 `PC1表` 依次取值放入 `ecx`，依次将 `PC1_Table[i] - 1` 处的 `in` 数组的值放入 `out[i]` 完成置换，输出变为56位。



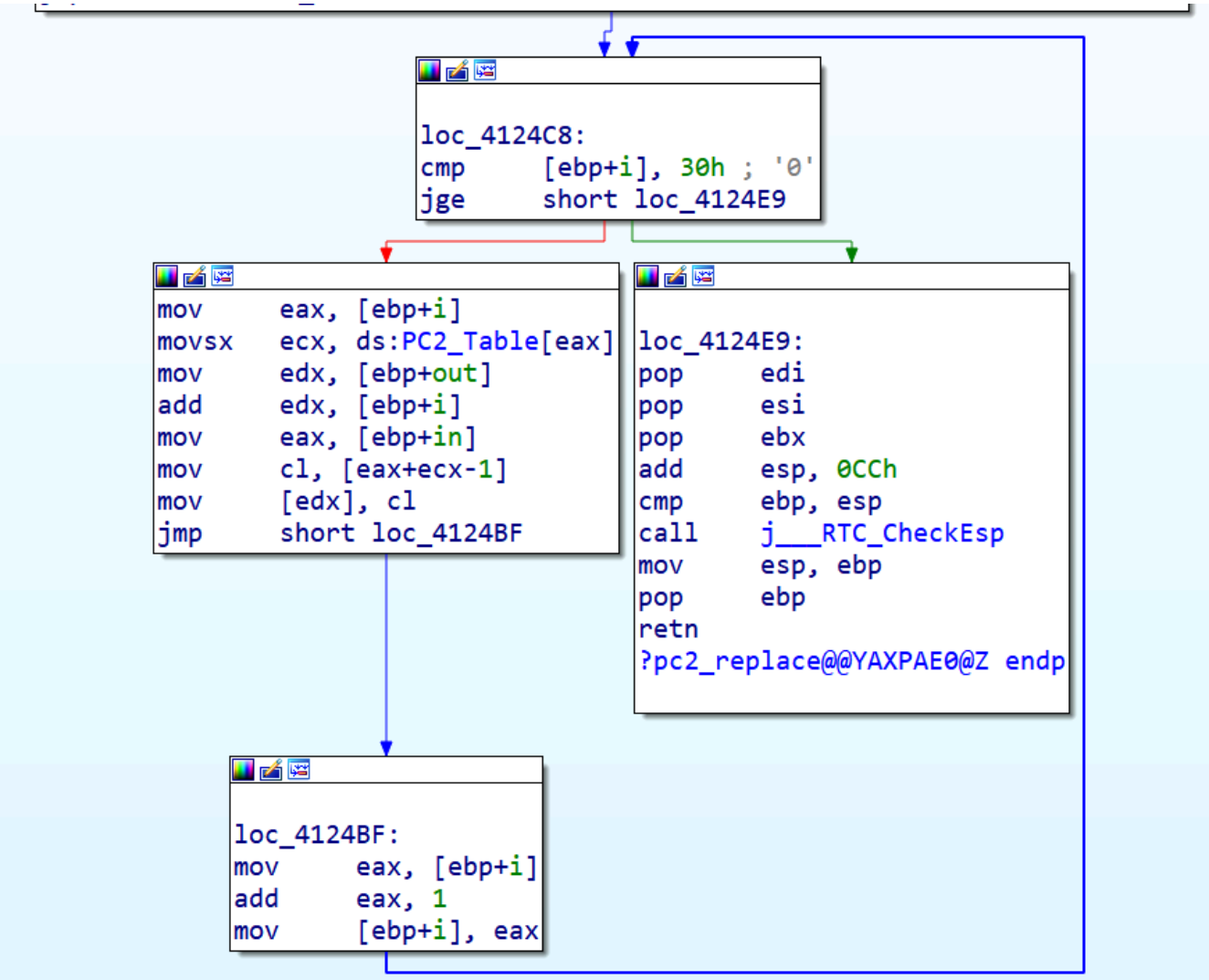
2. 根据 LOOP_TABLE 调用 shift_left() 进行循环左移，第一次传入前28位（[ebp+pc1_result]），第二次传入后28位（[ebp+pc1_result+1Ch]），然后进行 PC2 置换，最后放入 subkey 数组：



- `shift_left()`：使用数组操作模拟循环左移， `len` 为左移位数，右侧为小于28时，将对应位置的 `temp[i]` 放到 `out[j]` 里， `j` 自增，直到最后一个元素，然后进入左侧，此时 `j` 依然是 `out` 需要填充的位置， `i` 从0开始，将剩下的位放入 `out`；

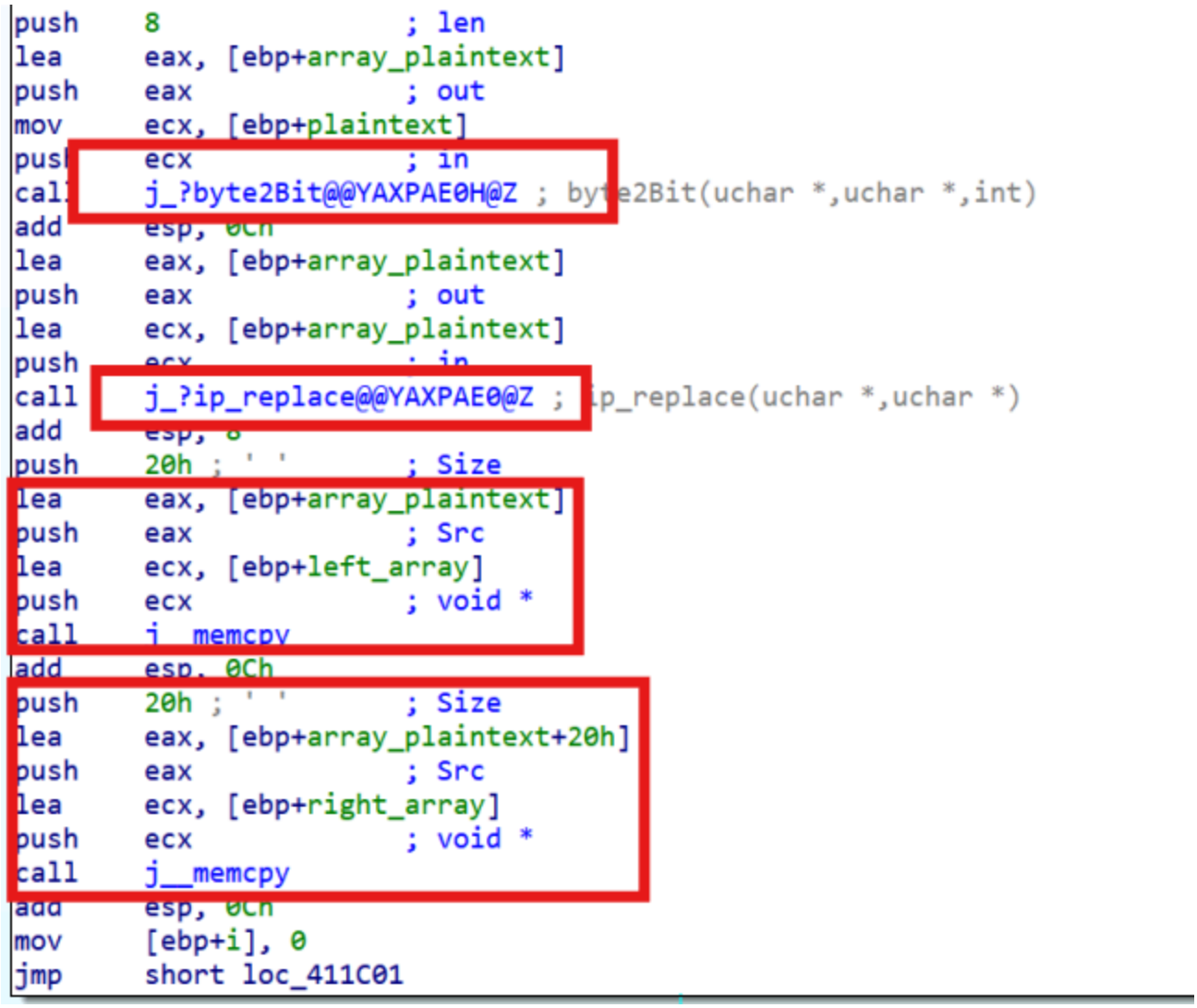


- `pc2_replace()`：过程同 `PC1` 置换，输出变成48位。



加密流程 `encryption()` 分析

1. 首先调用 `byte2Bit()` 转为数组，然后进行IP置换获得迭代初始数组，然后将左右两个32bit的数组分别复制到 `left_array` 和 `right_array`；



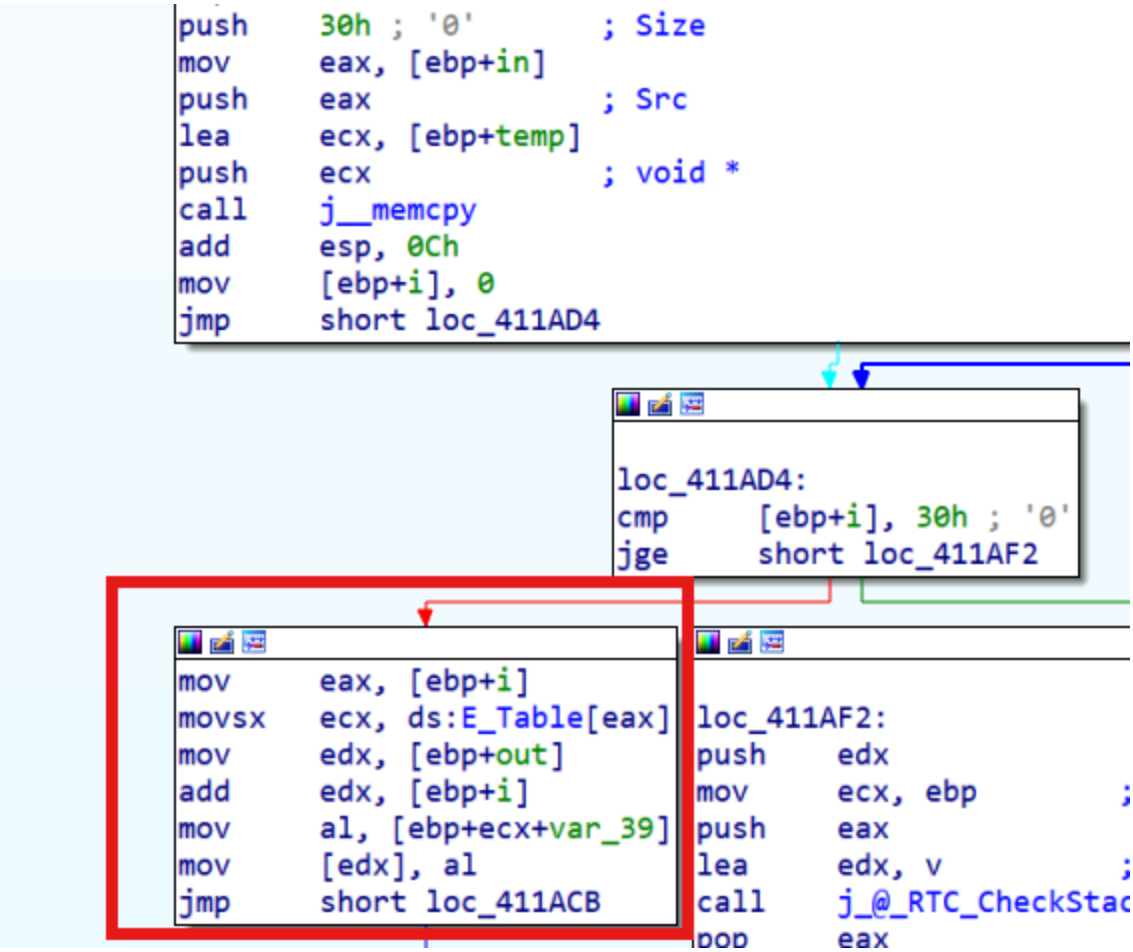
2. 15轮迭代：流程为先取出 `subkey[i]` 作为本轮密钥（48bit），与 `right_array` （in，32bit） 和 `f_result` （out，32bit）作为参数输入 `f` 函数，然后将输出结果和 `left_array` 输入 `byteXOR` 函数，而后将 `left_array` 更新为 `right_array` ， `right_array` 更新为 `f_result` ，重复15轮；

- `f_func()`：调用四个函数如下：

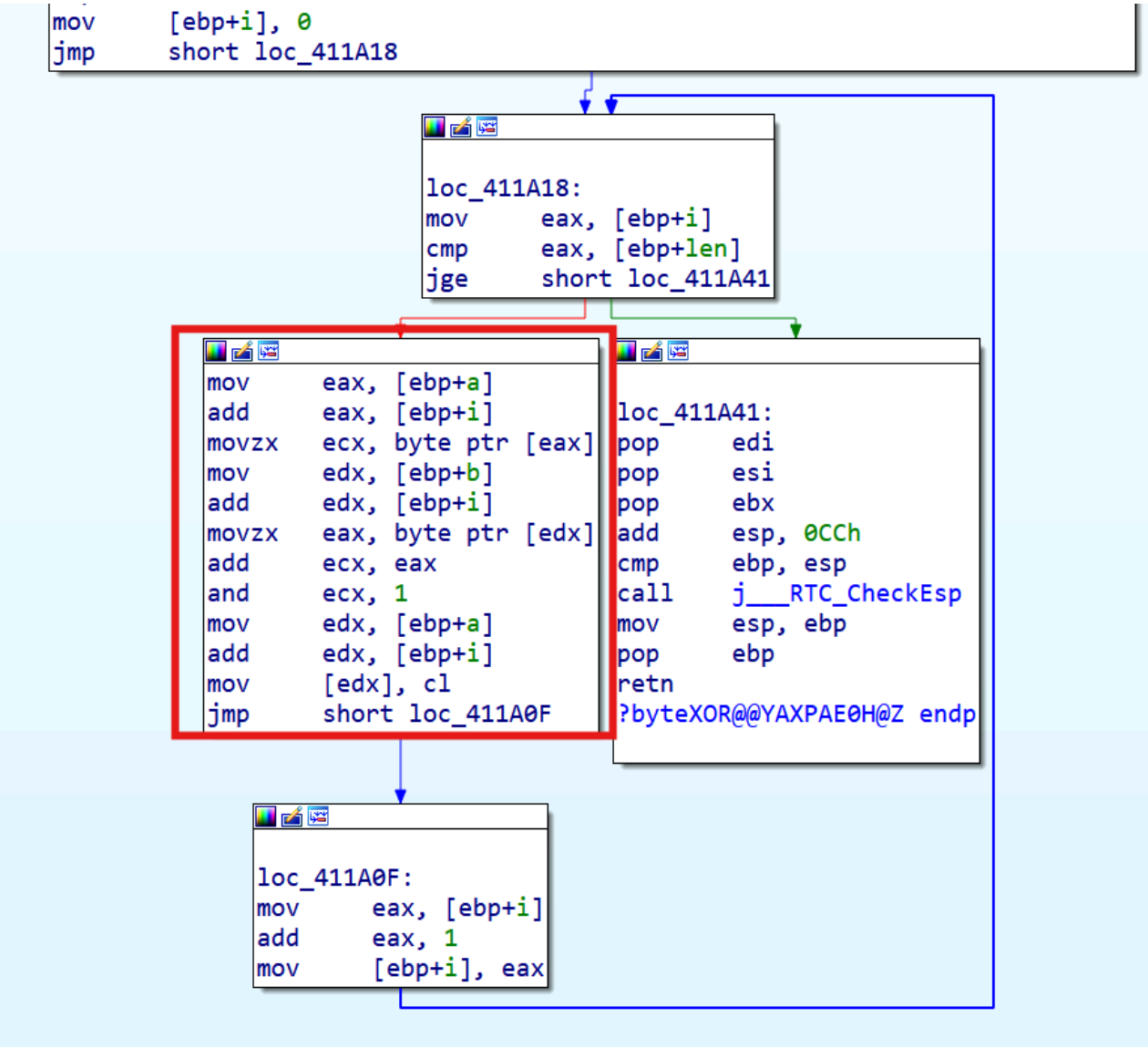
```
__$EncStackInitEnd_4:
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
mov     ecx, offset _6015FB55_main@cpp ; JMC_flag
call    j_@__CheckForDebuggerJustMyCode@4 ; __CheckForDebuggerJustMyCode(x)
nop
lea     eax, [ebp+expand_result]
push    eax             ; out
mov     ecx, [ebp+in]
push    ecx             ; in
call    j_?e_expand@@YAXPAE0@Z ; e_expand(uchar *,uchar *)
add     esp, 8
push    30h ; '0'       ; len
mov     eax, [ebp+ki]
push    eax             ; b
lea     ecx, [ebp+expand_result]
push    ecx             ; a
call    j_?byteXOR@@YAXPAE0H@Z ; byteXOR(uchar *,uchar *,int)
add     esp, 0Ch
lea     eax, [ebp+replace_result]
push    eax             ; out
lea     ecx, [ebp+expand_result]
push    ecx             ; in
call    j_?s_replace@@YAXPAE0@Z ; s_replace(uchar *,uchar *)
add     esp, 8
lea     eax, [ebp+replace_result]
push    eax             ; out
lea     ecx, [ebp+replace_result]
push    ecx             ; in
call    j_?p_replace@@YAXPAE0@Z ; p_replace(uchar *,uchar *)
add     esp, 8
push    20h ; ' '       ; Size
lea     eax, [ebp+replace_result]
push    eax             ; Src
mov     ecx, [ebp+out]
push    ecx             ; void *
call    j__memcpy
```

- e_expand()：输入参数为 in 和 expand_result 两个数组，将 in 放入 temp，根据 E_Table ，将 ebp+var_39+E_Table[i] 放入 out[i]，上面如下定义，即 var_39 比 temp 小1，所以实现内容为 out[i] = temp[E_Table[i]-1]，最终输出48bit；

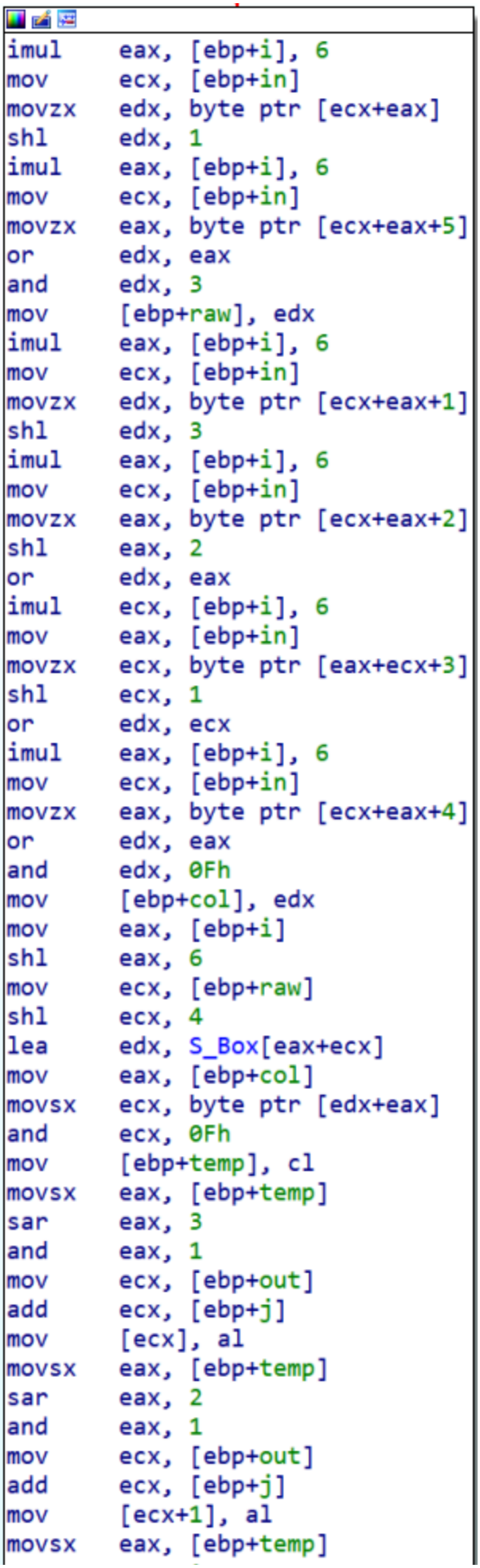
```
var_39= byte ptr -39h
temp= byte ptr -38h
```



- `byteXOR()`：参数为长度 `len` （48），密钥 `ki` （b） 和E扩展结果 `expand_result` （a），循环48轮，依次取 `a[i]` 、 `b[i]` ，相加后和 `1` 做与操作（取最低位），在放到 `a[i]` ，即对数组元素（1bit）做异或操作。



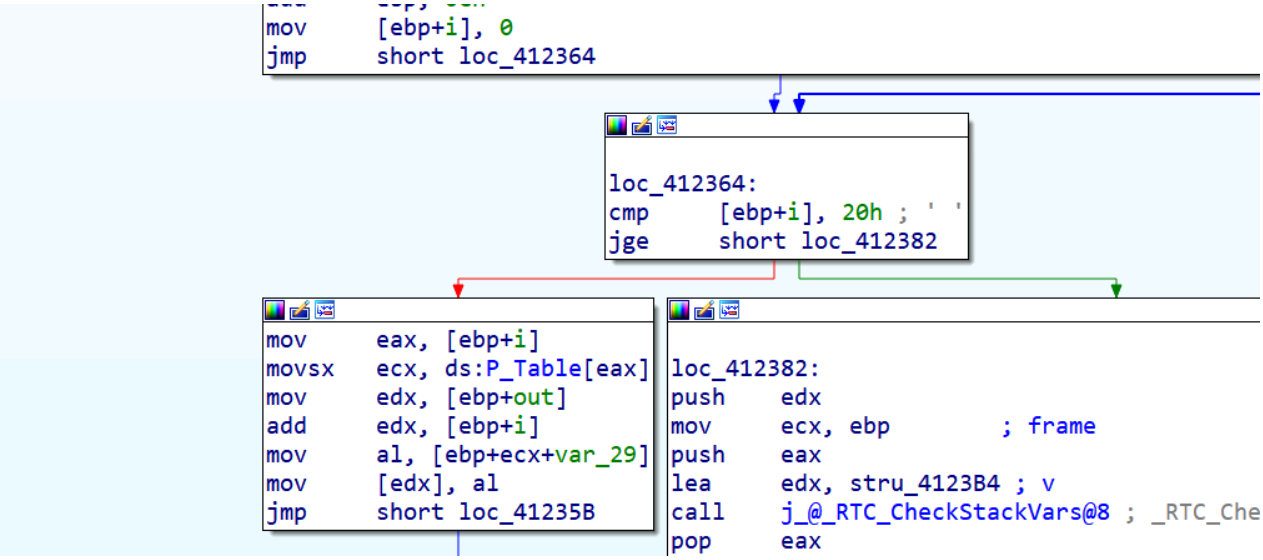
- `s_replace()`：参数为 `replace_result(out)` 和 `expand_result(in)`，共有8个S盒，循环8次，输入的48bit每6bit一组，每次循环如图操作。
  - 取 `in[i*6]` 左移1位（*2），取 `in[6*i+5]` 与上一结果做或操作，再和 `3` 做与操作，即 `(in[6 * i + 5] | (2 * in[6 * i])) & 3`，意义为取一组（6bit）中第一位和最后一位组成一个2bit的数，传入 `raw`；
  - 取 `in[i*6+1]` 左移3位（*8），`in[i*6+2]` 左移2位（*4），`in[i*6+3]` 左移1位（*2），`in[i*6+4]` 四个结果做或操作，然后和 `0xF` 做与操作，意义为取第二位到第四位组成一个4bit数，传入 `col`；
  - 取 `i*64`、`raw*16`、`col` 相加，作为S_box索引，显然是取 `S_Box[i][raw][col]` 放入 `temp`，然后和 `0xF` 做与操作取低四位；
  - 然后将结果依次右移3,2,1,0位再分别和 `1` 做与操作取最低位放入 `out[j]`, `out[j+1]`, `out[j+2]`, `out[j+3]`；
  - 最后 `j+=4`，进下一个循环，总体实现48bit转32bit（每一组6转4）。



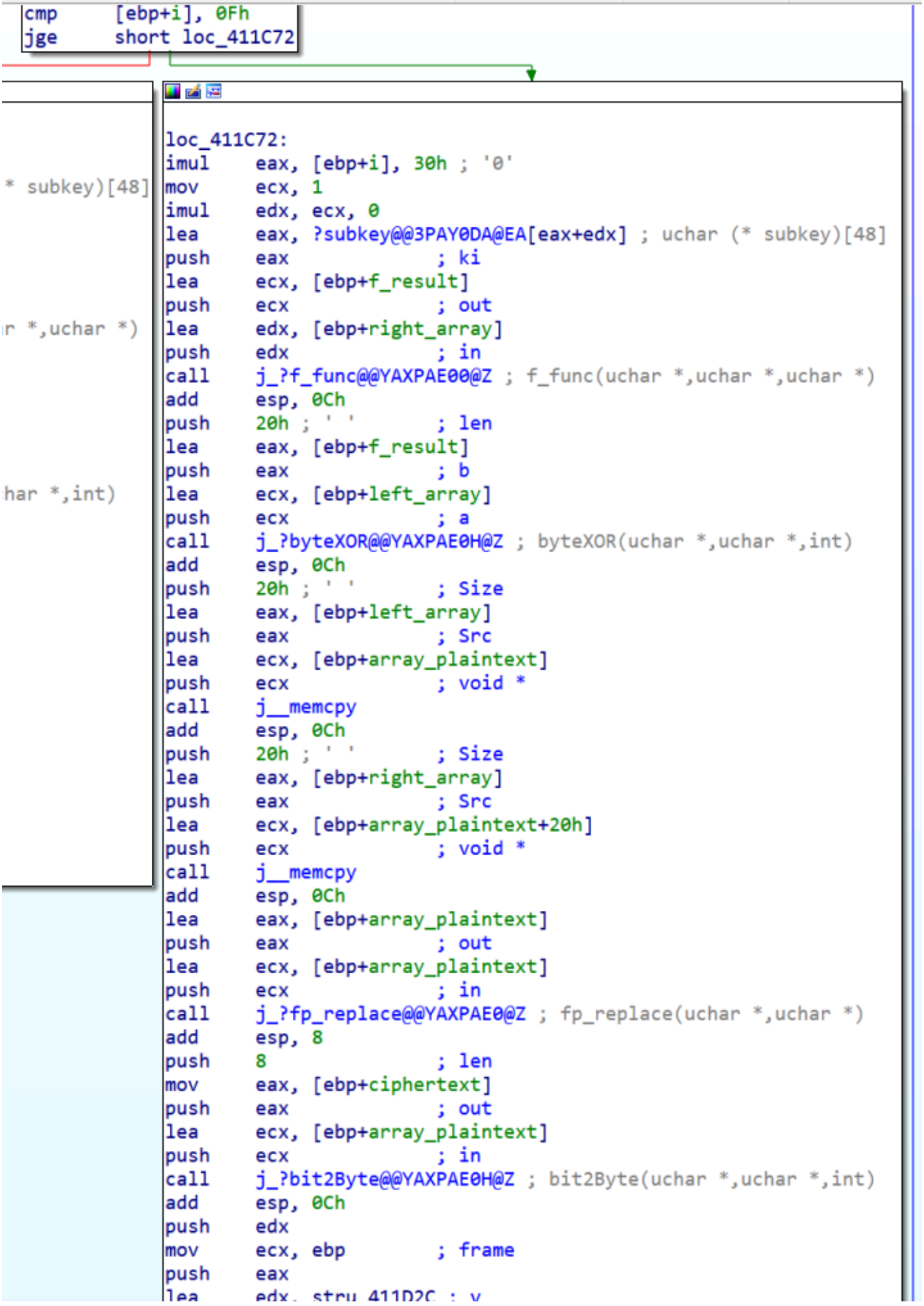
```
imul    eax, [ebp+i], 6
mov     ecx, [ebp+in]
movzx   edx, byte ptr [ecx+eax]
shl     edx, 1
imul    eax, [ebp+i], 6
mov     ecx, [ebp+in]
movzx   eax, byte ptr [ecx+eax+5]
or      edx, eax
and     edx, 3
mov     [ebp+raw], edx
imul    eax, [ebp+i], 6
mov     ecx, [ebp+in]
movzx   edx, byte ptr [ecx+eax+1]
shl     edx, 3
imul    eax, [ebp+i], 6
mov     ecx, [ebp+in]
movzx   eax, byte ptr [ecx+eax+2]
shl     eax, 2
or      edx, eax
imul    ecx, [ebp+i], 6
mov     eax, [ebp+in]
movzx   ecx, byte ptr [eax+ecx+3]
shl     ecx, 1
or      edx, ecx
imul    eax, [ebp+i], 6
mov     ecx, [ebp+in]
movzx   eax, byte ptr [ecx+eax+4]
or      edx, eax
and     edx, 0Fh
mov     [ebp+col], edx
mov     eax, [ebp+i]
shl     eax, 6
mov     ecx, [ebp+raw]
shl     ecx, 4
lea     edx, S_Box[eax+ecx]
mov     eax, [ebp+col]
movsx   ecx, byte ptr [edx+eax]
and     ecx, 0Fh
mov     [ebp+temp], cl
movsx   eax, [ebp+temp]
sar     eax, 3
and     eax, 1
mov     ecx, [ebp+out]
add     ecx, [ebp+j]
mov     [ecx], al
movsx   eax, [ebp+temp]
sar     eax, 2
and     eax, 1
mov     ecx, [ebp+out]
add     ecx, [ebp+j]
mov     [ecx+1], al
movsx   eax, [ebp+temp]
```

```
sar     eax, 1
and     eax, 1
mov     ecx, [ebp+out]
add     ecx, [ebp+j]
mov     [ecx+2], al
movsx   eax, [ebp+temp]
and     eax, 1
mov     ecx, [ebp+out]
add     ecx, [ebp+j]
mov     [ecx+3], al
mov     eax, [ebp+j]
add     eax, 4
mov     [ebp+j], eax
jmp     loc_412556
```

- `p_replace()`：参数为 `replace_result(out&in)`， 过程与E扩展之类的置换操作完全相同。



3. 第16轮迭代：流程同上，最后将结果分别复制到 `array_plaintext` 的前后32位，最后做IP逆置换（流程同IP置换），再调用 `bit2Byte()` 从数组转回字节流，安全检查后返回主逻辑。


cmp [ebp+i], 0Fh
jge short loc_411C72
loc_411C72:
imul eax, [ebp+i], 30h ; '0'
mov ecx, 1
imul edx, ecx, 0
lea eax, ?subkey@@3PAY0DA@EA[eax+edx] ; uchar (* subkey)[48]
push eax ; ki
lea ecx, [ebp+f_result]
push ecx ; out
lea edx, [ebp+right_array]
push edx ; in
call j_?f_func@@YAXPAE00@Z ; f_func(uchar *,uchar *,uchar *)
add esp, 0Ch
push 20h ; ' ' ; len
lea eax, [ebp+f_result]
push eax ; b
lea ecx, [ebp+left_array]
push ecx ; a
call j_?byteXOR@@YAXPAE0H@Z ; byteXOR(uchar *,uchar *,int)
add esp, 0Ch
push 20h ; ' ' ; Size
lea eax, [ebp+left_array]
push eax ; Src
lea ecx, [ebp+array_plaintext]
push ecx ; void *
call j__memcpy
add esp, 0Ch
push 20h ; ' ' ; Size
lea eax, [ebp+right_array]
push eax ; Src
lea ecx, [ebp+array_plaintext+20h]
push ecx ; void *
call j__memcpy
add esp, 0Ch
lea eax, [ebp+array_plaintext]
push eax ; out
lea ecx, [ebp+array_plaintext]
push ecx ; in
call j_?fp_replace@@YAXPAE0@Z ; fp_replace(uchar *,uchar *)
add esp, 8
push 8 ; len
mov eax, [ebp+ciphertext]
push eax ; out
lea ecx, [ebp+array_plaintext]
push ecx ; in
call j_?bit2Byte@@YAXPAE0H@Z ; bit2Byte(uchar *,uchar *,int)
add esp, 0Ch
push edx
mov ecx, ebp ; frame
push eax
lea edx, stru_411D2C ; v

```
call    j_@_RTC_CheckStackVars@8 ; _RTC_CheckStackVars(x,x)
pop     eax
pop     edx
pop     edi
pop     esi
pop     ebx
mov     ecx, [ebp+var_4]
xor     ecx, ebp        ; cookie
call    j_@__security_check_cookie@4 ; __security_check_cookie(x)
add     esp, 190h
cmp     ebp, esp
call    j___RTC_CheckEsp
mov     esp, ebp
pop     ebp
```