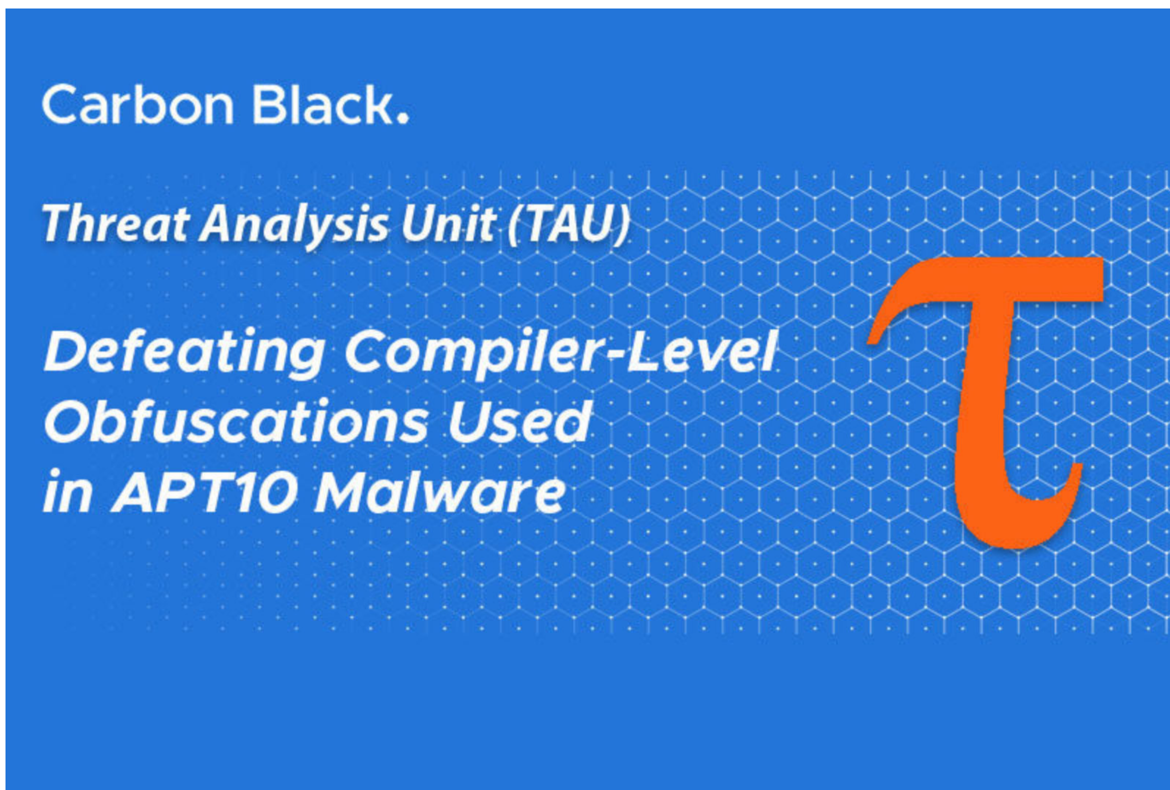
carbonblack.com/2019/02/25/defeating-compiler-level-obfuscations-used-in-apt10-malware

February 25, 2019



February 25, 2019 /
Takahiro Haruyama

Summary

The Carbon Black Threat Analysis Unit (TAU) recently analyzed a series of malware samples that utilized compiler-level obfuscations. For example, opaque predicates were applied to Turla mosquito and APT10 ANEL. Another obfuscation, control flow flattening, was applied to APT10 ANEL and Dharma ransomware packer.

ANEL (also referred to as UpperCut) is a RAT program used by APT10 and observed in Japan uniquely. According to SecureWorks, all ANEL samples whose version is 5.3.0 or later are obfuscated with opaque predicates and control flow flattening.

Opaque predicate is a programming term that refers to decision making where there is actually only one path. For example, this can be seen as calculating a value that will always return True. Control flow flattening is an obfuscation method where programs do not cleanly flow from beginning to end. Instead, a switch statement is called in an infinite loop having multiple code blocks each performing operations, as detailed later in this paper in Figure 10.

The obfuscations looked similar to the ones explained in Hex-Rays blog, but the introduced IDA Pro plugin HexRaysDeob didn't work for one of the obfuscated ANEL samples because the tool was made for another variant of the obfuscation.

TAU investigated the ANEL obfuscation algorithms then modified the HexRaysDeob code to defeat the obfuscations. After the modification, TAU was able to recover the original code.

The below image depicts, an example of an obfuscated function:

```
39  qmemcpy((void *)sbox, &g_blowfish_p_init, 0x48u);
40  memcpy((void *)(sbox + 72), &g_blowfish_ks0_table, 0x1000u);
41  v3 = 1527498016;
42  v28 = 0;
43  v29 = 0;
44  do
45  {
46    while ( 1 )
47    {
48      while ( 1 )
49      {
50        while ( 1 )
51        {
52          while ( v3 > 188596444 )
53          {
54            if ( v3 > 1527498015 )
55            {
56              if ( v3 <= 1831929903 )
57              {
58                if ( v3 == 1527498016 )
59                {
60                  v33 = v28;
61                  v31 = v29;
62                  v3 = 1023101967;
63                  if ( v29 < 18 )
64                    v3 = -699646159;
65                }
66                else if ( v3 == 1557812339 )
67                {
68                  v3 = -1555416444;
69                }
70              }
71              else
72              {
73                switch ( v3 )
74                {
75                  case 1831929904:
76                    v30 = v25;
77                    v3 = -312753338;
78                    if ( v25 < 4 )
79                      v3 = -1923977242;
80                    break;
81                  case 1886892414:
82                    v36 = v26;
83                    v3 = -1592561173;
84                    if ( v26 < 18 )
85                      v3 = 188596445;
86                    break;
87                  case 2017184256:
88                    v3 = -219835227;
89                    break;
90                }
91              }
92            }
93            else if ( v3 <= 723887935 )
94            {
95              if ( v3 == 188596445 )
96              {
97                v15 = dword_72DBB504;
98                v16 = dword_72DBB500;
99                v17 = -1917288831;
100               v18 = 1526512221;
101LABEL_39:
102               v19 = ~(v15 * (v15 - 1)) | 0xFFFFFFFE;
103               if ( (v19 == -1) != v16 < 10 )
104                 v17 = v18;
105               v3 = v17;
106               if ( v19 == -1 )
107                 v3 = v18;
108               if ( v16 >= 10 )
109                 v3 = v17;
110             }
111             else if ( v3 == 280961233 )
112             {
113               fn_blowfish_encdec(v35, v35, (_DWORD *)sbox);
114               v4 = (*v35 << 24) | (BYTE1(v37) << 16);
115               v5 = sbox + 72 + (v30 << 10);
116               *(_DWORD *)(v5 + 4 * v32) = HIBYTE(v37) & ((~v4 & 0x7AACE860 | v4 & 0x85530000) ^ (~(BYTE2(v37) << 8) & 0x7AACE860 |
117               v6 = ~((~(BYTE1(v38) << 16) & 0x6E85DBE8 | (BYTE1(v38) << 16) & 0x7A0000) ^ (~((unsigned __int8)v38 << 24) & 0x6E85DB
118               v7 = ~(BYTE2(v38) << 8);
119               v8 = (v6 & 0x1E5CF118 | ((~(BYTE1(v38) << 16) & 0x6E85DBE8 | (BYTE1(v38) << 16) & 0x7A0000) ^ (~((unsigned __int8)v38
120               *(_DWORD *)(v5 - (-4 - 4 * v32)) = HIBYTE(v38) & (v8 | ~(v7 | v6)) | HIBYTE(v38) ^ (v8 | ~(v7 | v6));
121               v3 = -1704058339;
122               v27 = v32 + 2;
123             }
```

Figure 1: obfuscated function example (all codes cannot be displayed in a screen)

The image below shows the same function once it has been deobfuscated:

```
13  qmemcpy((void *)sbox, &g_blowfish_p_init, 0x48u);
14  memcpy((void *)(sbox + 72), &g_blowfish_ks0_table, 0x1000u);
15  v9 = 0;
16  for ( i = 0; i < 18; ++i )
17  {
18    v4 = (*(unsigned __int8 *)(key + v9 % key_len) << 24) | (*(unsigned __int8 *)(key + (v9 + 1) % key_len) << 16) | (*(unsigned __
19    *(_DWORD *)(sbox + 4 * i) = v4 & ~*(_DWORD *)(sbox + 4 * i) | *(_DWORD *)(sbox + 4 * i) & ~v4;
20    v9 += 4;
21  }
22  v12 = 0;
23  v7 = 0;
24  v11 = 0;
25  while ( v7 < 18 )
26  {
27    fn_blowfish_encdec((unsigned __int8 *)&v11, &v11, (_DWORD *)sbox);
28    *(_DWORD *)(sbox + 4 * v7) = HIBYTE(v11) | (BYTE1(v11) << 16) ^ ((unsigned __int8)v11 << 24) | (BYTE2(v11) << 8);
29    *(_DWORD *)(sbox + 4 * v7 + 4) = (((unsigned __int8)v12 << 24) | (BYTE1(v12) << 16)) ^ (BYTE2(v12) << 8) ^ HIBYTE(v12) | ~(~HIB
30    v7 += 2;
31  }
32  for ( j = 0; j < 4; ++j )
33  {
34    for ( k = 0; k < 256; k += 2 )
35    {
36      fn_blowfish_encdec((unsigned __int8 *)&v11, &v11, (_DWORD *)sbox);
37      v3 = sbox + 72 + (j << 10);
38      *(_DWORD *)(v3 + 4 * k) = HIBYTE(v11) | (((unsigned __int8)v11 << 24) | (BYTE1(v11) << 16)) ^ (BYTE2(v11) << 8);
39      *(_DWORD *)(v3 - (-4 - 4 * k)) = HIBYTE(v12) | (BYTE1(v12) << 16) ^ ((unsigned __int8)v12 << 24) ^ (BYTE2(v12) << 8) | ~(~(BY
40    }
41  }
42  return (char *)&v11;
43}
```

Figure 2: de-obfuscated result of the same function

Details

HexRaysDeob is an IDA Pro plugin written by Rolf Rolles to address obfuscation seen in binaries. In order to perform the deobfuscation, the plugin manipulates the IDA intermediate language called microcode. If you aren't familiar with those structures (e.g, microcode data structures, maturity level, Microcode Explorer and so on), you should read his blog post. Rolles also provides an overview of each obfuscation technique in the same post.

HexRaysDeob installs two callbacks when loading:

- *optinsn_t* for defeating opaque predicates (defined as ObfCompilerOptimizer)
- *optblock_t* for defeating control flow flattening (defined as CFUnflattener)

Opaque Predicates

Before continuing, it is important to understand Hex-Rays maturity levels. When a binary is loaded into IDA Pro, the application will perform distinct layers of code analysis and optimization, referred to as maturity levels. One layer will detect shellcode, another optimizes it into blocks, another determines global variables, and so forth. The *optinsn_t::func* callback function is called in maturity levels from MMAT_ZERO (microcode does not exist) to MMAT_GLBOPT2 (most global optimizations completed). During the callback, opaque predicates pattern matching functions are called. If the code pattern is matched with the definitions, it is replaced with another expression for the deobfuscation. This is important to perform in each maturity level as the obfuscated code could be modified or removed as the code becomes more optimized. We defined two patterns for analysis of the ANEL sample.

```
switch (ins->opcode)
{
case m_bnot:
    iLocalRetVal = pat_BnotOrBnotConst(ins, blk);
    break;
case m_or:
    iLocalRetVal = pat_OrAndNot(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_OrViaXorAnd(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_OrNegatedSameCondition(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_LogicAnd1(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_MulSub2(ins, blk); // added
    break;
case m_and:
    iLocalRetVal = pat_AndXor(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_MulSub(ins, blk);
    break;
case m_xor:
    iLocalRetVal = pat_XorChain(ins, blk);
    if(!iLocalRetVal)
        iLocalRetVal = pat_LnotOrLnotLnot(ins, blk);
    if (!iLocalRetVal)
        iLocalRetVal = pat_LogicAnd1(ins, blk);
    break;
case m_lnot:
    iLocalRetVal = pat_LnotOrLnotLnot(ins, blk);
    break;
case m_setl:
case m_jl:
case m_jge:
    iLocalRetVal = pat_InitedVarCond10(ins, blk); // added
    break;
case m_mov:
```

Figure 3: opaque predicates pattern matching functions switch

Pattern 1: ~(x * (x – 1)) | -2

Below is an example of one of the opaque predicates patterns used by ANEL:

```
13  v2 = ~((_BYTE)dword_745BB58C * ((_BYTE)dword_745BB58C - 1)) | 0xFFFFFFFE;
```

Figure 4: opaque predicates pattern 1

The global variable value **dword_745BB58C** is either even or odd, so **dword_745BB58C * (dword_745BB58C – 1)** is always even. This results in the lowest bit of the negated value becoming 1. Thus, OR by -2 (0xFFFFFFFE) will always produce the value -1.

In this case, the pattern matching function replaces **dword_745BB58C * (dword_745BB58C – 1)** with 2.

Pattern 2: read-only global variable >= 10 or < 10

Another pattern is the following:

```
22  if ( dword_72DBB588 >= 10 )
23    v3 = v1;
24  v8 = dword_72DBB588 < 10;
```

Figure 5: opaque predicates pattern 2

The global variable value **dword_72DBB588** is always 0 because the value is not initialized (we can check it by is_loaded API) and has only read accesses. So the pattern matching function replaces the global variable with 0.

There are some variants with this pattern (e.g., the variable – 10 < 0), where the immediate constant can be different, like 9.

Data-flow tracking for the patterns

We also observed a pattern that was also using an 8-bit portion of the register. In the following example, the variable **v5** in pseudocode is a register operand (cl) in microcode. We need to check if the value comes from the result of x * (x – 1).

```
72  v5 = dword_72DBB504 * (dword_72DBB504 - 1);
73  v6 = 2017184256;
74  while ( 2 )
75  {
76    v7 = (~(_BYTE)v5 | 0xFFFFFFFE) == -1;
```

Figure 6: register operand (pseudocode) in pattern 1

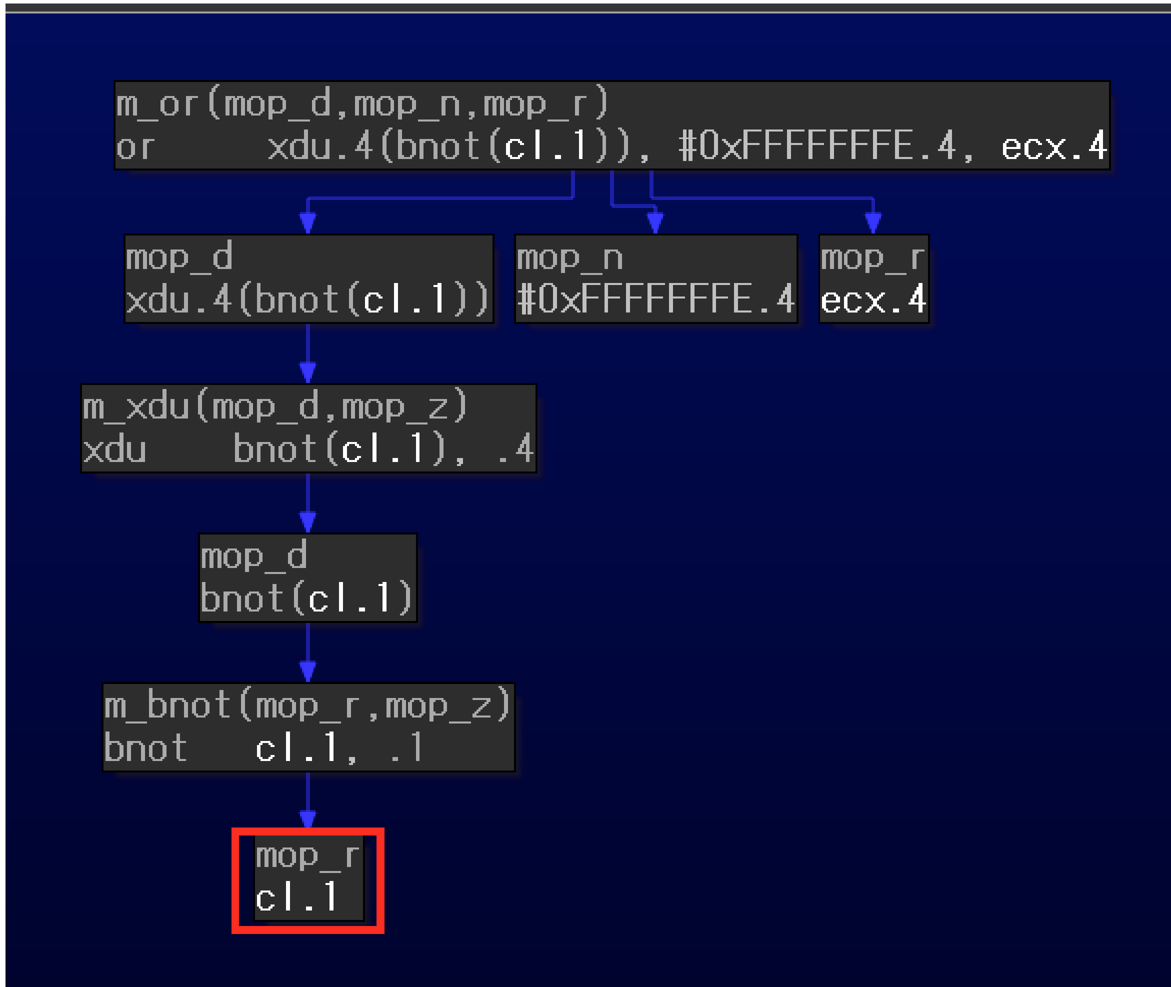


Figure 7: register operand (microcode) in pattern 1

In another example, the variable **v2** in pseudocode is a register operand (ecx) in microcode. We have to validate if a global variable with above-mentioned conditions is assigned to the register.

```
17    v2 = dword_72DBB5F8;
18    for ( i = 0x1C5CF5E5; v2 < 10 && i > 0x856E124; i = 0xD7B920EA )
19    {
```

Figure 8: register operand (pseudocode) in pattern 2

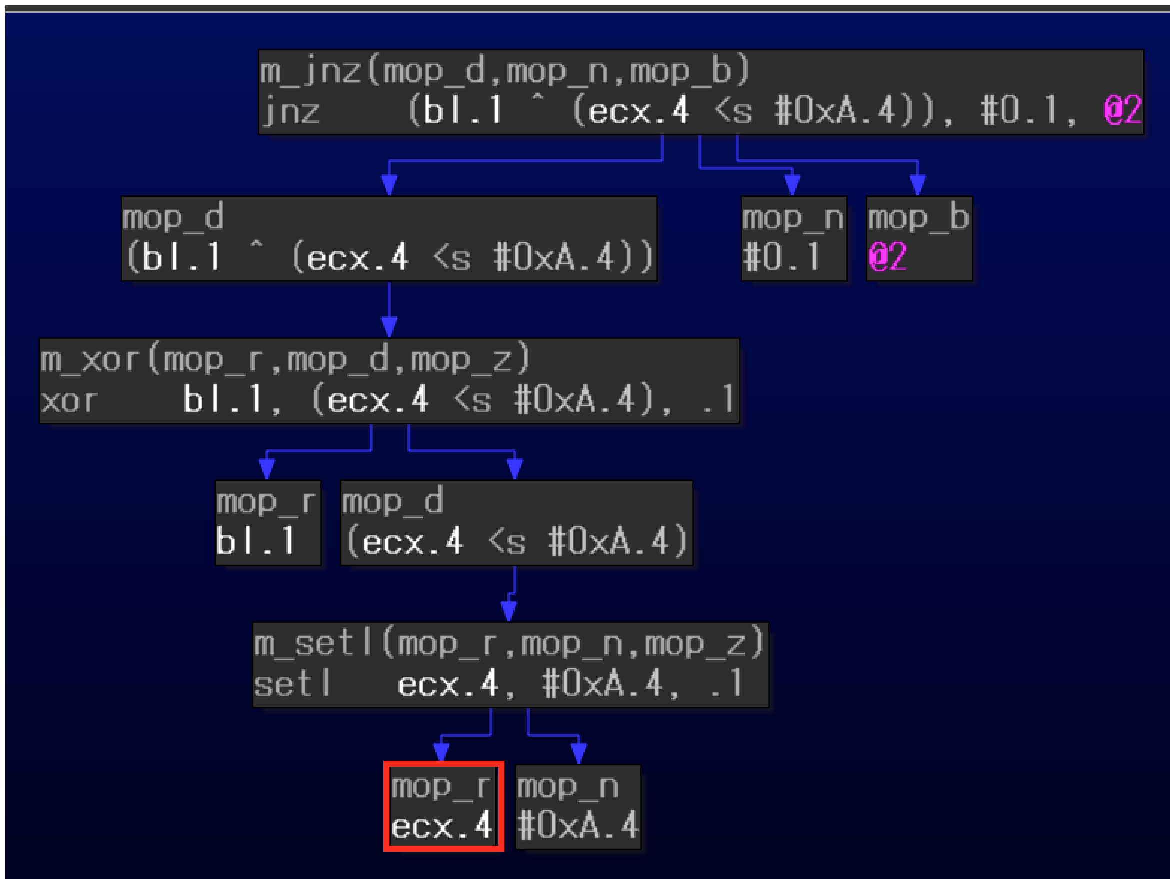


Figure 9: register operand (microcode) in pattern 2

Data-flow tracking code was added to detect these use-cases. The added code requires that the mblock_t pointer information is passed from the argument of *optinsn_t::func* to trace back previous instructions using the mblock_t linked list. However, the callback returns NULL from the mblock_t pointer if the instruction is not a top-level one. For example, Figure 9 shows **jnz** (m_jnz) is a top-level instruction and **setl** (m_setl) is a sub-instruction. If the setl is always sub-instruction during the optimization, we never get the pointer. To handle this type of scenario, the code was modified to catch and pass the mblock_t of the **jnz** instruction to the sub-instruction.

Control Flow Flattening

The original implementation calls the *optblock_t::func* callback function in MMAT_LOCOPT (local optimization and graphing are complete) maturity level. Rolles previously explained the unflattening algorithm in a Hex-Rays blog. For brevity I will quickly cover some key points to understand the algorithm at a high level.

Normally the call flow graph (CFG) of a function obfuscated with control flow flattening has a loop structure starting with yellow-colored “control flow dispatcher” like this, shown after the First Block:

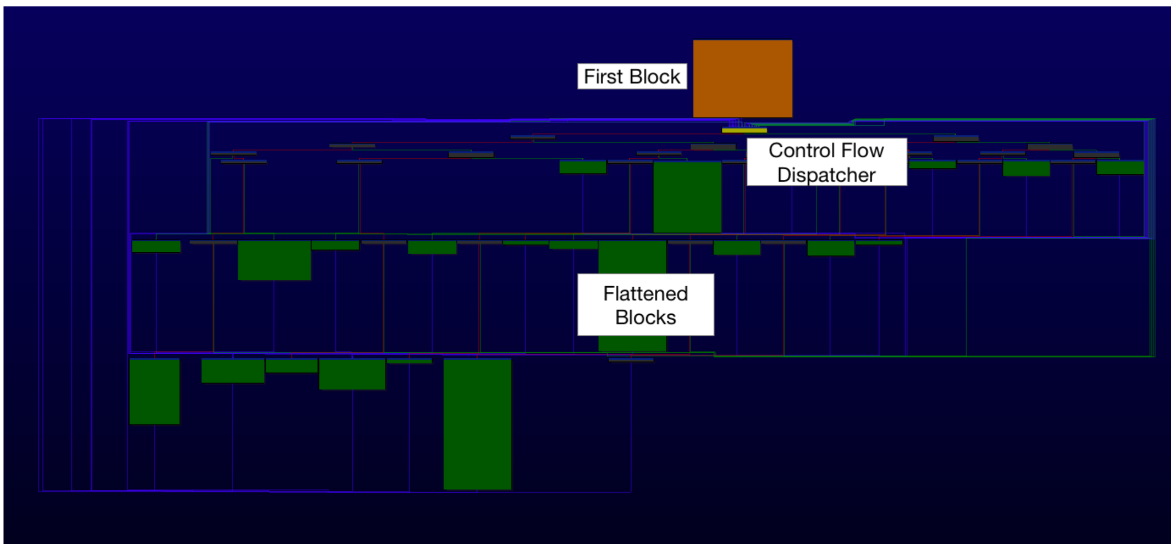


Figure 10: function obfuscated with control flow flattening

The original code is separated into the orange-colored “first block” and green-colored flattened blocks. The analyst is then required to resolve the correct next block and modify the destination accordingly.

The next portion of first block and each flattened block is decided by a “block comparison variable” with an immediate value. The value of the variable is assigned to a specific register in each block then compared in a control flow dispatcher and other condition blocks.

Figure 11: block comparison variable example (blue-highlighted eax in this case)

If the variable registers for the comparison and assignment are different, the assignment variable is called “block update variable” (which is further explained later).

The algorithm looks straightforward however some portions of the code had to be modified in order to correctly deobfuscate the code. This is further detailed below.

Unflattening in multiple maturity levels

As previously detailed, the original implementation of the code only works in MMAT_LOCOPT maturity level. Rolles said this was to handle another obfuscation called “Odd Stack Manipulations”, referred in his blog). However the unflattening of ANEL code had to be performed in the later maturity level since the assignment of block comparison variable heavily depends on opaque predicates.

As an example in the following obfuscated function, the **v3** and **v7** variables are assigned to the block comparison variable (**b_cmp**). However the values are dependent on opaque predicates results.

```
12  v1 = 0xD4A06334;
13  always_minus1 = ~((_BYTE)dword_72DBB58C * ((_BYTE)dword_72DBB58C - 1)) | 0xFFFFFFFE;
14  v3 = 0x7157F526;
15  if ( (always_minus1 == -1) != dword_72DBB588 < 10 )
16    v1 = 0x7157F526;
17  always_true = always_minus1 == -1;
18  result = value;
19  b_cmp = 0x4624F47C;
20  if ( !always_true )
21    v3 = v1;
22  if ( dword_72DBB588 >= 10 )
23    v3 = v1;
24  v8 = dword_72DBB588 < 10;
25  while ( 1 )
26  {
27    while ( b_cmp <= 0x4624F47B )
28    {
29      if ( b_cmp == 0xC504A26C )
30      {
31        b_cmp = v3;
32      }
33      else if ( b_cmp == -727686348 )
34      {
35        b_cmp = 0xC504A26C;
36      }
37    }
38    if ( b_cmp == 0x7157F526 )
39      break;
40    if ( b_cmp == 1176827004 )
41    {
42      v7 = 0xD4A06334;
43      if ( v8 != always_true )
44        v7 = 0xC504A26C;
45      b_cmp = v7;
46      if ( v8 )
47        b_cmp = 0xC504A26C;
48      if ( !always_true )
49        b_cmp = v7;
50    }
51  }
52  return result;
53}
```

Figure 12: simple obfuscated function example

Once the opaque predicates are broken, the loop code becomes simpler:

```
 6   result = value;
 7   for ( b_cmp = 0x4624F47C; ; b_cmp = 0xC504A26C )
 8   {
 9     while ( b_cmp <= 0x4624F47B )
10       b_cmp = 0x7157F526;
11     if ( b_cmp == 0x7157F526 )
12       break;
13   }
14   return result;
15 }
```

Figure 13: simple obfuscated function example (opaque predicates deleted)

Unflattening the code in later maturity levels like MMAT_GLBOPT1 and MMAT_GLBOPT2 (first and second pass of global optimization) caused additional problems. The unflattening algorithm requires mapping information between block comparison variable and the actual block number (mblock_t::serial) used in the microcode. In later maturity levels, some blocks are deleted by the optimization after defeating opaque predicates, which removes the mapping information.

In the example below, the blue-highlighted immediate value **0x4624F47C** is assigned to block comparison variable in the first block. The mapping can be created by checking the conditional jump instruction (**jnz**) in MMAT_LOCOPT.

jnz false case goes to block #9
(0x4624F47C => block #9)

Figure 14: mapping between block comparison variable 0x4624F47C and block number 9

Additionally here is no mapping information in MMAT_GLBOPT2 because the condition block that contains the variable has been deleted. So the next block of the first one in the level can not be determined.

Figure 15: mapping failure

To resolve that issue, the code was written to link the block comparison variable and block address in MMAT_LOCOPT, as the block number is changed in each maturity level. If the code can't determine the mapping in later maturity levels, it attempts to guess the next block number based on the address, considering each block and instruction addresses. The guessing is not 100% accurate however it works for the majority of obfuscated functions tested.

```
[I] MMAT_LOCOPT: found first block id 7 (ea=0x72d43b55), dispatcher id 17 (ea=0x72d43b8d)
[I] MMAT_LOCOPT: Comparison variable = esi
[I] MMAT_LOCOPT: register numbers: comparison variable=36 (esi), update variable=36 (esi)
[I] MMAT_LOCOPT: Inserting 4624f47c -> ea 72d43b72 into map
[I] MMAT_LOCOPT: Inserting 00000000 -> ea 72d43b80 into map
[I] MMAT_LOCOPT: Inserting 00000000 -> ea 72d43b86 into map
[I] MMAT_LOCOPT: Inserting 00000000 -> ea 72d43b87 into map
[I] MMAT_LOCOPT: Inserting c504a26c -> ea 72d43b8b into map
[I] MMAT_LOCOPT: Inserting d4a06334 -> ea 72d43ba8 into map
[I] MMAT_LOCOPT: Inserting 7157f526 -> ea 72d43bb8 into map
        Removed 0 vacuous GOTOs
[I] 2 comparisons, 2 numbers, 64 bits, 34 ones, 0.531250 entropy
[I] MMAT_GLBOPT2: found first block id 1 (ea=0x72d43b04), dispatcher id 3 (ea=0x72d43b8d)
[I] MMAT_GLBOPT2: Comparison variable = esi
[I] MMAT_GLBOPT2: register numbers: comparison variable=36 (esi), update variable=36 (esi)
[I] MMAT_GLBOPT2: Inserting 7157f526 -> block ID 6 into map
[I] FindBlockByKeyFromEA: block key 4624f47c (ea=72d43b72) in MMAT_LOCOPT is translated to block ID 2 in MMAT_GLBOPT2
[I] Next target resolved: 1 (cluster head 1) -> 2 (4624f47c)
[I] Erasing 72D43B42: mov    #0x4624F47C.4, esi.4
[I] The dispatcher predecessor = 1 (goto), cluster head = 1, destination = 2
[I] Block 2 was part of dominated cluster 2
[I] FindBlockByKeyFromEA: block key c504a26c (ea=72d43b8b) in MMAT_LOCOPT is translated to block ID 4 in MMAT_GLBOPT2
[I] Next target resolved: 2 (cluster head 2) -> 4 (c504a26c)
[I] Erasing 72D43B81: mov    #0xC504A26C.4, esi.4
[I] The dispatcher predecessor = 2 (goto), cluster head = 2, destination = 4
[I] Block 4 was part of dominated cluster 4
[I] Next target resolved: 4 (cluster head 4) -> 6 (7157f526)
[I] Erasing 72D43B8B: mov    #0x7157F526.4, esi.4{2}
[I] The dispatcher predecessor = 4 (goto), cluster head = 4, destination = 6
[I] Removed 2 blocks
```

Figure 16: the output log showing ea and block number translation

```
1 int __stdcall fn_just_ret_the_value(int value)
2 {
3   return value;
4 }
```

Figure 17: the result of deobfuscation in this case

Control flow handling with multiple dispatchers

Though the original implementation assumes an obfuscated function has only one control flow dispatcher, some functions in the ANEL sample have multiple control dispatchers. Originally the code called the *optblock_t::func* callback in MMAT_GLBOPT1 and MMAT_GLBOPT2, as the result was not correct in MMAT_CALLS (detecting call arguments). However, this did not work for functions with three or more dispatchers. Additionally, Hex-Rays kernel doesn't optimize some functions in MMAT_GLBOPT2 if it judges the optimization within the level is not required. In this case, the callback is executed just once in the implementation.

To handle multiple control flow dispatchers, a callback for decompiler events was implemented. The code catches the "hxe_prealloc" event (according to Hex-Rays, this is the final event for optimizations) then calls *optblock_t::func* callback. Typically this event occurs a few times to several times, so the callback can deobfuscate multiple control flow

flattenings. Other additional modifications were made to the code (e.g., writing a new algorithm for finding control flow dispatcher and first block, validating a block comparison variable, and so on).

After the modification, for example, the following functions with multiple control flow dispatchers can be unflattened.

In the case of two control flow dispatchers:

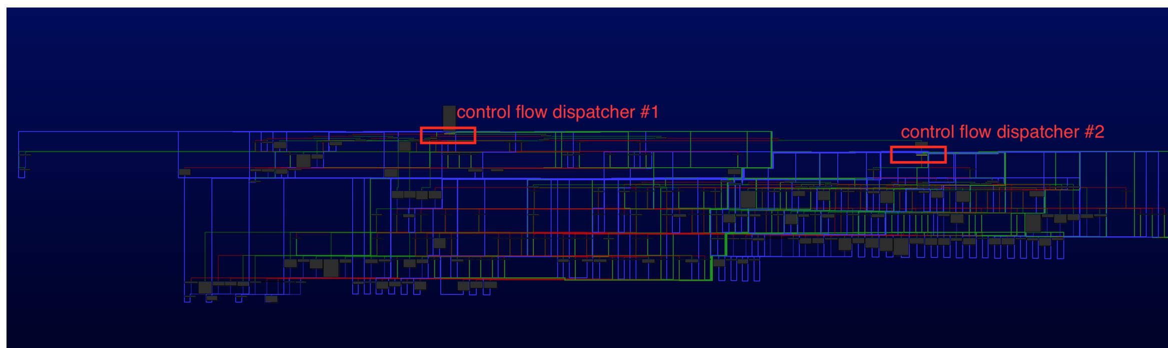


Figure 18: example #1 with two control flow dispatchers (graph)

```
168   v2 = 593200143;
169   v143 = (~((_BYTE)dword_6DAC2D18 * ((_BYTE)dword_6DAC2D18 - 1)) | 0xFFFFFFFE) == -1;
170   v144 = dword_6DAC2D14 < 10;
171   do
172   {
173     while ( 1 )
174     {
175       while ( 1 )
176       {
177LABEL_412:
178         while ( v2 > 368180951 )
179         {
180           if ( v2 > 1305087492 )
181           {
182             if ( v2 > 1743876098 )
183             {
184               if ( v2 == 1743876099 )
185               {
186                 v52 = -697111823;
187                 v53 = -159950636;
188                 v54 = (~((_BYTE)dword_6DAC2D18 * ~-(char)dword_6DAC2D18) | 0xFFFFFFFE) == -1;
189                 v2 = -159950636;
190LABEL_395:
191                 if ( v54 )
192                   v2 = v52;
193                 if ( dword_6DAC2D14 >= 10 )
194                   v2 = v53;
195                 if ( v54 ^ (dword_6DAC2D14 < 10) )
196                   v2 = v52;
197               }
198               else
199               {
200                 if ( v2 == 1865679192 )
201                 {
202                   v101 = dword_6DAC2D14;
203                   v102 = -134103021;
204                   v108 = ~((_BYTE)dword_6DAC2D18 * ((_BYTE)dword_6DAC2D18 - 1)) | 0xFFFFFFFE;
205                   if ( (v108 == -1) != dword_6DAC2D14 < 10 )
206                     v102 = -1388362230;
207                   v104 = v108 == -1;
208                   v105 = -1388362230;
209                   goto LABEL_405;
210                 }
211                 if ( v2 == 2113150305 )
212                 {
213                   v7 = FindNextFileA(v153, v148) == 0;
214                   v2 = -1792660797;
215                   if ( v7 )
216                     v2 = 792936358;
217                 }
218               }
219             }
220             else
221             {
222               switch ( v2 )
223               {
224                 case 1305087493:
225                   v114 = (char *)1305087493;
226                   v48 = (char *)malloc(0x104u);
227                   strcpy(v48, a1);
```

Figure 19: example #1 with two control flow dispatchers (before)

```
 79     v2 = (char *)malloc(0x104u);
 80     strcpy(v2, a2);
 81     strcat(v2, "*.*");
 82     FindFirstFileA(v2, (LPWIN32_FIND_DATAA)&v19);
 83   }
 84   v20 = 1313637808;
 85   v51 = (struct _WIN32_FIND_DATAA *)&v19;
 86   v57 = &v19;
 87   v58 = (char *)malloc(0x104u);
 88   v3 = v58;
 89   strcpy(v58, a2);
 90   strcat(v3, "*.*");
 91   v56 = FindFirstFileA(v3, v51);
 92   v48 = v56 != (HANDLE)-1;
 93   if ( v56 != (HANDLE)-1 )
 94   {
 95     while ( 1 )
 96     {
 97       v62 = v51->cFileName;
 98       if ( v51->cFileName[0] != 46 )
 99         break;
100       while ( !FindNextFileA(v56, v51) )
101 LABEL_2:
102         free(v63);
103     }
104     v63 = malloc(0x104u);
105     v49 = (char *)v63;
106     strcpy((char *)v63, a2);
107     strcat(v49, v62);
108     if ( (~LOBYTE(v51->dwFileAttributes) | 0xFFFFFFEF) != -1 )
109     {
110       v17 = v49;
111       *(_WORD *)&v17[strlen(v49)] = 92;
112       fn_main_with_anti_debug(v49, a3);
113       goto LABEL_2;
114     }
115     v52 = &v64;
116     v53 = &v60;
117     v35 = &v32;
118     v34 = v49;
119     v22 = fopen(v49, "rb+");
120     if ( v22 )
121     {
122       fseek(v22, 0, 2);
123       v33 = ftell(v22);
124       v28 = ((v33 < 1048577) ^ (v33 > 1023) | ~(v33 < 1048577 || v33 > 1023)) & 1;
125       if ( v28 )
126       {
```

Figure 20: example #1 with two control flow dispatchers (after)

In the case of three control flow dispatchers:

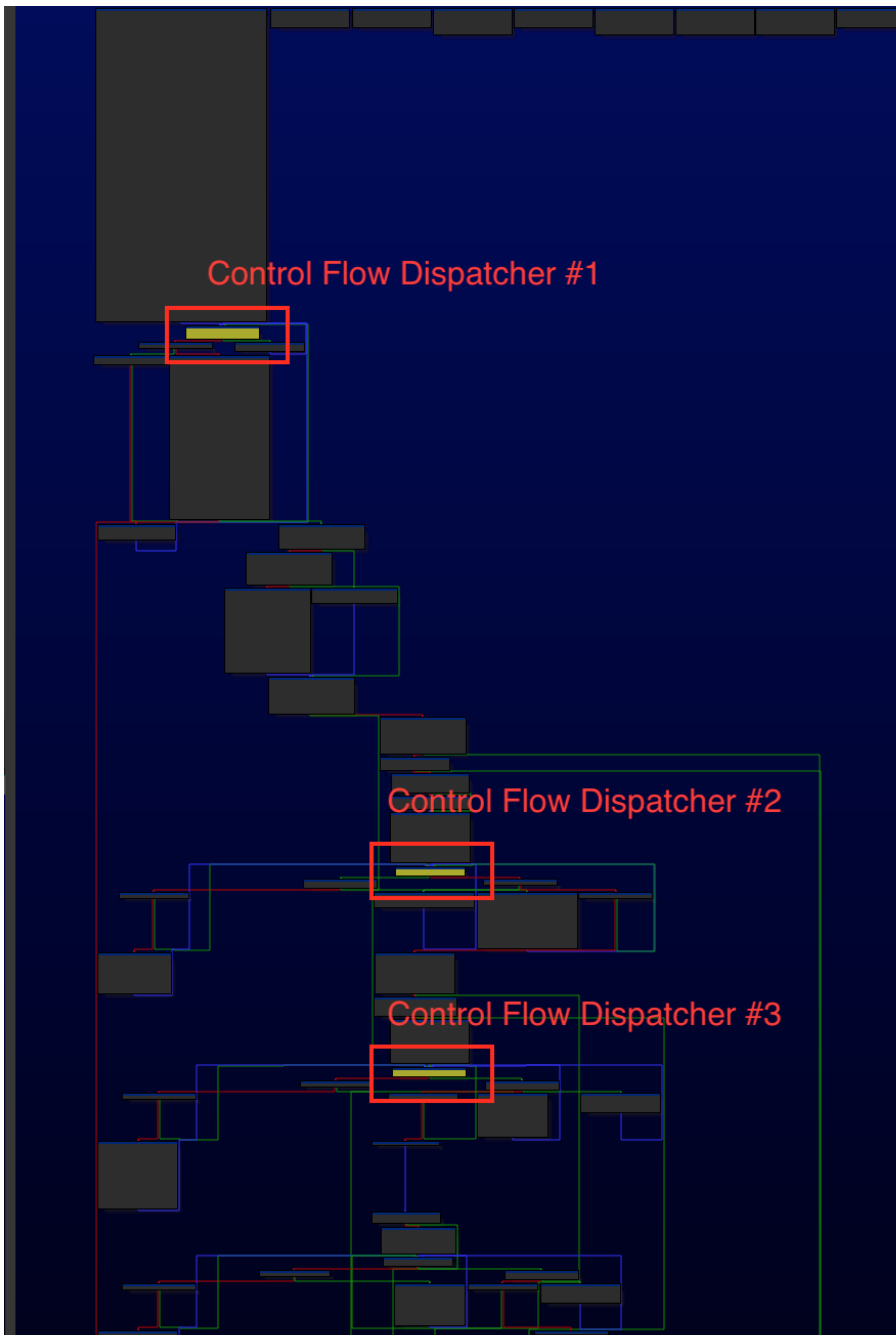


Figure 21: example #2 with three control flow dispatchers (graph)

```
355         {
356           while ( 1 )
357           {
358             while ( v28 <= 1206492458 )
359             {
360               if ( v28 == -1153504087 )
361               {
362                 v28 = 1206492459;
363                 if ( LOBYTE(v116[0]) )
364                   v28 = -202507057;
365                 if ( !(_BYTE)v117 )
366                   v28 = 1206492459;
367                 if ( LOBYTE(v116[0]) ^ (unsigned __int8)v117 )
368                   v28 = -202507057;
369               }
370               else if ( v28 == -202507057 )
371               {
372                 HIDWORD(v142) = 29;
373                 v29 = chg_xorloop_sub_10030E1A(&v107, (struc_bs *)&v130);
374                 v30 = 1206492459;
375                 if ( ((dword_72DBB3B4 * (dword_72DBB3B4 - 1) & (dword_72DBB3B4 * (dword_72DBB3B4 - 1) ^ 0xFFFFFFFE)) == 0) != dword_72DBB3B0 < 10 )
376                   v30 = 2013687272;
377                 v31 = v30;
378                 if ( !(dword_72DBB3B4 * (dword_72DBB3B4 - 1) & (dword_72DBB3B4 * (dword_72DBB3B4 - 1) ^ 0xFFFFFFFE)) )
379                   v31 = 2013687272;
380                 if ( dword_72DBB3B0 >= 10 )
381                   v31 = v30;
382                 v28 = v31;
383                 LOBYTE(v111) = v29 == 0;
384               }
385             }
386             if ( v28 != 1206492459 )
387               break;
388             HIDWORD(v142) = 29;
389             chg_xorloop_sub_10030E1A(&v107, (struc_bs *)&v130);
390             v28 = -202507057;
391           }
392         }
393         while ( v28 != 2013687272 );
394         if ( (_BYTE)v111 )
395         {
396           LOBYTE(v117) = (~((_BYTE)dword_72DBB3B4 * ((_BYTE)dword_72DBB3B4 - 1)) | 0xFFFFFFFE) == -1;
397           v32 = -1153504087;
398           LOBYTE(v116[0]) = dword_72DBB3B0 < 10;
399           do
400           {
401             while ( 1 )
402             {
403               while ( v32 <= 1206492458 )
404               {
405                 if ( v32 == -1153504087 )
406                 {
407                   v38 = 1206492459;
408                   if ( LOBYTE(v116[0]) != (_BYTE)v117 )
409                     v38 = -202507057;
410                   v32 = v38;
411                   if ( LOBYTE(v116[0]) )
412                     v32 = -202507057;
413                   if ( !(_BYTE)v117 )
414                     v32 = v38;
415                 }
416                 else if ( v32 == -202507057 )
417                 {
418                   HIDWORD(v142) = 29;
419                   v33 = chg_xorloop_sub_10030E1A((struc_bs *)&v121, (struc_bs *)&v129);
```

Figure 22: example #2 with three control flow dispatchers (before)

```
238     v21 = File_sub_100090E8((int)v8, v2, v20, (int)&v99[4]);
239     v92 = HIDWORD(v21);
240     v93 = v21;
241     nullsub_1();
242     if ( v99[4] )
243     {
244       if ( (v93 != 0) + v92 > 0 )
245       {
246         v98 = 0;
247         if ( GetFileTime(v99[4], 0, 0, (LPFILETIME)&v90) )
248         {
249           HIDWORD(v125) = 19;
250           v98 = _alldiv_w0(&v90);
251         }
252         CloseHandle(v99[4]);
253         HIDWORD(v125) = 19;
254         sub_72D64446(&v108);
255         LODWORD(v94) = 0;
256         LOBYTE(v99[0]) = 1;
257         v90 = 1;
258         v22 = -1041855547;
259         if ( v90 )
260           v22 = 1460462345;
261         if ( !LOBYTE(v99[0]) )
262           v22 = -1041855547;
263         if ( (unsigned __int8)v90 == LOBYTE(v99[0]) && v22 <= 1460462344 )
264         {
265           HIDWORD(v125) = 21;
266           fn_LoadLibrary_Advapi32((HMODULE *)&v94);
267         }
268         HIDWORD(v125) = 21;
269         fn_LoadLibrary_Advapi32((HMODULE *)&v94);
270         sub_72D64474((HMODULE *)&v94, (struc_bs *)&v90, (int)&v102);
271         chg_xorloop_sub_10012590(&v90, 0xC1E68BC5, (struc_bs *)&v104, 8u, 0x10u);
272         fn_bs_free_0((struc_bs *)&v90);
273         fn_FreeLibrary((HMODULE *)&v94);
274         if ( !v117 )
275           goto LABEL_65;
276         HIDWORD(v125) = 24;
277         LODWORD(v23) = sub_72D9B2A7((struc_bs *)&v118);
278         if ( v23 != __PAIR64__(v92, v93) )
279           goto LABEL_65;
280         sprintf((char *)&v94, "%lld", v98, 0);
281         fn_w_bs_make_from_str((struc_bs *)&v90, (char *)&v94);
282         LOBYTE(v100) = 1;
283         LOBYTE(v99[0]) = 1;
284         v24 = 1206492459;
285         if ( LOBYTE(v99[0]) )
286           v24 = -202507057;
287         if ( !(_BYTE)v100 )
288           v24 = 1206492459;
289         if ( LOBYTE(v99[0]) == (unsigned __int8)v100 )
290         {
291           if ( v24 <= 1206492458 )
292           {
293             HIDWORD(v125) = 29;
294             LOBYTE(v94) = chg_xorloop_sub_10030E1A((struc_bs *)&v90, (struc_bs *)&v113) == 0;
295             goto LABEL_41;
296           }
```

Figure 23: example #2 with three control flow dispatchers (after)

Implementation for various conditional/unconditional jump cases

As shown in Figure 24, the original implementation supports the following two cases of flattened blocks to find a block comparison variable for the next block (the cases are then simplified). In the second case, block comparison variable is searched in each block of endsWithJcc and nonJcc. If the next block is resolved, the CFG (specifically mblock_t::predset and mblock_t::succset) and the destination of **goto** jump instruction are updated.

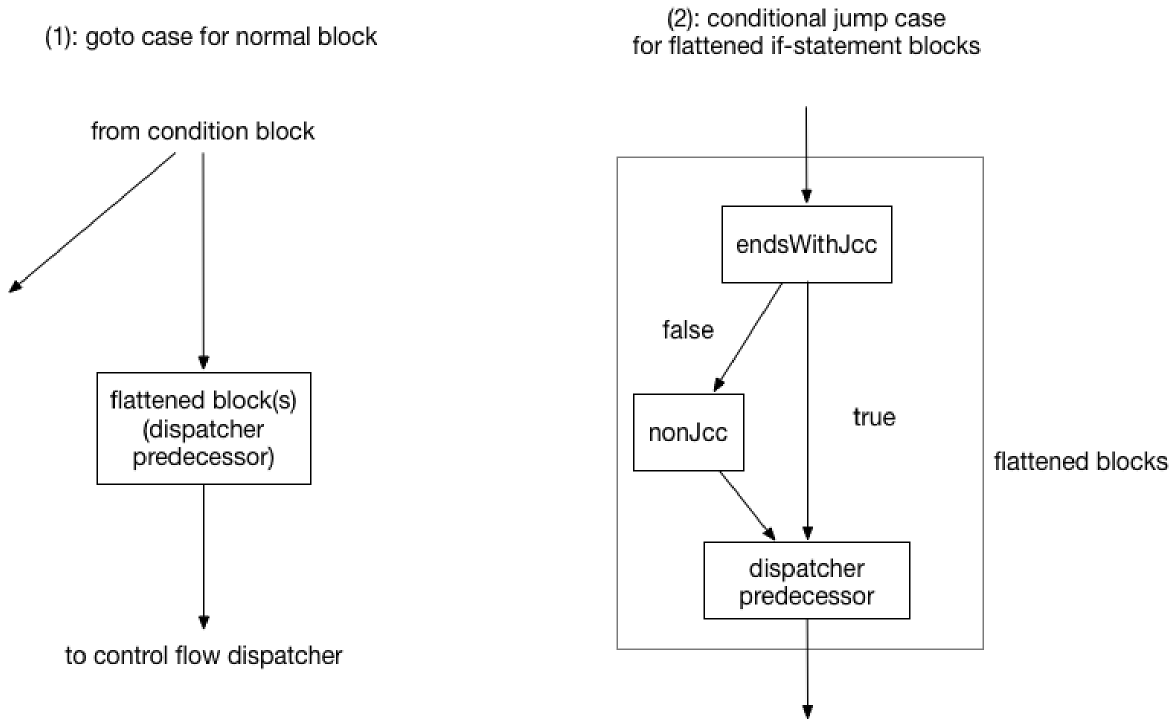


Figure 24: originally-supported two cases of blocks

I found and implemented three more cases in the ANEL sample. Of these, two cases are shown below:

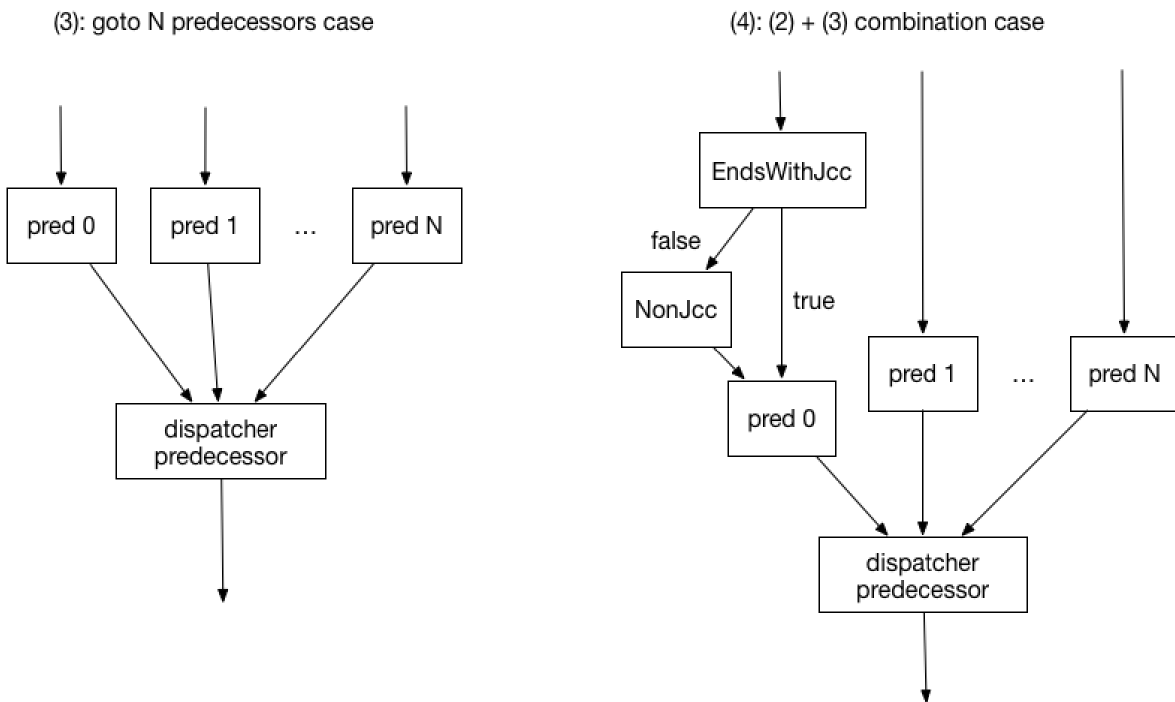


Figure 25: newly-supported two cases of blocks

The code tracks the block comparison variable in each predecessor and more (if any conditional blocks before the predecessor) to identify each next block for unflattening.

And, in the third case that was implemented, the block comparison variables are not assigned in the flattened blocks but rather the first blocks according to a condition. For example, the following microcode graph shows edi is assigned to esi (the block comparison variable in this case) in block number 7 but the edi value is assigned in block number 1 and 2.

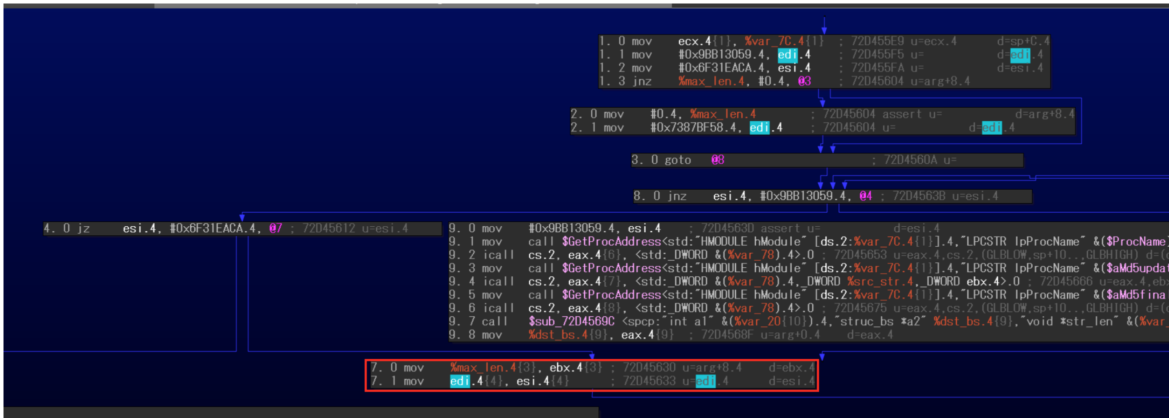


Figure 26: newly-supported case (assigned in first blocks)

If the immediate value for block comparison variable is not found in the flattened blocks, the new code tries to trace the first blocks to obtain the value and reconnects block number 1 and 2 as successors of block number 7, in addition to normal operations mentioned in the original cases.

Another example function did the same processing twice:

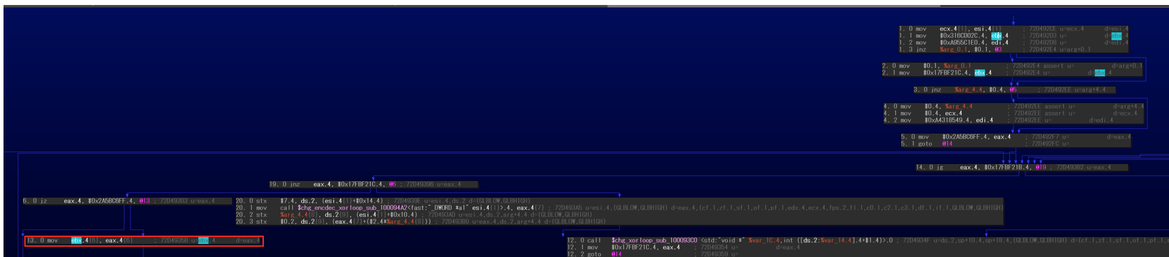


Figure 27: newly-supported case (assigned in first blocks twice #1)

Figure 28: newly-supported case (assigned in first blocks twice #2)

In this case, the code parses the structure in first blocks then reconnects each conditional blocks under the flattened blocks (#1 and #2 as successors of #13, #3 and #4 as successors of #11).

Last, but not least, in all cases explained here, the tail instruction of the dispatcher predecessor can be a conditional jump like **jnz**, not just **goto**. The modified code checks the tail instruction and if the true case destination is a control flow dispatcher, it updates the CFG and the destination of the instruction.

Other minor changes

The following changes are minor compared with above referenced ones.

- Additional jump instructions are supported when collecting block comparison variable candidates and mapping between the variable and ea or block number (**jnz/jle** in JZCollector, **jnz** in JZMapper)
- An entropy threshold adjustment due to check in high maturity level
- Multiple block tracking for getting block comparison variable

And the last change that was introduced in regards to the block update variable referred in the overview. Some functions in the ANEL sample utilize this, however the assignment is a little bit tricky:

Figure 29: block update variable usage with and instruction

By using the **and** instruction, the immediate values used in comparison look different from assigned ones. The modified code will consider this.

Evaluation

The modified tool was tested with an ANEL 5.4.1 payload dropped from a malicious document with the following hash (previously reported by FireEye):
3d2b3c9f50ed36bef90139e6dd250f140c373664984b97a97a5a70333387d18d

The code is able to deobfuscate 34 of 38 functions (89%). It should be noted every function is not always obfuscated. The failure examples are:

- Not yet implemented cases (e.g., a conditional jump of the dispatcher predecessor's tail instruction in goto N predecessors case, consecutive if-statement flattened blocks)
- An incorrect choice of control flow dispatcher and first block (algorithm error)

These fixes will be prioritized for future releases.

Additionally there is a known issue with the result (e.g., the remaining loop or paradoxical decompiled code), using the following IDAPython command in Output window:

idc.load_and_run_plugin("HexRaysDeob", 0xdead)

The command will instruct the code to execute only opaque predicates deobfuscation in the current selected function. This allows an analyst to quickly check if there are any lost blocks by control flow unflattening. For instance, in one of the failure cases, the pseudocode changes like this:

```
17   true1 = 1;
18   true2 = 1;
19   v2 = 0x1D3E02CA;
20   if ( true2 )
21     v2 = 0xC1A18C30;
22   if ( !true1 )
23     v2 = 0x1D3E02CA;
24   if ( true2 == true1 && v2 > 0x1D3E02C9 )      // never executed block
25   {
26     savedregs = 0x1D3E02CA;
27     v5 = fn_get_ptr_from_bs(src_bs);
28     StrToIntA((LPCSTR)v5);
29     v6 = _gmtime64((const __time64_t *)&v8);
30     strftime((char *)&v9, 0x50u, "%a, %d %b %Y %X", v6);
31     fn_w_bs_make_from_str(dst_bs, (char *)&v9);
32   }
33   savedregs = 0xC1A18C30;
34   v3 = fn_get_ptr_from_bs(src_bs);
35   StrToIntA((LPCSTR)v3);
36   v4 = _gmtime64((const __time64_t *)&v8);
37   strftime((char *)&v9, 0x50u, "%a, %d %b %Y %X", v4);
38   fn_w_bs_make_from_str(dst_bs, (char *)&v9);
39   return dst_bs;
40 }
```

Figure 30: one failure case pseudocode (before)

```
  v2 = 1226836615;
  true1 = 1;
  true2 = 1;
  while ( 1 )                                   // never execu
  {
    while ( v2 > 0x1D3E02C9 )
    {
      if ( v2 == 0x1D3E02CA )
      {
        savedregs = 0x1D3E02CA;
        v5 = fn_get_ptr_from_bs(src_bs);
        StrToIntA((LPCSTR)v5);
        v6 = _gmtime64((const __time64_t *)&v8);
        strftime((char *)&v9, 0x50u, "%a, %d %b %Y %X", v6);
        fn_w_bs_make_from_str(dst_bs, (char *)&v9);
        v2 = -1046377424;
      }
      else
      {
        v2 = 0x1D3E02CA;
        if ( true2 )
          v2 = 0xC1A18C30;
        if ( !true1 )
          v2 = 0x1D3E02CA;
        if ( true2 != true1 )
          v2 = 0xC1A18C30;
      }
    }
    if ( v2 != 0xC1A18C30 )
      break;
    savedregs = 0xC1A18C30;
    v3 = fn_get_ptr_from_bs(src_bs);
    StrToIntA((LPCSTR)v3);
    v4 = _gmtime64((const __time64_t *)&v8);
    strftime((char *)&v9, 0x50u, "%a, %d %b %Y %X", v4);
    fn_w_bs_make_from_str(dst_bs, (char *)&v9);
    v2 = -8892470;
  }
  return dst_bs;
```

Figure 31: one failure case pseudocode (after)

After the check, the original result can be restored by using the following command.

idc.load_and_run_plugin(“HexRaysDeob”, 0xf001)

Conclusion

The compiler-level obfuscations like opaque predicates and control flow flattening are started to be observed in the wild by analyst and researchers. Currently malware with the obfuscations is limited, however TAU expects not only APT10 but also other threat actors will start to use them. Unfortunately, in order to break the techniques we have to understand both of the obfuscation mechanisms and disassembler tool internals before we can automate the process.

TAU modified the original HexRaysDeob to make it work for APT10 ANEL obfuscations. The modified code is available publically here. The summary of the modifications is:

- New patterns and data-flow tracking for opaque predicates
- Analysis in multiple maturity levels, considering multiple control flow dispatchers and various jump cases for control flow flattening

The tool can work for almost all obfuscated functions in the tested sample. This implementation will deobfuscate approximately 89% of encountered functions. This provides researchers and analyst broad tool to attack this type of obfuscation, and if it adopted in other families. In should be noted that the tool may not work for the updated versions of ANEL if they are compiled with different options of the obfuscating compiler. Testing in multiple versions is important, so TAU is looking for newer versions ANEL samples. Please reach out to our unit if you have relevant samples or need assistance in deobfuscating the codes.

It's difficult to create a generic tool that can defeat every compiler-level obfuscated binary but experience and knowledge about IDA microcode can be useful for additional new tools.

TAGS:
APT10
/
Carbon Black TAU
/
malware