vb 2019 LONDON 2 – 4 October 2019

DEFEATING APT10 COMPILER-LEVEL OBFUSCATIONS

Takahiro Haruyama
Carbon Black, Japan

tharuyama@carbonblack.com

ABSTRACT

Compiler-level obfuscations, like opaque predicates and control flow flattening, are starting to be observed in the wild and are likely to become a challenge for malware analysts and researchers. Opaque predicates and control flow flattening are obfuscation methods that are used to limit malware analysis by defining unused logic, performing needless calculations, and altering code flow so that it is not linear. Manual analysis of malware utilizing these obfuscations is painful and time-consuming.

ANEL (also known as UpperCut) is a RAT used by APT10, typically targeting Japan. All recent ANEL samples are obfuscated with opaque predicates and control flow flattening. In this paper I will explain how to de-obfuscate the ANEL code automatically by modifying the existing *IDA Pro* plug-in HexRaysDeob.

Specifically, the following topics will be included:

- Disassembler tool internals (*IDA Pro* IL microcode)
- How to define and track opaque predicate patterns for their elimination
- How to break control flow flattening while considering various conditional/unconditional jump cases even if it depends heavily on the opaque predicate conditions and has multiple switch dispatchers.

The modified tool is available publicly and this implementation has been found to deobfuscate approximately 92% of encountered functions in the tested samples. Additionally, most of the failed functions can be properly deobfuscated in I*DA Pro 7.3*. This provides researchers with an approach with which to attack such obfuscations, which could be adopted by other families and other threat groups.

INTRODUCTION

The *Carbon Black* Threat Analysis Unit (TAU) analysed a series of malware samples that utilized compiler-level obfuscations. For example, opaque predicates were applied to Turla Mosquito [1] and APT10 ANEL samples. Another obfuscation, control flow flattening, was applied to APT10 ANEL samples and the Dharma ransomware packer.

ANEL is a RAT program used by APT10 and is observed solely in Japan. According to *Secureworks* [2], 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 flow cleanly from beginning to end.

```
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;
101 LABEL_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).

Instead, a switch statement is called in a loop with multiple code blocks, each of which performs operations, as detailed later in this paper (see Figure 10).

The obfuscations used by ANEL looked similar to the ones described in the *Hex-Rays* blog [3], but the *IDA Pro* plug-in HexRaysDeob [4] didn't work for one of the obfuscated ANEL samples because the tool had been 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.

Figure 1 (shown on the previous page) shows an example of an obfuscated function; Figure 2 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: Deobfuscated result of the function shown in Figure 1.

TECHNICAL DETAILS

HexRaysDeob is an *IDA Pro* plug-in written by Rolf Rolles to address obfuscation seen in binaries. In order to perform the deobfuscation, the plug-in 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 [3]. 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 on.

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

```
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:
case m_seto: // cause INTERR 50862 -> replace in later maturity level
    iLocalRetVal = pat_InitedVarCondImm(ins, blk); // added
    break;
case m_sets:
    iLocalRetVal = pat_InitedVarSubImmCond0(ins, blk); // added
    break;
case m_mov:
```

Figure 3: Opaque predicates pattern-matching functions switch.

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 mainly defined two patterns for analysis of the ANEL sample.

Pattern 1: ~(x * (x - 1)) | -2

Figure 4 shows an example of one of the opaque predicates patterns used by ANEL.

```
13  v2 = ~((_BYTE)dword_745BB58C * ((_BYTE)dword_745BB58C - 1)) | 0xFFFFFFFE;
14  v3 = 0x7157F526;
```

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 shown in Figure 5.

```
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 using the `is_loaded` API) and only has 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 may be different, such as 9.

Data-flow tracking for the patterns

We also observed a pattern that was also using an eight-bit portion of the register. In the example shown in Figures 6 and 7, 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.

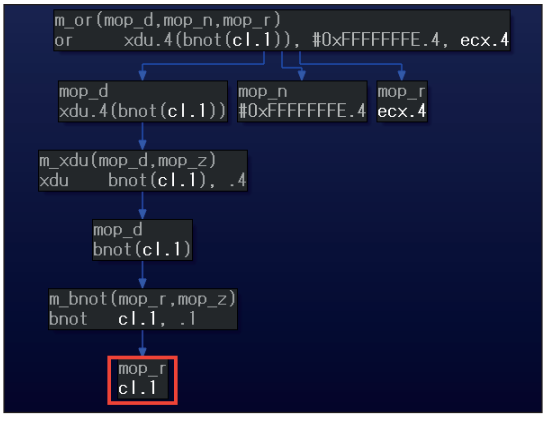

Figure 7: Register operand (microcode) in pattern 1.

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

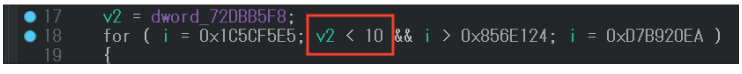

Figure 8: Register operand (pseudocode) in pattern 2.

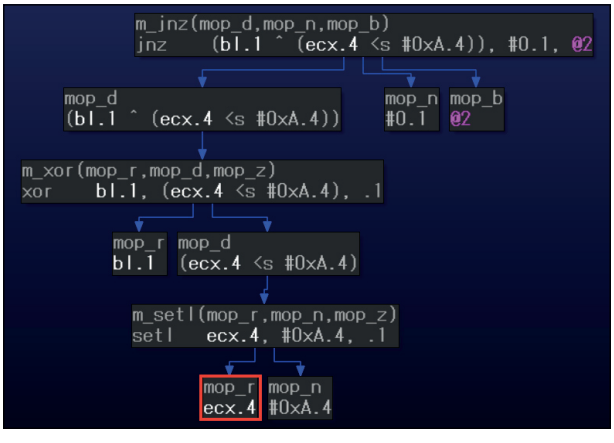

Figure 9: Register operand (microcode) in pattern 2.

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) as a top-level instruction and `setl` (m_setl) as a sub-instruction. If 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 the MMAT_LOCOPT (local optimization and graphing are complete) maturity level. Rolles previously described the unflattening algorithm in a *Hex-Rays* blog. For brevity, I will quickly cover some key points in order 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 that starts with the yellow-coloured 'control flow dispatcher' shown in Figure 10 after the First Block.

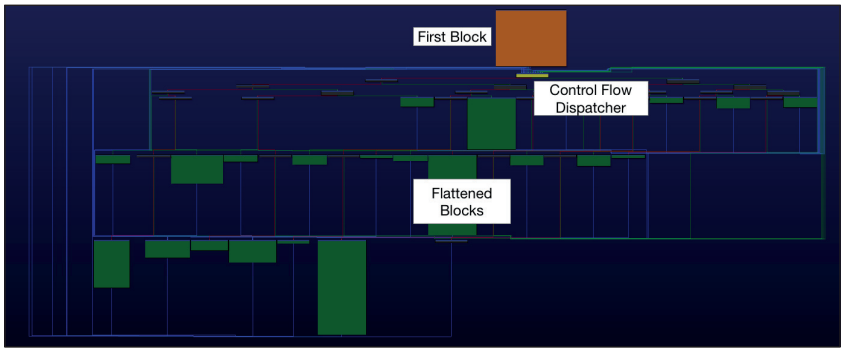

Figure 10: Function obfuscated with control flow flattening.

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

The next portion of the 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.

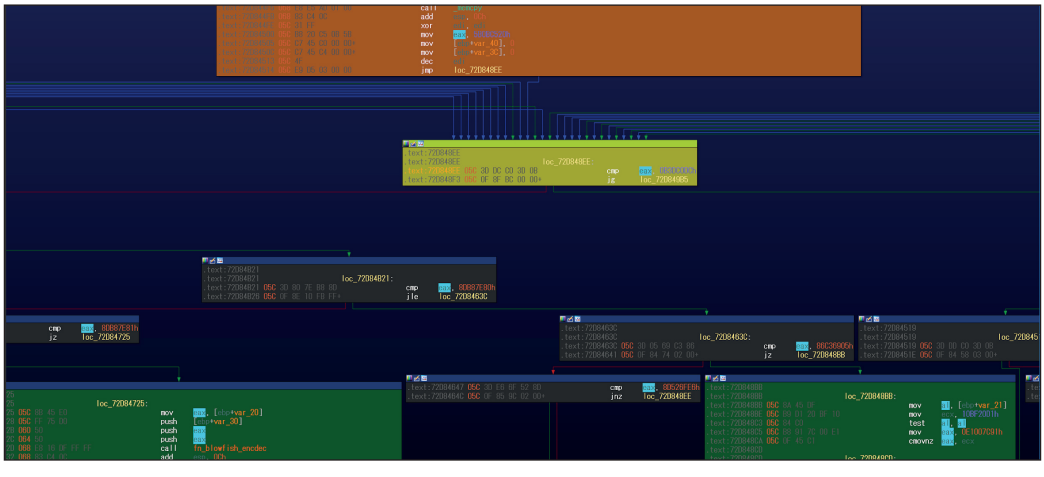

Figure 11: Block comparison variable example (blue-highlighted eax in this case).

If the variable registers for the comparison and assignment are different, the assignment variable is called the 'block update variable' (which will be described later).

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

Unflattening in multiple maturity levels

As previously described, the original implementation of the code only works in the MMAT_LOCOPT maturity level. Rolles said this was to handle another obfuscation called 'Odd Stack Manipulations', which he refers to in his blog. However, the unflattening of the ANEL code had to be performed in the later maturity level since the assignment of block comparison variables depends heavily on opaque predicates.

As an example, in the obfuscated function shown in Figure 12, the `v3` and `v7` variables are assigned to the block comparison variable (`b_cmp`). The values are dependent on the results of the opaque predicates.

```
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, as shown in Figure 13.

```
result = value;
for ( b_cmp = 0x4624F47C; ; b_cmp = 0xC504A26C )
{
  while ( b_cmp <= 0x4624F47B )
    b_cmp = 0x7157F526;
  if ( b_cmp == 0x7157F526 )
    break;
}
return result;
}
```

Figure 13: Simple obfuscated function example (opaque predicates deleted).

However, 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 the 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 shown in Figure 14, the blue-highlighted immediate value `0x4624F47C` is assigned to the block comparison variable in the first block. The mapping can be created by checking the conditional jump instruction (`jnz`) in MMAT_LOCOPT.

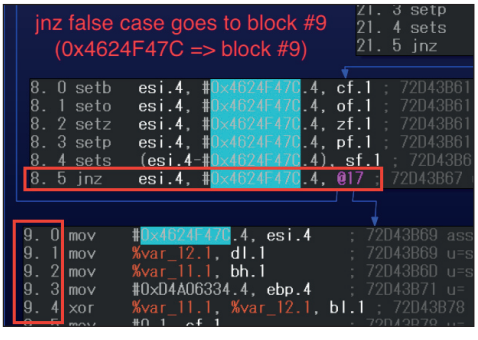

Figure 14: Mapping between block comparison variable 0x4624F47C and block number 9.

On the other hand, there 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 cannot be determined.

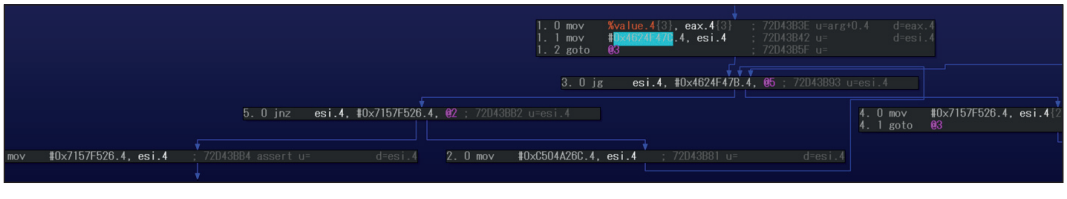

Figure 15: Mapping failure.

To resolve this 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 address. The guessing is not 100% accurate, however it works for nearly all of the 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 block address and number translation.

Figure 17 shows the final result of the deobfuscation in this case. The function just returns the argument value.

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

Figure 17: The result of the deobfuscation.

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 modified 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, the *Hex-Rays* kernel doesn't optimize some functions in MMAT_GLBOPT2 if it judges that 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 the `optblock_t::func` callback. Typically, this event occurs a few 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 the control flow dispatcher and first block, validating a block comparison variable, and so on).

Figures 18-23 show the functions with multiple control flow dispatchers that can be unflattened after the modification. Figures 18-20 show the case of two control flow dispatchers; Figures 21-23 show the case of seven 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) )
 101LABEL_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 )
```

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

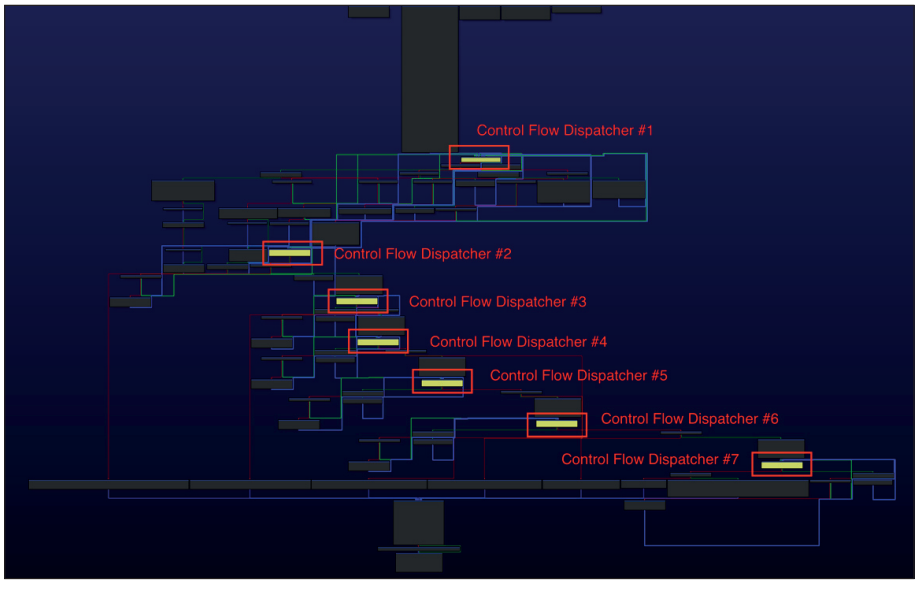

Figure 21: Example #2 with seven control flow dispatchers (graph).

```
266         v25 = fn_CMD_load_PE(v13, &v45, lpThreadParameter);
267       }
268       else
269       {
270         fn_store_qword(&cmd_id_to_compare, 1958040184, 660168479);
271         v30 = -882939367;
272         v51 = v49;
273         v52 = cmd_id_to_compare.ptr_or_str;
274         while ( 1 )
275         {
276           v31 = v30 & 0x7FFFFFFF;
277           if ( (v30 & 0x7FFFFFFF) == 630157743 )
278             break;
279           if ( v31 == 657560407 )
280           {
281             v30 = -1517325905;
282             LOBYTE(v13) = v50 == cmd_id_to_compare.field_4;
283           }
284           else if ( v31 == 1264544281 )
285           {
286             v30 = -1517325905;
287             if ( v51 == v52 )
288               v30 = -1489923241;
289             v13 = 0;
290           }
291         }
292         if ( v13 & 1 )
293         {
294           v56 = 1;
295           v25 = fn_CMD_upload_and_run(v13, &v45, lpThreadParameter);
296         }
297         else
298         {
299           fn_store_qword(&cmd_id_to_compare, -2047140637, -762289556);
300           v32 = -882939367;
301           v51 = v49;
302           v52 = cmd_id_to_compare.ptr_or_str;
303           while ( 1 )
304           {
305             v33 = v32 & 0x7FFFFFFF;
306             if ( (v32 & 0x7FFFFFFF) == 630157743 )
307               break;
308             if ( v33 == 657560407 )
309             {
310               v32 = -1517325905;
311               LOBYTE(v13) = v50 == cmd_id_to_compare.field_4;
312             }
313             else if ( v33 == 1264544281 )
314             {
315               v32 = -1517325905;
316               if ( v51 == v52 )
317                 v32 = -1489923241;
318               LOBYTE(v13) = 0;
319             }
320           }
321           if ( v13 & 1 )
322           {
323             v56 = 1;
324             v25 = fn_CMD_sleep(&v45, lpThreadParameter);
325           }
326           else
```

Figure 22: Example #2 with seven control flow dispatchers (before).

```
138       {
139         v44 = 1;
140         v18 = fn_CMD_load_PE(v20, &v33, lpThreadParameter);
141       }
142       else
143       {
144         fn_store_qword(&cmd_id_to_compare, 1958040184, 660168479);
145         v39 = v37;
146         v40 = cmd_id_to_compare.ptr_or_str;
147         if ( v37 == cmd_id_to_compare.ptr_or_str )
148           v21 = v38 == cmd_id_to_compare.field_4;
149         else
150           v21 = [illegible];
151         if ( v21 )
152         {
153           v44 = 1;
154           v18 = fn_CMD_upload_and_run(v21, &v33, lpThreadParameter);
155         }
156         else
157         {
158           fn_store_qword(&cmd_id_to_compare, -2047140637, -762289556);
159           v39 = v37;
160           v40 = cmd_id_to_compare.ptr_or_str;
161           if ( v37 == cmd_id_to_compare.ptr_or_str )
162             v22 = v38 == cmd_id_to_compare.field_4;
163           else
164             v22 = [illegible];
165           if ( v22 )
166           {
167             v44 = 1;
168             v18 = fn_CMD_sleep(&v33, lpThreadParameter);
169           }
170           else
171           {
172             fn_store_qword(&cmd_id_to_compare, -806312153, 1083996809);
173             v39 = v37;
174             v23 = cmd_id_to_compare.ptr_or_str;
175             v40 = cmd_id_to_compare.ptr_or_str;
176             if ( v37 == cmd_id_to_compare.ptr_or_str )
177               v24 = v38 == cmd_id_to_compare.field_4;
178             else
179               v24 = [illegible];
180             v44 = 1;
181             if ( v24 )
182             {
183               fn_CMD_take_screenshot_png(&v33);
184               v18 = -1000;
185             }
186             else
187             {
188               LOBYTE(v23) = v38 == cmd_id_to_compare.field_4;
189               fn_w_bs_free(&cmd_id_to_compare, v23);
190               v44 = 4;
191               v18 = fn_CMD_cmd_exe(&v33, lpThreadParameter, &cmd_id_to_compare);
192               fn_bs_free_0(&cmd_id_to_compare);
193             }
194           }
195         }
196       }
```

Figure 23: Example #2 with seven control flow dispatchers (after).

```
[I] MMAT_GLBOPT1: found first block id 3 (ea=0x7454e6ef), dispatcher id 14 (ea=0x7454e88f)
[I] MMAT_GLBOPT1: Comparison variable = esi
[I] MMAT_GLBOPT1: Update variable=12 (edx) is assigned by AND instruction with 7fffffff
[I] MMAT_GLBOPT1: register numbers: comparison variable=36 (esi), update variable=12 (edx)
[I] MMAT_GLBOPT1: Inserting 27319357 -> block ID 13 into map (the start ea 7454e888)
[I] MMAT_GLBOPT1: Inserting 4b5f6a19 -> block ID 10 into map (the start ea 7454e876)
[I] MMAT_GLBOPT1: Inserting 258f71af -> block ID 15 into map (the start ea 7454e89f)
[I] MMAT_GLBOPT1: Ignoring 27319357 -> block ID 23 due to multiple dispatchers (the start ea 7454e915)
[I] MMAT_GLBOPT1: Ignoring 4b5f6a19 -> block ID 20 due to multiple dispatchers (the start ea 7454e903)
[I] MMAT_GLBOPT1: Ignoring 27319357 -> block ID 33 due to multiple dispatchers (the start ea 7454e9a2)
[I] MMAT_GLBOPT1: Ignoring 4b5f6a19 -> block ID 30 due to multiple dispatchers (the start ea 7454e990)
[I] MMAT_GLBOPT1: Ignoring 27319357 -> block ID 43 due to multiple dispatchers (the start ea 7454ea2f)
[I] MMAT_GLBOPT1: Ignoring 4b5f6a19 -> block ID 40 due to multiple dispatchers (the start ea 7454ea1d)
[I] MMAT_GLBOPT1: Ignoring 27319357 -> block ID 53 due to multiple dispatchers (the start ea 7454eabc)
[I] MMAT_GLBOPT1: Ignoring 4b5f6a19 -> block ID 50 due to multiple dispatchers (the start ea 7454eaaa)
[I] MMAT_GLBOPT1: Ignoring 27319357 -> block ID 63 due to multiple dispatchers (the start ea 7454eb49)
[I] MMAT_GLBOPT1: Ignoring 4b5f6a19 -> block ID 60 due to multiple dispatchers (the start ea 7454eb37)
```

Figure 24: Ignoring duplicated block comparison variables.

Normally, block comparison variables used by the control flow flattening are unique in a function. Therefore, block numbers corresponding to the variables can be determined uniquely as well.

However, the function in the latter case contains duplicated block comparison variables due to multiple dispatchers. The modified code detects the duplications and applies the most likely variable.

Implementation for various conditional/unconditional jump cases

As shown in Figure 25, the original implementation supports two cases (1)-(2) of flattened blocks to find a block comparison variable for the next block (the cases are then simplified). In the second case, the 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 the `goto` jump instruction are updated.

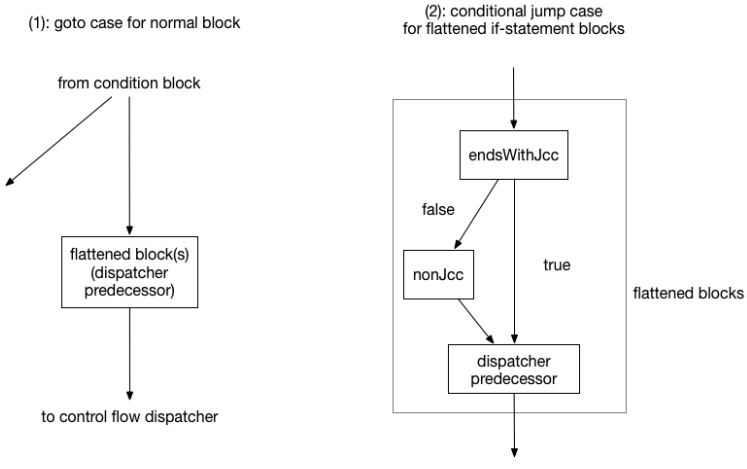

Figure 25: Originally supported two cases of blocks.

We found and implemented three more cases in the ANEL sample. Of these, two cases (3)-(4) are shown in Figure 26.

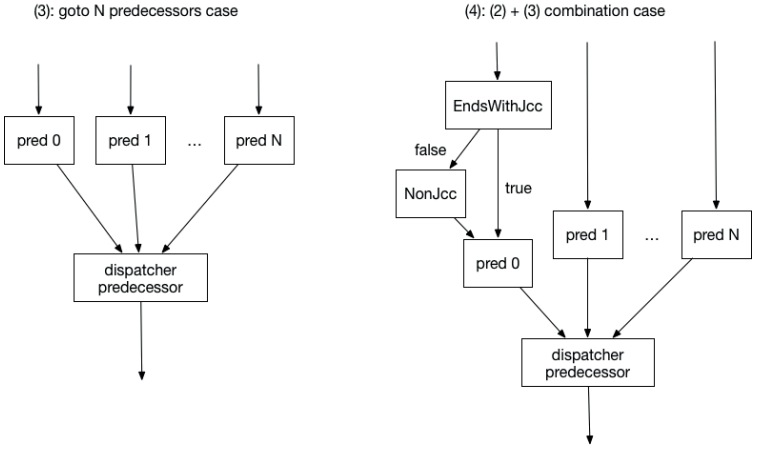

Figure 26: Newly supported two cases of blocks.

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

In the jump case (5) that was newly implemented, the block comparison variables are not assigned in the flattened blocks but rather the first blocks according to a condition. For example, the microcode graph depicted in Figure 27 shows that edi is assigned to esi (the block comparison variable in this case) in block number 7, but the edi value is assigned in block numbers 1 and 2.

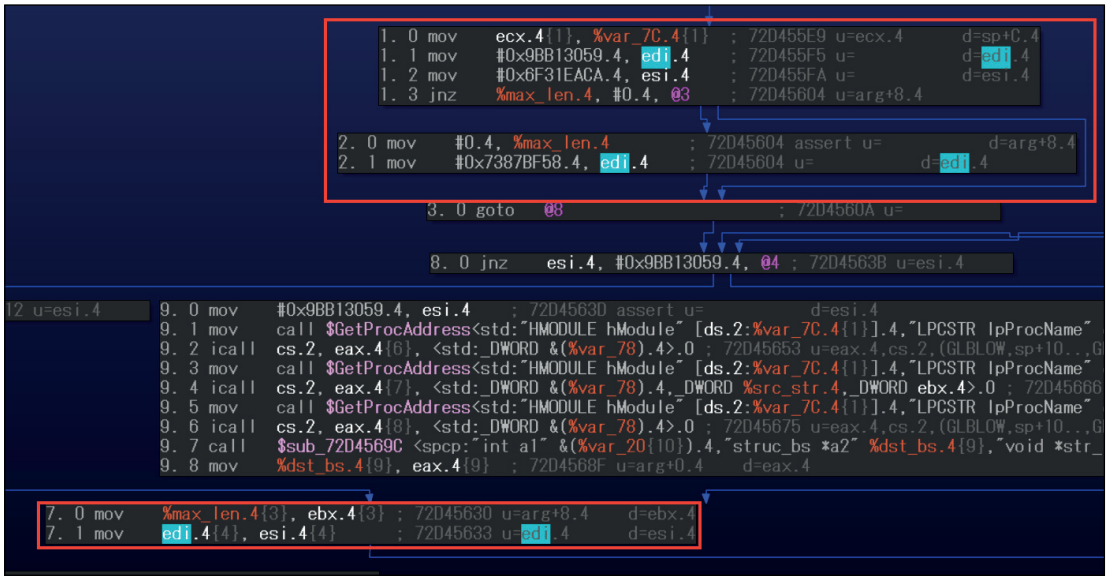

Figure 27: Newly supported case (assigned in first blocks).

If the immediate value for the 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 numbers 1 and 2 as successors of block number 7, in addition to the normal operations mentioned in the original cases.

Another example function, shown in Figures 28 and 29, did the same processing twice.

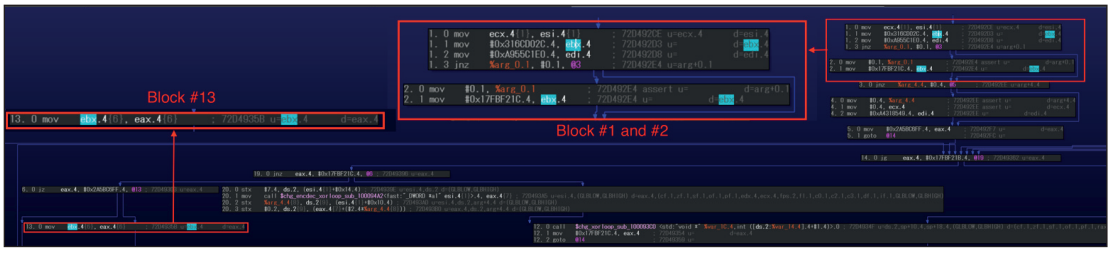

Figure 28: Newly supported case (assigned in first blocks twice #1).

Figure 29: Newly supported case (assigned in first blocks twice #2).

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

Last, but not least, in all cases described 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. However, the handling of conditional jumps in cases (2)-(4) requires more complicated operations and the latest *IDA* (version 7.2) at the time was not able to process them. It will be detailed below.

Other minor changes

The following changes are minor compared with the above referenced ones.

- Additional jump instructions are supported when collecting block comparison variable candidates and mapping between the variable and ea (linear address) 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 a block comparison variable

Additionally, two more changes were introduced in regards to the block comparison/update variables referenced in the overview. First, some functions in the ANEL sample utilize a block update variable, however the assignment is a little bit tricky, as shown in Figure 30.

Figure 30: Block update variable usage with and instruction.

By using the `and` instruction, the immediate values used in comparison look different from assigned ones. Second, in a different case, a small number of functions utilize dual block comparison variables, as shown in Figure 31.

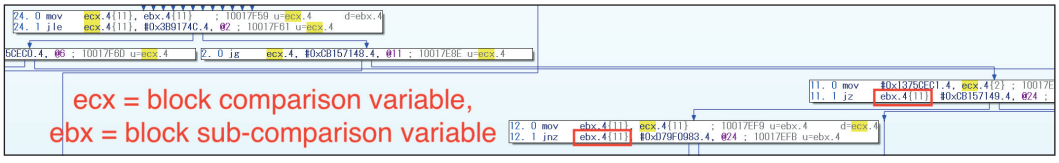

Figure 31: A function with dual block comparison variables (microcode).

The modified code will consider both of the cases.

REMAINING ISSUES AND IMPROVEMENTS IN IDA 7.3 BETA

The modified tool was tested on *IDA 7.2* with an ANEL 5.4.1 payload dropped from a malicious document with the following hash (previously reported by *FireEye* [5]):
3d2b3c9f50ed36bef90139e6dd250f140c373664984b97a97a5a70333387d18d

The current code was able to deobfuscate 92% of the obfuscated functions that we encountered. In the 8% of cases where deobfuscation failed, the failure was caused by any of the three following issues:

1. The next block number guessing algorithm failed.
2. *IDA* didn't propagate the results after defeating opaque predicate patterns.
3. There was no method to handle a conditional jump of a dispatcher predecessor with multiple predecessors.

The first issue has already been resolved but may be problematic in the future as the approach is not 100% accurate. The guessing algorithm will be improved every time a new issue in it is found. However, the other issues were reported to *Hex-Rays* and resulted in an *IDA 7.3* beta version to address them. In the following sections, the issues and their solutions will be discussed.

Additionally, the tool also worked for the following ANEL 5.5.0 rev1 payload loader DLL published by *Secureworks* [6]: f333358850d641653ea2d6b58b921870125af1fe77268a6fdfeda3e7e0fb636d.

Correct propagation of opaque predicates deobfuscation result

The *IDA 7.2* decompiler does not propagate aliased stack slots. In the example shown in Figure 32, the variables `true1` and `true2` are aliased. Thus the results after breaking opaque predicates are not propagated even if an immediate value 1 is assigned to them.

The *IDA 7.3* beta decompiler resolving this issue is able to deobfuscate the function correctly, as shown in Figure 33.

Handling a conditional jump of a dispatcher predecessor with multiple predecessors

As described previously, more complicated operations are required to handle the cases (2)-(4) of flattened blocks if the dispatcher predecessor's tail instruction is a conditional jump. For instance, in case (3), let's consider a dispatcher predecessor with two predecessors.

```
17  true1 = 1;                                  // opaque predicate result 1
18  true2 = 1;                                  // opaque predicate result 2
19  v2 = 0x1D3E02CA;
20  if ( true2 )
21    v2 = 0xC1A18C30;
22  if ( !true1 )
23    v2 = 0x1D3E02CA;
24  if ( true2 == true1 && v2 > 0x1D3E02C9 )    // never executed block (v2 == 0xC1A18C30)
25  {
26    savedregs = 0x1D3E02CA;
27    v5 = fn_get_ptr_from_bs(src_bs);
28    StrToIntA(v5);
29    v6 = _gmtime64(&v8);
30    strftime(&v9, 0x50u, "%a, %d %b %Y %X", v6);
31    fn_w_bs_make_from_str(dst_bs, &v9);
32  }
33  savedregs = 0xC1A18C30;
34  v3 = fn_get_ptr_from_bs(src_bs);
35  StrToIntA(v3);
36  v4 = _gmtime64(&v8);
37  strftime(&v9, 0x50u, "%a, %d %b %Y %X", v4);
38  fn_w_bs_make_from_str(dst_bs, &v9);
39  return dst_bs;
40}
```

Figure 32: Opaque predicates deobfuscation result propagation failure on IDA 7.2.

```
13
14  true1 = 1;                                  // opaque predicate result 1
15  true2 = 1;                                  // opaque predicate result 2
16  savedregs = 0xC1A18C30;
17  v2 = fn_get_ptr_from_bs(src_bs);
18  StrToIntA(v2);
19  v3 = _gmtime64(&v5);
20  strftime(&v6, 0x50u, "%a, %d %b %Y %X", v3);
21  fn_w_bs_make_from_str(dst_bs, &v6);
22  return dst_bs;
```

Figure 33: Opaque predicates deobfuscation result propagation success on IDA 7.3 beta.

Handling a `goto` case (unconditional jump case, in Figure 34) is straightforward. The implementation searches block comparison variables in `pred0` and `pred1` (predecessor #0 and #1) separately then resolves the next block numbers in microcode according to the variables. After that, it changes each destination in both CFG and instruction levels while appending the codes of a dispatcher predecessor to each predecessor. As a result, the dispatcher predecessor block will be eliminated.

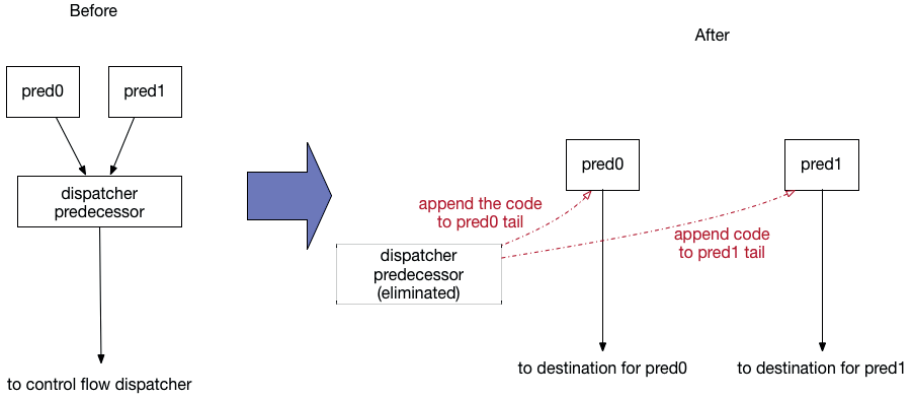

Figure 34: Before and after goto case with two predecessors.

However, in a conditional jump (Figure 35), the destination of the true case is replaced with resolved next block numbers, however two blocks of the dispatcher predecessor and its false case block should be copied for pred1 as a false case block number must be its conditional jump's block number plus 1 in *IDA* microcode (`dispatcher predecessor + 1` in this case). Therefore, the false case block cannot be shared by the two predecessors.

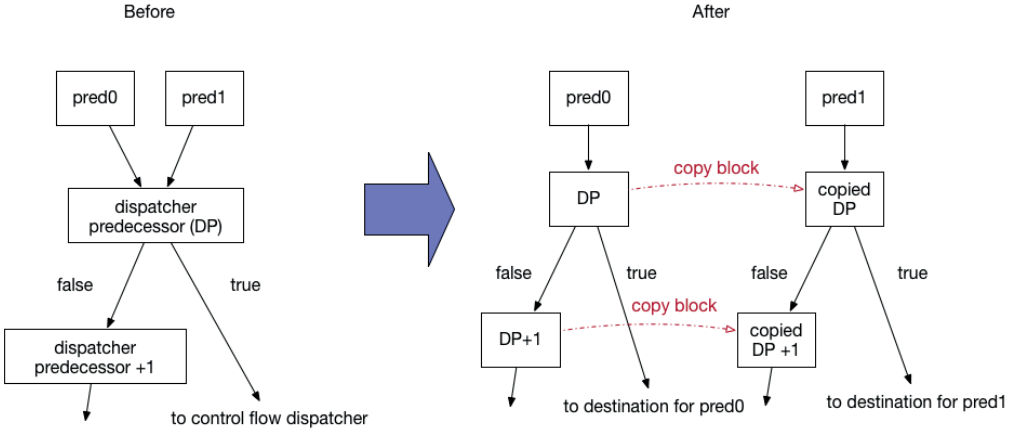

Figure 35: Before and after conditional jump case with two predecessors.

Additionally, in Figure 34, if `pred0` or `pred1` contains a conditional jump, the dispatcher predecessor will be copied in the same way regardless of its tail instruction because a conditional jump instruction cannot be overwritten by a `goto` one.

IDA 7.2 doesn't permit overlapped instructions by copying a microcode block (mblock_t) as many instructions must have distinct addresses. Duplicated instructions are allowed in *IDA 7.3* by clearing the flag `MBA2_NO_DUP_CALLS`. The latest code utilizes its flag and handles cases (2)-(4) with conditional jumps correctly.

Specifically, the code makes an empty block by using the `mbl_array_t::insert_block` API then copies instructions and other information such as flags and start/end addresses from the original block. The code also has to adjust CFGs and instructions of the blocks, passing control to the exit block whose block type is `BLT_STOP` if CFG updates by the API usage or the unflattening code cause a conflicted situation.

Workaround in control flow unflattening failure

If an obfuscated function contains any of the issues described in this section, the decompiled code result may be paradoxical or lose multiple code blocks. In this case, try to use the following IDAPython command in the 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 through control flow unflattening. For instance, Figures 36 and 37 show how the pseudocode changes in one of the failure cases.

```
1 size_t __cdecl fn_cause_crash(void *a1, void *a2, size_t a3)
2 {
3   bool v5; // [esp+57h] [ebp-19h]
4
5   if ( !a1 )
6     return a3;
7   v5 = (a3 & 0xF) != 0;
8   while ( !v5 )
9     ;
10  return 0;
11 }
```

Figure 36: One failure case pseudocode (before).

```
41    v5 = -1424257784;
42  v7 = 683910303;
43  v21 = v5;
44  if ( !a4 )
45    v6 = 639893090;
46  if ( !a5 )
47    v7 = 2094064936;
48  v22 = v7;
49  v27 = &v23;
50  v8 = -447934511;
51  while ( 1 )
52  {
53    while ( 1 )
54    {
55      while ( 1 )
56      {
57        while ( 1 )
58        {
59          while ( 1 )
60          {
61            v15 = v8;
62            if ( v8 <= 21690082 )
63              break;
64            if ( v8 > 1127530844 )
65            {
66              if ( v8 <= 1518054240 )
67              {
68                if ( v8 > 1278155936 )
69                {
70                  if ( v8 == 1278155937 )
71                  {
72                    memset(&v23, v39, v34);
73                    v25 = &v26[-528571521 - v34 + 528571522];
74                    v14 = memcmp(&v23, v25, v34) == 0;
75                    v8 = -35098432;
76                    if ( !v14 )
77                      v8 = 496748478;
78                  }
79                  else
80                  {
81                    v8 = 1127530845;
82                  }
83                }
84                else
85                {
```

Figure 37: 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

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

TAU modified the original *HexRaysDeob* plug-in to make it work for APT10 ANEL obfuscations. The modified code is available publicly [7]. 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.

This tool works for most obfuscated functions in the tested samples. This implementation can deobfuscate approximately 92% of encountered functions. Additionally, most of the failed functions will be properly deobfuscated in *IDA 7.3*.

It should be noted that the tool may not work for 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 version 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.

ACKNOWLEDGEMENT

First I acknowledge *Hex-Rays* for supporting the research patiently. Next, I appreciate Rolf Rolles for releasing the original version of HexRaysDeob. Last but not least, I would like to thank TAU's members, especially Jared Myers and Brian Baskin, for proofreading and giving a lot of feedback.

REFERENCES

[1] Diplomats in Eastern Europe bitten by a Turla mosquito. ESET. https://www.welivesecurity.com/wp-content/uploads/2018/01/ESET_Turla_Mosquito.pdf.

[2] APT10によるANELを利用した攻撃手法とその詳細解析. Secureworks. https://jsac.jpcert.or.jp/archive/2019/pdf/JSAC2019_6_tamada_jp.pdf.

[3] Rolles, R. Hex-Rays Microcode API vs. Obfuscating Compiler. http://www.hexblog.com/?p=1248.

[4] Rolles, R. Hex-Rays microcode API plugin for breaking an obfuscating compiler. https://github.com/RolfRolles/HexRaysDeob.

[5] APT10 Targeting Japanese Corporations Using Updated TTPs. FireEye. https://www.fireeye.com/blog/threat-research/2018/09/apt10-targeting-japanese-corporations-using-updated-ttps.html.

[6] ANELで日本を標的とした攻撃活動を行う攻撃者グループ BRONZE RIVERSIDEに関する調査報告. Secureworks. https://www.secureworks.jp/resources/at-bronze-riverside-updates-anel-malware.

[7] HexRaysDeob for APT10 ANEL. Carbon Black. https://github.com/carbonblack/HexRaysDeob.