2022-09-16

Unflattening ConfuserEx .NET Code in IDA

NCSC · **GovCERT.ch**

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Eidgenössisches Finanzdepartement EFD
Nationales Zentrum für Cybersicherheit NCSC
GovCERT.ch

Contents

This paper refers to a Ginzo sample, which can be downloaded from MalwareBazaar using MD5 5009e04920d5fb95f8a02265f89d25a5. Ginzo is an information stealer written in .NET, known to also steal cryptocurrency keys. However, we are not focusing on the malware itself, but instead have a closer look on one of it's *ConfuserEx*[1] obfuscation technique.

Initial Unpacking and Motivation

The sample has a first simple encryption layer that can be unwrapped by using `dnSpy` as a debugger, dumping the decrypted code to disk (e.g. using `x64dbg`), and patching the new code section back into the original sample. The resulting code still has long class/function names and *flattened* code, so the next usual step to try in such situations is to run `de4dot`[2] on it:

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Unknown Obfuscator (.../ginzo-patched.exe)
Cleaning .../ginzo-patched.exe
Renaming all obfuscated symbols
Saving .../ginzo-patched-cleaned.exe
ERROR: Error calculating max stack value. ...
Ignored 8 warnings/errors
Use -v/-vv option or set environment variable SHOWALLMESSAGES=1
to see all messages
```

The resulting .NET binary `ginzo-patched-cleaned.exe` now has readable symbol names, but unfortunately the ConfuserEx obfuscated code has not been unflattened; `de4dot` can actually unflatten some ConfuserEx samples, but not all of them. For these situations, there is an additional tool *ConfuserExSwitchKiller*[3], which can reorder the `switch` structures of our sample in its original way. However, such obfuscation schemes can change quickly, and it is very well possible to come across samples, where no ready-to-use unpacker tool exists. E.g., more than 10 years ago a banking trojan known as *Torpig*[4] made heavy use of a similar, if not even more advanced technique on x86 assembly level. That's why we'll study how to deal with such a switch obfuscator by writing our own Python script in *IDA Pro*[5] to unflatten code like this. Of course, the script needs to be modified for changed requirements. The main goal is to be able to decompile the unflattened code using `dnspy`, or at least produce enough information to follow the `switch` statements ourselves, without the need to do all tedious calculations manually.

Unfortunately, the tools available for .NET assembly reverse engineering are not as good as for native binaries (unmanaged code). The most widely used tool - `dnspy` - is no longer maintained for several years and does not allow to actually disassemble CIL - it only offers decompiled code. There are several .NET libraries that can disassemble .NET[6], which basically is what the famous `de4dot` deobfuscator tool is doing. This however forces reversere engineers to write such code in C# itself. Most reversers prefer to

[1] https://mkaring.github.io/ConfuserEx/ and https://github.com/yck1509/ConfuserEx
[2] https://github.com/de4dot/de4dot
[3] https://github.com/VAllens/ConfuserExSwitchKiller
[4] https://de.wikipedia.org/wiki/Torpig
[5] Interactive Disassembler, a well known reverse engineering tool, see https://hex-rays.com/
[6] E.g. https://github.com/0xd4d/dnlib or https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/

write Python code, and many are familiar with IDA, which offers a Python scripting interface. Also, IDA can disassemble CIL just fine, but does not offer a decompiler for it. More seriously, IDA can't directly assemble/patch/modify CIL code. So, the .NET support in IDA is not (yet?) where we'd like it to be, but it nevertheless offers the required features for our job. We had to use some workarounds to deal with the lacking .NET support in IDA. But as .NET becomes more and more popular in malware - similar to malware written in the Go language - we need to find a way to deal with IDAs shortcomings.

Code Flattening After Decompilation

If we just export the binary to a project using `dnspy` and study at the decompiled code, it looks as follows in a simple situation - we often see dozens of `case` blocks instead:

```
internal void method_0()
{
   uint num = 1U;
   for (;;)
   {
      IL_91:
      uint num2 = 2666110116U;
      for (;;)
      {
         uint num3;
         switch ((num3 = (num2 ^ 2708435411U)) % 6U)
         {
         case 0U:
            goto IL_91;
         case 1U:
            num2 = (num3 * 1468221071U ^ 3530660000U);
            continue;
         case 2U:
            this.struct0_0[(int)num].method_0();
            num2 = 2183023298U;
            continue;
         case 4U:
            num2 = (((ulong)num < (ulong)(1L << (this.int_0 & 31))) ? \
                   3536790039U : 2440591868U);
            continue;
         case 5U:
            num += 1U;
            num2 = (num3 * 2529799854U ^ 3879142487U);
            continue;
         }
         return;
      }
   }
}
```

The basic algorithm works like this:

- One *state variable* is created (`num2` with an alias `num3`, created by the decompiler). In a more generic scenario, several, if not dozens such variables can be used as state variables, declared and initialized step by step at different locations, spread all over the `case` blocks. An example for this is the aforementioned Torpig trojan. This introduces the additional problem for a deobfuscator to differ between state variables and normal variables. But fortunately, ConfuserEx only uses one state variable.

- At the start of every loop iteration, the next *state* is calculated using this variable (e.g. `(num2 ^ 2708435411U))% 6U`), always using the same operation: the state variable is updated using an XOR operation with a constant, and then a modulo operation with the total number of states tells us which state to go to next. Because this calculation happens during runtime, it is not trivial to find out which state follows one given state, without actually running or emulating the code. So, the code structure (different kind of loops, conditional statements, etc) disappears and is replaced as one single loop containing a single `switch`. Such a technique is known as *code flattening*.
- The code for each state (inside the `case` code fragments) starts with some (optional) "real" code, followed by a "control" code fragment, updating the state variable in one of four different manners, which we call *suffix* types:
  - Suffix type `SIMPLE`: The state value is replaced by a completely new one (e.g. `num2 = 2183023298U`)
  - Suffix type `MXOR`: The state value is multiplied with a constant and then XORed with another constant (e.g. `num2 = (num3 * 1468221071U ^ 3530660000U)`)
  - Suffix type `BRANCH`: A condition is checked, then the state variable is set to a new value depending on the outcome. `dnspy` shows this as ternary operation (`... ? ... : ...`), e.g. `num2 = (((ulong)num < (ulong)(1L << (this.int_0 & 31)))? 3536790039U : 2440591868U)` in `case 4` above. The condition is "real" code, the rest is "control" code. In this case, the state variable is overwritten, similar to `SIMPLE`
  - Suffix type `BRANCH_MXOR`: This is a combination of `BRANCH` and `MXOR`. Above sample does not show this suffix type, but here is an example from a more complex case: `num2 = (((num < 16777216U)? 4284753547U : 2545021605U)^ num3 * 3864922473U)`. Again, a ternary operator checks a condition and chooses a constant depending on the outcome. This value is then XORed with the result of the multiplication of the old state variable with a constant - note that multiplication has a higher precedence than bitwise XOR in C#. This formula is identical to the one used with the `MXOR` type, except the XOR constant depends on the condition. One problem in analysis of such a state machine is that we can't calculate the next state - or next two states in case of one of the two branch types - by just statically looking at the code of one of the `case` blocks. While we do know the state itself - this is the `case` label after all - we do not know the content of the state variable before the modulo operation was applied, because modulo is a non-reversible operation. But the state variable content is needed to calculate the next state; knowledge of the state alone does not suffice. So, such a state machine needs to be fully emulated, using some sort of backtracking, to gain a full analysis.
- One state (`case 0` in above example) - acts as *initial* state, which usually just assigns the initial value (`num2 = 2666110116U`). The decompiler shows this state outside the `switch` with an additional `goto` label (`IL_91`) and creates a second `for (;;)` loop. However, this is just the way the decompiler shows this code in a try to make it look structured, but failing here. Replacing the `goto` in the `case` by a `goto` from the outside directly into the `case` code would probably work better. This would also allow to drop one of the two nested loops. Note that this initial state does not need to be of type `SIMPLE`, it can also be of type `BRANCH`.
- One or more state acts as *end* state (or leaf), which leaves the loop. In the above code, this is `case 3`. Here it does not show up as an explicit `case` label - but it does on level of MSIL instructions, as shown below. Instead, the `switch` is left for `case 3` as default action, and the following `return` statement

terminates the loop. Several end states are possible.

An analysis of above state machine (performed by the actual script described later on) results in:

```
Function Struct1::method_0 at VA 2310
  Switch loop at VA 2378: XOR 2708435411
  Entry state 0
  0 [Address 23A1 - 23A1 START]
  1 [Address 236B - 2377]
  4 [Address 232A - 2350]
  IF:
    2 [Address 2353 - 2364]
    5 [Address 2317 - 2327]
    4 ^
  ELSE:
    3 [Address 23A8 - 23A8 END]
```

The acronym `VA` stands for *virtual address*, the memory address of code and data after being loaded, which is also shown in IDA. `VA`s need to be distinguished from the actual *raw file offset* in the sample. This shows that code runs from the initial state 0 (which usually does not contain any "real" code) to state 1 (containing no "real" code at all), then to state 4, where a branch to either state 2 or state 3 (the end state) occurs. State 2 leads to state 5 and then loops back to state 4 (the loop-back situation is indicated by a `^` character). State 4 probably implements a kind of `while` loop. Such an analysis allows us to *unflatten* the code manually, resulting in something similar to:

```
internal void method_0()
{
   uint num = 1u;
   while ((ulong)num < (ulong)(1L << (this.int_0 & 31)))
   {
      this.struct0_0[(int)num].method_0();
      num += 1u;
   }
}
```

This code was actually produced by `dnspy` after the deobfuscation script was applied, but in such simple cases a manual reassembly works just fine. Of course, this could better be expressed as a `for` loop.

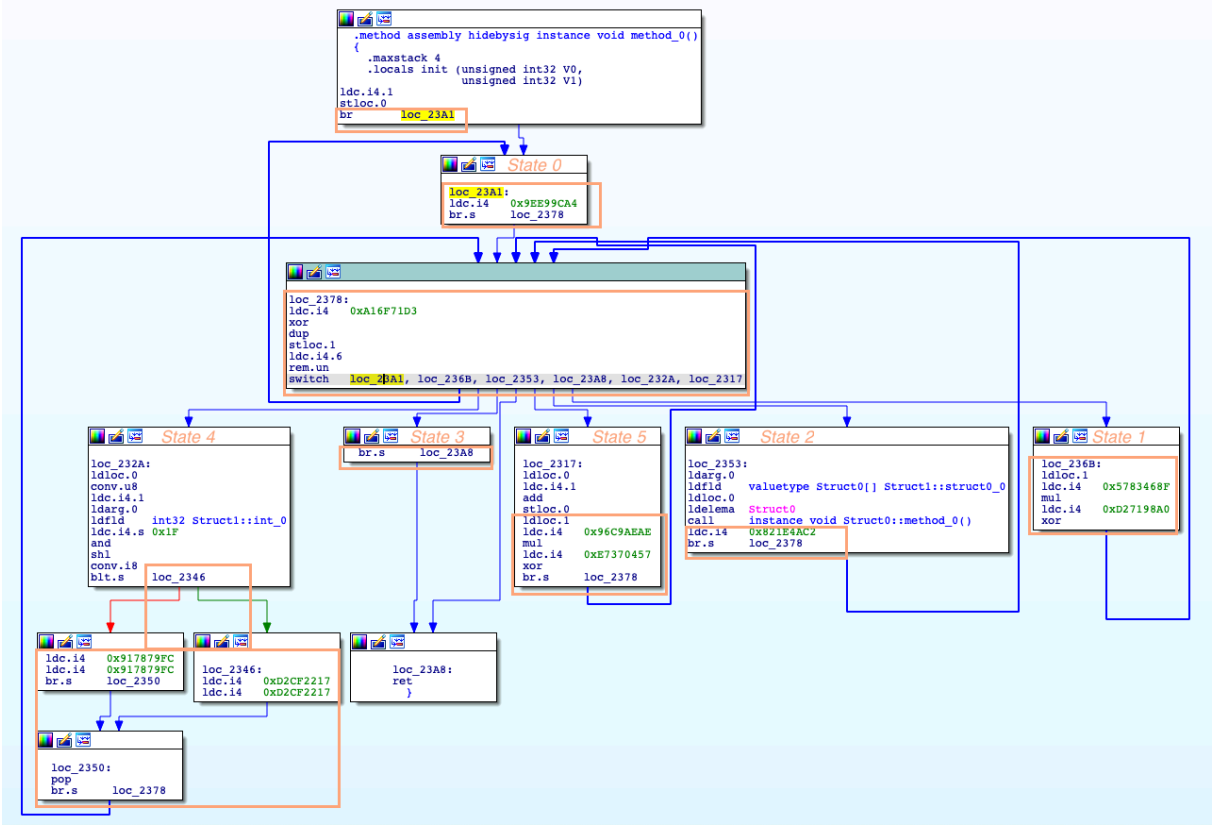
While one could write a deobfuscator using decompiled code as input using text processing, that would not be a very effective approach for several reasons:

- The decompiler might make different decisions in different similar situations, leading to the necessity to differ many strange situations and using complex regular expressions.
- The decompiler also often erroneously re-declares local variables inside the `case` blocks because the real control flow is not known to it, so the scopes of local variables become messy.
- `while`-loops would never be simplified to `for`-loops, because this is done by heuristics in `dnspy`, unless we re-implement those in our deobfuscator.
- This approach would just not be very satisfying….

A much better strategy is to deal with this on the level of *Microsoft Intermediate Language (MSIL)*, also known as the *Common Intermediate Language (CIL)*, .NET's assembly instruction set. The obfuscation structures are much clearer there, though still not as clear as we'd expect and wish - it does still seem to contain many "compiler optimization"-like code fragmentation. This also suggests the obfuscator itself operates on source code, and not on CIL instructions level directly.

Analysis of Code Flattening in CIL

Let's look at the flattening code elements on CIL level. CIL is a simple stack-based instruction set[7]. Op-codes are all just one byte, followed by optional operands. This is a screenshot of above method as it appears in IDA:



Control code fragments are surrounded by orange frames, and the actual `case` values for the states are written in orange over the relevant blocks. The "spider in the web" of the structure is the `switch` statement in the middle with the preceding loop iteration code implementing the shared next-state-variable-calculation - it expects the previous value of the state variable on the stack. This code fragment actually appears in two different small variations. In this more common case, we have:

```
ldc.i4   0xA16F71D3   // push 2666110116 on the stack
                      // (above the previous state variable)
```

[7] https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes?view=net-6.0

```
xor                     // XOR with previous state variable
dup                     // Now we have 2 copies of the new state variable
stloc.1                 // store in local variable 1 (reserved for the
                        // state variable) - the other copy remains on stack
ldc.i4.6                // calculate state variable modulo 6 (number of
                        // states), giving the next state
rem.un                  // modulo operator
switch   loc_23A1, loc_236B, loc_2353, loc_23A8, loc_232A, loc_2317
```

Some variations we see in other places:

- `ldc.i4.6` pushes the value `6` on the stack. Because `6` is a small number, a special opcode using only one byte can be used. For larger values, this can show as `ldc.i4.s 9` (for values up to 127, using two bytes), or even `ldc.i4 0x100` (using five bytes). These two cases both occasionally happen, so we must consider them. Of course, the obviously "random" constant(s) (like `ldc.i4 0xA16F71D3`) could also be expressed in one of the other variations for small values, but this is probably rare, so we ignore that possibility for simplicity. It could have consequences as how many bytes remain available for patching in our own code (one, two, or five bytes). The probability for these random constants to be "small" by accident is roughly 1 to 17 million (24 bits).

- In some rare cases (e.g. at VA `3F12`), we see something similar to:

  ```
  ldc.i4   0x98EFB846
  xor
  ldc.i4.4
  rem.un
  switch   loc_3F3D, loc_3F47, loc_3F0D, loc_3EF1
  ```

  In this case, the state variable is not stored in a local variable of its own. As the new state variable is actually eaten away by `rem.un`, this is only possible if all state variable calculations in the subsequent switch blocks are of types `SIMPLE` or `BRANCH`, so the previous values need not be remembered. This modification looks like a compiler optimization due to the fact the state variable is not read again later on. Once more, this suggests the obfuscator does its work on source code level. We only see this in very small code blocks.

One interesting and helpful observation is that the `switch` statement is always followed by an unconditional branch to the end state, `br.s loc_23A8` above (marked with `State 3` as it is the default if nothing matches), which branches to the final `ret` statement. So, the tagging as "*state 3*" above is not completely true, as it in fact only jumps there. This instruction does not seem to be required though, as the switch instruction above is complete and no `default` is needed (it's label for state 3 actually also points to `ret`), but it help our deobfuscator to identify the "normal" exit state.

For the different types, we see different code patterns at the end of the blocks, ignoring the end state 3:

- `SIMPLE` (e.g. state 2):

```
ldc.i4   0x821E4AC2   // push completely new state variable on the stack
```

- `MXOR` (e.g. state 5 or state 1)

```
ldloc.1               // push previous state variable on the stack
ldc.i4   0x96C9AEAE   // push multiplier constant on stack
mul
ldc.i4   0xE7370457   // push xor constant on stack
xor                   // leaves new state variable on the stack
```

- `BRANCH` (e.g. state 4)

```
  blt.s    loc_2346   // blt.s is a "real" instruction, but jump
                      // target is "control" code
  ldc.i4   0x917879FC // ELSE part: put 2 copies of the "else" new
                      // state variable on the stack
  ldc.i4   0x917879FC
  br.s     loc_2350
loc_2346:             // IF part: put 2 copies of the "if" new
                      // state variable on the stack
  ldc.i4   0xD2CF2217
  ldc.i4   0xD2CF2217
loc_2350:
  pop                 // code flows combine, one copy is removed
                      // (duplication and pop happen for unknown reasons)
  br.s     loc_2378   // jump back to loop
```

This code also appears in a second layout with additional fragmentation, e.g. at VA 2773:

```
  bge.un   loc_26E6
  br.s     loc_275C
  ret                 // unrelated code in the same fct behind
  ...

loc_26E6:
  ldc.i4   0xBA2C2B6D
  ldc.i4   0xBA2C2B6D
loc_26F0:
  pop
  br.s     loc_2737
  ...

loc_275C:
  ldc.i4   0x8873AC65
  ldc.i4   0x8873AC65
  br.s     loc_26F0
```

The three sections can be torn apart and even appear in different order. The only relevant difference is the additional unconditional branch in the ELSE part after the conditional branch. This again looks like a compiler optimization and suggests obfuscation happened on source code level. These are the only two layouts we found in the sample.

- `BRANCH_MXOR` (e.g. at VA 2829)

```
  blt.un.s loc_2837     // same code as in BRANCH, leaving an XOR constant
                        // (depending on test) on the stack
  ldc.i4   0x97B1F2A5
  ldc.i4   0x97B1F2A5
  br.s     loc_2841
loc_2837:
  ldc.i4   0xFF64268B
  ldc.i4   0xFF64268B
loc_2841:
  pop
  ldloc.s  5            // Previous state variable
  ldc.i4   0xE65E0969   // multiplier constant
  mul
  xor
```

  Again, an unconditional branch could follow the first conditional jump. However, we did not observe that other layout in the sample.

There are several additional caveats we noticed during analysis, which make automated deobfuscation harder in some cases - but as these border cases are rare, we can deal with them manually and let the deobfuscator script do the major part:

Caveat: Fragmentation of State Code

The code inside one state is sometimes itself fragmented, using additional `br`/`br.s` instructions. However, it is relatively easy to deal with this by just following these branches while traversing the state code.

Caveat: Non-Linear Code Fragments

One would expect every "real" code fragment inside a `case` block to be a single building block, i.e. without additional conditional branches. Unfortunately, this is not always the case - it seems the obfuscator does not obfuscate every normal conditional branch, but instead leaves some of these inside one `case` block untouched. We deal with this situation by just following one side of the branch in order to find the end of a state. This usually works, but there seem to be very few cases where actual non-obfuscated branches lead to different follower states or otherwise inconsistent behavior.

Caveat: More Than One `switch` in a Function

Sometimes, more than one obfuscation-`switch` appear in the same function. This is not a major problem, but it can make the initial or end state detection harder (except for the default end state). We try to cover that by defining the appearance of a `switch` instruction inside a state as an indication the state to be a leaf. This would fail if one obfuscation switch appeared inside an actual (non-leaf) state code of an outer switch obfuscation. We did not actually see such a situation, but it would make deobfuscation harder.

Caveat: Shared Code

In rare situations, two different states share some common control code and branch into this before the next loop iteration starts. We cannot deal with these situations at the moment, and they might result in code that `dnspy` can no longer decompile. This can also happen in case of some strange interaction with `try-except` constructs (e.g. at VA `53D6`).

Caveat: Inconsistent State Machine

Whenever the same state appears as successor of more that one other state, probably some kind of loop is implemented. This is a common situation. We expect all re-entries into such re-used states to have the *same value of the state variable* (or at least values that lead to the same state paths thereafter). Should this not be the case, we call the state machine *inconsistent*. Fortunately, all state machines in our sample turned out to be consistent. It is hard to think of an obfuscator algorithm producing an inconsistent state machine - it would have to somehow melt two different states into one. A very advanced state obfuscator might be able to actually realize this kind of magic. The situation is comparable to compilers: any standard C code results in some type of consistent assembly code, but assembly code in general can be inconsistent and not decompilable back into reasonale C code. Any state machine we come across could in theory turn out to be inconsistent. We deal with the problem by just detecting it, emitting an error line, and leaving the code unchanged.

There might be situations where an obfuscator could actually intentionally produce inconsistent states, just to break debofuscator scripts. This could be done in code that is never actually be executed, e.g. because some conditions leading to it will never be true, comparable to junk code. It might also be possible that actual state code is of a nature that repetitions don't do any harm, so repeating a state in a different context might be acceptable, if it is only used as a link state to another state value which is not yet used. The redundant repeating of the state code would then act like an empty state. This kind of obfuscation would be quite advanced though, and fortunately, we did not need to take care of it in this sample. We're not aware of such a technique actually used by code obfuscators.

We occasionally observed unused states, i.e. states that were never reached during emulation of a state machine. This does not make the state machine inconsistent, and it is safe to just ignore unused states; but it is still a slightly unusual observation. Whereever we checked, these unused states actually did not contain any real code, only control code. So they might very well be obfuscator artifacts.

Parsing and Analysis

As mentioned, we use IDA-Python for parsing. To make things easier, we use the excellent *sark* library[8] as disassembler wrapper. We define several structures to encapsulate the suffix type `SType` (a simple enum), information about the actual suffix code `Suffix` (most importantly the XOR and multiplication constants), and all information about an actual state `Block`, and finally a `Switch` structure:

[8] https://github.com/tmr232/Sark

```
# SType: the different code constructs which calculate the next
# state variable value
class SType(Enum):
    NA = auto()
    SIMPLE = auto()
    MXOR = auto()
    BRANCH = auto()
    BRANCH_MXOR = auto()


# Suffix collect type+context of ctrl code at end of  blocks
@dataclass
class Suffix:
    addr: int = 0      # address of first control instruction
    sType: SType = SType.NA  # actual code type
    val: int = -1      # value used for "if" / "else"
    val_else: int = -1
    xor: int = 0       # common xor/mult constants (MXOR)
    mult: int = 0
    # actual control instructions (for later NOPing):
    ctrls: List[sark.code.line.Line] = field(default_factory=list)


# Block encodes information about one switch block
@dataclass
class Block:
    state: int  # case tag (idx of block in switch)
    start: int  # start- and end-VA of data
    end: int = 0
    is_start: bool = False
    is_end: bool = False
    is_ret: bool = False
    next_state = -1
    next_state_else = -1
    enter_value = -1
    enter_values: Set[int] = field(default_factory=set)
    suffix = Suffix()  # used to calculate next state


@dataclass
class Switch:
    patcher: Patcher      # used later to patch code
    cont_addr: int = 0    # VA for loop calc
    end_addr: int = 0     # VA of block which ends loop
    switch_addr: int = 0  # VA of switch statement
    # entry_states gets the blocks that are initially traversed
    entry_states: Dict[int, List[int]] = field(default_factory=dict)
    # control values for state calculation:
    xor_value: int = 0
    nbr_blocks: int = 0
    failed: bool = False  # set to true if unusual cases
    blocks: List[Block] = field(default_factory=list)
```

Main Parsing Loop

Using above structure, the main code parsing loop starts with:

```
for fct in sark.functions():
    buff: List[sark.code.line.Line] = []  # reverse order
    pattern = bytearray()  # Allows us to find the function
```

```
    switches: List[Switch] = []
    lines: Dict[int, sark.code.line.Line] = {}

    print(f"\nFunction {fct.name} at VA {fct.ea:X}")
    for l in fct.lines:
        buff.insert(0, l)
        lines[l.ea] = l
        ops = l.insn.operands
        pattern.extend(l.bytes)

        # detect switch statement with preceding modulus:
        if l.insn.mnem == "switch" and \
           buff[1].insn.mnem == "rem.un" and \
           buff[2].insn.mnem.startswith("ldc.i4"):
```

We do keep the actual original byte data of the code in a byte array `pattern`. This is done because IDA currently does not support code patching for .NET - we can only read CIL instructions. While it would certainly possible to actually mimic the interpretation of .NET headers in Python to map `VA`s to actual raw file offsets, we decided for the simpler and pragmatic way to just search for `pattern` in the actual binary data of the executable in order to find the `delta` - which by the way is not constant for every function. There's also a caveat about this, which will be discussed further down.

Next, we check if one of the two known code fragments precede this, read and memorize the corresponding XOR value and the continuation address - i.e. the address jumped back to at each loop iteration after the new state variable was set:

```
# 2 opcode sequences are possible:
#   ldc / xor / dup / stloc / ldc / rem / switch: usual case, state variable is stored as
#      local var
#   ldc / xor / ldc / rem / switch: in some simple cases (e.g. VA 3f1a), the state
#      variable if kept on stack
if len(buff) > 4 and buff[3].insn.mnem == "xor" and buff[4].insn.mnem == "ldc.i4":
    xor_value = buff[4].insn.operands[0].imm
    cont_addr = buff[4].ea  # this is where jumps to the next loop occur
elif len(buff) > 6 and buff[5].insn.mnem == "xor" and buff[6].insn.mnem == "ldc.i4":
    xor_value = buff[6].insn.operands[0].imm
    cont_addr = buff[6].ea
else:
    raise Exception("Unknown xor construct before switch")
```

As mentioned earlier, we ignore the really improbable situation where the XOR value were so small that an `ldc.i4.s` or even an implicit instruction suffices. For our sample, the resulting exception in such a situation never triggered.

However, when reading the number of states, we need to consider all possibilities. Finally, the `Switch` object can be created:

```
# number of states is pushed in the previous ldc instruction. That one can embed the
# immediate value into
# the opcode for small values:
if buff[2].insn.mnem.startswith("ldc.i4.") and len(buff[2].insn.operands) == 0:
    # Implicit instruction, value is not stored in an operand, extract it from the
```

```
    # mnemonic as string:
    nbr_blocks = int(buff[2].insn.mnem.split(".")[-1])
else:
    # Explicit operand (".s" or full)
    nbr_blocks = buff[2].insn.operands[0].imm

switch = Switch(patcher, switch_addr=l.ea, cont_addr=cont_addr, xor_value=xor_value,
                nbr_blocks=nbr_blocks)
```

As additional confirmation, we assert an unconditional branch to the end state follows, which can be a far or a near one. `switch.end_addr` is set to where this branch jumps to, which is the default end block address:

```
if l.next.insn.mnem in ("br", "br.s"):
    switch.end_addr = l.next.insn.operands[0].addr
else:
    raise Exception("no br after switch")
```

Finally, we must extract the entry addresses for every `case` block. Unlike native assembly, CIL has a `switch` instruction with a *variadic* number of operands, meaning the list of operands is not always of the same length. As IDA's instruction model does not support variadic instructions, we can only read a comma-separated string of all labels like one single operand. To get the actual addresses instead of the symbolic labels, we rely on the fact these are of the form `loc_`+ "hexadecimal address". As the `loc_` prefix requires 4 characters, this can be expressed in the following way, which admittedly is an ugly hack - remember not to rename any labels before applying the script:

```
# Create switch blocks for each label (we must parse these as text labels, e.g.
# "loc_3F47", where 3f47 is the VA)
for state, l in enumerate(ops[0].text.split(',')):
    switch.blocks.append(Block(state, int(l.strip()[4:], 16)))
```

And we add the resulting switch construct: `switches.append(switch)`

After the loop parsed all lines of the function, we come back to each such detected `switch`, as we need to find more information about the actual blocks, such as: Where do they end? Are they end or start states? What suffix code is used? What are the relevant constants?

```
for switch in switches:
    start_found = False   # Will be set to true if start states could be detected
    for block in switch.blocks:
        if block.start not in lines:
            raise Exception(f"Block address {block.start:X} has no instruction")
        l = lines[block.start]
        crefs = list(l.crefs_to)
```

We create a list of *code references* to the first instruction of the block (`crefs`). For normal states, this list

only contains one instruction, namely the `switch` instruction. Start states should additionally get a code reference from where the loop is entered, and the normal end state should get another reference from the branch instruction following the `switch`. But we can check for the normal end state by directly comparing to `switch.end_addr` assigned in the previous step:

```
# The block that the branch after "switch" jumps to is a leaf (e.g. does not loop back):
if switch.end_addr == l.ea:
    block.is_end = True
    block.end = l.ea
    continue
```

Otherwise, if there are any cross references that differ from the switch address, we know this is a start state. Unfortunately, two obfuscation `switch`es in the same function occasionally share a state (e.g. `switch` at VA `53D6` and `switch` at `5462` both point to the same block on VA `53FB`). This is still an unclear case, and for the moment we consider these as end states:

```
# We need to find the initial states. Normal states have only one xref (the switch
# statement itself),
# while the initial state is also referenced by the jump-in instructions.
for a in crefs:
    if a != switch.switch_addr:   #  not in (switch.switch_addr, switch.end_addr):
        if lines[a].insn.mnem == "switch":
            # if 2 switch statements link top the same state (sigh),
            # we assume it's a common end state
            block.is_end, block.is_start = True, False
            break
        if block.state not in switch.entry_states:
            switch.entry_states[block.state] = []
        switch.entry_states[block.state].append(a)
        block.is_start = True
        start_found = True
```

Now we traverse the code for this block until we reach its end. Certain instructions are interpreted as markers that we reached an end state; we don't know at the moment if `switch` instructions themselves indicate an end state. They probably do, but we can't rely on this. We emit a warning if such a such a `switch` instruction is not followed by a branch. An example is the `switch` at VA `AC74`, which does not seem to have any end state at all (maybe an endless loop), and is jumped in from another `switch` at VA `AF73` (to address `AC73`). We consider these cases as end states. Note that branches which don't return to the loop entry (`cont_addr`) are just followed and assumed to occur due to code fragmentation:

```
# Traverse the code for this block:
while True:
    # returns end the block immediately:
    if l.insn.mnem == "ret" and not block.is_end:
        block.end = l.ea
        block.is_end = True
        break

    # embedded switch are tricky - we follow the branch behind:
    if l.insn.mnem == "switch":  # ... maybe we can assume these are always end nodes,
```

```
                                # but not sure...
            l = l.next
            if l.insn.mnem not in ("br", "br.s"):
                print(f"  INFO: Strange embedded switch at {l.ea:X} - we assume it's an end node")
                block.is_end, block.is_start = True, False
                break
            else:
                l = lines[l.insn.operands[0].addr]
                continue

        # If we hit the first instruction of the xor-part, this is the last block before,
        # where the branch lacks:
        if l.ea == switch.cont_addr:
            block.end = l.prev.ea
            # extract the next-block-constants (code suffix of block):
            block.suffix = find_suffix(l.prev, block.start)
            break

        # ... but most blocks end in a branch to the xor-part:
        elif l.insn.mnem in ("br", "br.s", "leave", "leave.s"):
            if l.insn.operands[0].addr == switch.cont_addr:
                block.end = l.prev.ea
                block.suffix = find_suffix(l.prev, block.start)
                # We consider the terminating branch also as control instruction:
                block.suffix.ctrls.append(l)
                break
            # branches that don't return are BRANCH-type instructions that we just follow
            # (can be fragmented)
            l = lines[l.insn.operands[0].addr]
            continue

        # Should not really happen (but maybe it does): last instruction of function also
        # ends the block
        elif l.ea + l.size not in lines:
            break

        l = lines[l.ea + l.size]
```

Keep in mind that all conditional jump conditions in this traversal are considered to fail - that's how the `l.next` method works. Hence, only one branch is traversed, knowing that in most cases both branches will eventually coalesce. Also, while most states end in a `br`/`br.s` instruction for the next loop iteration, in one case we probably see a fall-through (the block located just before this loop).

Detection of the Suffix Code Type and Extraction of the Constants (`find_suffix` function)

The `find_suffix` function is responsible for finding the control code part of the block, telling us about the suffix type, and extracting the relevant constants; all of this will be returned in a new `Suffix` instance. `find_suffix` also stores references to every control instruction into its `ctrls` field; these will be later `nop`-ed out in the patching stage. To do it's work, `find_suffix` relies on the parameter `l`, pointing to the last instruction of the block previous to the final branch to the next loop iteration (or just the last instruction in case of a fall-through state). It does so by walking backwards from that instruction using a temporary variable `l2`. Because the control code is at most 11 instructions long, it stops when this threshold is reached, or when the block's start-address or the `switch` instruction is hit. These up to 11 instructions are put into a list `ctrls`, like before in reverse order:

```
    l2 = l
    tree_else_addr = -1  # for branches, the else part usually follows, but could be
                         # branched to as well
    # byte sequence of previous instruction (so actually the one behind), allows to detect
    # 2 identical ldc instructions
    prev_bytes = b''
    for i in range(11):  # suffixes are never longer
        if l2.insn.mnem == "switch":
            break
        ctrls.append(l2)
        if l2.ea == block_start_addr:
            break
```

The variable `tree_else_addr` will be used to correctly link the IF-part of a branch to the ELSE-part - see below for more details. Notice that we keep the binary representation of the previous instruction in a variable `prev_bytes`; this allows us to easily detect the two subsequent `ldc.i4` clones used in the `BRANCH` suffix types.

We skip a short piece of code here (shown and explained three code paragraphs further down), which asserts the correct flow in case of `BRANCH` suffix type.

The code cross-reference approach is used once more to decide if we can just go backwards normally, or need to take care of a branch; this also allows us to follow back unconditional branches without preceding fall-through:

```
crefs = list(l2.crefs_to)
if len(crefs) == 1:
    prev_bytes = l2.bytes
    l2 = sark.Line(crefs[0])
    continue
# Only instructions used in standard branch construct (b.. / ldc /  )
if len(crefs) != 2:
    raise Exception(f"Only 1 or 2 crefs allowed at {l2.ea:X}")
```

If we have more than one reference, we must either be at the `pop` instruction of one of the two `BRANCH` suffix types, or one of their `ldc.i4` instructions. As explained below, the latter case should already be dealt with at this point though. In case of the `pop` instruction, in all observed layouts - fragmented or not - the IF part with its 2 `ldc` instructions immediately precedes the `pop` instruction, while the ELSE part ends in an unconditional `br` to the `pop` instruction. The address of this branch can be extracted using `list(set(crefs)- set([l2.prev.ea]))[0]`, which just removes the final instruction of the IF part from the code references and so should point to the `br` of the ELSE part; this address is stored in a local variable `tree_else_addr`, initialized to -1, for one of the next loop iterations:

```
# a BRANCH combines at "pop" instruction; immediately before is the if part (2 ldc's)
# note: theoretically, a branch cold appear in between
if l2.insn.mnem == "pop":
    tree_else_addr = list(set(crefs) - set([l2.prev.ea]))[0]
    prev_bytes = l2.bytes
    l2 = l2.prev
```

```
        continue
    else:
        prev_bytes = l
        l2 = l2.prev
```

When going back beyond the `pop` instruction, we use the fact `tree_else_addr` was set to a value different from -1 (meaning a `pop`instruction was seen) in order to detect the first of the two `ldc` clones in the IF branch - at this point we must jump to the just memorized ELSE branch. This needs to be done earlier in the loop (before `crefs` was assigned, where the skipped code was mentioned), and we also set `tree_else_addr` back to -1:

```
if tree_else_addr >= 0 and l2.insn.mnem == "ldc.i4" and l2.bytes == prev_bytes:
    # 2 identical ldc.i4 instructions detected
    prev_bytes = l2.bytes
    # after this, we jump to the else tree (linking to the pop)
    l2 = sark.Line(tree_else_addr)
    tree_else_addr = -1
    continue
```

When the collection of the control code is completed, we can use the resulting `ctrls` list to differ the types:

- `SIMPLE` is easily identified by its `ldc.i4` instruction at the end - it also is the only actual control instruction:

```
if ctrls[0].insn.mnem == "ldc.i4":
    s.sType = SType.SIMPLE
    s.val = l.insn.operands[0].imm
    s.addr = l.ea
    s.ctrls.append(ctrls[0])
```

- `MXOR` is also easy to detect: we just allow for different local variables for the state variable. Keep in mind that all 5 instructions checked for are considered control instructions (and suffix's `addr` is set to the first of them):

```
elif ctrls[0].insn.mnem == "xor" and ctrls[1].insn.mnem == "ldc.i4" and \
     ctrls[2].insn.mnem == "mul" and \
     ctrls[3].insn.mnem == "ldc.i4" and ctrls[4].insn.mnem.startswith("ldloc."):
    s.sType = SType.MXOR
    s.xor = ctrls[1].insn.operands[0].imm
    s.mult = ctrls[3].insn.operands[0].imm
    s.addr = ctrls[4].ea
    s.ctrls.extend(ctrls[0:5])
```

- `BRANCH` needs a bit more flexibility - note that `ctrls[6]` is just checked for starting with the letter `b`, as it can be the relevant conditional branch, or an additional unconditional fragmentation branch just after it, depending on the layout. Everything except this one or two branches are considered control code (this has to do with the special patching situation explained later on), but depending on the

situation, we must set the start address differently:

```
elif  ctrls[0].insn.mnem == "pop" and \
      ctrls[1].insn.mnem == "ldc.i4" and ctrls[2].insn.mnem == "ldc.i4" and \
      ctrls[3].insn.mnem in ("br", "br.s") and \
      ctrls[4].insn.mnem == "ldc.i4" and ctrls[5].insn.mnem == "ldc.i4" and \
      ctrls[6].insn.mnem.startswith("b"):
    s.sType = SType.BRANCH
    s.val = ctrls[1].insn.operands[0].imm
    s.val_else = ctrls[4].insn.operands[0].imm
    if ctrls[6].insn.mnem in ("br", "br.s"):
        if ctrls[7].insn.mnem.startswith("b"):
            # Fragmentation case, we start one instruction earlier
            s.addr = ctrls[7].ea
        else:
            raise Exception(f"br should be preceeded by conditional branch at VA {ctrls[6].
                ea}")
    else:
        s.addr = ctrls[6].ea
    s.ctrls.extend(ctrls[0:6])
```

- `BANCH_MXOR` is dealt with in the same way. As a side note, we did not actually see the fragmented layout for this suffix type:

```
 elif ctrls[0].insn.mnem == "xor" and ctrls[1].insn.mnem == "mul" and \
      ctrls[2].insn.mnem == "ldc.i4" and ctrls[3].insn.mnem.startswith("ldloc.") and \
      ctrls[4].insn.mnem == "pop" and \
      ctrls[5].insn.mnem == "ldc.i4" and ctrls[6].insn.mnem == "ldc.i4" and \
      ctrls[7].insn.mnem in ("br", "br.s") and \
      ctrls[8].insn.mnem == "ldc.i4" and ctrls[9].insn.mnem == "ldc.i4" and \
      ctrls[10].insn.mnem.startswith("b"):
   s.sType = SType.BRANCH_MXOR
   s.val = ctrls[5].insn.operands[0].imm
   s.val_else = ctrls[8].insn.operands[0].imm
   s.mult = ctrls[2].insn.operands[0].imm
   if ctrls[10].insn.mnem in ("br", "br.s"):
       if ctrls[11].insn.mnem.startswith("b"):
           s.addr = ctrls[11].ea
       else:
           raise Exception(f"br should be preceeded by conditional branch at VA {ctrls[6].ea
               }")
   else:
       s.addr = ctrls[10].ea
   s.ctrls.extend(ctrls[0:10])
```

Emulation

After a `Switch` is fully parsed and an initial state actually found, it can be emulated by recursive code; this is done for every possible initial state, if there should be more than one:

```
for initial_state in switch.entry_states.keys():
    print(f"  Entry state {initial_state}")
    switch.failed = not switch.emulate(initial_state, value=-1, prev_block=None,
```

```
                                        do_print=True, indent=1)
```

`Switch`'es emulation method returns `True` if it has worked without issues. Situations like inconsistent state variables return `False`. The emulation method is pretty straightforward, so we won't describe it in detail; if things work, the `next_state` and `next_state_else` properties of every block should be filled in correctly. It also allows to print the state machine using `do_print` and `indent`.

Patching

The actual patching step is the most critical one. As already mentioned, IDA can't directly patch CIL code. Instead, we must patch directly into the binary. For this purpose, a `Patcher` class is defined, reading the original binary and writing the modified binary to the same filename with an additional `.dec` suffix:

```
class Patcher(object):
    bin_data: bytes
    delta: int

    # Read binary data from disk when instantiated:
    def __init__(self) -> None:
        with open(idaapi.get_root_filename(), "rb") as f:
            self.bin_data = f.read()
        self.delta = 0

    def write(self) -> None:
        with open(idaapi.get_root_filename() + ".dec", "wb") as f:
            f.write(self.bin_data)
```

The `delta` value stores the difference between raw file offsets and IDA's virtual addresses. Unfortunately, it needs to be recalculated for each method. Also, as we use sark's `.next` property to walk through all instructions of a method, this might sometimes skip some "code caves" (unused junk instructions), so we can't expect the byte pattern to be precisely identical in the binary to find the offset. So we decided for a workaround to find a match as far as possible:

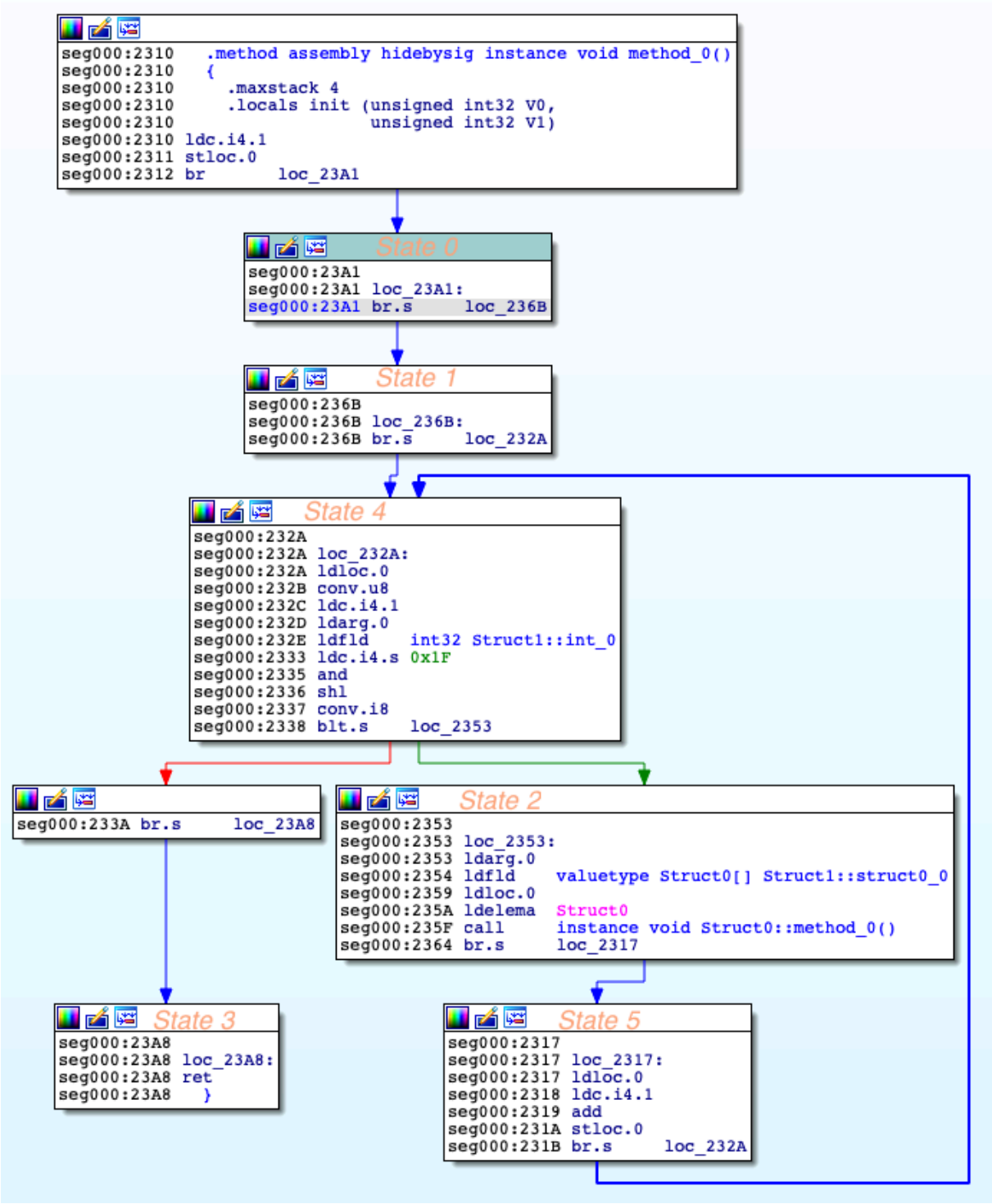
```
    # set_fct: find function code in binary data and calculate "delta" for this function
    # (VA-physical mapping)
    def set_fct(self, start_va_addr: int, pattern: bytes) -> None:
        while True:
            # Somethimes instructions are skipped that sark does not see, so not the full
            # pattern need to match. Decrease until it matches.
            # TODO: Make sure match is still unique
            fct_offset = self.bin_data.find(pattern)
            if fct_offset >= 0:
                break
            pattern = pattern[:-1]
        self.delta = fct_offset - start_va_addr
```

When trying to unflatten code, different approaches can be chosen. The cleanest would be to re-arrange the whole code of a method. We decided against this as we expect many potential problems doing so. For

once, all branches in the method must be re-calculated, and care needs to be taken to differ between near and far branches. Also, embedded structural elements like try-except-constructs could be broken by just moving code of a function around.

An easier approach is be to leave all "real" code where it is, and just replace the start of all "control" code fragments we previously identified by branches to the next calculated states. This will result in code with non-optimal layout, fragmentation, and non-required branches - the main question however is whether `dnspy` will be able to decompile such code or not. We can't conclusively answer this question, but our experiments showed that this works quite well, as long as all "other" control code (the code not actually used for our replacement branches) is `nop`-ed out, i.e. overwritten by `00` bytes (CIL's opcode for `nop`). `dnspy` does not seem to be able to detect and ignore unreachable instructions, so we can't just let these artifacts lie around, but are instead forced to clean things up.

To show the intended result, the code shown in the previous image looks like this after patching - notice that all control code is now gone, but actual code blocks remain at their addresses. We mark the same `State` tags to make comparison easier; e.g. state 1, which does not contain any actual code - it's like an empty state - is now reduced to a single branch to the next state 4. Also, at the end of state 4, we actually left the conditional branch where it was (at addresses `2338`) and overwrote the first `ldc.i4 0x917879FC` instruction of the ELSE part by a branch to state 3 (at VA `233A`, jumping to `23A8`)



To show the nop-ed out part, this is the same code in non-graph view:

```
seg000:2310   {
seg000:2310     .maxstack 4
```

```
seg000:2310     .locals init (unsigned int32 V0,
seg000:2310                   unsigned int32 V1)
seg000:2310     ldc.i4.1
seg000:2311     stloc.0
seg000:2312     br       loc_23A1
seg000:2317
seg000:2317 loc_2317:
seg000:2317     ldloc.0
seg000:2318     ldc.i4.1
seg000:2319     add
seg000:231A     stloc.0
seg000:231B     br.s     loc_232A
seg000:231D     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
seg000:232A
seg000:232A loc_232A:
seg000:232A
seg000:232A     ldloc.0
seg000:232B     conv.u8
seg000:232C     ldc.i4.1
seg000:232D     ldarg.0
seg000:232E     ldfld    int32 Struct1::int_0
seg000:2333     ldc.i4.s 0x1F
seg000:2335     and
seg000:2336     shl
seg000:2337     conv.i8
seg000:2338     blt.s    loc_2353
seg000:233A     br.s     loc_23A8
seg000:233C     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
seg000:233C     .byte 0, 0
seg000:2353
seg000:2353 loc_2353:
seg000:2353     ldarg.0
seg000:2354     ldfld    valuetype Struct0[] Struct1::struct0_0
seg000:2359     ldloc.0
seg000:235A     ldelema  Struct0
seg000:235F     call     instance void Struct0::method_0()
seg000:2364     br.s     loc_2317
seg000:2366     .byte 0, 0, 0, 0, 0
seg000:236B
seg000:236B loc_236B:
seg000:236B     br.s     loc_232A
seg000:236D     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
seg000:236D     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
seg000:236D     .byte 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
seg000:23A1
seg000:23A1 loc_23A1:
seg000:23A1     br.s     loc_236B
seg000:23A3     .byte 0, 0, 0, 0, 0
seg000:23A8
seg000:23A8 loc_23A8:
seg000:23A8     ret
seg000:23A8   }
```

And indeed, `dnspy` can correctly decompile this method the way shown above.

CIL Branch Instructions

To understand the patching code, a short detour into CIL is required. Unlike native assembly, the first byte of each instruction is always the opcode - no two- or more-byte opcodes exist. CIL knows exactly 26 different branch opcodes: one unconditional and 12 conditional ones, and each of them in a far variant

(4 byte operand) and a near variant (1 byte operand with `.s` suffix in the mnemonic), so occupying 5 or 2 bytes. The far- and near-variant opcodes of each branch always differ by exactly 13. E.g. we can modify the opcode of `blt` to `blt.s` by subtracting 13 from the opcode, and vice versa. The opcodes for `br` and `br.s` are `0x38` and `0x2b`, again with a difference of 13. As in native assembly, the encoded operand is a 2-complement delta to the address of the otherwise following instruction, so near branches have a range of -128 to 127. When patching code, we must take care of far and near branches because we must not overwrite non-control code; this would happen if we changed a near branch to a longer far branch requiring three additional bytes.

These small helper functions support us with patching code - they should be self-explanatory:

```
# Some helper functions to deal with branch functions:
# 0x2b - 0x37 are near (.s) variants, 0x38 - 0x44 far variants
# (difference with same condition is always 13)
# 0x38 / 0x2b are the unconditional jumps
def is_branch(opcode: int) -> bool:
    return 0x2b <= opcode <= 0x44


def is_near(opcode: int) -> bool:
    return 0x2b <= opcode <= 0x37


def is_far(opcode: int) -> bool:
    return 0x38 <= opcode <= 0x44


def make_far(opcode: int) -> int:
    if is_far(opcode):
        return opcode
    return opcode + 13


def make_near(opcode: int) -> int:
    if is_near(opcode):
        return opcode
    return opcode - 13


def is_in_near_range(jump_addr: int, target_addr: int) -> bool:
    return -128 <= target_addr - (jump_addr + 2) <= 127


# branch_delta: calculate delta from jump_addr to target_addr and encode in 1
# (if near) or 4 bytes
def branch_delta(jump_addr, target_addr, near: bool) -> bytes:
    if near:
        return struct.pack("B", (target_addr - (jump_addr + 2)) & 0xff)
    return struct.pack("<I", (target_addr - (jump_addr + 5)) & 0xffffffff)
```

With this in mind, `Patcher` implements methods to patch branches, either by overwriting with an unconditional branch (if `overwrite_unconditional` is set to `True`), or by replacing the target address of an existing conditional branch, as well as to `nop` out instructions and code blocks - the latter one uses recursion in case of conditional branches:

```
    # patch_branch: patch in a branch to the destination address at the given address.
    # if overwrite_unconditional is true, a br or br.s is encoded.
    # otherwise, the existing branch condition is kept (just made far or near if required)
    # returns the number of bytes required (5 for long, 2 for short)
    def patch_branch(self, jump_addr: int, target_addr: int,
                     overwrite_unconditional=False) -> int:
        opcode = self.bin_data[jump_addr + self.delta]
        if overwrite_unconditional:
            opcode = 0x38  # this is a br instruction
        if not is_branch(opcode):
            raise Exception(f"ERROR: there is no branch at address {jump_addr:X}")

        near = is_in_near_range(jump_addr, target_addr)
        if near:
            inst_size = 2
            opcode = make_near(opcode)
        else:
            inst_size = 5
            opcode = make_far(opcode)

        self.bin_data = self.bin_data[:jump_addr + self.delta] + \
                        opcode.to_bytes(1, "little") + \
                        branch_delta(jump_addr, target_addr, near) + \
                        self.bin_data[jump_addr + self.delta + inst_size:]

        return inst_size

    def nop_inst(self, l: sark.code.line.Line) -> None:
        addr = l.ea + self.delta
        self.bin_data = self.bin_data[:addr] + \
                        (b'\x00' * l.size) + \
                        self.bin_data[addr + l.size:]

    def nop_code(self, l: sark.code.line.Line, end_address: int) -> None:
        while True:
            self.nop_inst(l)
            if l.ea == end_address:
                return
            if l.insn.mnem in ("br", "br.s"):
                l = sark.Line(l.insn.operands[0].addr)
                continue
            if l.insn.mnem.startswith("b"):
                self.nop_code(sark.Line(l.insn.operands[0].addr), end_address)
            l = l.next
```

The main code creates one `Patcher` instance, which is referenced to in every `Switch` instance as a property `patcher`, because `patch` is a method of the `Switch` class.

Avoiding Overwriting Real Code When Patching

As mentioned before, we must be careful to never overwrite actual code. So let's look at the different suffix types and layouts:

- `SIMPLE`: The control code for a `SIMPLE` block is always a `ldc.i4 ...` instruction, which requires 5 bytes. As explained earlier, theoretically a 2-byte `ldc.i4.s` for 1-byte values or even an implicit 1-byte instruction for very small values could be used; but as this happens very rarely, we think it's safe to ignore these or deal with these situations manually. As our branch instruction requires 2 or 5

bytes as well (depending if it's near or far), we're safe.
- `MXOR`: The same goes here, control code is even longer (and we never saw it fragmented), so overwriting is safe
- `BRANCH`: This is the tricky part. We must differ between the 2 layouts.
  - If no fragmentation is used, we're safe as well: We'll have a 2- or 5-byte conditional jump, followed by a two 5 byte `ldc` instructions of the ELSE part, so 12 or 17 bytes. We must overwrite this in the worst case with a far conditional and a far unconditional jump, so 10 bytes.
  - If fragmentation is used, the situation is different. In the worst case, we only have a 2-byte conditional near branch followed by a 2-byte unconditional near branch (4 bytes), so we can't fill in 10 bytes without potentially overwriting real code behind. As a workaround, we decided to leave these 2 branches untouched, and instead overwrite the first `ldc.i4` instruction of the IF- and ELSE-part, where we know we have 5 (even 10) bytes available at each location. This results in additional branches, but as this does not seem to be a problem for `dnspy`, we can live with it. Note that the fact these 2 branch instructions remain as they are is the reason we do not consider them as control instructions, so they are not `nop`ed out in the `patch` method. Note that we could use the same approach for the no-fragmentation case, but decided to not do this, mainly because it would be more complicated to implement it without offering advantages.
- `BRANCH_MXOR`: This is basically identical to `BRANCH`.

Finally, we can implement `Switch`'s `patch` method. Note that we do not need to change jumps into the initial states, as they already point to the right place. Patching occurs in several stages:

- `nop` out the main loop construct (where the main loop's XOR is calculated), up to and including the branch that follows the `switch` instruction. As this code was never fragmented, we can go for the easy way here.
- For all blocks, except end blocks:
  - `nop` out all control instructions (keep in mind that the conditional and optional unconditional branch instructions of a `BRANCH`/`BRANCH_MXOR` type are not affected by this).
  - Should a state not have been actually used in the emulation, an `INFO` warning is printed: this is not exactly an error, but still unusual. Then the whole "real code" block is `nop`-ed out as well.
  - For `SIMPLE` and `MXOR` suffixes (where `block.next_state_else == -1`), just patch over an unconditional branch to the calculated next state - we know this can't overwrite any real code.
  - For the two `BRANCH` types we differ between the fragmented case (if a `br`/`br.s` follows) and the monolithic case. In the first case, we overwrite the two first `ldc.i4` instructions of the 2 branches; we find these by extracting the jump addresses of the conditional and the unconditional branch. Otherwise, we modify the conditional branch to one new target, and put an unconditional branch behind.

```
    # patch: patch in all required branch and nop codes:
    def patch(self, lines: Dict[int, sark.code.line.Line]) -> None:
        # nop out switch loop as it could trigger unwanted crossreferences:
        l = lines[switch.cont_addr]
        while True:
            self.patcher.nop_inst(l)
            if is_branch(l.bytes[0]):
                break
```

```
            l = l.next

        for block in switch.blocks:
            if block.is_end or block.is_ret:
                continue

            # nop out all control instructions:
            for ctrl_inst in block.suffix.ctrls:
                self.patcher.nop_inst(ctrl_inst)

            # nop out all unusuaed states:
            if block.next_state == -1:
                print(f"INFO: block {block.state} at VA {block.start} is "
                      f"unused - code will be nopped out")
                self.patcher.nop_code(lines[block.start], block.end)
                continue

            # and patch in branches:
            if block.next_state_else == -1:
                # State has a unique follower (is not a branch): patch in a
                # unconditional br instruction
                self.patcher.patch_branch(block.suffix.addr,
                                          switch.blocks[block.next_state].start,
                                          overwrite_unconditional=True)
            else:
                # State has an actual branch:
                if lines[block.suffix.addr].next.insn.mnem in ("br", "br.s"):
                    # Followed by a br/br.s: fragmentation. Long branches might not fit,
                    # so put these over subsequent ldc instructions:
                    self.patcher.patch_branch(
                           lines[block.suffix.addr].insn.operands[0].addr,
                           switch.blocks[block.next_state].start,
                           overwrite_unconditional=True)
                    self.patcher.patch_branch(
                           lines[block.suffix.addr].next.insn.operands[0].addr,
                           switch.blocks[block.next_state_else].start,
                           overwrite_unconditional=True)
                else:
                    # Otherwise (2 ldc instructions follow) there is enough space
                    size = self.patcher.patch_branch(
                           block.suffix.addr,
                           switch.blocks[block.next_state].start,
                           overwrite_unconditional=False)
                    self.patcher.patch_branch(
                           block.suffix.addr + size,
                           switch.blocks[block.next_state_else].start,
                           overwrite_unconditional=True)
```

Results and Lookout

The deobfuscator script only reports one function, `Class6::smethod_1` as failed because the initial content of the state variable could not be found out. It actually happens at VA `5322` (`ldc.i4 0xB4A56FF0`), but the situation is a bit complicated due to several `switch` constructs combined with a `try`. Another function, `GClass14::method_5`, reports several unusues states and results in code `dnspy` can't handle. This is a quite long function and needs forther study. It seems to contain a `switch` statement inside a non-end state of another `switch` statement, which explains the current inability to deal with it.

All other 90 obfuscated functions seem to have worked - we did not check if they all make sense though.

There were occasional unusued states reported, but whereever we checked, these states indeed did not contain any real code, so this might just be a normal obfuscator artifact.

Source code is available on https://github.com/govcert-ch/ConfuserEx_IDAPython/