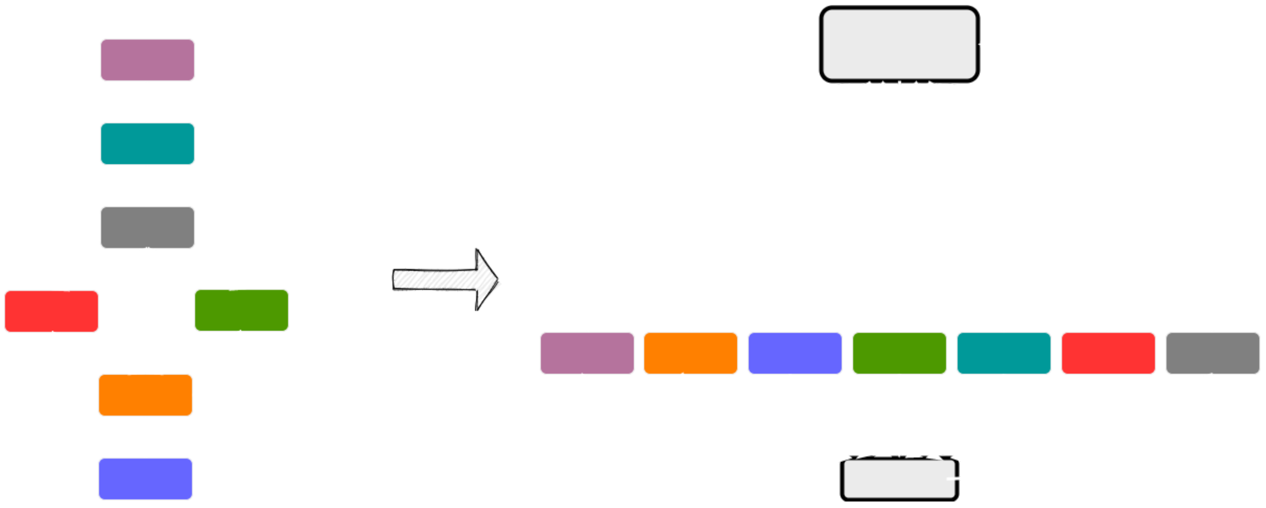
MODeflattener - Miasm's OLLVM Deflattener

mrt4ntr4.github.io/MODeflattener/

Suraj Malhotra June 25, 2021

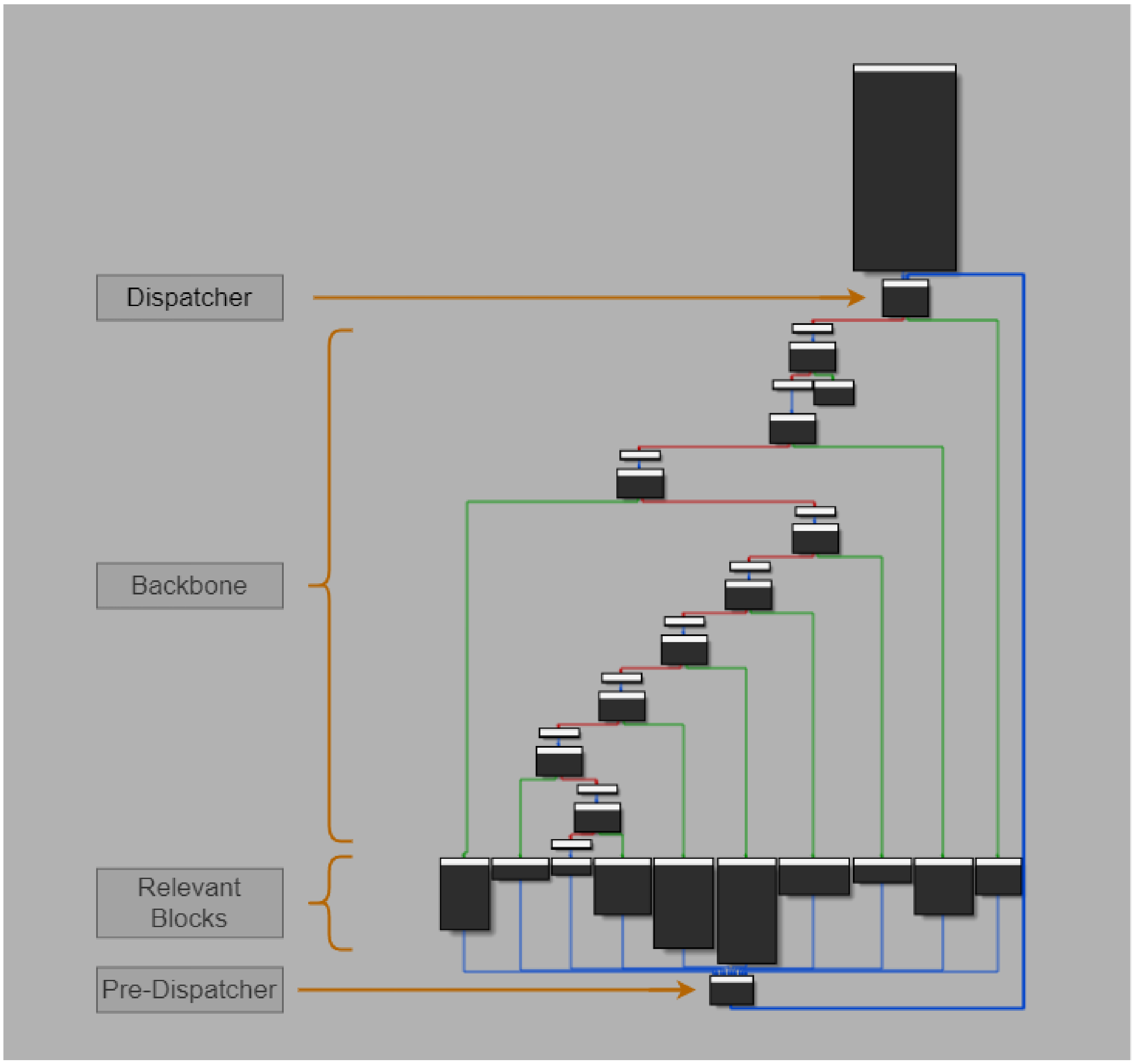
So recently a challenge(Layers) from 3kCTF featured control flow flattening using OLLVM. Although I did know about control flow flattening I hadn't encountered it personally. And as I've been experimenting with miasm for the past few days I thought of developing a tool to deal with it.

Control Flow Flattenning



Control flow flattening is an interesting and clever technique to make a reverse engineer's day difficult. It basically puts all the basic blocks in a function at the same level and destroys the control flow. It then uses a dispatcher to reconstruct the control flow along with a control/state variable which is updated at the end of each block.

The following illustration depicts various parts of a flattened function.



Various obfuscators employ their own version of control flow flattening transformations :

- OLLVM
- Tigress
- Hellscape

> CAUTION : Only Static Analysis used !
> As we are primarly focusing on OLLVM for now we can just use some static analysis techniques to deobfuscate it, but for developing a universal tool to deal with this kind of obfuscation we need to look more into dynamic approaches.
> *For example, this looks interesting.*

Getting Flattening Information

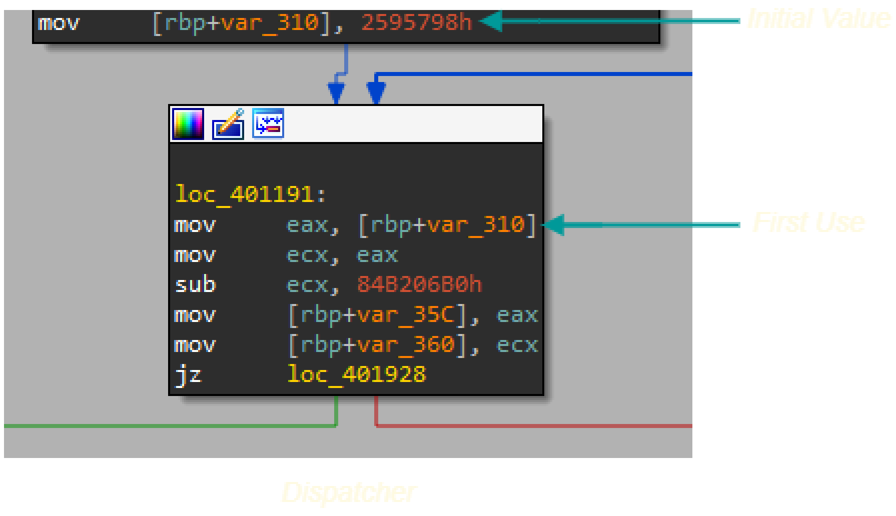
To start off, we need to identify the relevant blocks and the dispatcher. To find the dispatcher we need to first find the pre-dispatcher. This is because we rely on the fact that the pre-dispatcher has the maximum number of predecessors, it is easy to identify. Later we can just get the first successor of the pre-dispatcher to get the dispatcher, easy!

```
def get_cff_info(asmcfg):
    preds = {}
    for blk in asmcfg.blocks:
        offset = asmcfg.loc_db.get_location_offset(blk.loc_key)
        preds[offset] = asmcfg.predecessors(blk.loc_key)
    pre_dispatcher = sorted(preds, key=lambda key: len(preds[key]), reverse=True)
[0]
    dispatcher =
asmcfg.successors(asmcfg.loc_db.get_offset_location(pre_dispatcher))[0]
    dispatcher = asmcfg.loc_db.get_location_offset(dispatcher)
```

Also we now already have the relevant blocks as they are just the predecessors to the pre-dispatcher we just found!

State Variable

The state variable is responsible for maintaining the control flow in the flattened function.



```
mov     [rbp+var_310], 2595798h

loc_401191:
mov     eax, [rbp+var_310]
mov     ecx, eax
sub     ecx, 848206B0h
mov     [rbp+var_35C], eax
mov     [rbp+var_360], ecx
jz      loc_401928
```

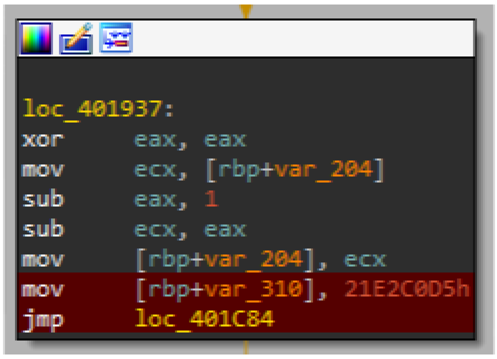
The state variable is always initialized before the dispatcher and is used in the first line of the dispatcher. We can use this information to get the state variable automatically.

Relevant Blocks

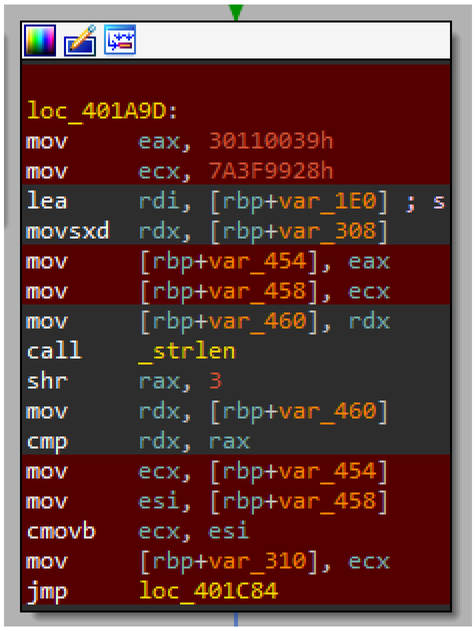
The blocks that are aligned at the same level in the disassembly graph and include the useful code are known as relevant blocks. These blocks update the value of the state variable at the very end.

Note: Currently we also add the *tail of the backbone* to our relevant blocks as we are just depending on the predecessors of the pre-dispatcher. Basically the tail is used if the state variable value doesn't satisfy any condition in the backbone. It doesn't update the state variable and only has a jump to pre-dispatcher. So if we don't find any code related to modification of the state variable in a relevant block we mark this as tail.

Having most of the information we can now proceed with deflattening the flow.
Basically we have two types of relevant blocks:



```
loc_401937:
xor     eax, eax
mov     ecx, [rbp+var_204]
sub     eax, 1
sub     ecx, eax
mov     [rbp+var_204], ecx
mov     [rbp+var_310], 21E2C0D5h
jmp     loc_401C84
```



```
loc_401A9D:
mov     eax, 30110039h
mov     ecx, 7A3F9928h
lea     rdi, [rbp+var_1E0] ; s
movsxd  rdx, [rbp+var_308]
mov     [rbp+var_454], eax
mov     [rbp+var_458], ecx
mov     [rbp+var_460], rdx
call    _strlen
shr     rax, 3
mov     rdx, [rbp+var_460]
cmp     rdx, rax
mov     ecx, [rbp+var_454]
mov     esi, [rbp+var_458]
cmovb   ecx, esi
mov     [rbp+var_310], ecx
jmp     loc_401C84
```

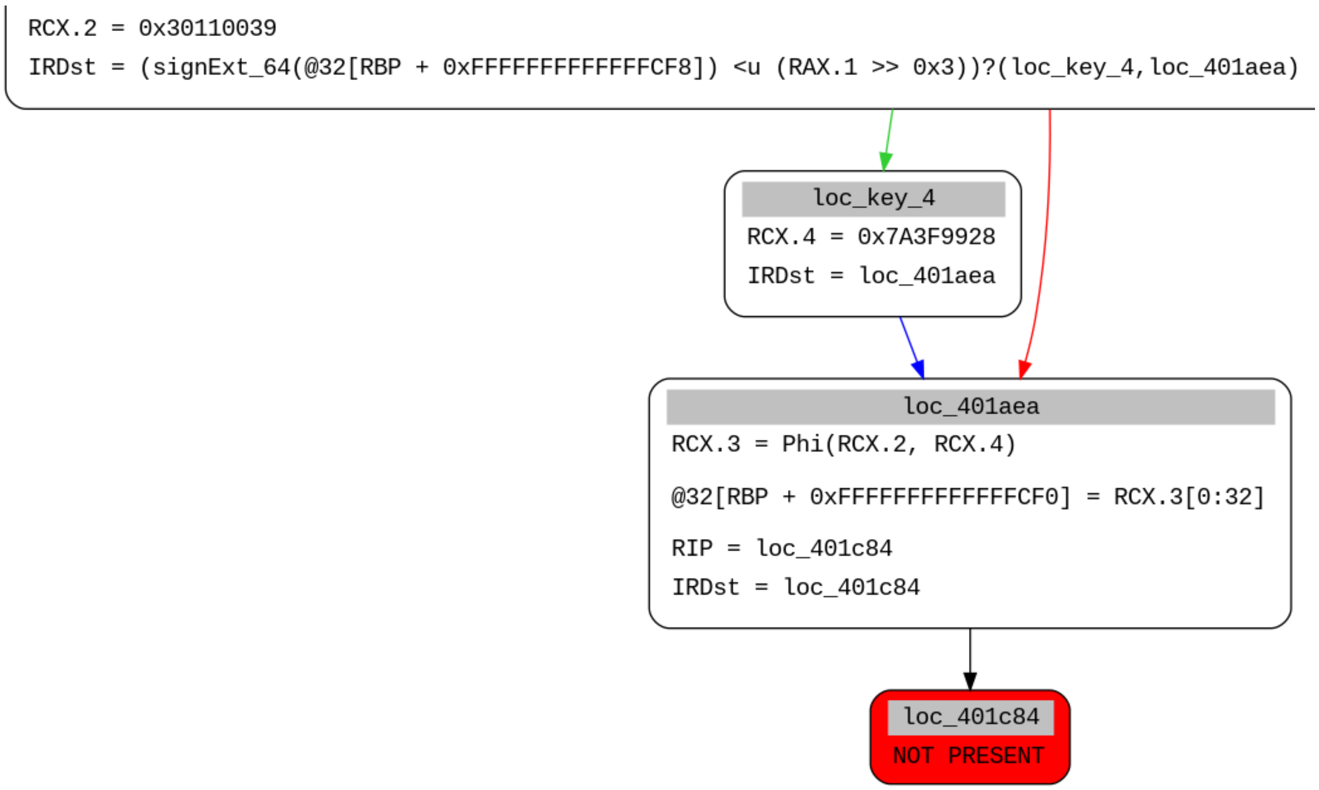
- **Simple**
  Block without any conditions, so the state variable is always updated with the same value. Only one instruction is used to modify the state variable.

- **Conditional**
  Blocks with conditional statements and loops. Here the state variable could have two possible values depending on whether the condition results in a true or false. These often end with a `cmov` instruction. Several instructions are used to modify the state variable.

Use of SSA Expressions

We further simplify the IR to SSA to deal with the conditional relevant blocks.
We only make use of `do_propagate_expressions` ssa simplification pass.

```
head = loc_db.get_offset_location(addr)
ssa_simplifier = IRCFGSimplifierSSA(lifter)
ssa = ssa_simplifier.ircfg_to_ssa(ircfg,
head)
ssa_simplifier.do_propagate_expressions(ssa,
head)
```



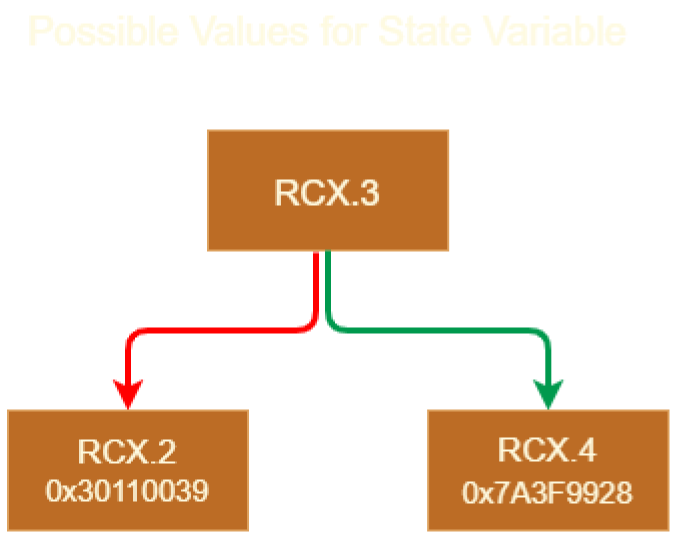
```
RCX.2 = 0x30110039
IRDst = (signExt_64(@32[RBP + 0xFFFFFFFFFFFFFCF8]) <u (RAX.1 >> 0x3))?(loc_key_4,loc_401aea)
```

```
loc_key_4
RCX.4 = 0x7A3F9928
IRDst = loc_401aea
```

```
loc_401aea
RCX.3 = Phi(RCX.2, RCX.4)
@32[RBP + 0xFFFFFFFFFFFFFCF0] = RCX.3[0:32]
RIP = loc_401c84
IRDst = loc_401c84
```

```
loc_401c84
NOT PRESENT
```

In the SSA form we observe a Phi operation which basically means that one of the variables arriving from different predeccesors is chosen depending on which path the control flow took.



In the above example we observe that if the condition is *true* the state variable is assigned the value = 0x7A3F9928 (RCX.4) and if *false* value = 0x30110039 (RCX.2).

We get the block where the phi variables are assigned using the following code and then get their values from those blocks.

```
if irblock_has_phi(irblock):
    for dst, sources in viewitems(irblock[0]):
        phi_vars = sources.args
        parent_blks =
get_phi_sources_parent_block(
            ircfg,
            irblock.loc_key,
            phi_vars
        )
```

We get the following information from a relevant block:

```
0x401a9d: {'cond': 'CMOVB',
           'false_next':
0x30110039,
           'true_next':
0x7a3f9928}
```

Now we need to iterate over the backbone blocks and get their destinations if it includes condition regarding any of the possible state variable values. We can map these to the original state variable values and make use of it to correct the flow.

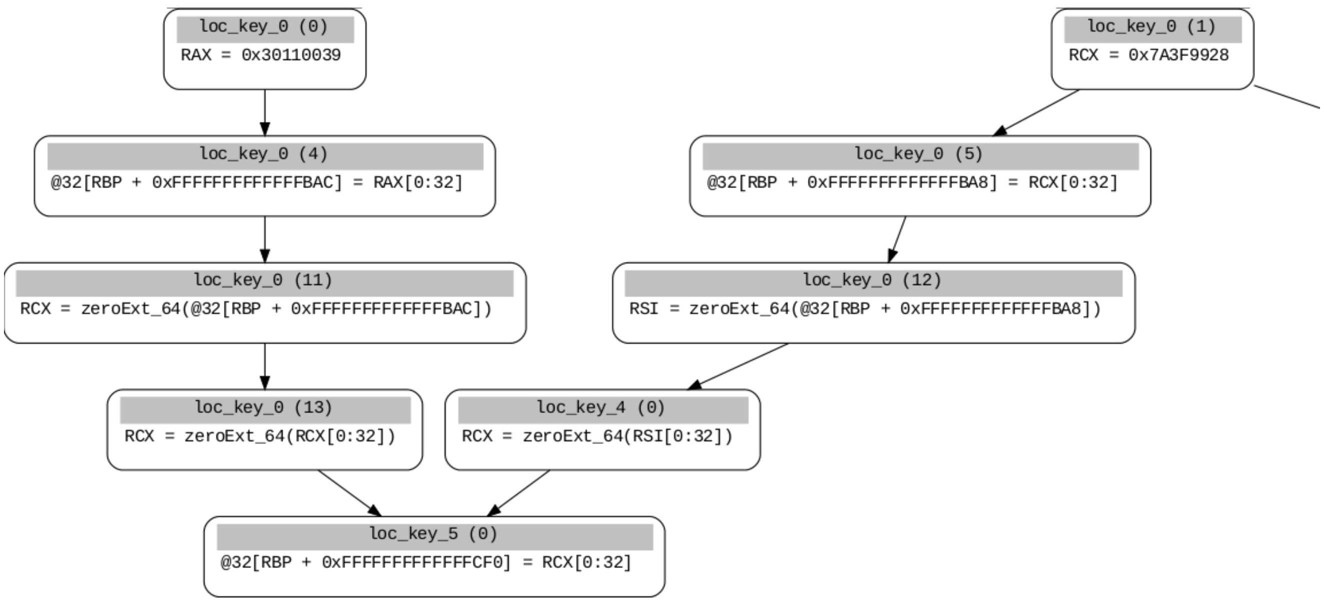
```
if isinstance(arg, ExprInt):
    if int(arg) in val_list:
        cmp_val = int(arg)
        var, locs = irblock[-1].items()[0]
        true_dst =
main_ircfg.loc_db.get_location_offset(locs.src1.loc_key)
        backbone[hex(cmp_val)] = hex(true_dst)
```

After resolving these values from the backbone, the final result looks like this.

```
0x401a9d: {'cond': 'CMOVB',
           'false_next':
0x401bb0,
           'true_next':
0x401af5}
```

Removing Useless Instructions

Modeflattener finds the addresses to nop out using the def-use graph for the state variable. This is one of the data flow analysis feature provided by miasm. This algorithm returns all the instructions affecting the state variable and therefore we call these as useless instructions. Read more about it here


loc_key_0 (0)
RAX = 0x30110039
loc_key_0 (1)
RCX = 0x7A3F9928
loc_key_0 (4)
@32[RBP + 0xFFFFFFFFFFFFFBAC] = RAX[0:32]
loc_key_0 (5)
@32[RBP + 0xFFFFFFFFFFFFFBA8] = RCX[0:32]
loc_key_0 (11)
RCX = zeroExt_64(@32[RBP + 0xFFFFFFFFFFFFFBAC])
loc_key_0 (12)
RSI = zeroExt_64(@32[RBP + 0xFFFFFFFFFFFFFBA8])
loc_key_0 (13)
RCX = zeroExt_64(RCX[0:32])
loc_key_4 (0)
RCX = zeroExt_64(RSI[0:32])
loc_key_5 (0)
@32[RBP + 0xFFFFFFFFFFFFFCF0] = RCX[0:32]

The state variable is always located in one of the leaves in the graph, we can easily get all of its parents using the following code.

```
def find_state_var_usedefs(ircfg, search_var):
    var_addrs = set()
    reachings = ReachingDefinitions(ircfg)
    digraph = DiGraphDefUse(reachings)

    for leaf in digraph.leaves():
        if leaf.var == search_var:
            for x in (digraph.reachable_parents(leaf)):
                var_addrs.add(ircfg.get_block(x.label)
[x.index].instr.offset)
    return var_addrs
```

Patching and Reconstructing the Control Flow

While cleaning these useless instructions we have to keep in mind that the call instructions will get affected by this as they are based on relative offsets.
We can fix it using the following code :

```
rel = lambda addr, patch_addr: hex(addr - patch_addr)

for instr in instrs:
    #omitting useless instructions
    if instr.offset not in nop_addrs:
        if instr.is_subcall():
            #generate asm for fixed calls with relative addrs
            patch_addr = start_addr + len(final_patch)
            tgt =
loc_db.get_location_offset(instr.args[0].loc_key)
            call_patch_str = "CALL %s" % rel(tgt, patch_addr)
            call_patch = asmb(call_patch_str, loc_db)
            final_patch += call_patch
        else:
            #add the original bytes
            final_patch += instr.b
```

At last we need to generate a patch for jumps and reconstruct the control flow.
For a simple relevant block we only need a single patch.

```
asmb = lambda patch_str, loc_db: mn_x86.asm(mn_x86.fromstring(patch_str, loc_db,
32))[0]
patch_addr = start_addr + len(final_patch)

n_addr = link['next']
patch = "JMP %s" % rel(n_addr, patch_addr)
jmp_patches = asmb(patch, loc_db)
```

We have two instruction patches for a conditional relevant block. We replace the conditional move with a conditional jump to the true address and add another jump in succession to the false address.

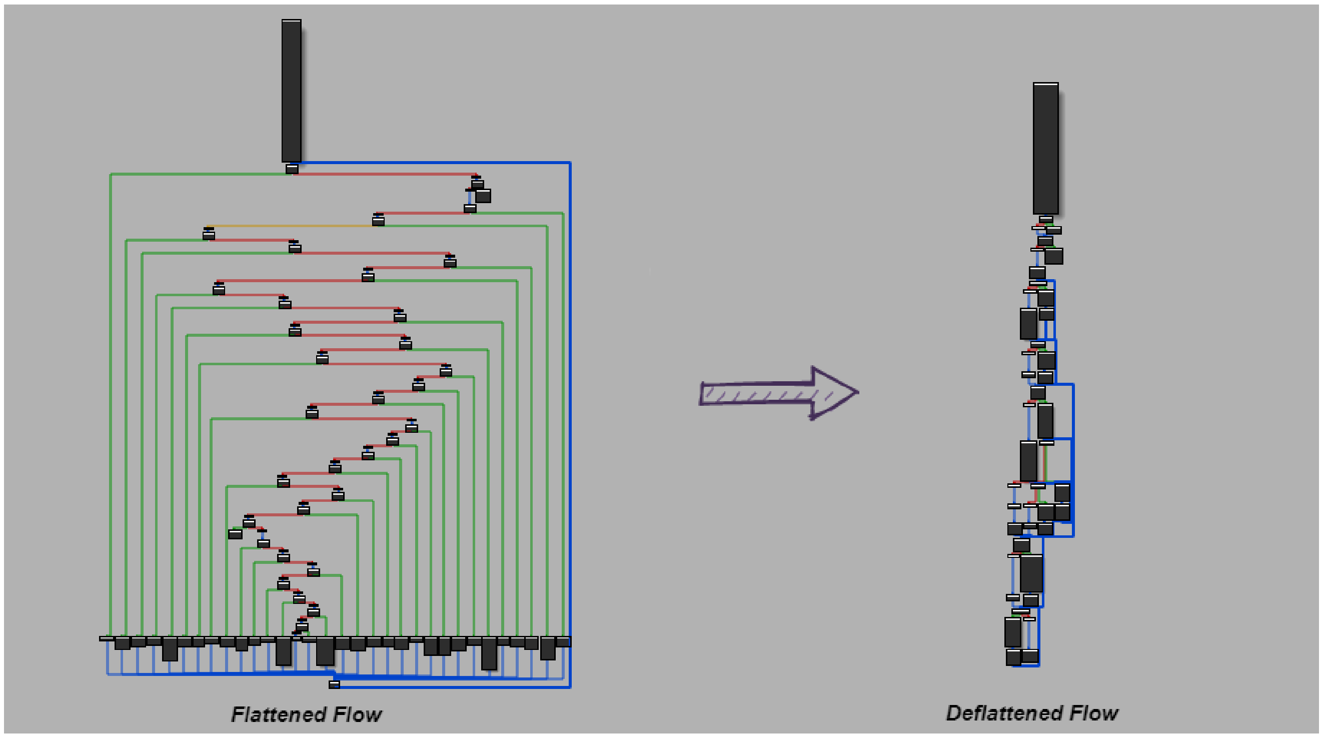
```
t_addr = link['true_next']
f_addr = link['false_next']
jcc = link['cond'].replace('CMOV', 'J')

patch1_str = "%s %s" % (jcc, rel(t_addr,
patch_addr))
jmp_patches += asmb(patch1_str, loc_db)
patch_addr += len(jmp_patches)

patch2_str = "JMP %s" % (rel(f_addr,
patch_addr))
jmp_patches += asmb(patch2_str, loc_db)
```

We can nop out the backbone as it is useless now.

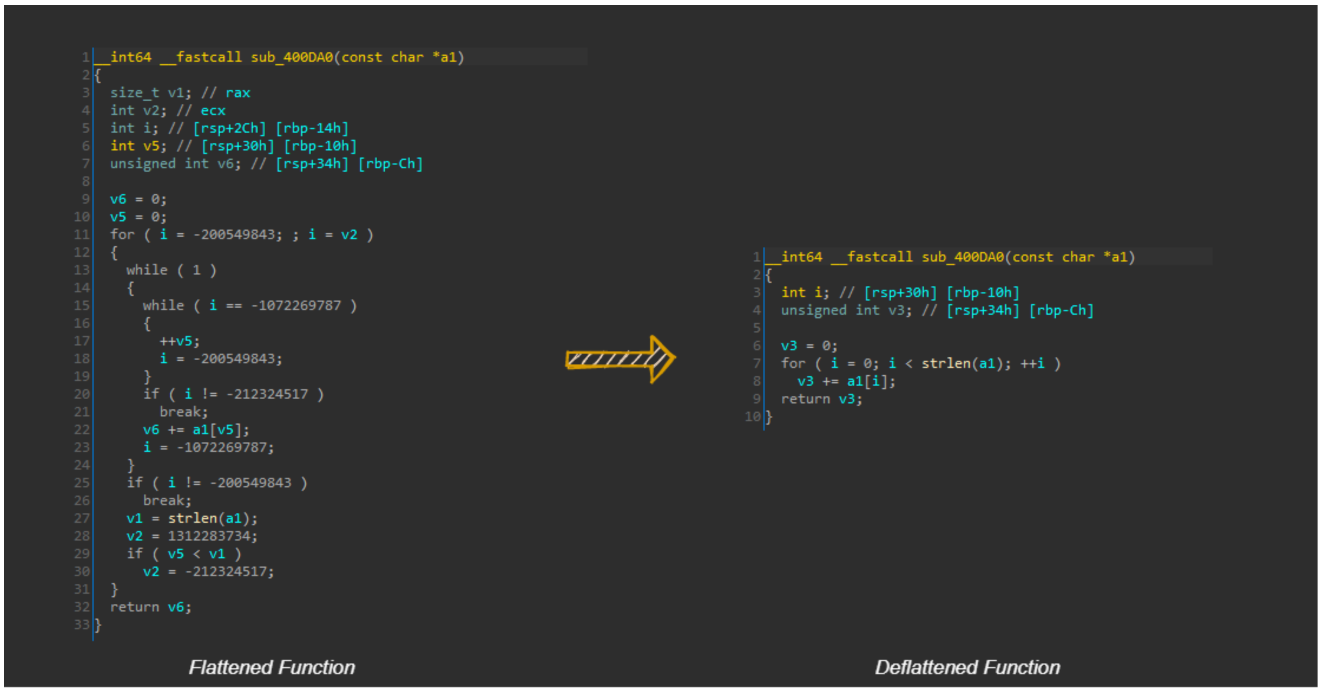
```
backbone_start, backbone_end = dispatcher, tail.offset +
tail.l
nop_len = backbone_end - backbone_start
patches[backbone_start] = b"\x90" * nop_len
```

Final Results

Graph View



F5 View



```
__int64 __fastcall sub_400DA0(const char *a1)
{
  size_t v1; // rax
  int v2; // ecx
  int i; // [rsp+2Ch] [rbp-14h]
  int v5; // [rsp+30h] [rbp-10h]
  unsigned int v6; // [rsp+34h] [rbp-Ch]

  v6 = 0;
  v5 = 0;
  for ( i = -200549843; ; i = v2 )
  {
    while ( 1 )
    {
      while ( i == -1072269787 )
      {
        ++v5;
        i = -200549843;
      }
      if ( i != -212324517 )
        break;
      v6 += a1[v5];
      i = -1072269787;
    }
    if ( i != -200549843 )
      break;
    v1 = strlen(a1);
    v2 = 1312283734;
    if ( v5 < v1 )
      v2 = -212324517;
  }
  return v6;
}
```

Flattened Function

```
__int64 __fastcall sub_400DA0(const char *a1)
{
  int i; // [rsp+30h] [rbp-10h]
  unsigned int v3; // [rsp+34h] [rbp-Ch]

  v3 = 0;
  for ( i = 0; i < strlen(a1); ++i )
    v3 += a1[i];
  return v3;
}
```

Deflattened Function

Get MODeflattener

> I've open sourced the tool on my github. I've added some samples to test it as well. Try it out!
> https://github.com/mrT4ntr4/MODeflattener

Bonus

- Tim Blazytko's flattening heuristic script
  While disassembling the specified function we can look out for other functions used by it and can make use of this script to automatically detect whether it is a flattened one and try to deobfuscate it. This has already been integrated into the tool!

- nop-hider idapython script
  This script hides the nop instructions from IDA graph view as the backbone is converted into a long nop chain after deobfuscation.

References

Dissecting LLVM Obfuscator - RPISEC
Automated Detection of Control-flow Flattening - Tim Blazytko