VB 2023

Deobfuscating virtualized malware using Hex-Rays Decompiler

Georgy Kucherin
Kaspersky GReAT

Obfuscations make malware analysis a difficult thing

Static analysis

Code becomes unreadable

Dynamic analysis

Prevented with anti-sandbox techniques

How obfuscations can be tackled

Identify the obfuscator and search for a deobfuscation tool

Use a debugger to dump unobfuscated code from memory

Advanced actors know all these classic methods

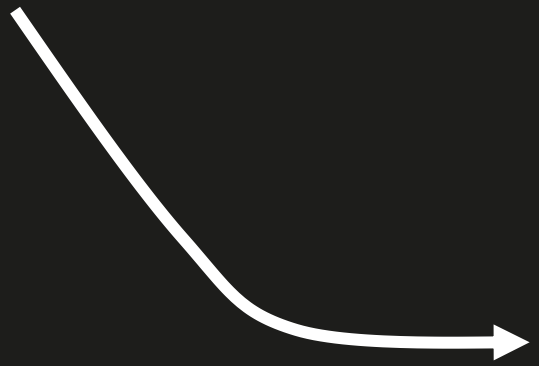
FinFisher's main function

```
push    ebp
mov     ebp, esp
sub     esp, 0C7Ch
push    ebx
push    esi
push    edi
push    735FBBCh
push    edx
xor     edx, edx
pop     edx
jz      loc_401930
```

Function prologue

```
push    ebp
mov     ebp, esp
sub     esp, 0C7Ch
push    ebx
push    esi
push    edi
push    735FBBCh
push    edx
xor     edx, edx
pop     edx
jz      loc_401930
```

This jump is always taken



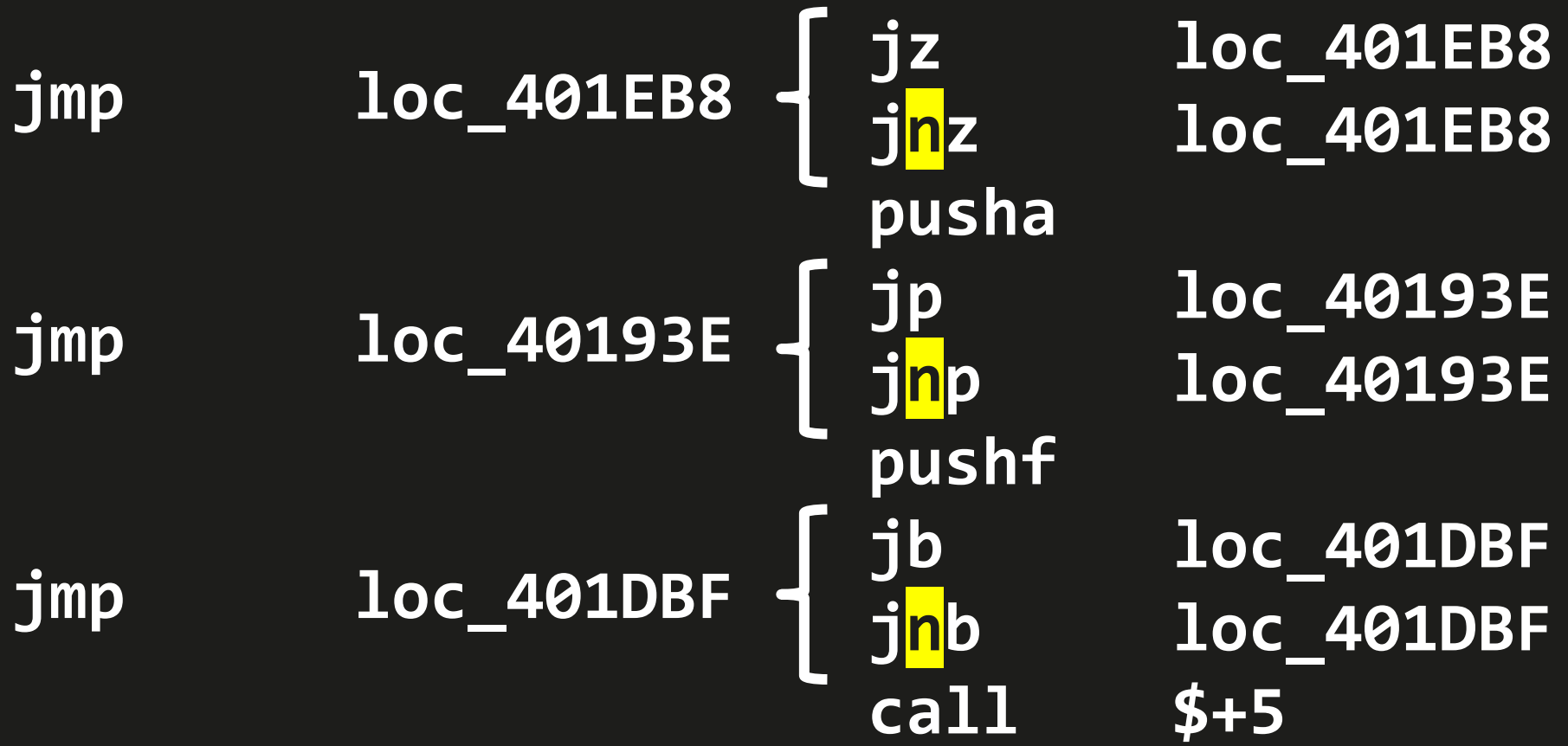
```
push    ebp
mov     ebp, esp
sub     esp, 0C7Ch
push    ebx
push    esi
push    edi
push    735FBBCh
push    edx
xor     edx, edx
pop     edx
jz      loc_401930
```

```
loc_401930:     jz      loc_401EB8
                jnz     loc_401EB8

loc_401EB8:     pusha
                jp      loc_40193E
                jnp     loc_40193E

loc_40193E:     pushf
                jb      loc_401DBF
                jnb     loc_401DBF

loc_401DBF:     call    $+5
                ...
```

Another obfuscation: pairs of opposite jumps

```
jz      loc_401EB8
jnz     loc_401EB8
pusha
jp      loc_40193E
jnp     loc_40193E
pushf
jb      loc_401DBF
jnb     loc_401DBF
call    $+5
...
```

A simple search & replace



Search & replace script (by Rolf Rolles)

Code | Blame | Raw

```
6     class deobX86Hook(idaapi.IDP_Hooks):

def ev_ana_insn(self, insn):
        b1 = idaapi.get_byte(insn.ea)
        if b1 >= 0x70 and b1 <= 0x7F:
                d1 = idaapi.get_byte(insn.ea+1)
                b2 = idaapi.get_byte(insn.ea+2)
                d2 = idaapi.get_byte(insn.ea+3)
                if b2 == b1 ^ 0x01 and d1-2 == d2:
                        idaapi.put_byte(insn.ea, 0xEB)
                        idaapi.put_word(insn.ea+2, 0x9090)
```

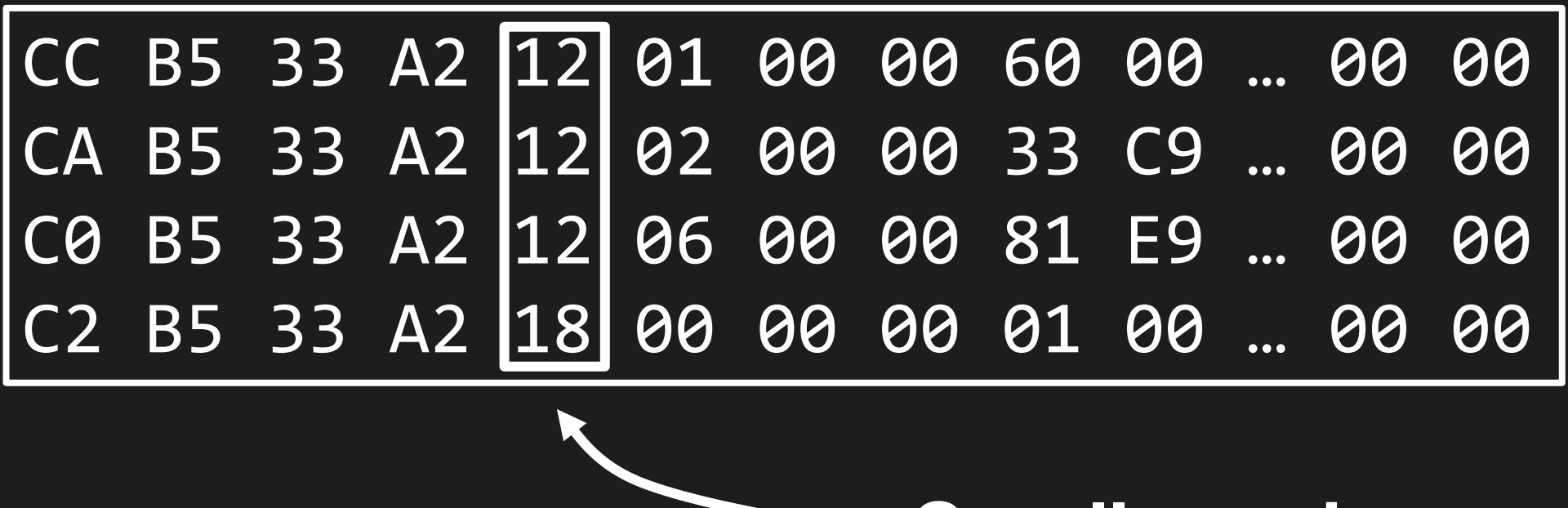
We see this after running it

```
49  v26 = a1;
50  v7 = __readeflags();
51  if ( !sub_4027EA() )
52  {
53    v8 = sub_40260C(4530);
54    qmemcpy(v8, byte_40A789, 0x11B2u);
55    v20 = v8;
56    sub_402456(v8, 0x11B2, xorKey);
57    v20(4, &dword_40A0A0);
58    v21 = sub_40260C(0x10000);
59    (sub_402859)(v21);
```

This code decrypts a blob of data
It consists of bytearrays of size 24

```
CC B5 33 A2 12 01 00 00 60 00 … 00 00
CA B5 33 A2 12 02 00 00 33 C9 … 00 00
C0 B5 33 A2 12 06 00 00 81 E9 … 00 00
C2 B5 33 A2 18 00 00 00 01 00 … 00 00
```

This code decrypts a blob of data
It consists of bytearrays of size 24



That number is an index of this array

```
arrayOfFuncs    dd offset sub_4017B7, offset sub
                dd offset sub_403AC7, offset sub
                dd offset sub_403D5C, offset sub
                dd offset unk_403EBD, offset unk
                dd offset unk_403FD0, offset unk
                dd offset sub_405934, offset sub
                dd offset sub_404347, offset sub
                dd offset unk_405179, offset unk
                dd offset unk_4056E4, offset sub
                dd offset unk_4057EA, offset unk
                dd offset unk_405C77, offset unk
```

Functions in this array are quite small

```
int __usercall sub_4017B7@<eax>(int a1@<e
{
  *(a1 + 8) = 0;
  *a1 += 24;
  return (*(a1 + 16))(0);
}
```

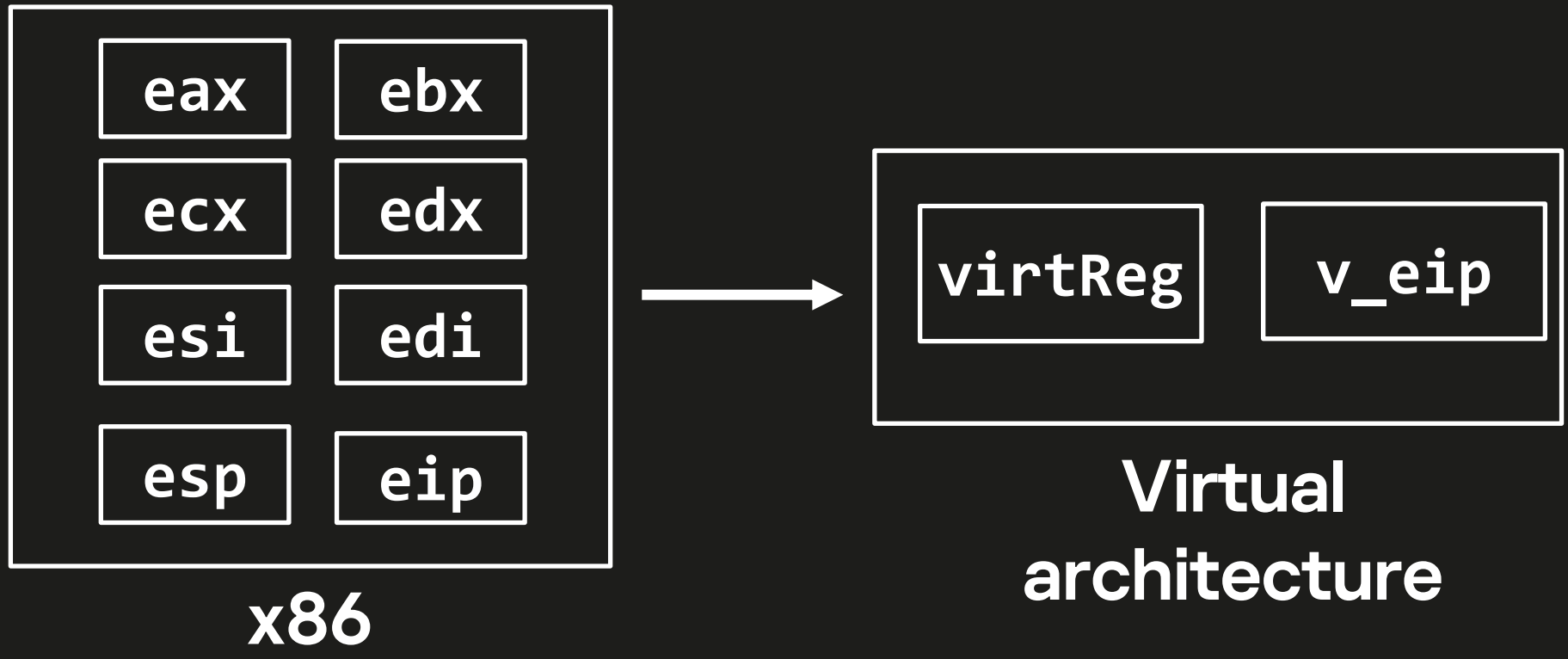
Functions in this array are quite small

```
int __usercall sub_4052CB@<eax>(int a1@<ebx>)
{
  if ( ((**(a1 + 44) & 0x80) != 0) == ((**(a1 + 44)
    return (*(a1 + 24))();
  else
    return (*(a1 + 28))();
}                                                    //
```

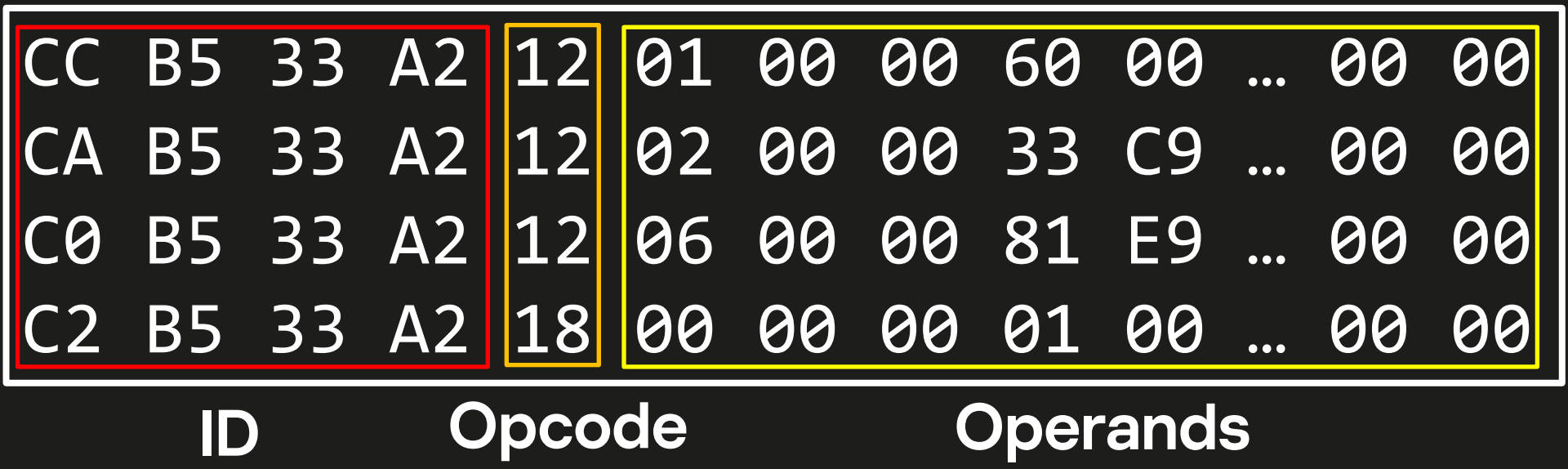
Functions in this array are quite small

```
int __usercall sub_406060@<eax>(int a1@<ebx>)
{
  qmemcpy((*(a1 + 44) - 4), *(a1 + 44), 0x24u);
  *(a1 + 44) -= 4;
  *(*(a1 + 44) + 36) = *(a1 + 8);
  *a1 += 24;
  return (*(a1 + 16))();
}
```

These functions emulate an entire computer architecture



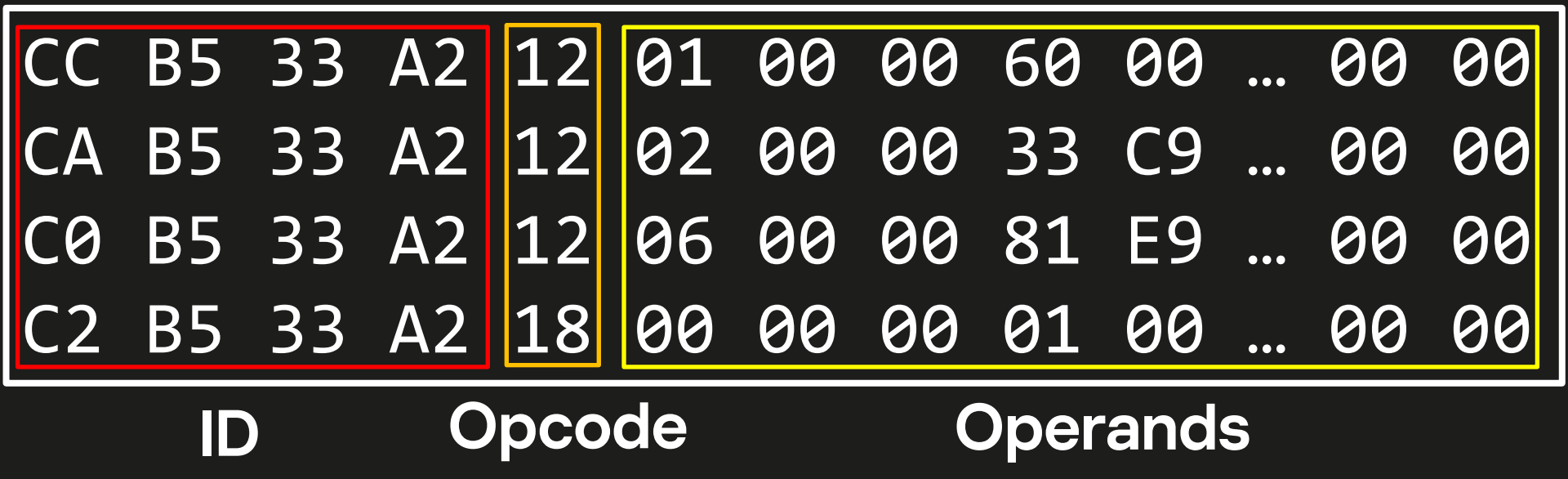
These bytes are instructions of the virtual architecture



These functions emulate instructions

```
arrayOfFuncs    dd offset sub_4017B7, offset sub
                dd offset sub_403AC7, offset sub
                dd offset sub_403D5C, offset sub
                dd offset unk_403EBD, offset unk
                dd offset unk_403FD0, offset unk
                dd offset sub_405934, offset sub
                dd offset sub_404347, offset sub
                dd offset unk_405179, offset unk
                dd offset unk_4056E4, offset sub
                dd offset unk_4057EA, offset unk
                dd offset unk_405C77, offset unk
```

How can you translate these bytes to something sensible?



How can you translate these bytes to something sensible?

- Existing tools are quite complex
- So, let’s try to create our own devirtualizer!
- Let’s also do all the work in IDA

What’s the strategy?

Let’s first
take a closer look
at the virtual instructions

Instruction type 1: jump instructions

```
  var = arg1->savedRegs->flags;
  if ( (var & 0x40) != 0 || (var & 1) != 0 )
    return (arg1->nextInstruction)();
  else
    return (arg1->targetInstruction)();
}
```

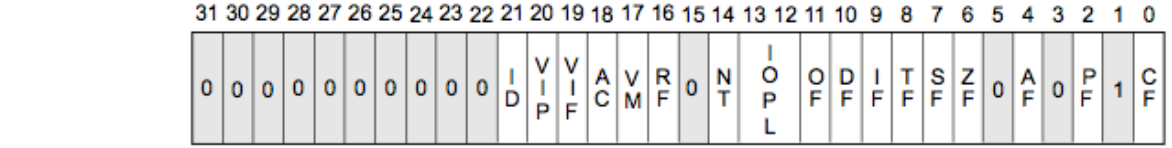
Instruction type 1: jump instructions

```
var = arg1->savedRegs->flags;
if ( (var & 0x40) != 0 || (var & 1) != 0 )
  return (arg1->nextInstruction)();
else
  return (arg1->targetInstruction)();
}
```

0x40 is ??, 0x01 is ??

Instru ons

```
var =
if (                                    = 0 )
   retu
else
   retu
}
```



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

Instruction type 1: jump instructions

```
  var = arg1->savedRegs->flags;
  if ( (var & 0x40) != 0 || (var & 1) != 0 )
    return (arg1->nextInstruction)();
  else
    return (arg1->targetInstruction)();
}
```

0x40 is ZF, 0x01 is CF

Instruction type 1: jump instructions

71 cb	JNO rel8	D	Valid	Valid	Jump short if not overflow (OF=0).
7B cb	JNP rel8	D	Valid	Valid	Jump short if not parity (PF=0).
79 cb	JNS rel8	D	Valid	Valid	Jump short if not sign (SF=0).
75 cb	JNZ rel8	D	Valid	Valid	Jump short if not zero (ZF=0).
70 cb	JO rel8	D	Valid	Valid	Jump short if overflow (OF=1).
7A cb	JP rel8	D	Valid	Valid	Jump short if parity (PF=1).
7A cb	JPE rel8	D	Valid	Valid	Jump short if parity even (PF=1).
7B cb	JPO rel8	D	Valid	Valid	Jump short if parity odd (PF=0).
78 cb	JS rel8	D	Valid	Valid	Jump short if sign (SF=1).
74 cb	JZ rel8	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 cw	JA rel16	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 cd	JA rel32	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 cw	JAE rel16	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.
0F 83 cd	JAE rel32	D	Valid	Valid	Jump near if above or equal (CF=0).

Instruction type 1: jump instructions

```
  var = arg1->savedRegs->flags;
  if ( (var & 0x40) != 0 || (var & 1) != 0 )
    return (arg1->nextInstruction)();
  else
    return (arg1->targetInstruction)();
}
```

0x40 is ZF, 0x01 is CF:
“Jump if above” (JA) instruction

Instruction type 1: jump instructions

```
void __usercall cmd_Jnz(VMContext *a1@<ebx>)
{
  if ( (a1->savedRegs->flags & 0x40) != 0 )
    __asm { jmp     dword ptr [ebx+1Ch] }
  __asm { jmp     dword ptr [ebx+18h] }
}
```

0x40 is JZ

“Jump if not zero” (JNZ) instruction

Instruction type 1: jump instructions

```
int __usercall cmd_Jo@<eax>(VMContext *a1@<ebx>)
{
  if ( (a1->savedRegs->flags & 0x800) != 0 )
    return (a1->targetInstruction)();
  else
    return (a1->nextInstruction)();
}
```

0x800 is JO

“Jump if overflow” (JO) instruction

Instruction type 2: virtual register stuff

```
void __usercall cmd_Add(VMContext *a1@<e
{
  a1->virtReg += *a1->instruction.data;
  ++a1->instructionIndex;
  __asm { jmp     dword ptr [ebx+10h] }
}
```

Add number to virtual register

Instruction type 2: virtual register stuff

```
int __usercall cmd_Shl@<eax>(VMContext
{
  a1->virtReg <<= *a1->instruction.data
  ++a1->instructionIndex;
  return (a1->cmd_complete)();
}
```

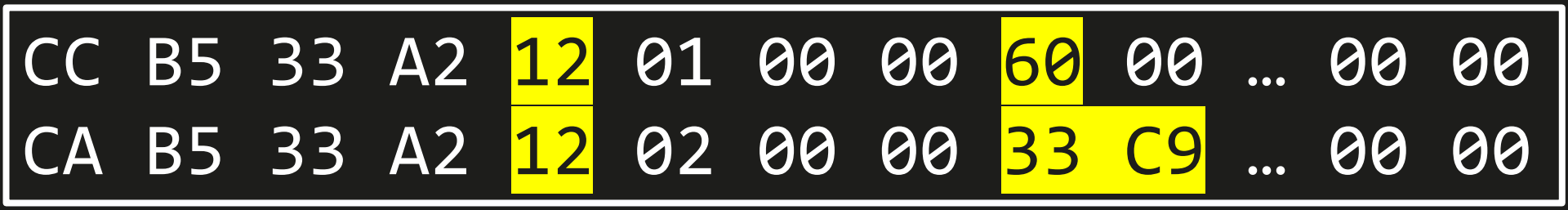
Shift virtual register left

Instruction type 2: virtual register stuff

```
int __usercall cmd_Ld@<eax>(VMContext *a
{
  a1->virtReg = *a1->virtReg;
  ++a1->instructionIndex;
  return (a1->cmd_complete)();
}
```

Dereference virtual register

Instruction type 3: execute x86 code



60: pushad

33 C9: xor ecx, ecx

A lot of virtual instructions resemble x86 ones

- So, let’s try to translate virtual instructions to x86
- Translating jump instructions is an easy task
- But what about virtual register instructions?

Translating virtual register instructions

```
shl virtReg, 3
```

Translating virtual register instructions

```
shl virtReg, 3
```

```
shl
eax, 3
```

Replace
virtual register
with a real one

Translating virtual register instructions

```
shl virtReg, 3
```

```
shl
eax, 3
```

Replace virtual register with a real one

```
shl
dword ptr [virtReg], 3
```

Or place the virtual register in memory

Looking at the first option

Is it a good idea to replace virtReg with eax?

```
exec {mov eax, 3}
mov virtReg, 0x400300
mov [virtReg], eax
```

Value in 0x400300:
0x3

Looking at the first option

Is it a good idea to replace virtReg with eax?

```
exec {mov eax, 3}
mov virtReg, 0x400300
mov [virtReg], eax
```

Value in 0x400300:
0x3

```
mov eax, 3
mov eax, 0x400300
mov [eax], eax
```

Value in 0x400300:
0x400300

Translating virtual register instructions

```
shl virtReg, 3
```

```
shl
eax, 3
```

Replace virtual register with a real one

```
shl
dword ptr [virtReg], 3
```

Or place the virtual register in memory

Translation examples

```
shl virtReg, 3
```

```
shl dword ptr [virtReg], 3
```

Translation examples

```
add virtReg, 3
```

```
add dword ptr [virtReg], 3
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     [virtReg], [[virtReg]]
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     [virtReg], [[virtReg]]
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     reg, [mem] - OK
mov     [mem], reg - OK
mov     [mem], [mem] - bad
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     eax, [virtReg]
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     eax, [virtReg]
mov     eax, [eax]
```

Translation examples

```
mov     virtReg, [virtReg]
```

```
mov     eax, [virtReg]
mov     eax, [eax]
mov     [virtReg], eax
```

Translation examples

```
mov       virtReg, [virtReg]
```

```
push      eax
mov       eax, [virtReg]
mov       eax, [eax]
mov       [virtReg], eax
pop       eax
```

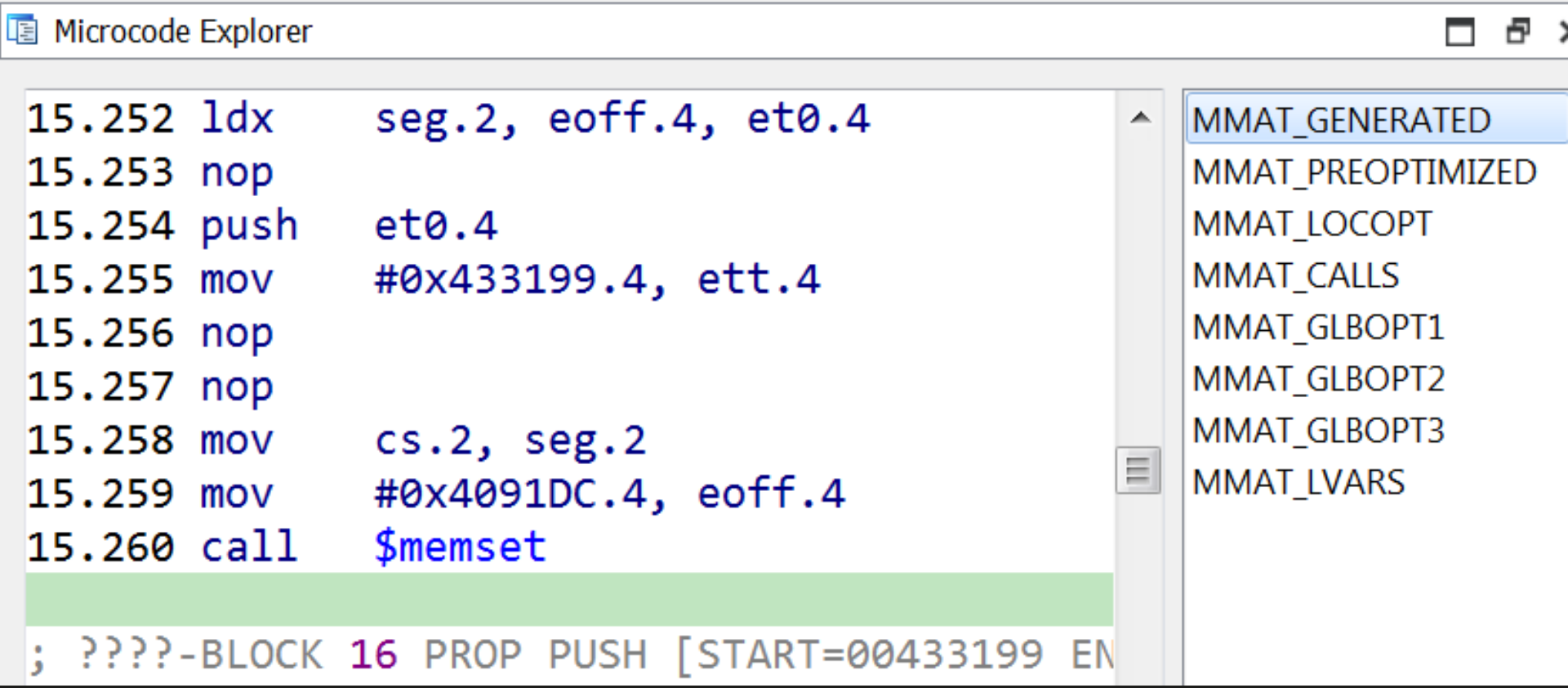
Now we can:

- Iterate over virtual instructions
- Write their translations to our IDA database

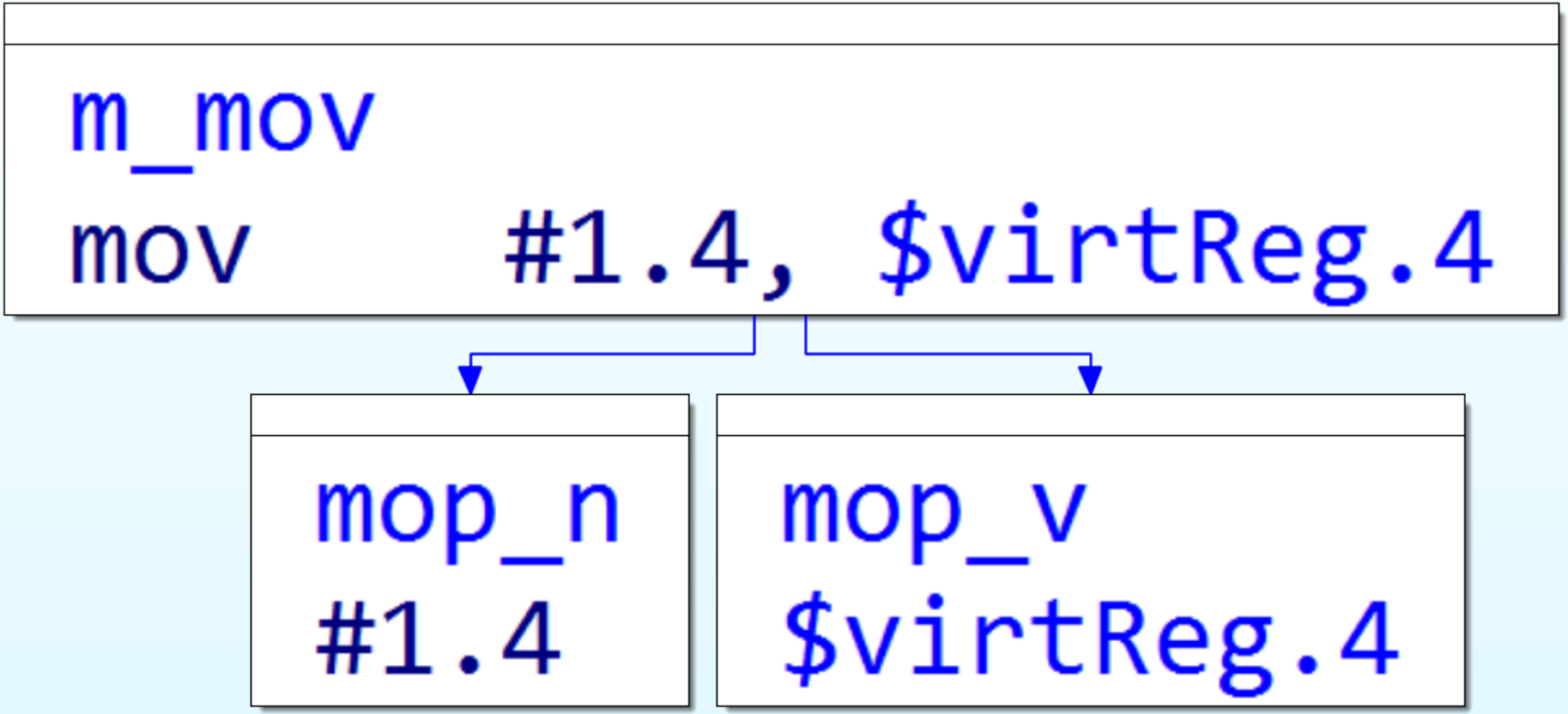
Example code

```
elif insn_opcode == VMOpcodes.DerefVirtReg:
    x86_insn_bytes[0] = 0x50 # Write the "push eax" instruction
    x86_insn_bytes[1] = 0xa1 # Write the "mov eax, [virtReg]" instruction
    x86_insn_bytes[2:2+4] = virt_reg_addr.to_bytes(4, 'little', signed=True)
    x86_insn_bytes[6] = 0x8b # Write the "mov eax, [eax]" instruction
    x86_insn_bytes[7] = 0x00
    x86_insn_bytes[8] = 0xa3 # Write the "mov [virtReg], eax" instruction
    x86_insn_bytes[9:9+4] = virt_reg_addr.to_bytes(4, 'little', signed=True)
    x86_insn_bytes[13] = 0x58 # Write the "pop eax" instruction
# Adds an x86 register to a virtual register
```

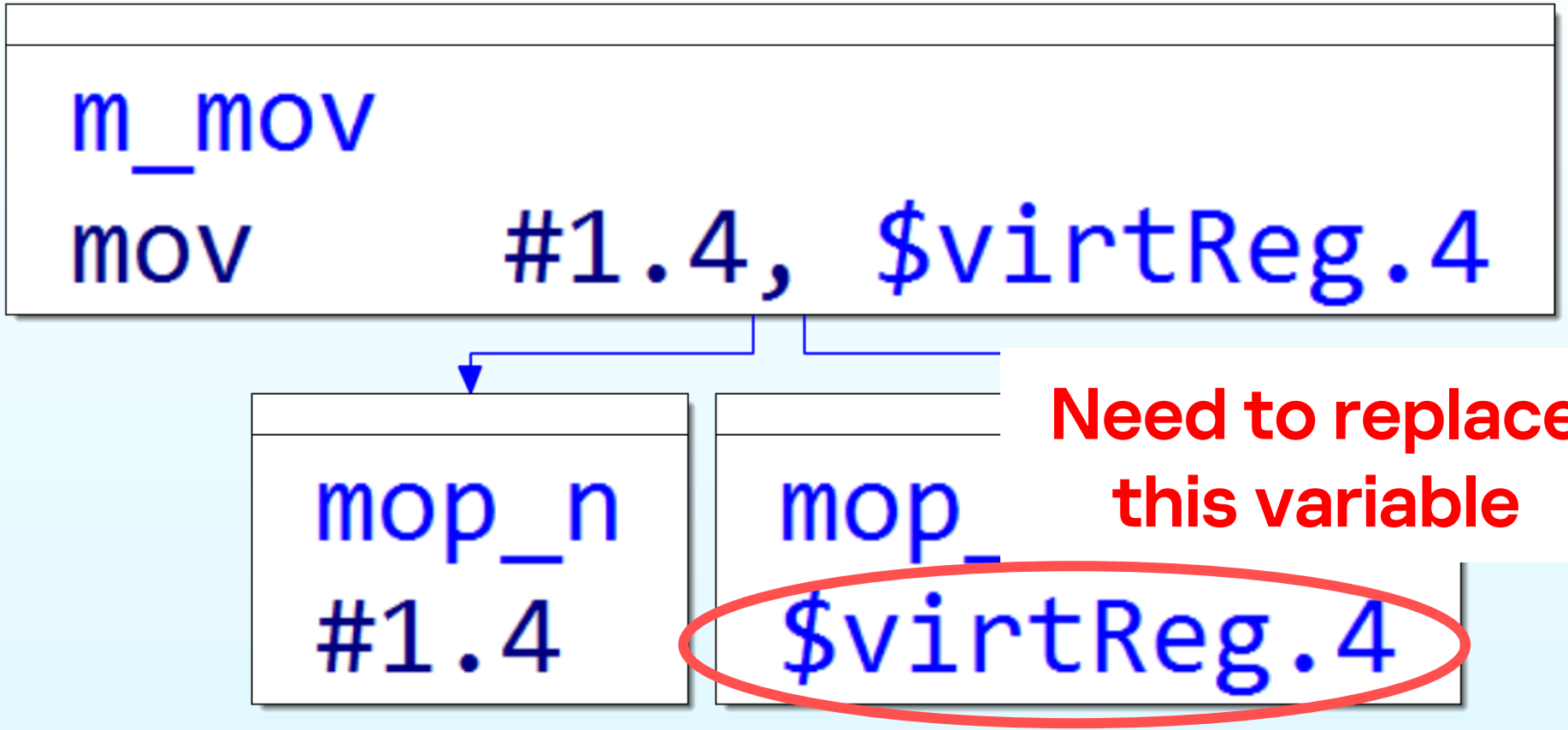
Looking at the function's microcode



Looking at the function's microcode



Looking at the function's microcode



Kernel registers

- x86: eax, ebx, ecx, edx, esi, edi, ebp, esp
- Microcode: any number of registers
- We can tell IDA to treat the virtReg variable as a register

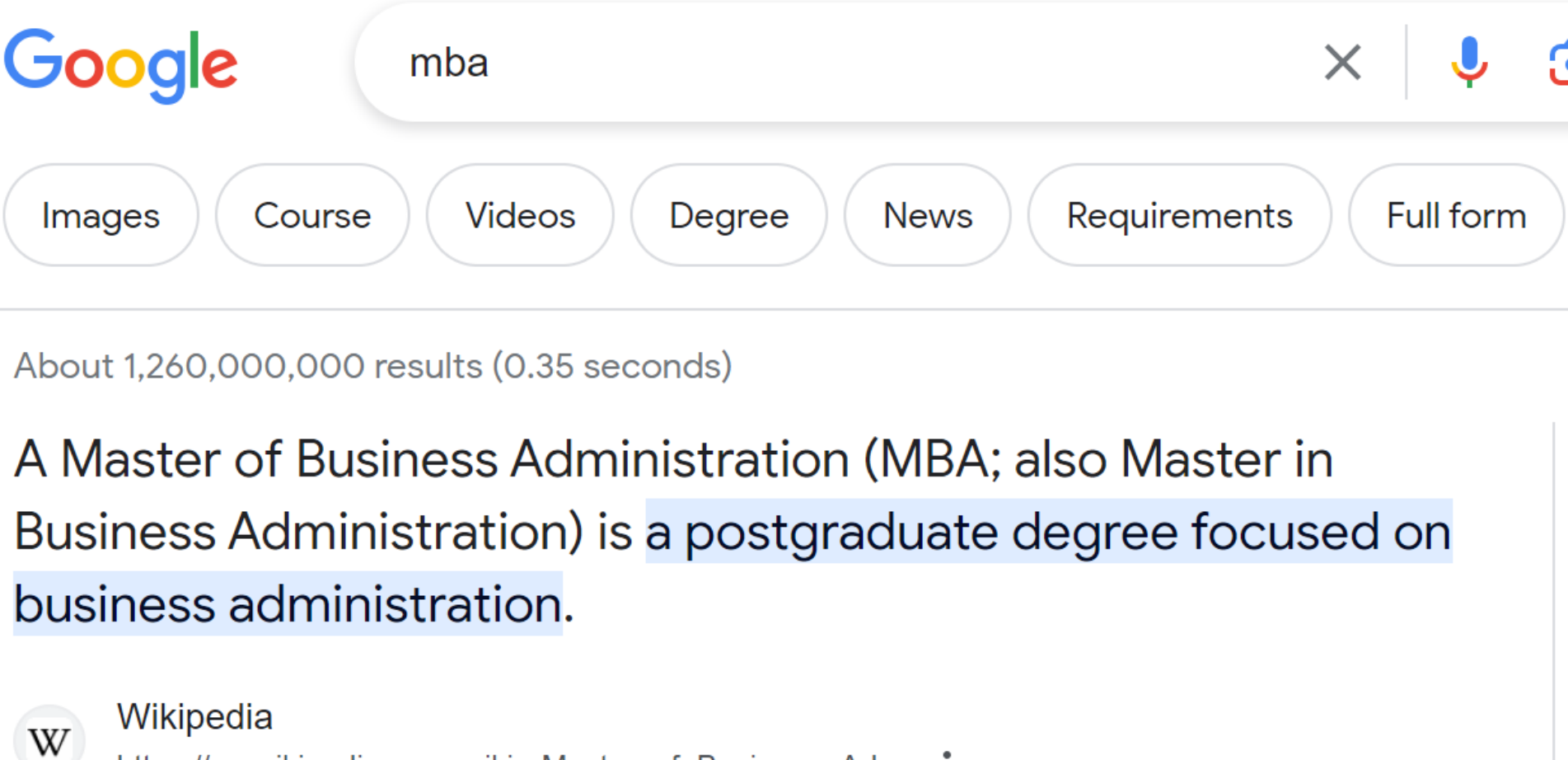
Kernel registers

```
class DecompilerHook(ida_hexrays.Hexrays_Hooks):
    def preoptimized(self, *args):
        global kernel_register
        mba = args[0]
        kernel_register = mba.alloc_kreg(4)
        if kernel_register != ida_hexrays.mr_none:
            repl = OperandReplacer()
        mba.for_all_ops(repl)
        return 0
```

Kernel registers

```
class DecompilerHook(ida_hexrays.Hexrays_Hooks):
    def preoptimized(self, *args):
        global kernel_register
        mba = args[0]
        kernel_register = mba.alloc_kreg(4)
        if kernel_register != ida_hexrays.mr_none:
            repl = OperandReplacer()
        mba.for_all_ops(repl)
        return 0
```

Kernel registers



Kernel registers

```
class DecompilerHook(ida_hexrays.Hexrays_Hooks):
    def preoptimized(self, *args):
        global kernel_register
        mba = args[0]
        kernel_register = mba.alloc_kreg(4)
        if kernel_register != ida_hexrays.mr_none:
            repl = OperandReplacer()
            mba.for_all_ops(repl)
        return 0
```

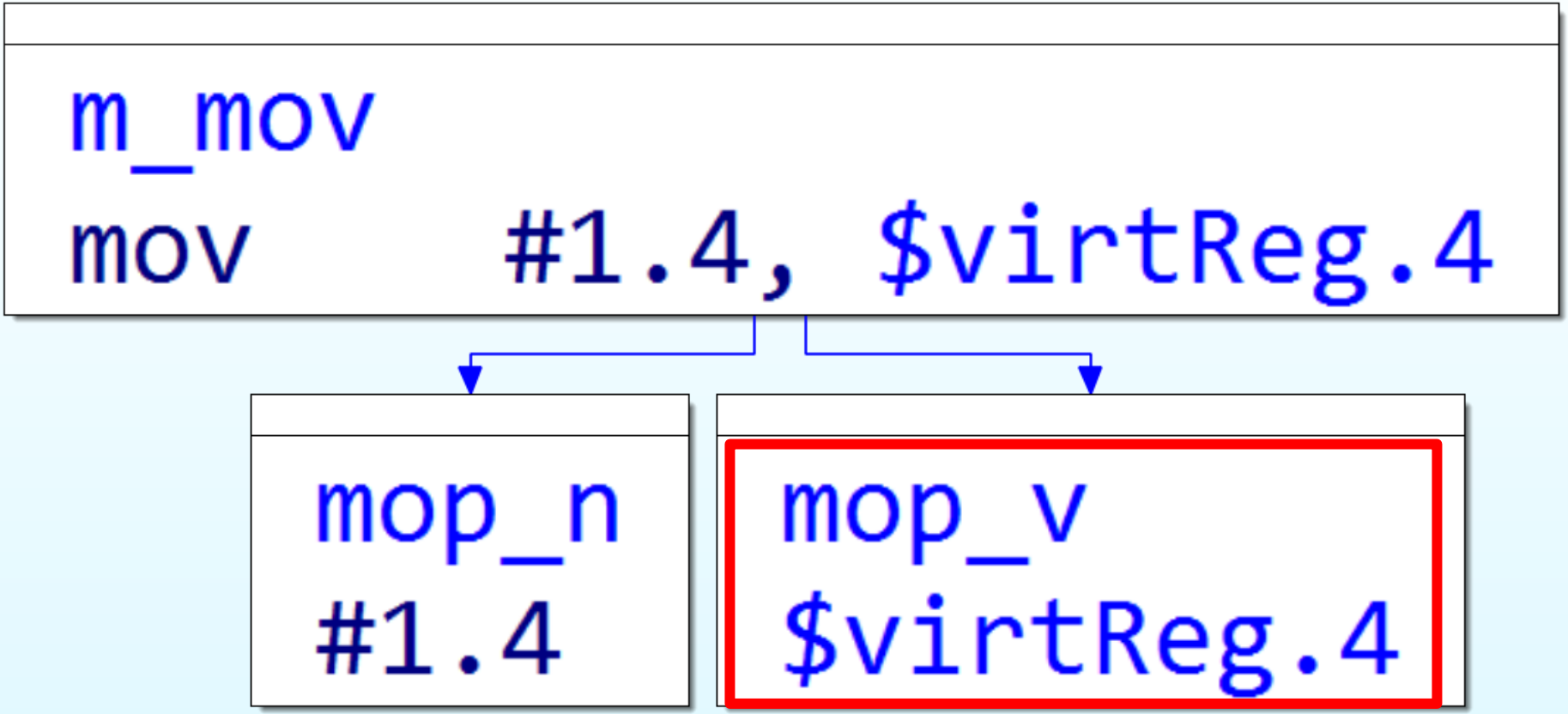
Kernel registers

```
class DecompilerHook(ida_hexrays.Hexrays_Hooks):
    def preoptimized(self, *args):
        global kernel_register
        mba = args[0]
        kernel_register = mba.alloc_kreg(4)
        if kernel_register != ida_hexrays.mr_none:
            repl = OperandReplacer()
            mba.for_all_ops(repl)
        return 0
```

Kernel registers

```
class OperandReplacer(ida_hexrays.mop_visitor_t):
    def visit_mop(self, op, op_type, is_target):
        global kernel_register
        if op.t == ida_hexrays.mop_v \
        and op.g == register_addr:
            op.make_reg(kernel_register, op.size)
            if self.blk:
                self.blk.mark_lists_dirty()
        return 0
```

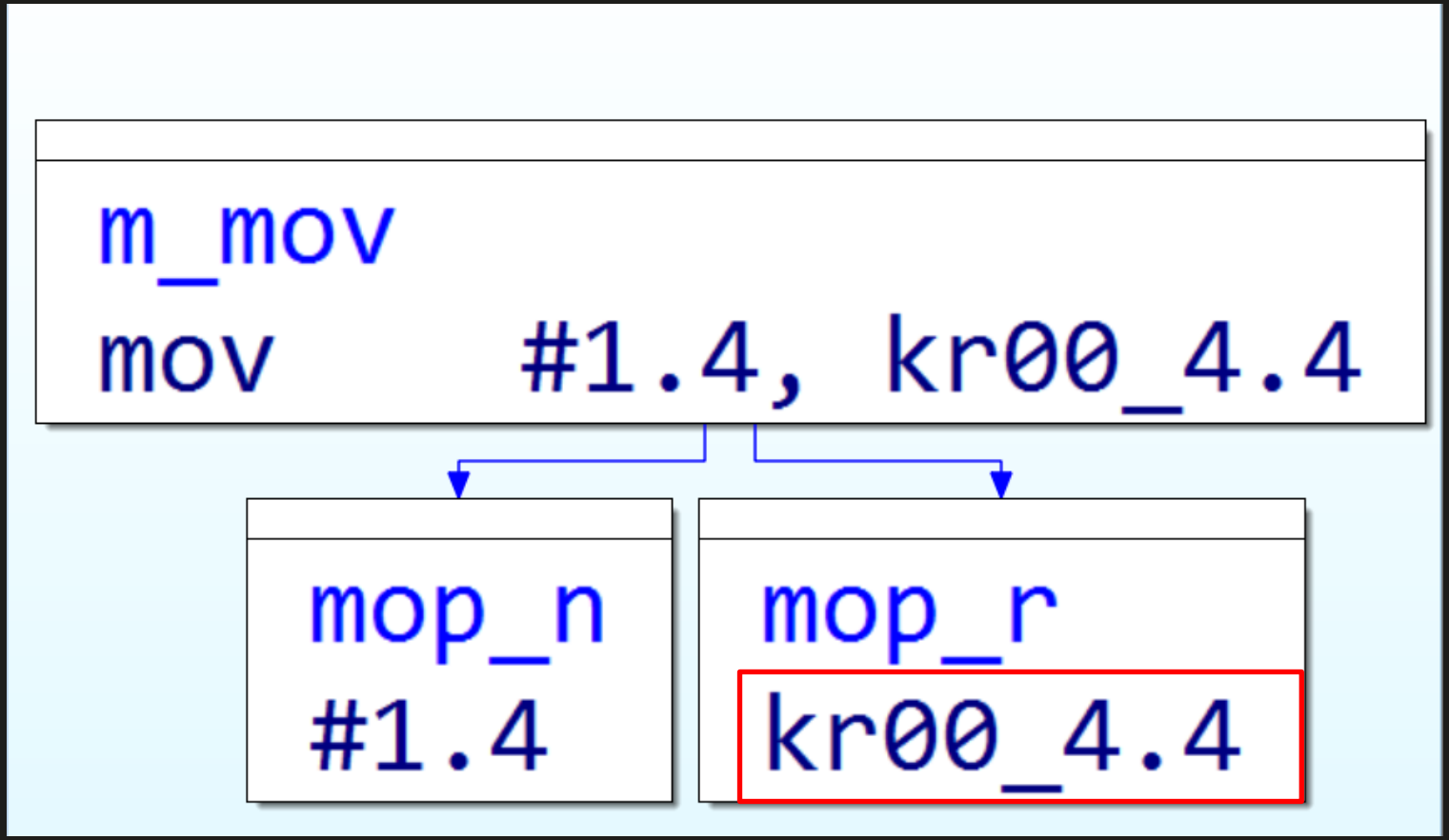
Looking at the function's microcode



Kernel registers

```
class OperandReplacer(ida_hexrays.mop_visitor_t):
    def visit_mop(self, op, op_type, is_target):
        global kernel_register
        if op.t == ida_hexrays.mop_v \
        and op.g == register_addr:
            op.make_reg(kernel_register, op.size)
            if self.blk:
                self.blk.mark_lists_dirty()
        return 0
```

After replacement



Conclusions

- Virtualization can be very complex to tackle
- Use of IDA and the Hex-Rays decompiler helped us save time
- Total code size: about 240 lines

Thank you!

The devirtualizer code is available at

https://github.com/gkucherin/finspy_devirtualizer

Please feel free to ask me questions!

Email: georgy.kucherin@gmail.com

Twitter: @kucher1n