THE 25-YEAR-OLD INTEL PATENT THAT KILLED YOUR DISASSEMBLERS ft. colby57

Hello everyone, it's been a while since my last post, and today we're going to fix that.

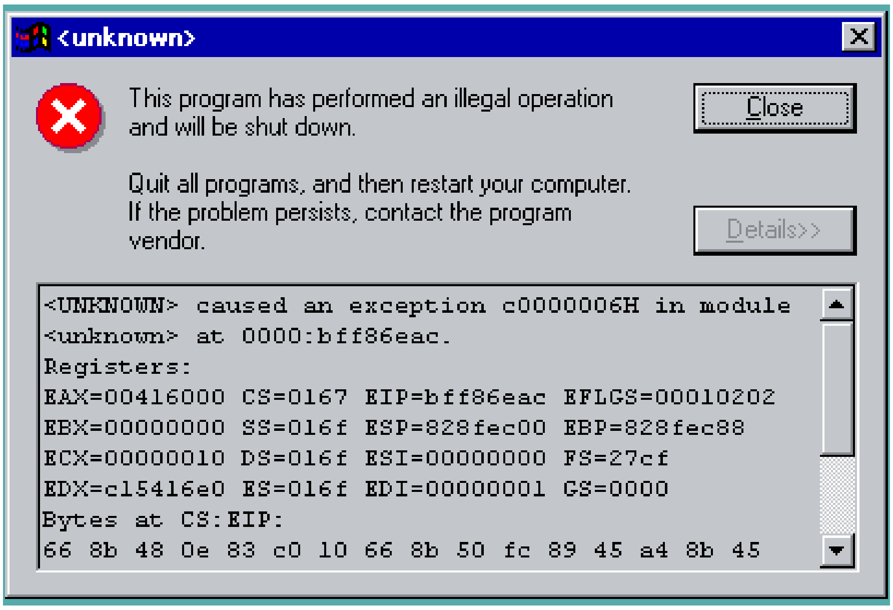
Our story began with a message from my friend-reverser colby57: *"The first time I saw this, I thought I was about to crash, but everything was valid"*. The attachment was a shell sample. Nothing special at first glance, but in the middle of the code, the disassembler simply broke off, showing "garbage" bytes further on. The processor, however, executed this code without any issues.

This "garbage" turned out to be two opcodes: `0F 1A` and `0F 1B`. There are actually more of them, but only two will be mentioned.

And this isn't a bug in a specific sample. It's a systemic failure of the entire reverse engineering industry, laid down by an Intel patent almost 30 years ago.

Digging into the past: where did these ghosts come from?

The roots of our problem go back to 1997, to one of Intel's most elegant patents, created at a time when the world was preparing for the Y2K problem. Back then, Intel faced an eternal headache - backward compatibility. Engineers added bunches of cool instructions to new processors, but developers wouldn't touch them, fearing their programs wouldn't run on computers released a couple of years earlier.



This program has performed an illegal operation and will be shut down.

Quit all programs, and then restart your computer. If the problem persists, contact the program vendor.

```
<UNKNOWN> caused an exception c0000006H in module
<unknown> at 0000:bff86eac.
Registers:
EAX=00416000 CS=0167 EIP=bff86eac EFLGS=00010202
EBX=00000000 SS=016f ESP=828fec00 EBP=828fec88
ECX=00000010 DS=016f ESI=00000000 FS=27cf
EDX=c15416e0 ES=016f EDI=00000001 GS=0000
Bytes at CS:EIP:
66 8b 48 0e 83 c0 10 66 8b 50 fc 89 45 a4 8b 45
```

The solution proposed in patent **US5,701,442** was brilliantly simple. "What if we reserve an entire range of opcodes - from `0x0F18` to `0x0F1F` - as just placeholders?" In the patent, they were called **Hintable NOPs**.

On processors of that time, they did absolutely nothing. But at any moment, any placeholder could "come alive" and turn into a new, useful instruction. On older computers, such an instruction would execute as a harmless NOP, on newer ones - as a powerful tool.

And so it happened. `0x0F18` turned into the **PREFETCH** family of instructions, and `0x0F1E` after twenty years became the heart of CET protection technology in the form of **ENDBR64**. Most opcodes from this range were eventually either utilized or, at the very least, correctly recognized by disassemblers.

Except for two. Opcodes **`0F 1A`** and **`0F 1B`** became real ghosts. They remained in the shadows, forgotten by everyone except the silicon itself and the pages of that very patent.

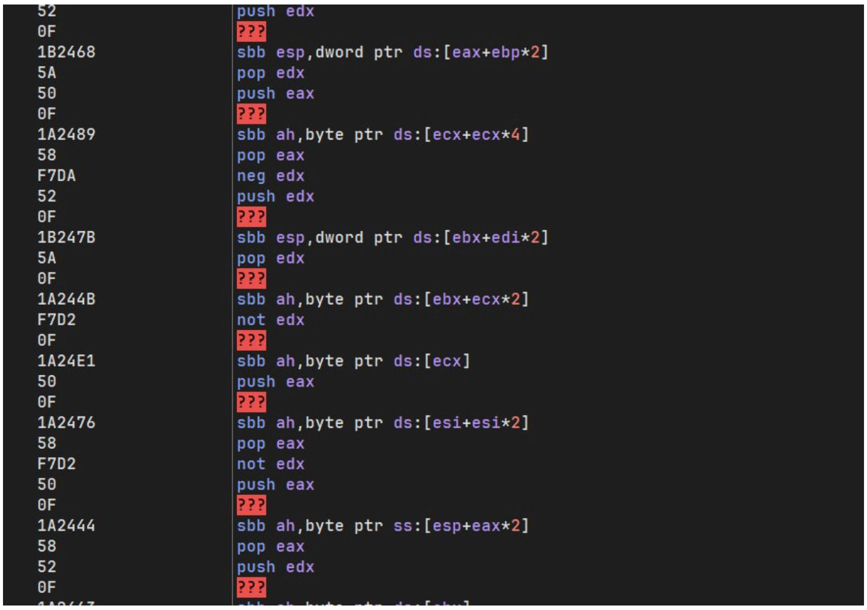
Live test: breaking everything

Enough theory. My good friend wrote a simple PoC binary containing these instructions, and we decided to run it through our pack of standard tools. Let's look at this together.

GitHub repository with PoC

First patient: x64dbg

We load our binary into the debugger. Here they are, our "garbage" bytes. x64dbg honestly shows us `???`, unable to recognize the instruction.



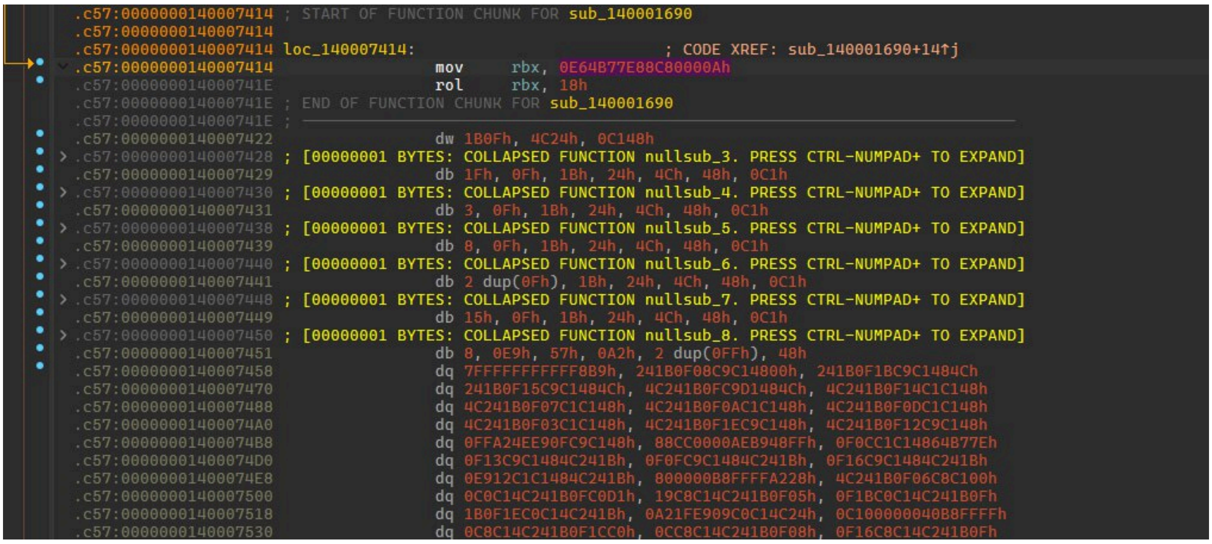
```
52                   push edx
0F                   ???
1B2468               sbb esp,dword ptr ds:[eax+ebp*2]
5A                   pop edx
50                   push eax
0F                   ???
1A2489               sbb ah,byte ptr ds:[ecx+ecx*4]
58                   pop eax
F7DA                 neg edx
52                   push edx
0F                   ???
1B247B               sbb esp,dword ptr ds:[ebx+edi*2]
5A                   pop edx
0F                   ???
1A244B               sbb ah,byte ptr ds:[ebx+ecx*2]
F7D2                 not edx
0F                   ???
1A24E1               sbb ah,byte ptr ds:[ecx]
50                   push eax
0F                   ???
1A2476               sbb ah,byte ptr ds:[esi+esi*2]
58                   pop eax
F7D2                 not edx
50                   push eax
0F                   ???
1A2444               sbb ah,byte ptr ss:[esp+eax*2]
58                   pop eax
52                   push edx
0F                   ???
```

The EIP arrow points to the line with bytes 0F 1A C0. In the disassembler column on this line are three question marks.

But the most interesting thing is the processor's behavior. We step (F7), and... EIP calmly jumps over this "unknown" instruction and continues. No crash, no error. For the CPU, it's valid, executable code. For the debugger - some nonsense

Second patient: IDA Pro

And what will our beloved IDA say? We open the binary and see a depressing picture.

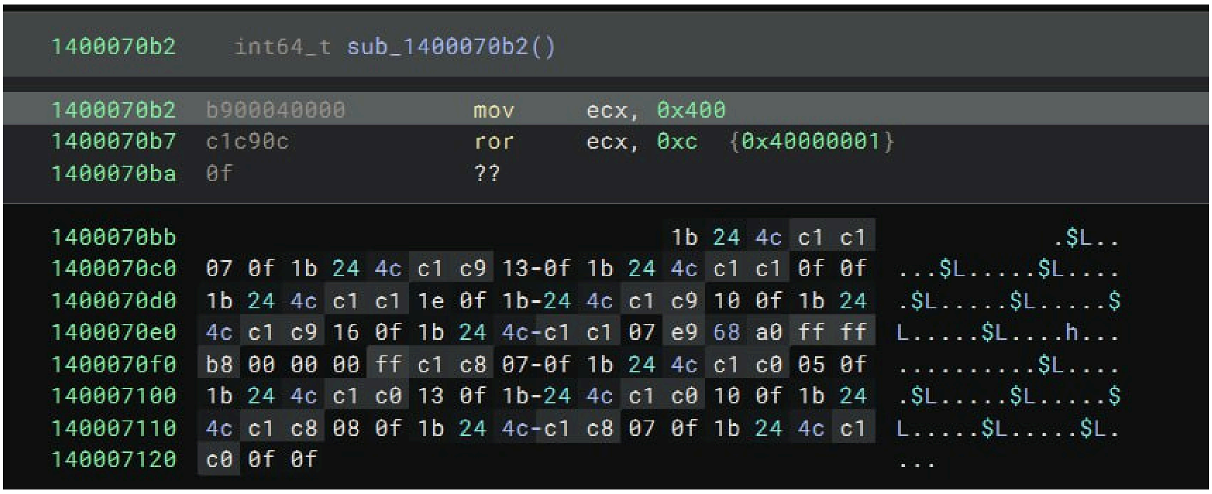


```
.c57:0000000140007414 ; START OF FUNCTION CHUNK FOR sub_140001690
.c57:0000000140007414
.c57:0000000140007414 loc_140007414:                          ; CODE XREF: sub_140001690+14↑j
.c57:0000000140007414                 mov     rbx, 0E64B77E88C80000Ah
.c57:000000014000741E                 rol     rbx, 18h
.c57:000000014000741E ; END OF FUNCTION CHUNK FOR sub_140001690
.c57:000000014000741E ; ───────────────────────────────────────────────────────────
.c57:0000000140007422                 dw 1B0Fh, 4C24h, 0C148h
.c57:0000000140007428 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_3. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007429                 db 1Fh, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007430 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_4. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007431                 db 3, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007438 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_5. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007439                 db 8, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007440 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_6. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007441                 db 2 dup(0Fh), 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007448 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_7. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007449                 db 15h, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007450 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_8. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007451                 db 8, 0E9h, 57h, 0A2h, 2 dup(0FFh), 48h
.c57:0000000140007458                 dq 7FFFFFFFFFFF8B9h, 241B0F08C9C14800h, 241B0F1BC9C1484Ch
.c57:0000000140007470                 dq 241B0F15C9C1484Ch, 4C241B0FC9D1484Ch, 4C241B0F14C1C148h
.c57:0000000140007488                 dq 4C241B0F07C1C148h, 4C241B0F0AC1C148h, 4C241B0F0DC1C148h
.c57:00000001400074A0                 dq 4C241B0F03C1C148h, 4C241B0F1EC9C148h, 4C241B0F12C9C148h
.c57:00000001400074B8                 dq 0FFA24EE90FC9C148h, 88CC0000AEB948FFh, 0F0CC1C14864B77Eh
.c57:00000001400074D0                 dq 0F13C9C1484C241Bh, 0F0FC9C1484C241Bh, 0F16C9C1484C241Bh
.c57:00000001400074E8                 dq 0E912C1C1484C241Bh, 800000B8FFFFA228h, 4C241B0F06C8C100h
.c57:0000000140007500                 dq 0C0C14C241B0FC0D1h, 19C8C14C241B0F05h, 0F1BC0C14C241B0Fh
.c57:0000000140007518                 dq 1B0F1EC0C14C241Bh, 0A21FE909C0C14C24h, 0C100000040B8FFFFh
.c57:0000000140007530                 dq 0C8C14C241B0F1CC0h, 0CC8C14C241B0F08h, 0F16C8C14C241B0Fh
```

IDA doesn't just not understand the instruction. It decides that **the executable code ends here** and data begins. The function analysis breaks off. All subsequent code simply ceases to exist for the static analyzer. This is very bad, but it's business as usual for it.

Third patient: Binary Ninja & Ghidra

No miracle happened here either. Binary Ninja marks the instruction as `??`. Ghidra shows `undefined`. The same story - the industry's leading tools see valid code as an error



```
1400070b2    int64_t sub_1400070b2()

1400070b2  b900040000          mov     ecx, 0x400
1400070b7  c1c90c              ror     ecx, 0xc  {0x40000001}
1400070ba  0f                  ??

1400070bb                                            1b 24 4c c1 c1              .$L..
1400070c0  07 0f 1b 24 4c c1 c9 13-0f 1b 24 4c c1 c1 0f 0f  ...$L.....$L....
1400070d0  1b 24 4c c1 c1 1e 0f 1b-24 4c c1 c9 10 0f 1b 24  .$L.....$L.....$
1400070e0  4c c1 c9 16 0f 1b 24 4c-c1 c1 07 e9 68 a0 ff ff  L.....$L....h...
1400070f0  b8 00 00 00 ff c1 c8 07-0f 1b 24 4c c1 c0 05 0f  ..........$L....
140007100  1b 24 4c c1 c0 13 0f 1b-24 4c c1 c0 10 0f 1b 24  .$L.....$L.....$
140007110  4c c1 c8 08 0f 1b 24 4c-c1 c8 07 0f 1b 24 4c c1  L.....$L.....$L.
140007120  c0 0f 0f                                         ...
```

How can we use this?

So what do these two opcodes give an attacker in practice?

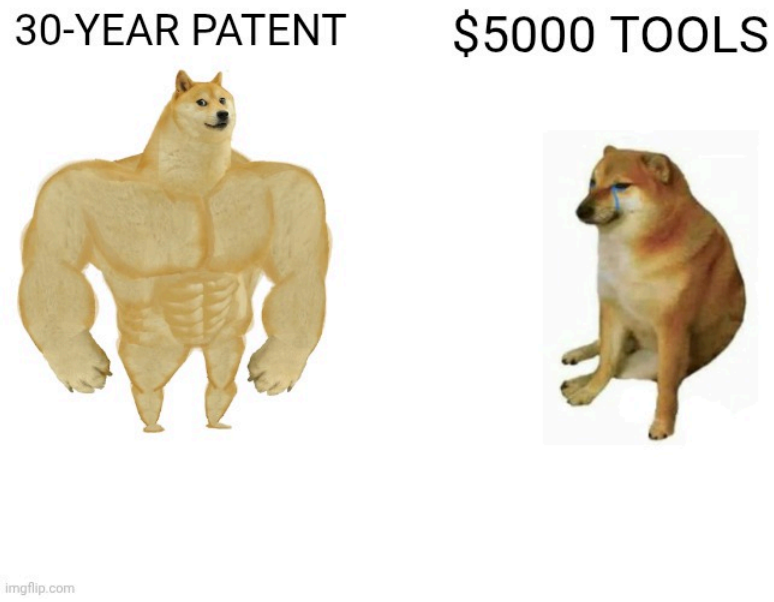
The most obvious - **breaking analysis**. As we saw in IDA, one such NOP - and the function analysis breaks off. All subsequent code simply ceases to exist for the disassembler.

A higher-level example: **creating fake execution graphs**. After a `jmp`, a combination of `0F 1B E8...` is placed, which the disassembler will mistakenly take as a `CALL`, sending the analyst down a false trail that leads nowhere.

And this is just the tip of the iceberg; thousands of different applications can be devised.

An industry problem. A human problem.

But all these tricks are just consequences. The real problem is that a fundamental, documented part of the x86 architecture remained invisible to our most advanced tools for **decades**.



Think about it. This isn't some exotic feature from secret manuals. Not an undocumented feature of a new processor. These are basic instructions described in a patent almost 30 years ago. And all this time they just... existed. Right under our noses.

The sample that started it all revealed not a vulnerability in the processor, but a huge problem in our approach.

That's all for now. Thanks for reading.