Anti Disassembly

Bui Viet Dung

June 26, 2025

1 Understand Disassembly

Disassemblers parse bytes as machine instructions, starting from a known entry point into human-readable assembly code. When a program is compiled, the code you write (C, C++, etc) is converted by the compiler into machine code – sequences of bytes like:

```
B8 04 00 00 00
```

This actually means:

```
mov eax, 4
```

But as humans, we cannot understand the bytes directly, so we disassemble them using tools like :

- IDA Pro
- Ghidra
- Objdump
- Radare2
- x64Dbg, WinDbg

There are 2 types of disassebly agothirms popular :

- Linear disassembly
- Flow-Oriented Disassembly

2 Understand Anti-Disassembly

Anti-disassembly refers to techniques used by malware authors or advanced programmers to trick disassemblers like IDA pro or Ghidra into producing incorrect or misleading assembly code during analysis. Disassemblers do not truly understand code - they interpret byte by byte using assumptions :

- They assume where code start
- They assume how instruction are aligned

- They try follow control flow

Malware author exploit this by :

- Insert invalid or misleading bytes
- Jumping into middle of instructions
- Embedding data inside code
- Self modify or obfuscated control flow

3 Anti-Disassembly Techniques

Disassemblers (like IDA Pro, Ghidra, objdump, etc.) don't truly "understand" code — they follow a set of rules and assumptions to convert bytes into readable assembly.

Malware exploits those assumptions to confuse the disassembler and make it produce incorrect or misleading output. In other words, the malware intentionally creates situations where the disassembler is likely to guess wrong.

The chapter focuses on simple but effective tricks that break the most common assumptions made by disassemblers. These are basic techniques that take advantage of how disassemblers interpret bytecode — especially in how they assume where instructions start and end, or which parts of a file are code vs. data.

More complex anti-disassembly methods use runtime information — things that only exist when the program is actually running (like register values, self-modifying code, memory layout, etc.) — and this makes them harder or even impossible for static disassemblers to analyze properly.

Some techniques create code that cannot be fully disassembled using normal static disassembly — such as using:

- Obfuscated control flows
- Overlapping instructions
- Encrypted code that decrypts at runtime
- Encrypted code that decrypts at runtime
- Polymorphic or metamorphic code

3.1 Jump Instructions with the Same Target

One of the most common anti-disassembly techniques found in real-world malware is the use of two consecutive conditional jump instructions that both point to the same destination.
For example, consider the following pair:

```
jz  loc_512
jnz loc_512
```

In this case, no matter the result of the condition flags, execution will always jump to `loc_512`. The combination of `jz` (jump if zero) and `jnz` (jump if not zero) effectively behaves like an unconditional `jmp`, since one of them will always be taken.

However, the disassembler typically analyzes one instruction at a time, and it does not interpret the logic behind the combined behavior. As a result, when it sees the second instruction (`jnz`), it assumes that the conditional branch might not be taken and proceeds to disassemble the "false branch" — the code that comes immediately after it.

In practice, this false branch is never executed, and is often filled with junk instructions or misleading data intended to confuse static analysis tools and reverse engineers.
following code on figure 1 shows IDA Pro's first interpretation of a piece of code protected with this technique

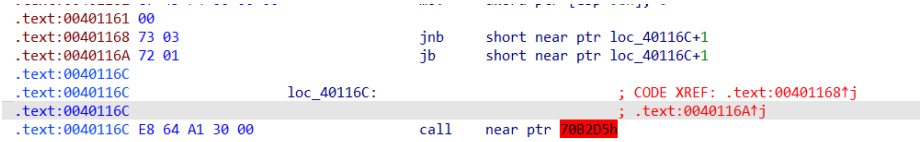

Figure 1: A piece of code protected with this technique

In this example, the instruction that appears immediately after the two conditional jumps may seem like a `call` instruction, beginning with the byte `0xE9`. However, this is not the case. Both conditional jumps actually point to a location one byte *after* the `0xE9` byte — meaning they land *inside* the instruction rather than at its beginning.

When this fragment is viewed in IDA Pro, the code cross-references shown at `loc_401161` will appear in red, rather than the standard blue. This visual indicator occurs because the disassembler detects that control flow is jumping to the middle of an instruction — something normally invalid in structured code. As a malware analyst, encountering this type of irregular cross-reference is often your first sign that anti-disassembly techniques are being used in the binary under analysis.

The following is disassembly of the same code on figure 2 , but this time fixed with the D key, to turn the byte immediately following the jnz instruction into data, and the C key to turn the bytes at `loc_401161` into instructions.

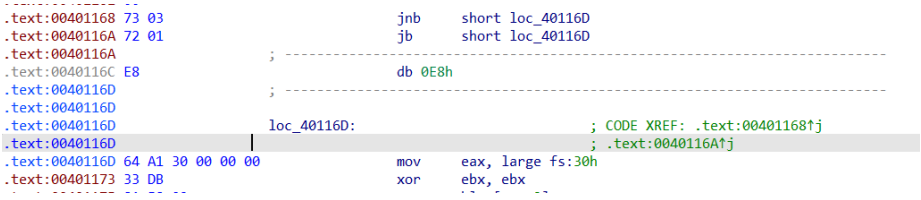

Figure 2: Fixed with the D key, C key with the byte after jno instrcution

The column on the left in these examples shows the bytes that constitute the instruction. Display of this field is optional, but it's important when learning

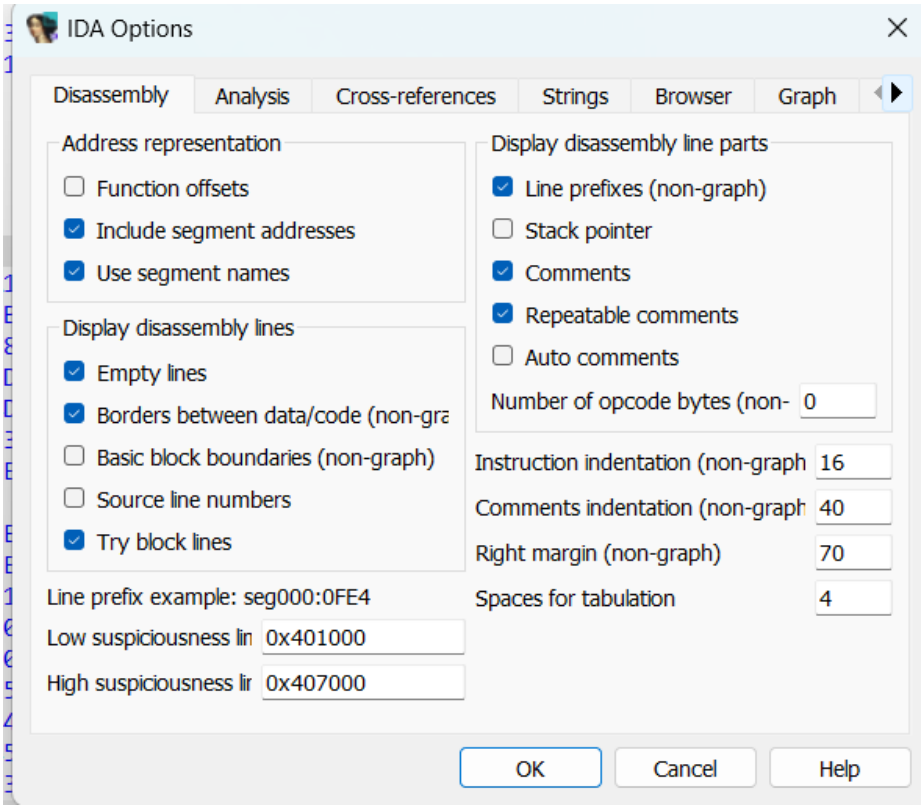

Figure 3: Opcode display in IDA pro

anti-disassembly. To display these bytes (or turn them off), select Options -¿ General. The Number of Opcode Bytes option allows you to enter a number for how many bytes you would like to be displayed.

3.2 A Jump Instruction with a Constant Condition

Another anti-disassembly technique commonly found in the wild is composed of a single conditional jump instruction placed where the condition will always be the same. The following code uses this technique:

```
33 C0                xor     eax, eax
74 01                jz      short near ptr loc_4011C4+1
        loc_4011C4:                          ; CODE XREF: 004011C2j
                                             ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94       jmp     near ptr 94A8D521h
```

Figure 4: Jump Instruction with a Constant Condition Technique

Notice that this code begins with the instruction xor eax, eax. This instruction will set the EAX register to zero and, as a byproduct, set the zero flag. The next instruction is a conditional jump that will jump if the zero flag is set.

In reality, this is not conditional at all, since we can guarantee that the zero flag will always be set at this point in the program. As discussed previously, the disassembler will process the false branch first, which will produce conflicting code with the true branch, and since it processed the false branch first, it trusts that branch more. As you've learned, you can use the D key on the keyboard while your cursor is on a line of code to turn the code into data, and pressing the C key will turn the data into code. Using these two keyboard shortcuts, a malware analyst could fix this fragment and have it show the real path of execution, as follows:

```
33 C0                   xor     eax, eax
74 01                   jz      short near ptr loc_4011C5
        ; ---------------------------------------------------------------------
E9                      db 0E9h
        ; ---------------------------------------------------------------------
        loc_4011C5:                             ; CODE XREF: 004011C2j
                                                ; DATA XREF: .rdata:004020ACo
58                      pop     eax
C3                      retn
```

Figure 5: After fix with D and C key

Figure 6 shows this example graphically.

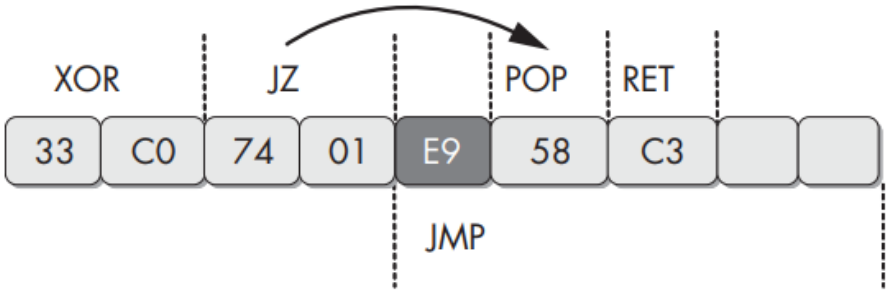

Figure 6: False conditional of xor followed by a jz instruction

3.2.1 Impossible Disassembly

In earlier sections, we explored examples where disassemblers such as IDA Pro initially failed to disassemble code correctly — but with manual adjustments (e.g., using the `D` and `C` keys), we were able to fix the output and obtain accurate results.

However, in some advanced scenarios, it becomes fundamentally impossible for traditional disassemblers to represent the actual instructions executed at runtime using a normal linear or control-flow-based listing. We refer to these scenarios as **impossible disassembly**. This term is not entirely precise — technically, disassembling is still possible, but doing so would require a completely different way of representing code, beyond what current tools offer.

Most anti-disassembly tricks rely on inserting a **rogue byte** — a carefully chosen data byte placed right after a conditional jump. This rogue byte is intended to confuse disassemblers. For example, the rogue byte might correspond to the opcode of a multi-byte instruction like `0xE8` (the `CALL` instruction). When the disassembler tries to decode from the rogue byte, it misinterprets the actual code that follows.

In many of these cases, the rogue byte is not executed and can safely be ignored when analyzing the real control flow.

But what if the rogue byte *is* executed at runtime?

This leads to a more complex situation: a single byte may belong to *multiple valid instructions*, depending on which execution path is taken. In other words, the same byte may be used differently across different code paths.

This is a form of **overlapping instruction encoding**, which modern processors can handle without issue — because the CPU decodes instructions at runtime from wherever the instruction pointer points. But static disassemblers, which map each byte to a single instruction, cannot represent this overlap.

No commercial disassembler (including IDA Pro, Ghidra, or Radare2) currently supports showing the same byte as being part of two or more distinct instructions. As a result, when faced with such overlapping logic, they are forced to pick just one disassembly path — which hides the alternative one.

This discrepancy is at the heart of the term *impossible disassembly*: it's not that the processor can't execute the code — it's that disassemblers can't accurately show all possible execution interpretations within a single static listing.

Figure 7 shows an example. The first instruction in this 4-byte sequence is a 2-byte jmp instruction. The target of the jump is the second byte of itself. This doesn't cause an error, because the byte FF is the first byte of the next 2-byte instruction, inc eax.

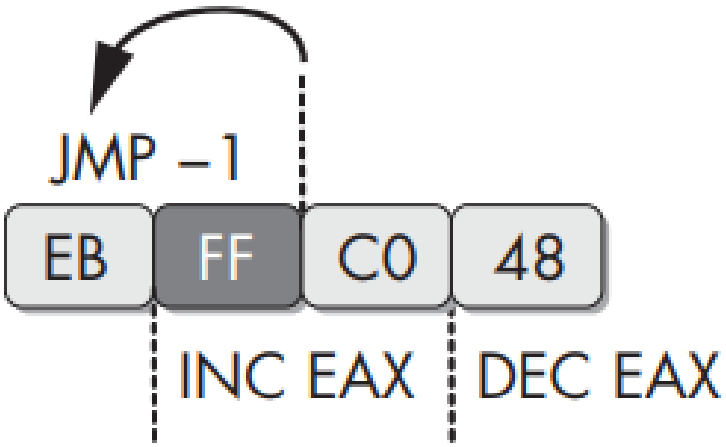

Figure 7: Inward-pointing jmp instruction

For a glimpse of the complexity that can be achieved with these sorts of instruction sequences, let's examine a more advanced specimen. Figure 8 shows

an example that operates on the same principle as the prior one, where some bytes are part of multiple instructions.

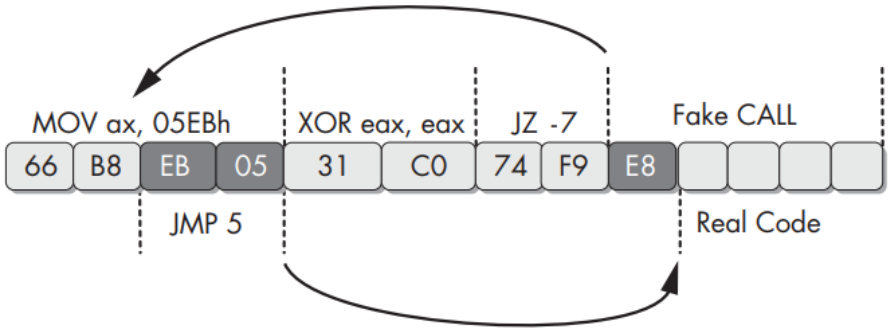

Figure 8: Multilevel inward-jumping sequence

The first instruction in this sequence is a 4-byte mov instruction. The last 2 bytes have been highlighted because they are both part of this instruction and are also their own instruction to be executed later. The first instruction populates the AX register with data. The second instruction, an xor, will zero out this register and set the zero flag. The third instruction is a conditional jump that will jump if the zero flag is set, but it is actually unconditional, since the previous instruction will always set the zero flag. The disassembler will decide to disassemble the instruction immediately following the jz instruction, which will begin with the byte 0xE8, the opcode for a 5-byte call instruction. The instruction beginning with the byte E8 will never execute in reality. The disassembler in this scenario can't disassemble the target of the jz instruction because these bytes are already being accurately represented as part of the mov instruction. The code that the jz points to will always be executed, since the zero flag will always be set at this point. The jz instruction points to the middle of the first 4-byte mov instruction. The last 2 bytes of this instruction are the operand that will be moved into the register. When disassembled or executed on their own, they form a jmp instruction that will jump forward 5 bytes from the end of the instruction. When first viewed in IDA Pro, this sequence will look like the following:

```
66 B8 EB 05             mov     ax, 5EBh
31 C0                   xor     eax, eax
74 F9                   jz      short near ptr sub_4011C0+1
                loc_4011C8:
E8 58 C3 90 90          call    near ptr 98A8D525h
```

Figure 9: Multilevel inward-jumping sequence ida pro

Since there is no way to clean up the code so that all executing instructions are represented, we must choose the instructions to leave in. The net side effect of this anti-disassembly sequence is that the EAX register is set to zero. If you manipulate the code with the D and C keys in IDA Pro so that the only

instructions visible are the xor instruction and the hidden instructions, your result should look like the following.

```
66                      byte_4011C0     db 66h
B8                                      db 0B8h
EB                                      db 0EBh
05                                      db    5
                ; ---------------------------------------------------------
31 C0                                   xor     eax, eax
                ; ---------------------------------------------------------
74                                      db 74h
F9                                      db 0F9h
E8                                      db 0E8h
                ; ---------------------------------------------------------
58                                      pop     eax
C3                                      retn
```

Figure 10: Enter Caption

This is a somewhat acceptable solution because it shows only the instructions that are relevant to understanding the program. However, this solution may interfere with analysis processes such as graphing, since it's difficult to tell exactly how the xor instruction or the pop and retn sequences are executed. A more complete solution would be to use the PatchByte function from the IDC scripting language to modify remaining bytes so that they appear as NOP instructions.

This example has two areas of undisassembled bytes that we need to convert into NOP instructions: 4 bytes starting at memory address 0x004011C0 and 3 bytes starting at memory address 0x004011C6. The following IDAPython script will convert these bytes into NOP bytes (0x90):

```
def NopBytes(start, length):
   for i in range(0, length):
     PatchByte(start + i, 0x90)
   MakeCode(start)

NopBytes(0x004011C0, 4)
NopBytes(0x004011C6, 3)
```

Figure 11: IDA python script nop-out

This code takes the long approach by making a utility function called NopBytes to NOP-out a range of bytes. It then uses that utility function against the two ranges that we need to fix. When this script is executed, the resulting disassembly is clean, legible, and logically equivalent to the original:

3.3 NOP-ing Out Instructions with IDA Pro

With a little IDA Python knowledge, we can develop a script that allows malware analysts to easily NOP-out instructions as they see fit. The following

```
90                      nop
90                      nop
90                      nop
90                      nop
31 C0                   xor     eax, eax
90                      nop
90                      nop
90                      nop
58                      pop     eax
C3                      retn
```

Figure 12: Enter Caption

```
import idaapi

idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')

AddHotkey("Alt-N", "n_key")

def nopIt():

      start = ScreenEA()
      end = NextHead(start)
      for ea in range(start, end):
            PatchByte(ea, 0x90)
      Jump(end)
      Refresh()
```

Figure 13: IDA Python script to create hot-key

script establishes the hotkey ALT-N. Once this script is executed, whenever the user presses ALT-N, IDA Pro will NOP-out the instruction that is currently at the cursor location. It will also conveniently advance the cursor to the next instruction to facilitate easy NOP-outs of large blocks of code.

3.4 Reference

All of these I reference from **Practical_Malware_Analysis.pdf**.