An Effective Approach To Software Obfuscation

Yu-Jye Tung

@yujyet

What Is Software Obfuscation?

A software protection mechanism through program transformation (source-level, compilation-level, or binary-level) that…

- makes the corresponding executable binary more difficult to analyze
- without changing program's core functionalities *(intended observable behaviors)*.

Notable aside: compilation-level transformation is the most flexible of the 3.

Collberg. A Taxonomy of Obfuscating Transformations. 1997.
Collberg. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. 1998.

What Is Software Obfuscation?

A software protection mechanism through program transformation (source-level, compilation-level, or binary-level) that…

- makes the corresponding executable binary more difficult to analyze
- without changing program's core functionalities *(intended observable behaviors)*.

Notable aside: compilation-level transformation is the most flexible of the 3.

In respect to the transformation's **potency**, **resilience**, and **stealth**.

Analysis is performed by the reverse engineering process.

Collberg. A Taxonomy of Obfuscating Transformations. 1997.
Collberg. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. 1998.

More Definitions... ☹

Potency: strength of transformation against manual analysis

Resilience: strength of transformation against automated analysis

Stealth: strength of transformation against initial detection

Manual Analysis

More Definitions... 😢

Potency: strength of transformation against manual analysis

Resilience: strength of transformation against automated analysis

Stealth: strength of transformation against initial detection

Angr

BINSEC

B2R2

BinaryAnalysisPlatform / **bap**

Automated Analysis

More Definitions... ☹

BINSEC

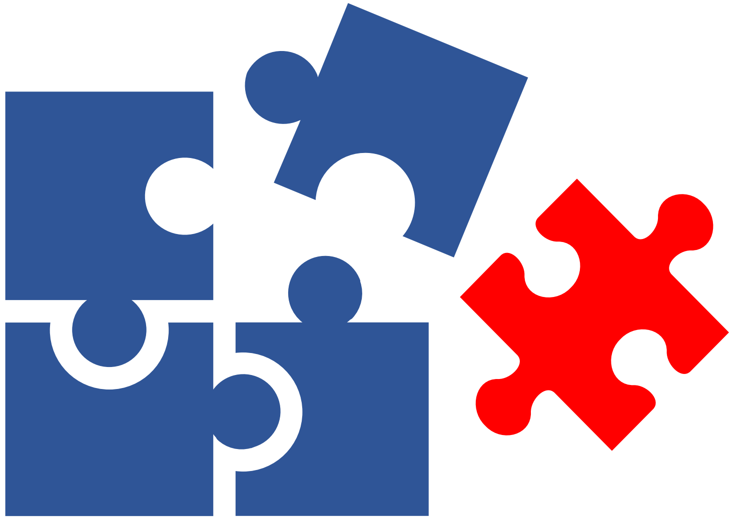
```
binsec@ddcd8df6e0b8:~/connect$ binsec -bw -bw-opaque -bw-k 10  anagram_ollvm
[bw:info] Checking all predicates in anagram_ollvm for opacity
[disasm:info] Using section until 8048e01
[disasm:result] Linear disassembly from 080483b0 to 08048e01
[bw:result] Predicate jbe 0x8048419 @ 0x080483fd is opaque
            (then: clear; else: opaque)
[bw:result] Predicate jz 0x8048419 @ 0x08048406 is opaque
            (then: opaque; else: opaque)
[bw:result] Predicate jz 0x8048453 @ 0x08048436 is opaque
            (then: clear; else: opaque)
[bw:result] Predicate jz 0x8048453 @ 0x0804843f is opaque
            (then: opaque; else: opaque)
...
```

More Definitions... 😢

Potency: strength of transformation against manual analysis

Resilience: strength of transformation against automated analysis

Stealth: strength of transformation against initial detection



Initial Detection

Software Obfuscation != Cryptography

The protection offered by software obfuscation does not have the same mathematical guarantee as cryptography.

In other words, the strength of transformation's potency, resilience, and stealth can be reduced.

What Is Software Obfuscation?

A software protection mechanism through program transformation (source-level, compilation-level, or binary-level) that…

- makes the corresponding executable binary more difficult to analyze
- without changing program’s core functionalities *(intended observable behaviors)*.

Notable aside: compilation-level transformation is the most flexible of the 3.

In respect to the transformation's **potency**, **resilience**, and **stealth.**

Analysis is performed by the reverse engineering process.

Collberg. A Taxonomy of Obfuscating Transformations. 1997.
Collberg. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. 1998.

What Is Software Obfuscation?

A software protection mechanism through program transformation (source-level, compilation-level, or binary-level) that…

- makes the corresponding executable binary more ~~difficult~~ to analyze
- without changing program's core functionalities *(intended observable behaviors)*.

Notable aside: compilation-level transformation is the most flexible of the 3.

time-consuming

Analysis is performed by the reverse engineering process

Collberg. A Taxonomy of Obfuscating Transformations. 1997.
Collberg. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. 1998.

The "Time-Consuming" Aspect

A transformation's potency, resilience, and stealth combine to ultimately answer this question: **how much more time-consuming did the transformation makes it for reverse engineering?**

End goal: **make analysts give up.**

More time
consuming
==
More frustrating for
the analysts

The "Time-Consuming" Aspect

A transformation's potency, resilience, and stealth combine to ultimately answer this question: **how much more time-consuming did the transformation makes it for reverse engineering?**

End goal: **make analysts give up.**

More time
consuming
==
More frustrating for
the analysts

Deobfuscation Process

1. Identifying the obfuscation technique (stealth)

2. Performing the relevant deobfuscation steps (potency, resilience)

Effects Of Modern Obfuscation

1. Identifying the obfuscation technique (stealth)

2. Performing the relevant deobfuscation steps (potency, resilience)

Notable Examples:

- Control-flow graph flattening
- Virtualization obfuscation

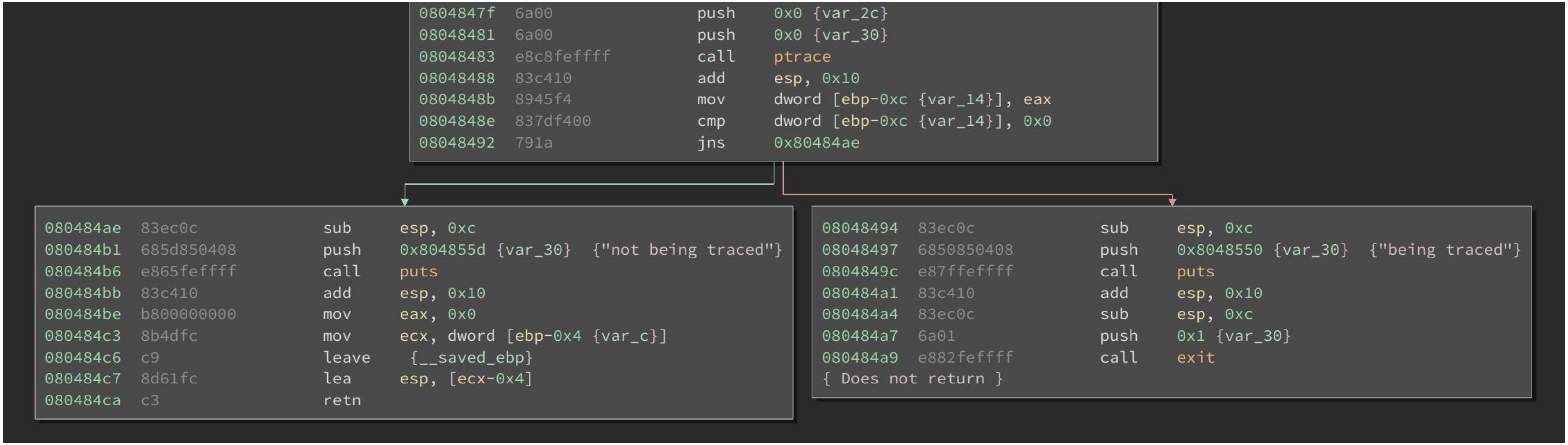
Stealth is ignored!

Modern Obfuscation = Noisy!

1. Easy to identify
   (low stealth)

Control-Flow Graph (CFG) Flattening: Theory

Control-Flow Graph (CFG): representation of a function's disassembly (instructions) where program flow is also represented.



```
0804847f  6a00          push    0x0 {var_2c}
08048481  6a00          push    0x0 {var_30}
08048483  e8c8feffff    call    ptrace
08048488  83c410        add     esp, 0x10
0804848b  8945f4        mov     dword [ebp-0xc {var_14}], eax
0804848e  837df400      cmp     dword [ebp-0xc {var_14}], 0x0
08048492  791a          jns     0x80484ae
```

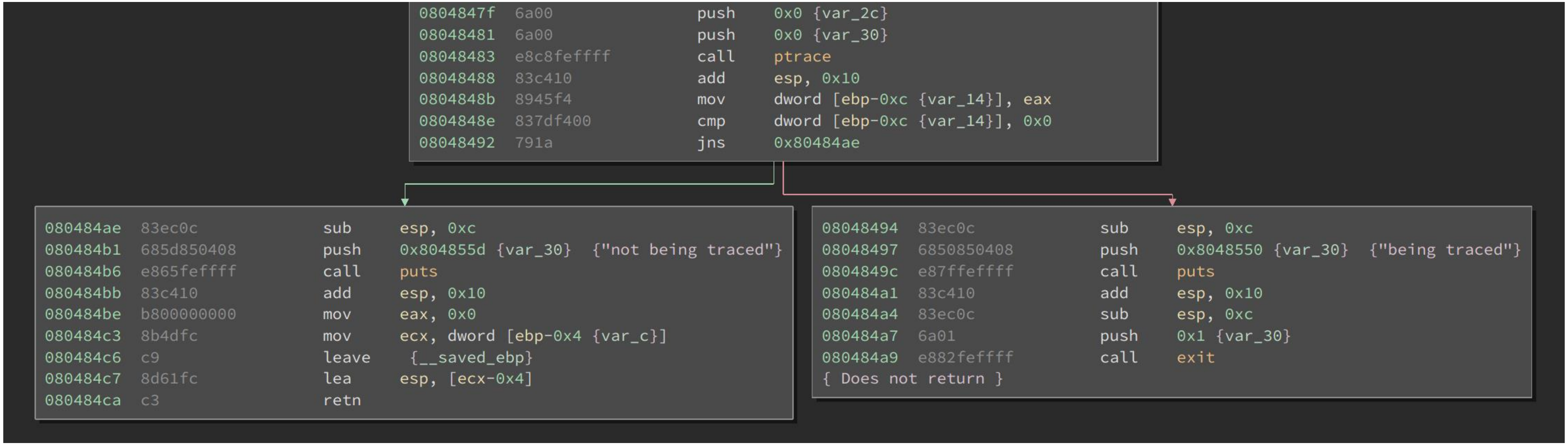
```
080484ae  83ec0c        sub     esp, 0xc
080484b1  685d850408    push    0x804855d {var_30}  {"not being traced"}
080484b6  e865feffff    call    puts
080484bb  83c410        add     esp, 0x10
080484be  b800000000    mov     eax, 0x0
080484c3  8b4dfc        mov     ecx, dword [ebp-0x4 {var_c}]
080484c6  c9            leave    {__saved_ebp}
080484c7  8d61fc        lea     esp, [ecx-0x4]
080484ca  c3            retn
```

```
08048494  83ec0c        sub     esp, 0xc
08048497  6850850408    push    0x8048550 {var_30}  {"being traced"}
0804849c  e87ffeffff    call    puts
080484a1  83c410        add     esp, 0x10
080484a4  83ec0c        sub     esp, 0xc
080484a7  6a01          push    0x1 {var_30}
080484a9  e882feffff    call    exit
{ Does not return }
```

Control-Flow Graph (CFG) Flattening: Theory

Control-Flow Graph (CFG): representation of a function's disassembly (instructions) where program flow is also represented.

Why is CFG representation helpful?



```
0804847f  6a00        push  0x0 {var_2c}
08048481  6a00        push  0x0 {var_30}
08048483  e8c8feffff  call  ptrace
08048488  83c410      add   esp, 0x10
0804848b  8945f4      mov   dword [ebp-0xc {var_14}], eax
0804848e  837df400    cmp   dword [ebp-0xc {var_14}], 0x0
08048492  791a        jns   0x80484ae
```

```
080484ae  83ec0c      sub    esp, 0xc
080484b1  685d850408  push   0x804855d {var_30}  {"not being traced"}
080484b6  e865feffff  call   puts
080484bb  83c410      add    esp, 0x10
080484be  b800000000  mov    eax, 0x0
080484c3  8b4dfc      mov    ecx, dword [ebp-0x4 {var_c}]
080484c6  c9          leave   {__saved_ebp}
080484c7  8d61fc      lea    esp, [ecx-0x4]
080484ca  c3          retn
```

```
08048494  83ec0c      sub   esp, 0xc
08048497  6850850408  push  0x8048550 {var_30}  {"being traced"}
0804849c  e87ffeffff  call  puts
080484a1  83c410      add   esp, 0x10
080484a4  83ec0c      sub   esp, 0xc
080484a7  6a01        push  0x1 {var_30}
080484a9  e882feffff  call  exit
{ Does not return }
```

Control-Flow Graph (CFG) Flattening: Theory

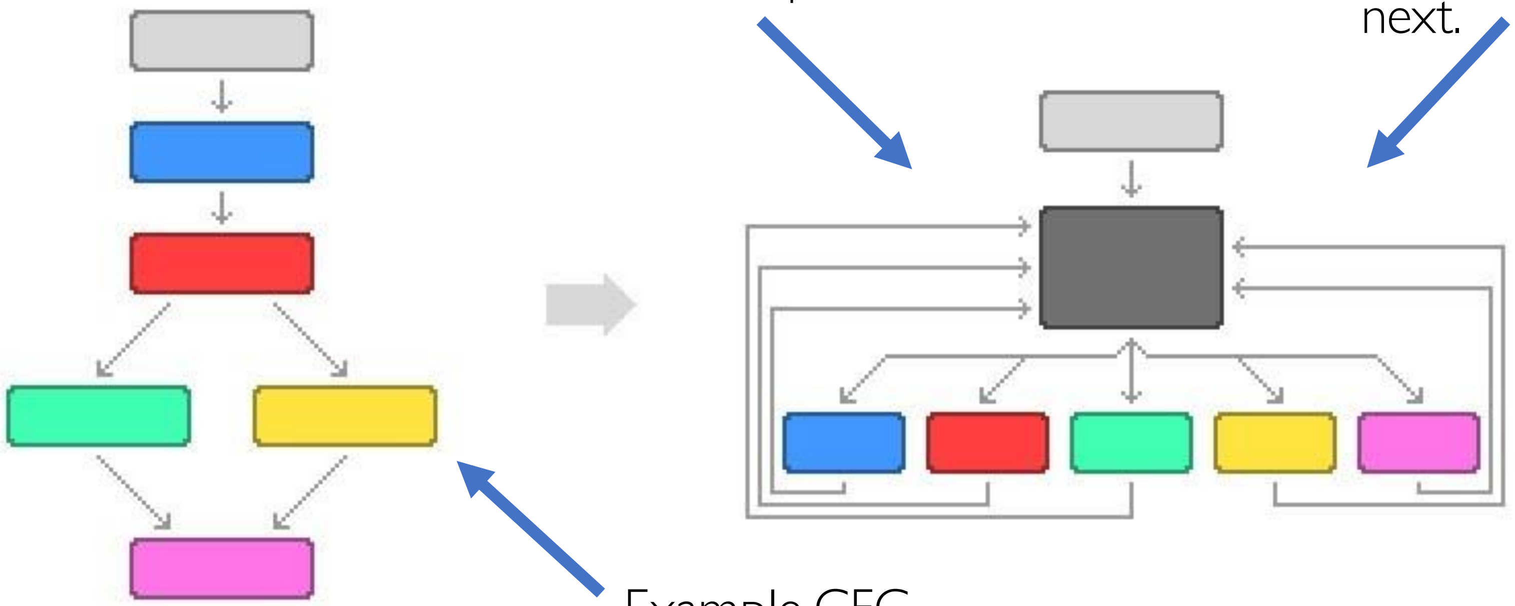
Control-Flow Graph (CFG): representation of a function's disassembly (instructions) where program flow is also represented.

Why is CFG representation helpful?

Control-flow graph increases disassembly's **glance value.**

For example, one can recognize high-level programming constructs (e.g., if/while/for/switch statements) by just a quick glance of the disassembly.

Control-Flow Graph (CFG) Flattening: Theory



Turn every CFG into a CFG with that shape.

The dispatcher (in black) decides which original basic block to execute next.

Example CFG.

Control-flow graph flattening removes the increased glanced value the CFG representation provides, such as:

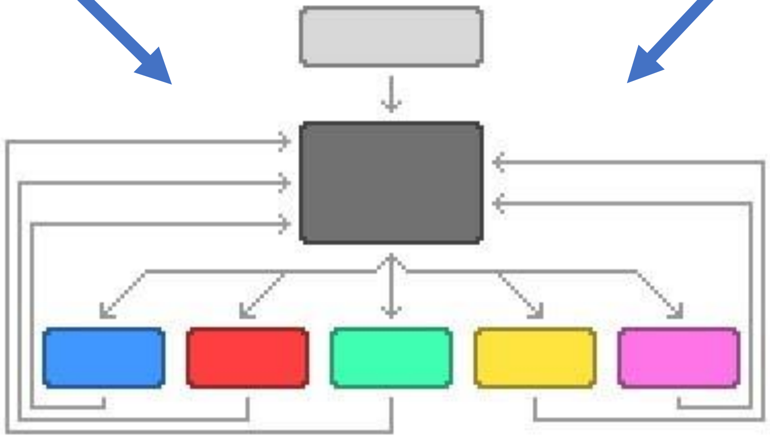
- shapes indicating high-level programming constructs
- spatial locality of basic blocks assists cognitive reasoning on the semantics of a disassembly sequence

Jscrambler. Jscrambler 101 – Control Flow Flattening. 2017. https://blog.jscrambler.com/jscrambler-101-control-flow-flattening/

Control-Flow Graph (CFG) Flattening: Theory

Turn every CFG into a CFG with that shape.

The dispatcher (in black) decides which original basic block to execute next.



Low Stealth!

Jscrambler. Jscrambler 101 – Control Flow Flattening. 2017. https://blog.jscrambler.com/jscrambler-101-control-flow-flattening/

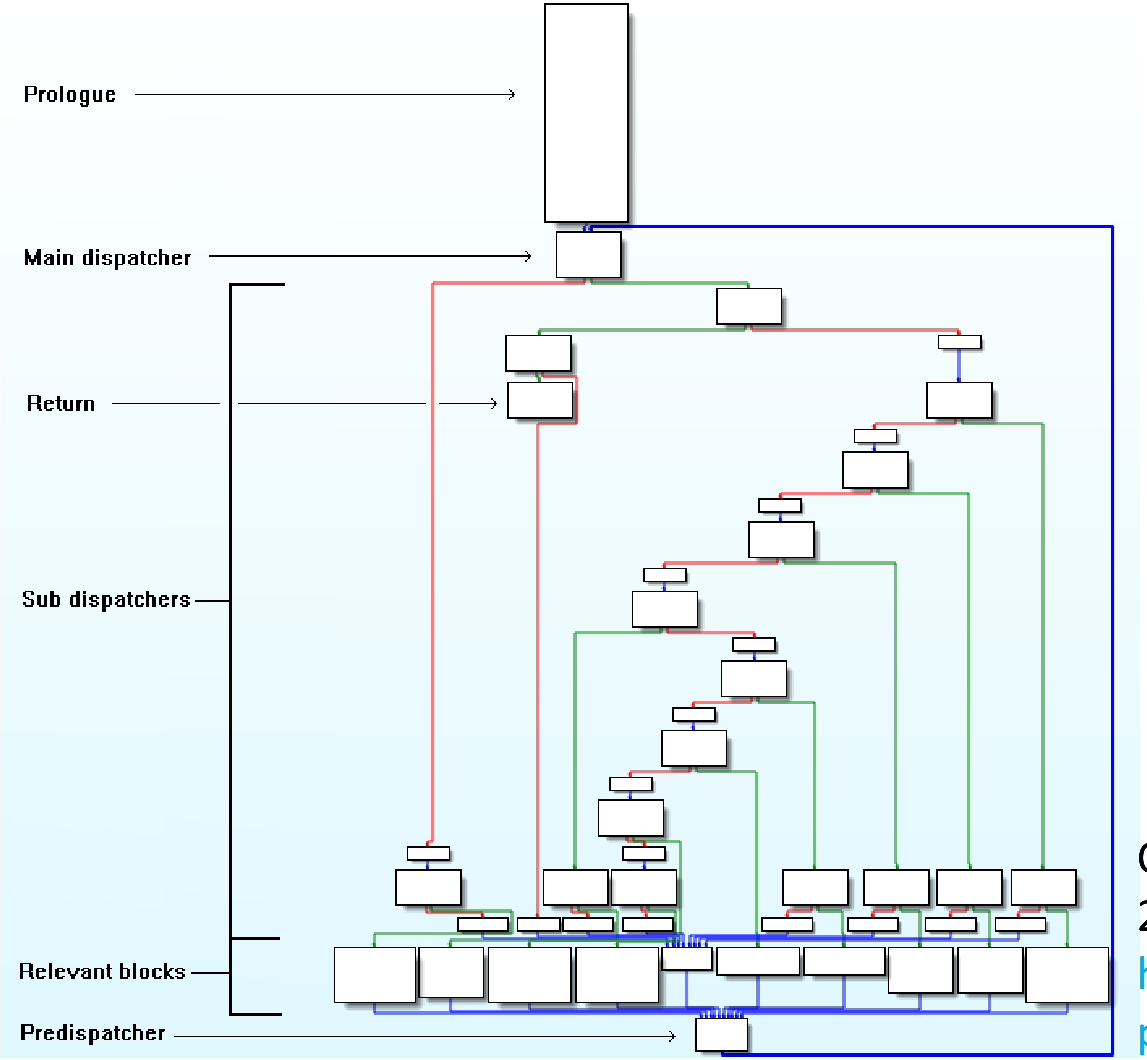
Modern Obfuscation = Noisy!

1. Easy to identify
   (low stealth)

But it doesn't matter if deobfuscation takes a long time, right?

Real-world implementations leave behind distinctive footprints to allow for ad-hoc approaches to deobfuscation.

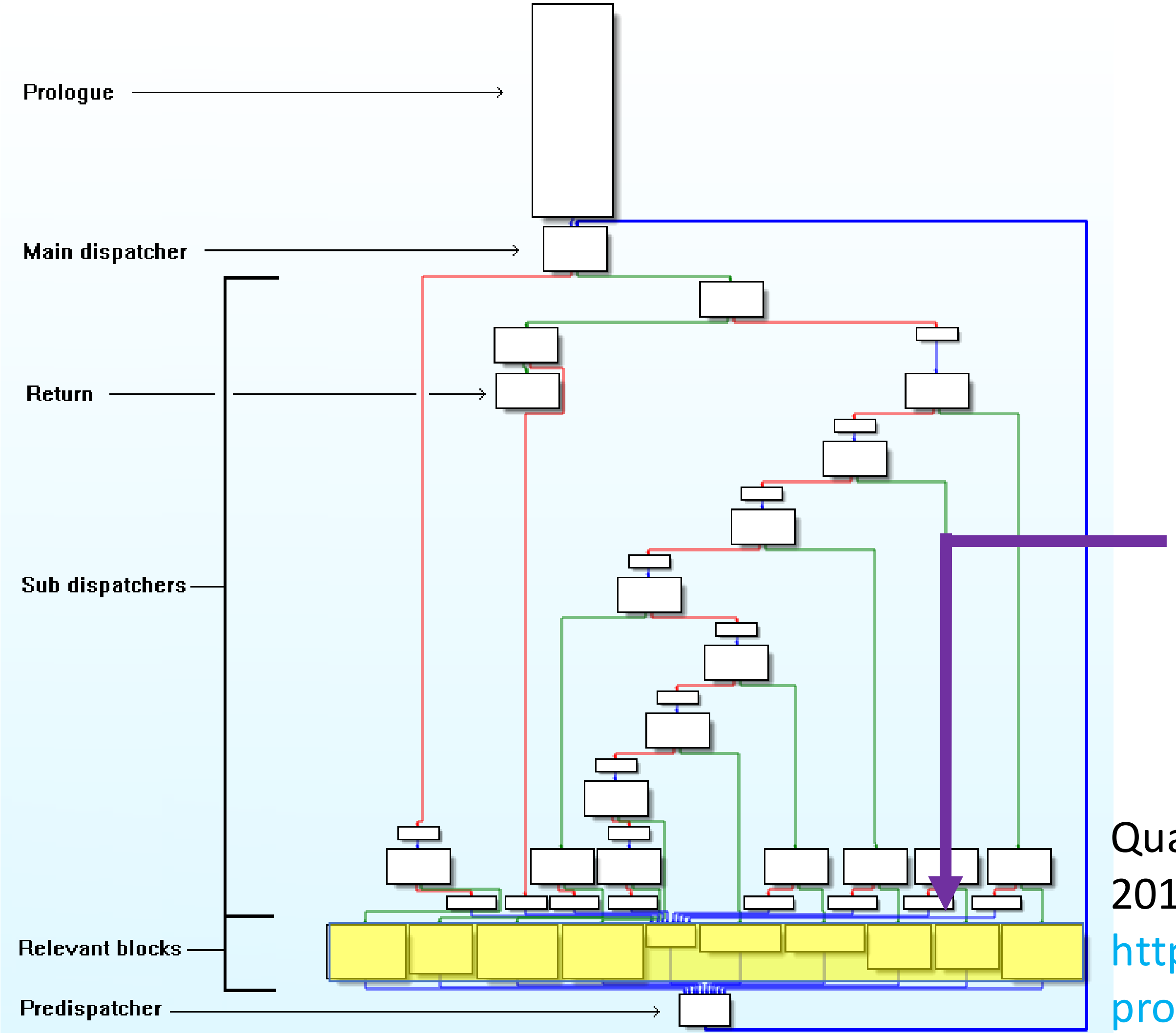
Control-Flow Graph Flattening: OLLVM



An original basic block will always end with setting a local variable to a constant corresponding to the next original basic block the dispatcher needs to execute.

Quarkslab. Deobfuscation: Recovering An OLLVM-Protected Program. 2014. https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html

Control-Flow Graph Flattening: OLLVM



An original basic block will always end with setting a local variable to a constant corresponding to the next original basic block the dispatcher needs to execute.

Figuring out the constant corresponding to these basic blocks allow us to reconstruct original CFG

Quarkslab. Deobfuscation: Recovering An OLLVM-Protected Program. 2014. https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html

Solution

Instead of focusing on making the obfuscation technique harder to break (potency, resilience), **also** focusing on making it harder to identify (stealth).

Respect each property that makes up the "time-consuming" aspect.

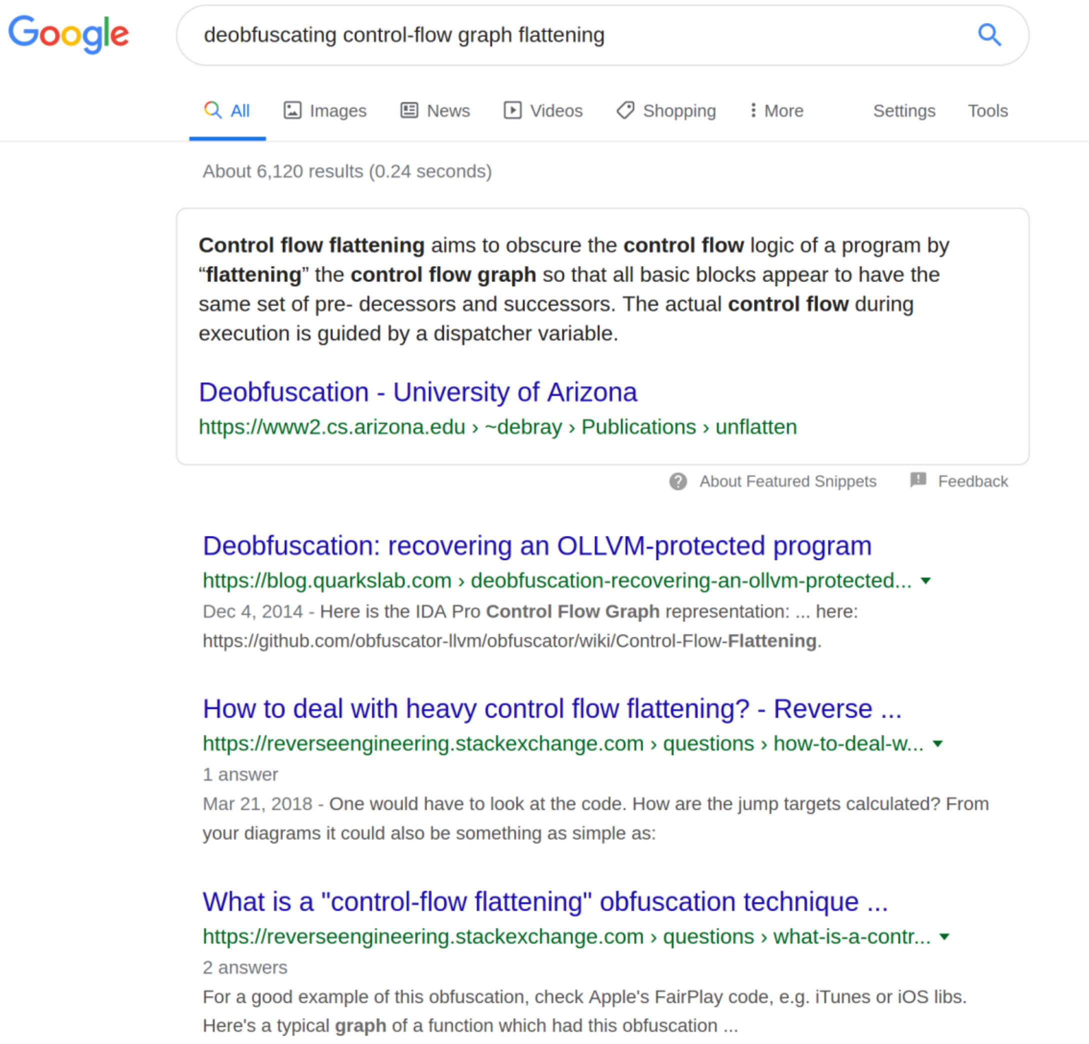
What's More Frustrating?

- Understanding what the problem is but not how to solve it?

What's More Frustrating?

- Understanding what the problem is but not how to solve it?

Google? If there're solutions online that solve similar problems, learn the general approach to tackle that problem



deobfuscating control-flow graph flattening

About 6,120 results (0.24 seconds)

Control flow flattening aims to obscure the **control flow** logic of a program by "**flattening**" the **control flow graph** so that all basic blocks appear to have the same set of pre- decessors and successors. The actual **control flow** during execution is guided by a dispatcher variable.

Deobfuscation - University of Arizona
https://www2.cs.arizona.edu › ~debray › Publications › unflatten

Deobfuscation: recovering an OLLVM-protected program
https://blog.quarkslab.com › deobfuscation-recovering-an-ollvm-protected... ▾
Dec 4, 2014 - Here is the IDA Pro **Control Flow Graph** representation: ... here: https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-**Flattening**.

How to deal with heavy control flow flattening? - Reverse ...
https://reverseengineering.stackexchange.com › questions › how-to-deal-w... ▾
1 answer
Mar 21, 2018 - One would have to look at the code. How are the jump targets calculated? From your diagrams it could also be something as simple as:

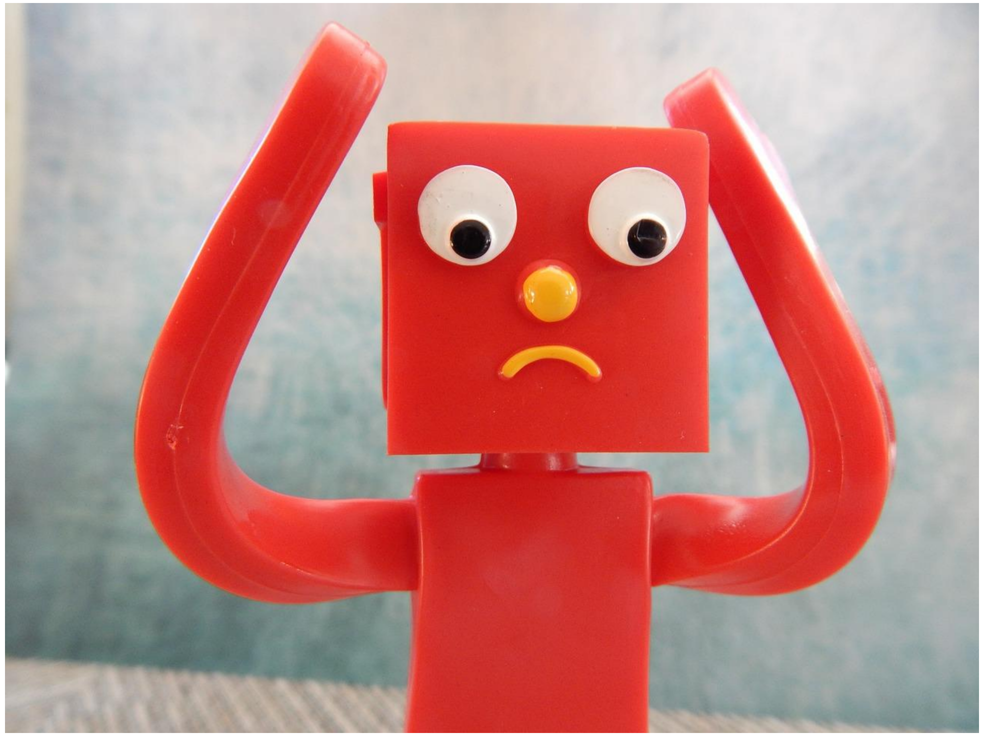
What is a "control-flow flattening" obfuscation technique ...
https://reverseengineering.stackexchange.com › questions › what-is-a-contr... ▾
2 answers
For a good example of this obfuscation, check Apple's FairPlay code, e.g. iTunes or iOS libs. Here's a typical **graph** of a function which had this obfuscation ...

What’s More Frustrating?

- Not understanding or even aware what the problem is?

What's More Frustrating?

- Not understanding or even aware what the problem is?



Inconspicuous Obfuscation

If analysts aren't aware of what was obfuscated, it makes them...

1. Make the wrong assumptions about what the code is doing
2. Falling deeper into the rabbit hole (aka **reversing hell**)

Inconspicuous Obfuscation

If analysts aren't aware of what was obfuscated, it makes them...

1. Make the wrong assumptions about what the code is doing
2. Falling deeper into the rabbit hole (aka **reversing hell**)

Only stealth (not potency or resilience) can achieve this!

Inconspicuous Obfuscation: Example

The Return of Disassembly Desynchronization

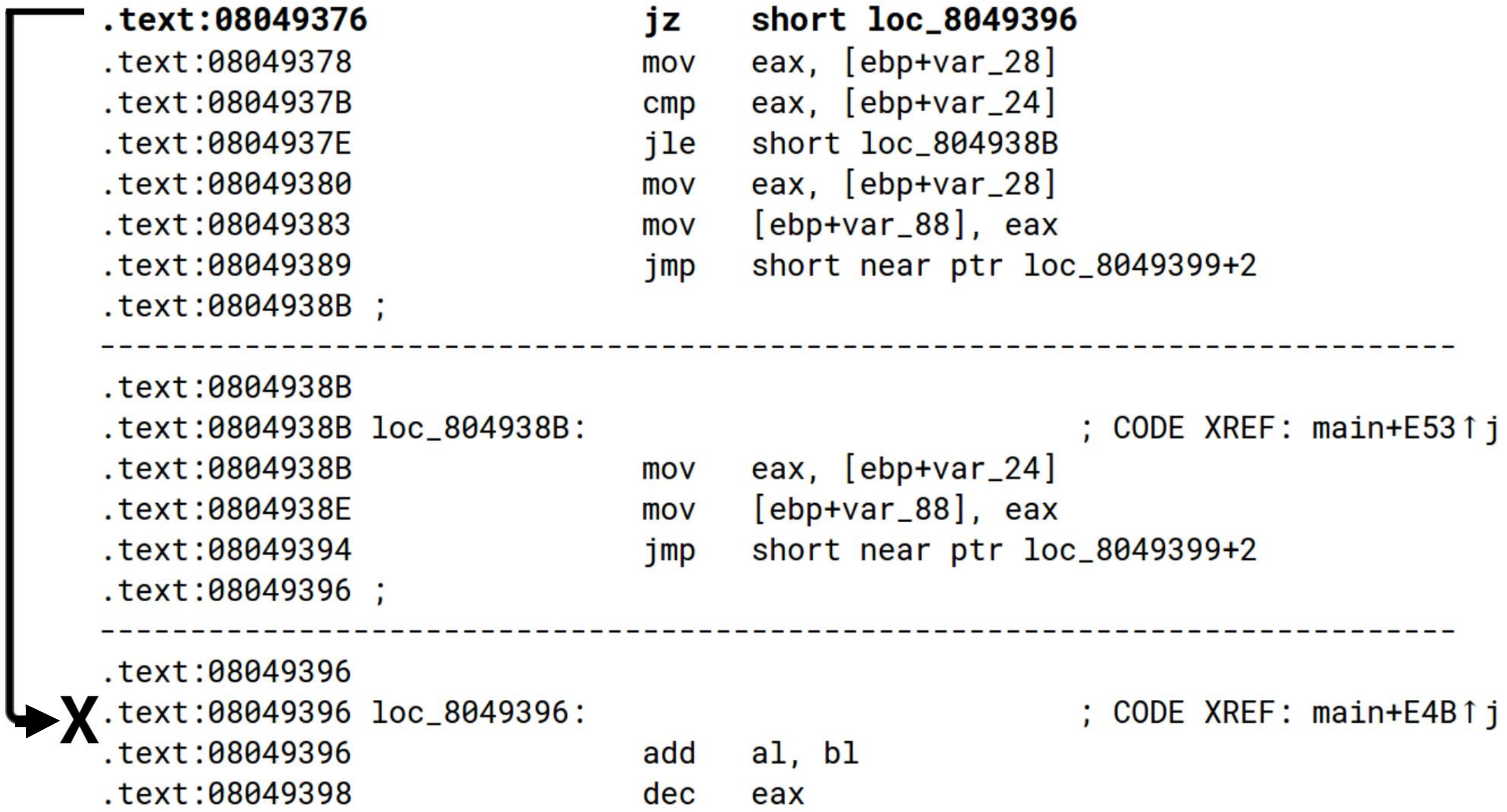
github.com/yellowbyte/analysis-of-anti-analysis

Summary: We take advantage of the assumption IDA Pro makes to detect opaque predicates to create even stealthier opaque predicates.

Stealth is important too!

Disassembly Desynchronization

An umbrella term for software obfuscation techniques whose main goal is to degrade the accuracy of the retrieved disassembly.



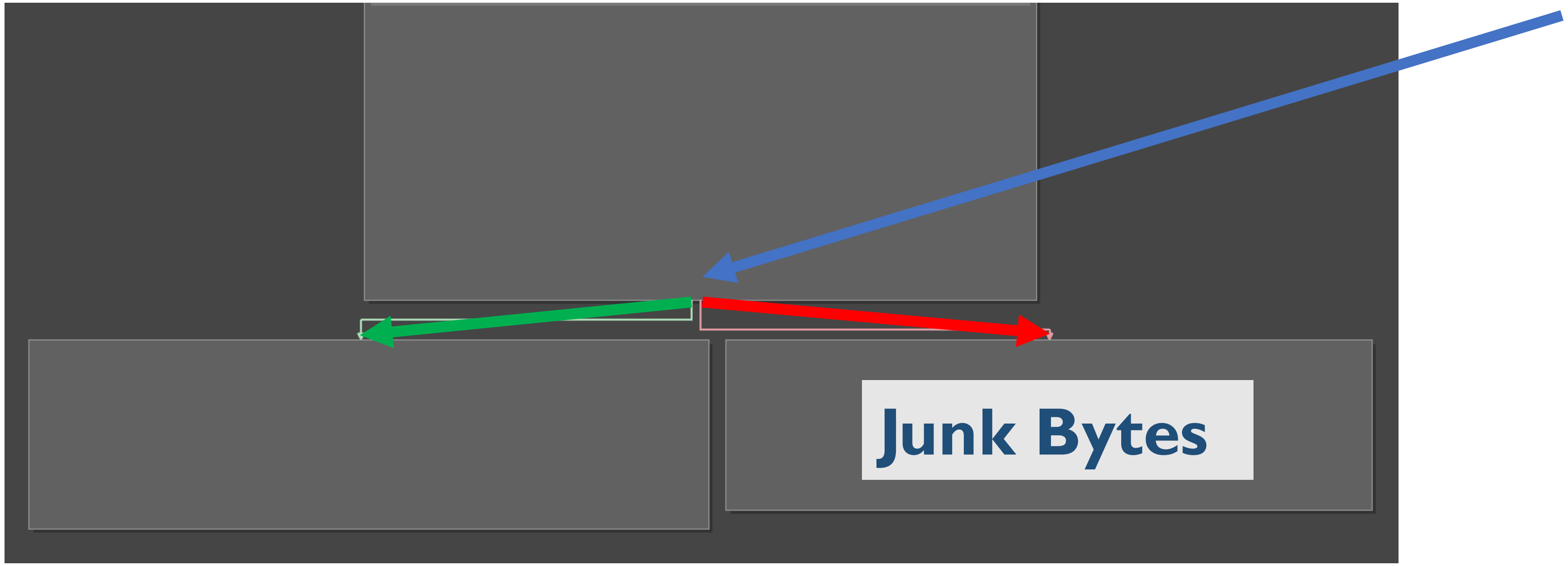
```
.text:08049376                jz     short loc_8049396
.text:08049378                mov    eax, [ebp+var_28]
.text:0804937B                cmp    eax, [ebp+var_24]
.text:0804937E                jle    short loc_804938B
.text:08049380                mov    eax, [ebp+var_28]
.text:08049383                mov    [ebp+var_88], eax
.text:08049389                jmp    short near ptr loc_8049399+2
.text:0804938B ;
---------------------------------------------------------------------------
.text:0804938B
.text:0804938B loc_804938B:                            ; CODE XREF: main+E53↑j
.text:0804938B                mov    eax, [ebp+var_24]
.text:0804938E                mov    [ebp+var_88], eax
.text:08049394                jmp    short near ptr loc_8049399+2
.text:08049396 ;
---------------------------------------------------------------------------
.text:08049396
.text:08049396 loc_8049396:                            ; CODE XREF: main+E4B↑j
.text:08049396                add    al, bl
.text:08049398                dec    eax
```

Inaccurate Disassembly

Opaque Predicates

Definition: Conditional branches that are always true or false. One of their branches is unreachable so junk bytes (data bytes) can be inserted.

Predicate always evaluate to True



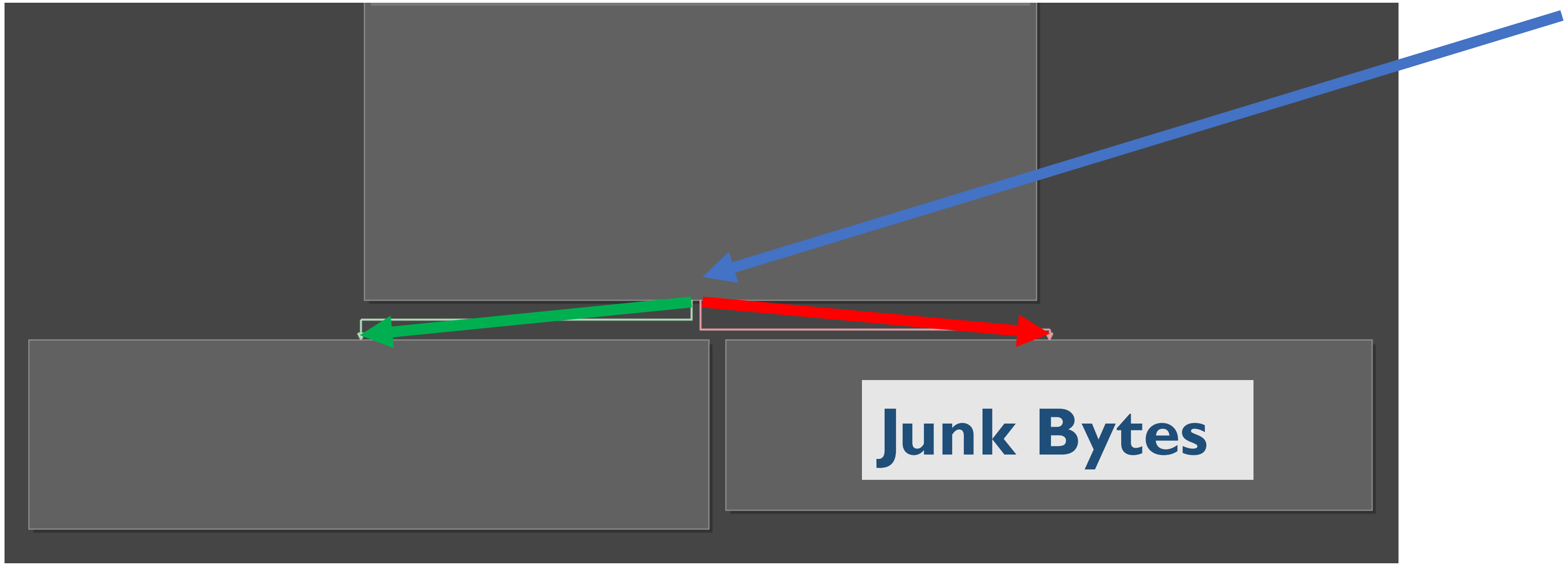
"Opaque Predicates"

Opaque Predicates

BINSEC

Vector35 / **OpaquePredicatePatcher**

Can both branches be executed?

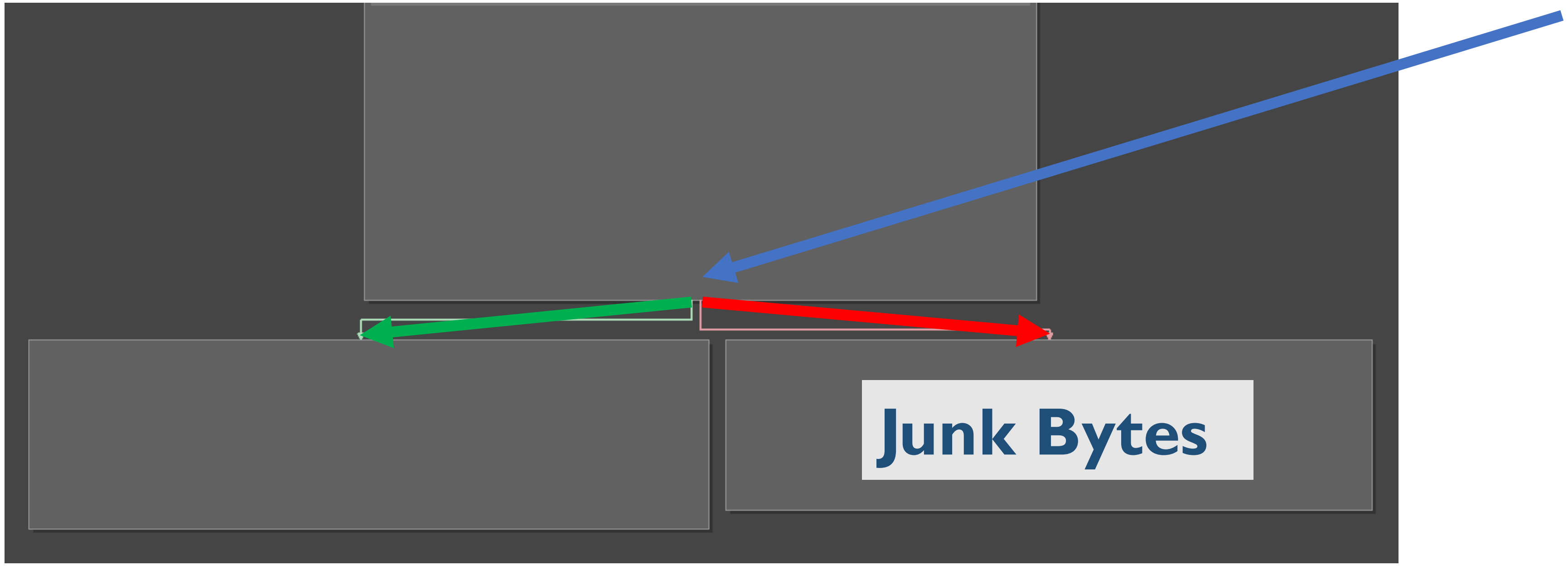


"Opaque Predicates"

Opaque Predicates

Since identifying opaque predicates is non-trivial, IDA Pro takes a heuristic-based approach to identify them.

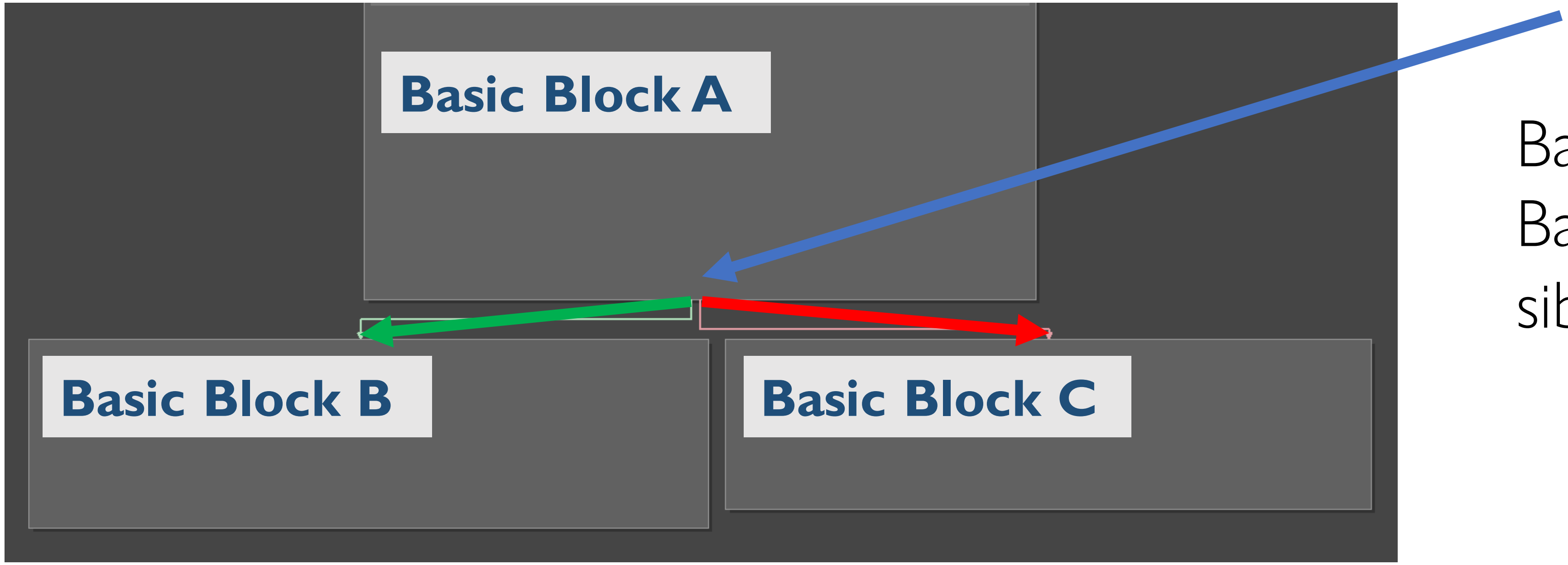
Predicate always evaluate to True



"Opaque Predicates"

Opaque Predicates

Initial Detection: If IDA Pro detects overlapped instructions in sibling basic blocks, it will assume the conditional branch is an opaque predicate.



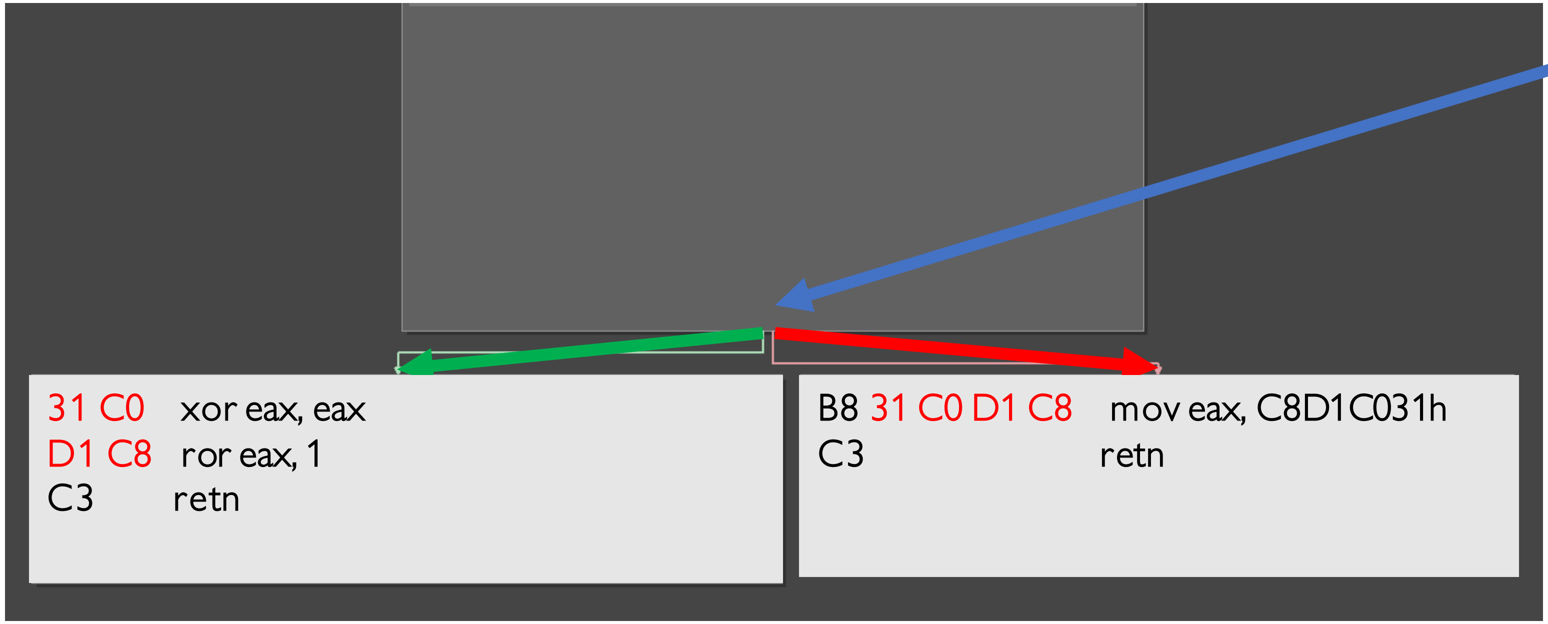
Predicate always evaluate to True

Basic Block B and Basic Block C are siblings.

Opaque Predicates

Initial Detection: If IDA Pro detects **overlapped instructions** in sibling basic blocks, it will assume the conditional branch is an opaque predicate.

Predicate always evaluate to True



```
31 C0    xor eax, eax
D1 C8    ror eax, 1
C3       retn
```

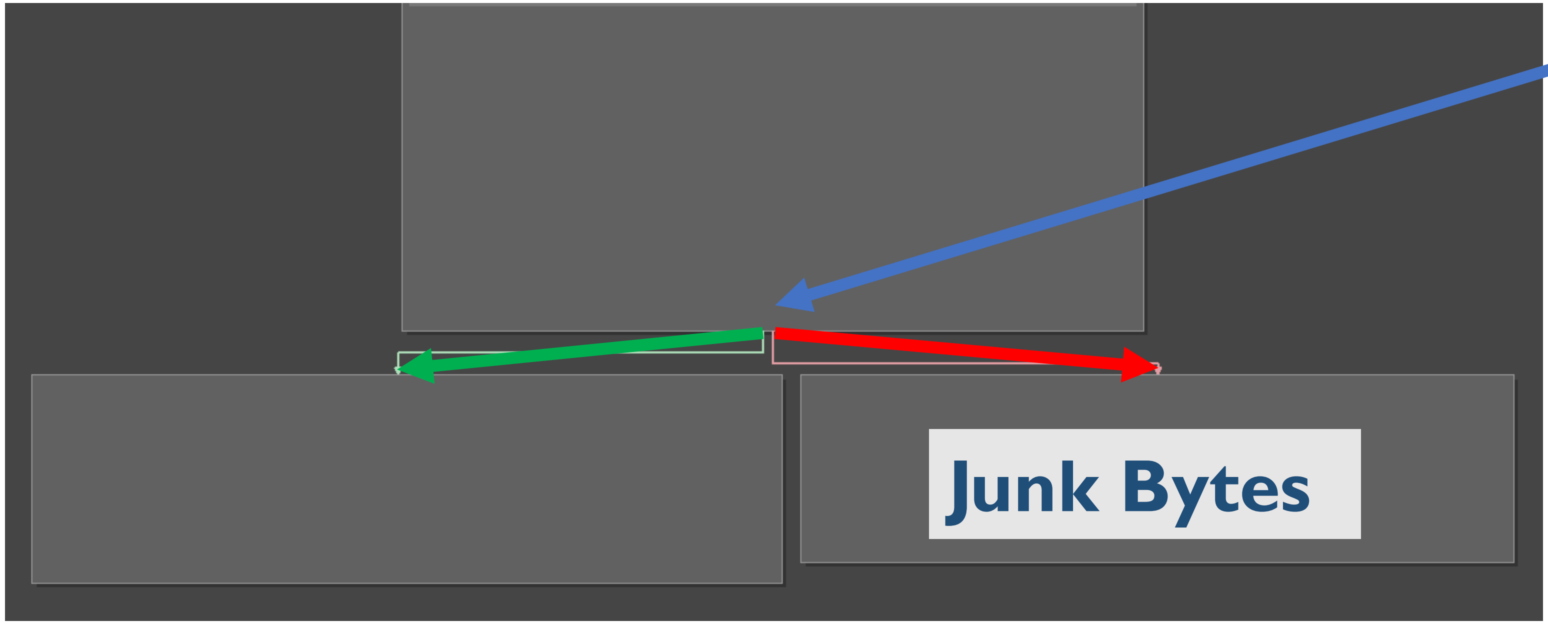
```
B8 31 C0 D1 C8    mov eax, C8D1C031h
C3                retn
```

Basic Block B and Basic Block C are siblings.

Opaque Predicates

Leaking Assumption: It will always assume an opaque predicate looks like this:

IDA Pro can detect



Predicate always evaluate to True

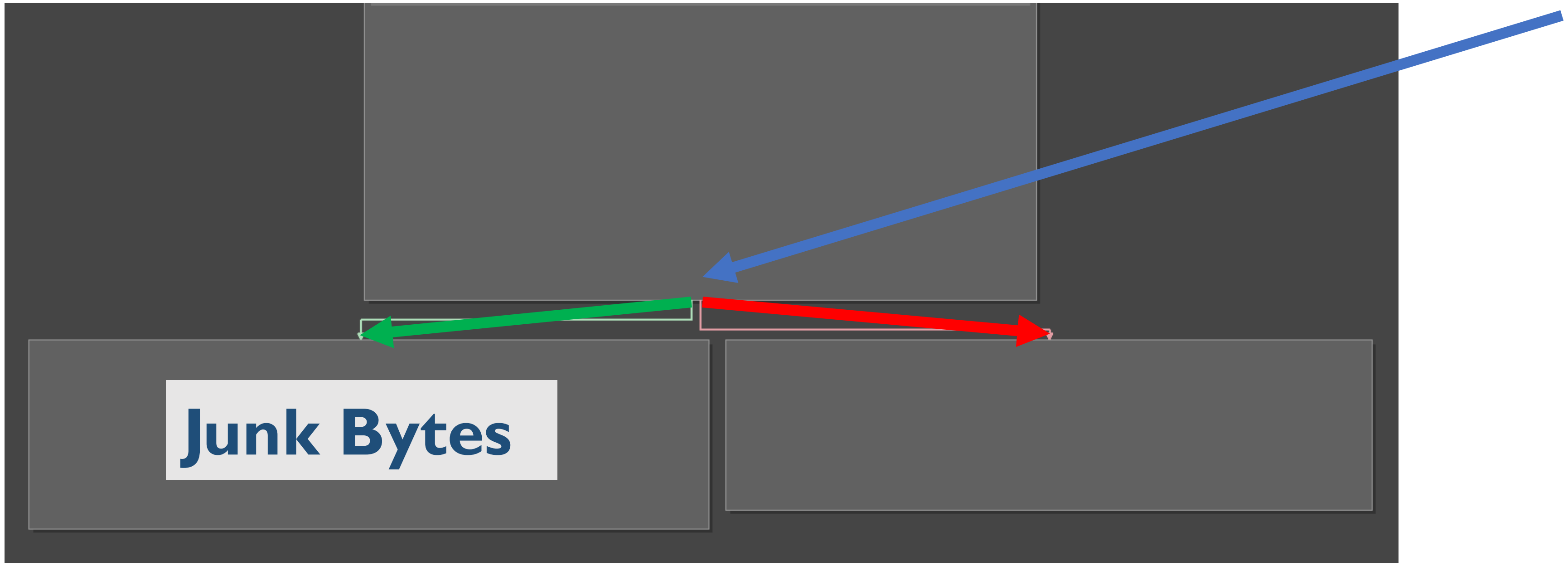
"Opaque Predicates"

Opaque Predicates

But an opaque predicate can also look like this:

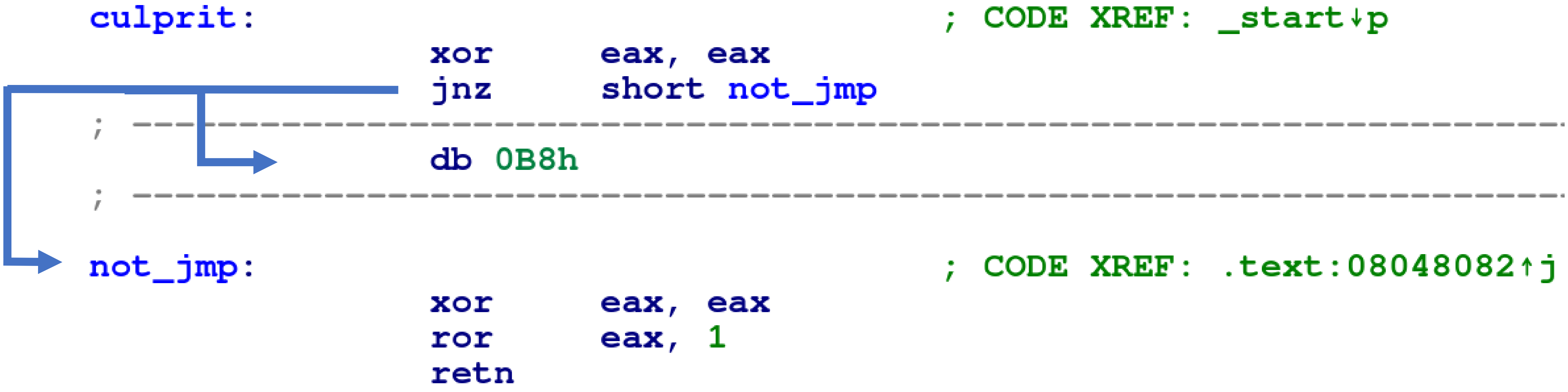
IDA Pro cannot detect

Predicate always evaluate to False



"Opaque Predicates"

Hiding Genuine Instruction: Displayed



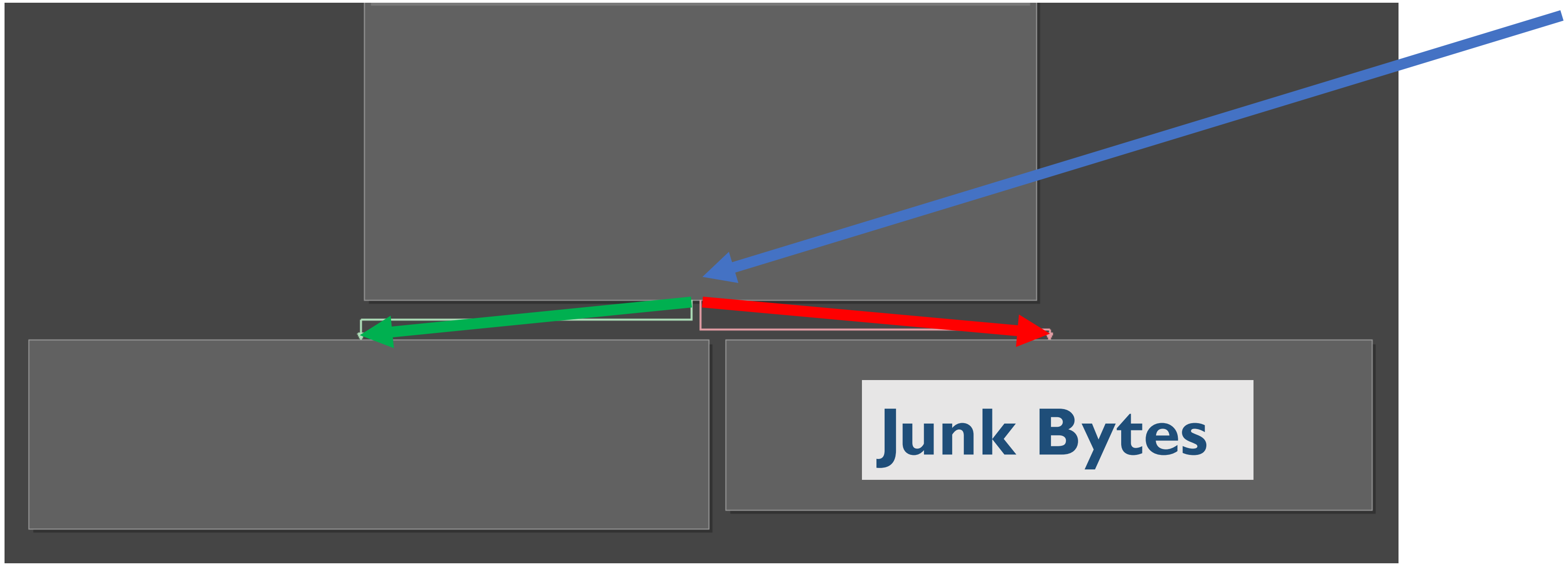
```
culprit:                                ; CODE XREF: _start↓p
                xor     eax, eax
                jnz     short not_jmp
; ---------------------------------------------------------------------------
                db 0B8h
; ---------------------------------------------------------------------------

not_jmp:                                ; CODE XREF: .text:08048082↑j
                xor     eax, eax
                ror     eax, 1
                retn
```

IDA's disassembly of the culprit function shows that it will return 0 when in reality it returns a nonzero value.

Hiding Genuine Instruction: Displayed

When IDA detects sibling basic blocks with overlapped instructions, it will assume that the opaque predicate looks like this:

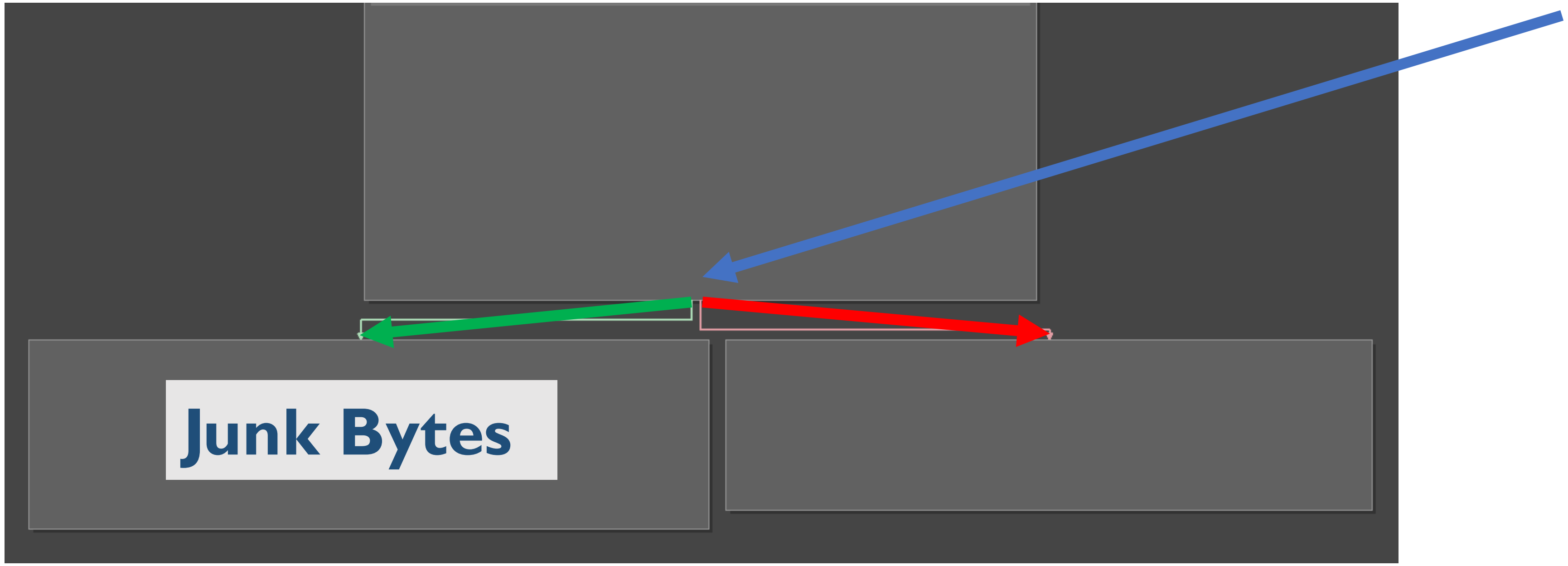


Predicate always evaluate to True

"Opaque Predicates"

Hiding Genuine Instruction: Displayed

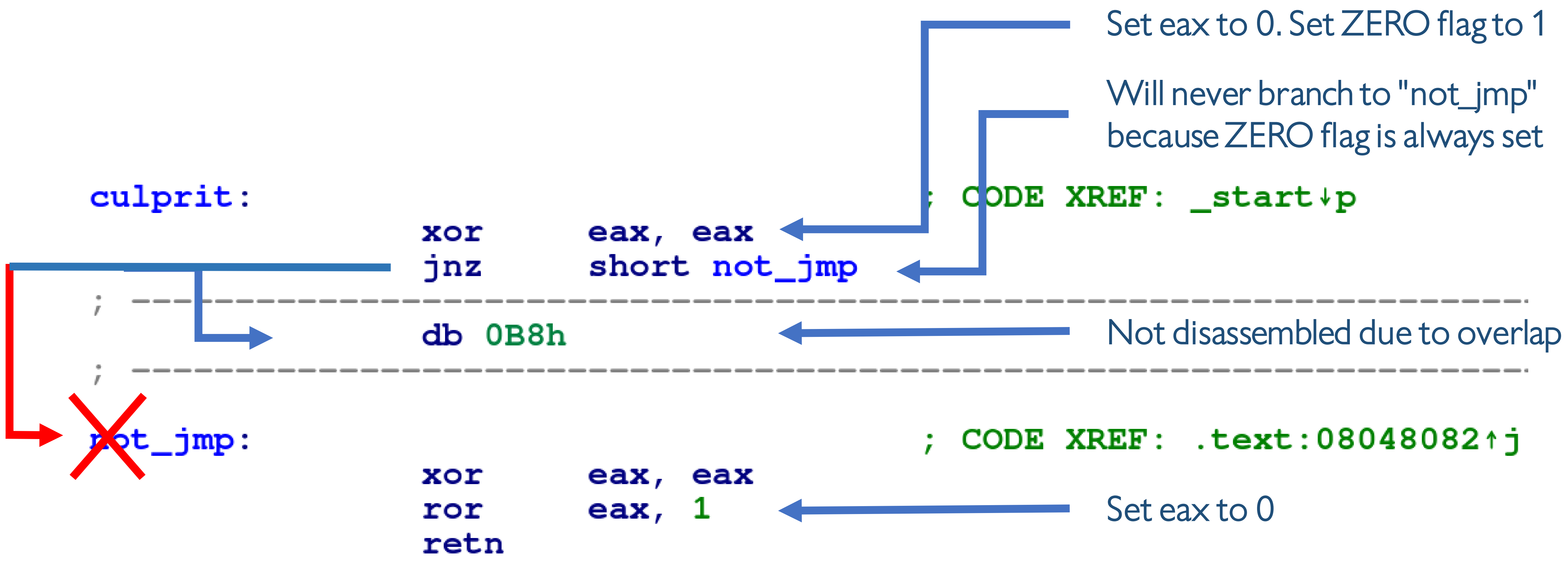
But our example opaque predicate instead looks like this:



Predicate always evaluate to False

"Opaque Predicates"

Hiding Genuine Instruction: Displayed



```
culprit:                                        ; CODE XREF: _start↓p
                xor     eax, eax
                jnz     short not_jmp
; -----------------------------------------------------------------
                db 0B8h
; -----------------------------------------------------------------
not_jmp:                                        ; CODE XREF: .text:08048082↑j
                xor     eax, eax
                ror     eax, 1
                retn
```

Set eax to 0. Set ZERO flag to 1

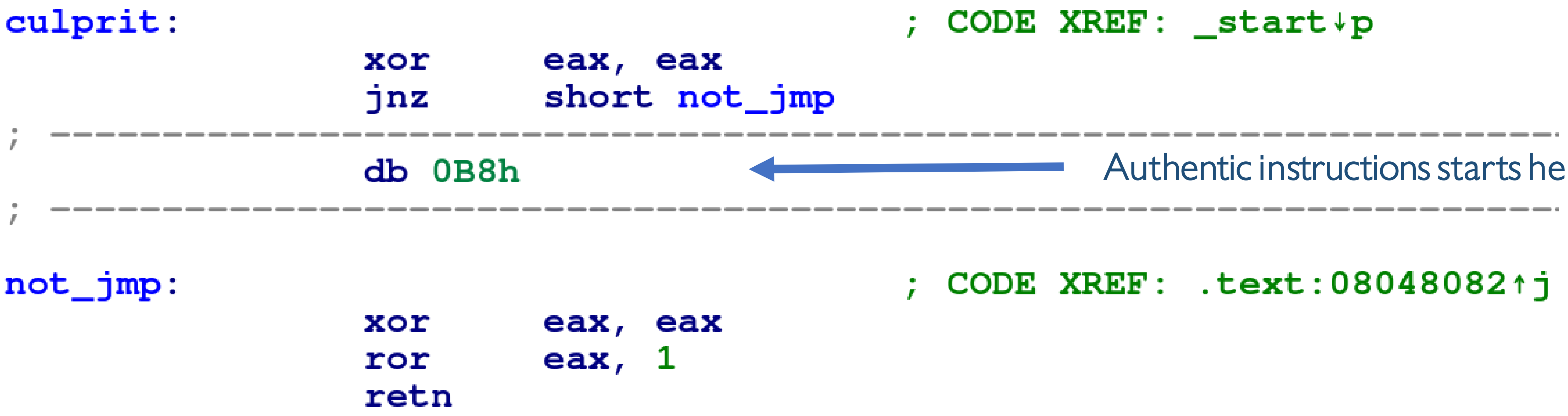
Will never branch to "not_jmp" because ZERO flag is always set

Not disassembled due to overlap

Set eax to 0

IDA's disassembly of the culprit function shows that it will return 0 but at runtime it returns a nonzero value.

Hiding Genuine Instruction: Displayed



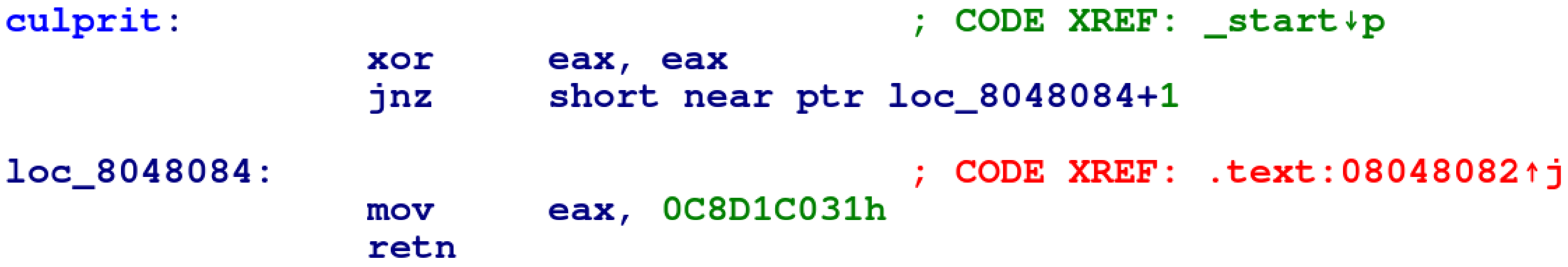
```
culprit:                                ; CODE XREF: _start↓p
                xor     eax, eax
                jnz     short not_jmp
; ---------------------------------------------------------------------------
                db 0B8h
; ---------------------------------------------------------------------------

not_jmp:                                ; CODE XREF: .text:08048082↑j
                xor     eax, eax
                ror     eax, 1
                retn
```

Authentic instructions starts here!

IDA's disassembly of the culprit function shows that it will return 0 but at runtime it returns a nonzero value.

Hiding Genuine Instruction: Executed



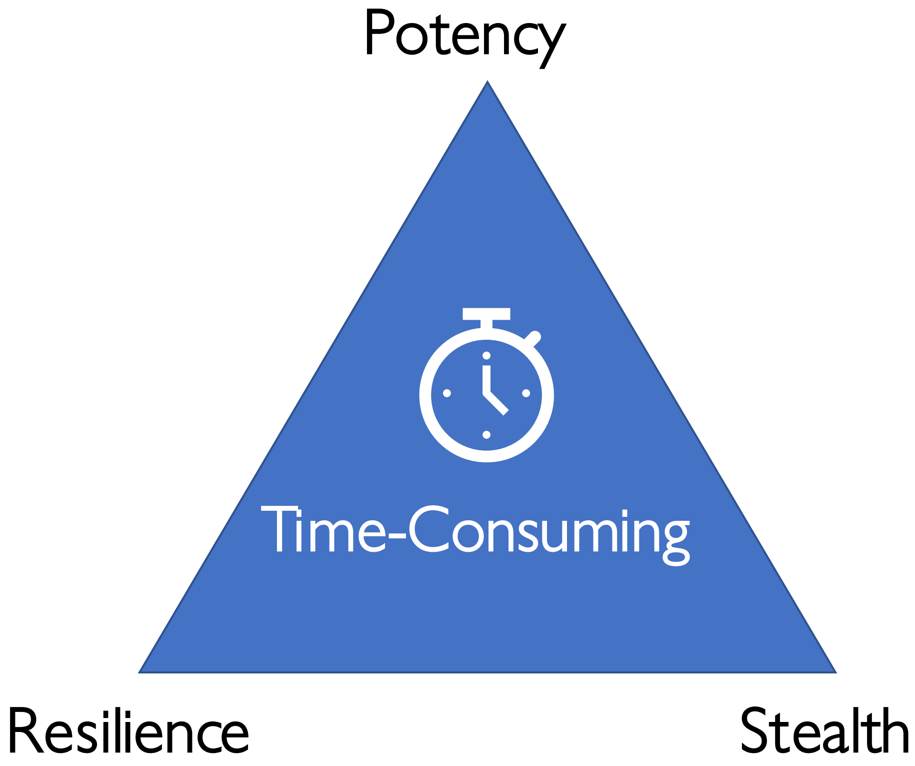
```
culprit:                                        ; CODE XREF: _start↓p
                xor     eax, eax
                jnz     short near ptr loc_8048084+1

loc_8048084:                                    ; CODE XREF: .text:08048082↑j
                mov     eax, 0C8D1C031h
                retn
```

Parent function of culprit can display convoluted behaviors if culprit returns 0 to confuse a reverser.

Main Takeaway

In implementing obfuscation, try to respect each property that makes up the "time-consuming" aspect!



Potency

Time-Consuming

Resilience

Stealth