A Heuristic Approach to Detect Opaque Predicates that Disrupt Static Disassembly

By: Yu-Jye Tung, Ian G. Harris
**Opaque Predicates**

**Definition:** conditional branches that always evaluate to true or false. Thus, one of their branches is unreachable at runtime (a.k.a *superfluous branch*).
Opaque Predicates

The **damage** is what's inserted into the unreachable basic blocks introduced by opaque predicates' superfluous branches.
Opaque Predicates' Damage

- Code Bloat
- Disassembly Desynchronization

"Opaque Predicates"

Invariant expression evaluates to True
Other Approaches

- Dynamic Symbolic Execution
- Value-Set Analysis
- Pattern Matching
- Machine Learning
- Statistical Analysis

Other approaches to detecting opaque predicates include:

**Dynamic Symbolic Execution**


Classification of Opaque Predicates

- **Trivial**
  - Invariant expression is constructed inside a basic block.

- **Weak**
  - Invariant expression is constructed throughout a function.

- **Strong**
  - Invariant expression is constructed across multiple functions.

- **Full**
  - Invariant expression is constructed across multiple processes.

Our Detection Method

We detect opaque predicates by identifying the superfluous branch whose target basic block contains the damage. Currently, we focus on when the damage is disassembly desynchronization.
How Our Method Identifies Damage

Our method can correctly identify the superfluous branch by analyzing each conditional branch's outgoing basic blocks for illogical behaviors.
Our Rules To Identify Illogical Behaviors

1. $B \leftarrow$ set of basic blocks originating from a conditional branch
2. $rules \leftarrow \{
3. nonexistence_memory_address,
4. unreasonable_memory_offset,
5. abrupt_basic_block_end,
6. unimplemented_BNIIls_percentage,
7. privileged_instruction_usage,
8. memory_pointer_constraints,
9. defined_but_unused,
10. \}
11:
12. \textbf{for each} $b \in B \textbf{ do}
13. \hspace{1em} \textit{illogical_basic_block} \leftarrow \texttt{false}
14. \hspace{1em} \textbf{for each} $r \in rules \textbf{ do}
15. \hspace{2em} \textbf{if} $r(b) \textbf{ then}
16. \hspace{3em} \textit{illogical_basic_block} \leftarrow \texttt{true}
17. \hspace{3em} \textbf{break}
18. \hspace{2em} \textbf{end if}
19. \hspace{1em} \textbf{end for}
20. \hspace{1em} \textbf{if} \textit{illogical_basic_block} \textbf{ then}
21. \hspace{2em} \textbf{print} ”\textit{b’s origin is an opaque predicate}”
22. \hspace{2em} \textbf{end if}
23. \textbf{end for}
Nonexistence Memory Address

• Target address of a control-flow altering instruction must be in the executable section of mapped address space.
• Memory location used to store written data must be in writable section of mapped address space.
Unreasonable Memory Offset

- A memory offset should not be extremely large or small.
- A data structure in high-level programming languages (e.g., array, structure) is accessed by an offset from the beginning of the data structure when compiled to machine code.
Abrupt Basic Block End

- An incomplete basic block cannot be part of the disassembly.
- A basic block is an incomplete basic block if it does not have a unique exit point, with explicit outgoing edges or implicit outgoing edges.

<table>
<thead>
<tr>
<th>bytes in hex</th>
<th>corresponding disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>inc edx</td>
</tr>
<tr>
<td>d0afeb158b85</td>
<td>shr byte [edi-0x7a74ea15], 0x1</td>
</tr>
<tr>
<td>60</td>
<td>pushad</td>
</tr>
<tr>
<td>ff</td>
<td>??</td>
</tr>
</tbody>
</table>
Unimplemented BNILs Percentage

- A basic block is illogical if it contains too many instructions that BinaryNinja’s lifter cannot lift to BNILs.

```
180 @ 08048612  unimplemented  {out dx, al}
181 @ 08048613  if (flag:d) then 275 else 279

"LLIL"
```
Privileged Instruction Usage

• A user space program cannot execute a privileged instruction, or any instruction that can only be executed in the most privileged level.

"Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand)."
Memory Pointer Constraints

• A memory pointer should only be stored or accessed in a full-length register and never a sub-register (e.g., AX instead of EAX in x86).

• A memory pointer is restricted from operation by × and ÷ in the set of primitive arithmetic operators {+, −, ×, ÷}.

• A memory pointer should not store its own memory address to itself.

• If a memory pointer is a stack pointer, it cannot be directly assigned a constant since a stack pointer keeps track of current stack frame.
Defined But Unused

• Every defined variable should have a subsequent instruction that uses it.

"None of the status flags that TEST affects (SF, ZF, and PF ) are used"
Main Limitation

Detecting opaque predicates in the presence of the obfuscation technique 
**junk code insertion**.

- Inserts carefully selected code into the instruction stream such that the 
inserted code will not affect program functionalities.

```
mov eax, 1
mov eax, 3
```

Our dataflow rule, *defined_but_unused*, will erroneously identify a basic 
block containing junk code as exhibiting illogical behaviors.
Evaluation

We implement our method as a BinaryNinja plugin. 

github.com/yellowbyte/opaque-predicates-detective

RQ1

• What is the performance of our tool on protected code (TP, FN, F1)?

RQ2

• What is the error rate of our tool on unprotected code?
Evaluation: RQ2

We use all 109 GNU core utilities' executable binaries compiled with GCC at optimization level O0, O1, O2, and O3 as ground truth.

Of the 436 combined GNU core utilities’ executable binaries across the four optimization levels, our tool has 61 false positive identifications.

All 61 false positive identifications are found when analyzing executable binaries compiled at optimization level O0 since unoptimized binaries can naturally contain junk code and the *defined_but_unused* rule causes false identification in the presence of junk code.
Evaluation: Dataset

We evaluate our tool by inserting trivial, weak, and strong opaque predicates generated by Tigress into the obfuscation benchmark provided by Banescu.

tigress.wtf
github.com/tum-i22/obfuscation-benchmarks

Note: we discard source files in benchmark that are randomly generated by Tigress since randomly generated programs are unrealistic examples.
**Evaluation: RQ1**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Classification</th>
<th>Total Conditionals</th>
<th>TP/Total Opaque Predicates</th>
<th>Detection Percentage</th>
<th>FP</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Tool</td>
<td>trivial</td>
<td>2465</td>
<td>221/297</td>
<td>74.41%</td>
<td>40</td>
<td>79.21%</td>
</tr>
<tr>
<td></td>
<td>weak</td>
<td>4657</td>
<td>212/297</td>
<td>71.38%</td>
<td>33</td>
<td>78.22%</td>
</tr>
<tr>
<td></td>
<td>strong</td>
<td>757</td>
<td>26/33</td>
<td>78.78%</td>
<td>2</td>
<td>85.24%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>7879</td>
<td>459/627</td>
<td>73.20%</td>
<td>75</td>
<td>79.06%</td>
</tr>
</tbody>
</table>

Accuracy of our tool on detecting trivial, weak, and strong opaque predicates.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Classification</th>
<th>Total Conditionals</th>
<th>TP/Total Opaque Predicates</th>
<th>Detection Percentage</th>
<th>FP</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Tool</td>
<td>trivial</td>
<td>2465</td>
<td>174/297</td>
<td>58.58%</td>
<td>31</td>
<td>69.32%</td>
</tr>
<tr>
<td></td>
<td>weak</td>
<td>4657</td>
<td>155/297</td>
<td>52.18%</td>
<td>23</td>
<td>65.26%</td>
</tr>
<tr>
<td></td>
<td>strong</td>
<td>757</td>
<td>20/33</td>
<td>60.60%</td>
<td>2</td>
<td>72.72%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>7879</td>
<td>349/627</td>
<td>55.66%</td>
<td>56</td>
<td>67.63%</td>
</tr>
</tbody>
</table>

Accuracy of our tool on detecting trivial, weak, and strong opaque predicates without defined\_but\_unused rule.
If the inserted junk bytes create multiple unreachable basic blocks and our rules detect illogical behaviors in an unreachable basic block that does not contain the start of the junk bytes sequence.

"2f a0 29 ab 61 4b 72"

<table>
<thead>
<tr>
<th>Basic Block A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2f</td>
<td>das</td>
</tr>
<tr>
<td>a029ab614b</td>
<td>mov al, byte [0x4b61ab29]</td>
</tr>
<tr>
<td>72eb</td>
<td>jb 0x8048752</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Block B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0a837d90010f</td>
<td>or al, byte [ebx+0xf01907d]</td>
</tr>
<tr>
<td>8ed7</td>
<td>mov ss, di</td>
</tr>
<tr>
<td>0100</td>
<td>add dword [eax], eax</td>
</tr>
<tr>
<td>008b459099c1</td>
<td>add byte [ebx-0x3e666fbb], cl</td>
</tr>
<tr>
<td>Ea1f01d083e001</td>
<td>jmp far 0x83d0011f, 0x1e0</td>
</tr>
</tbody>
</table>
Summary

An invariant expression in a conditional branch is not the only identifier for an opaque predicate; it can also be identified through its superfluous branch.

Here we present the first approach to detect opaque predicates by identifying corresponding superfluous branches. [github.com/yellowbyte/opaque-predicates-detective](https://github.com/yellowbyte/opaque-predicates-detective)

This novel approach allows us to detect opaque predicates that disrupt disassembly regardless of how the invariant expression is constructed.