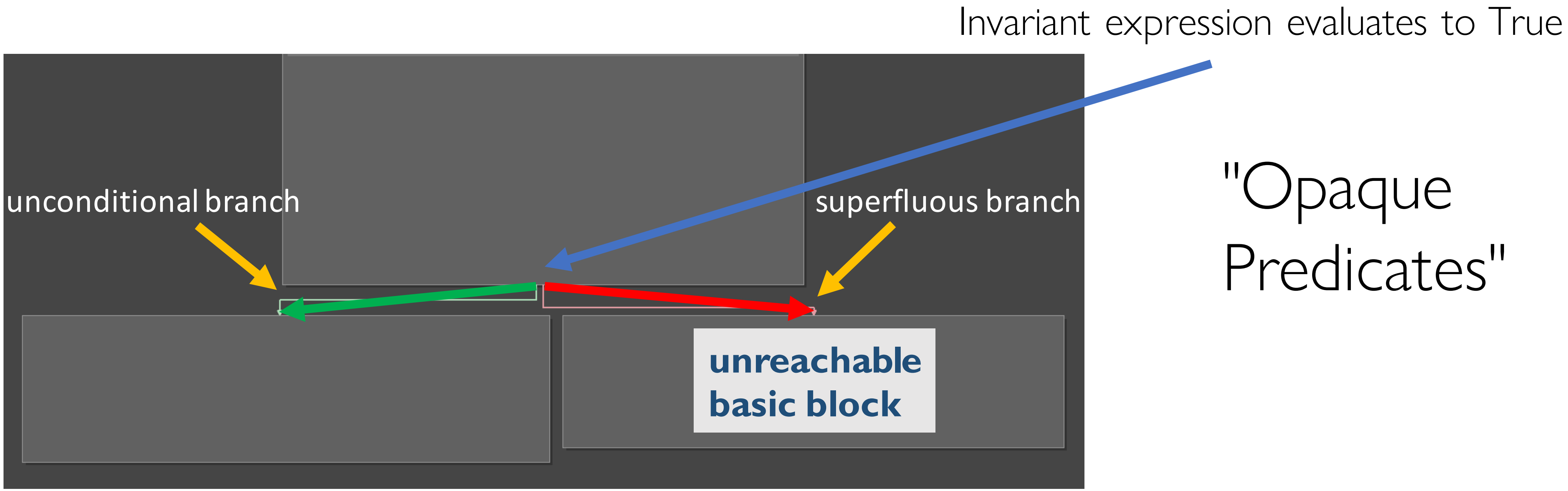
A Heuristic Approach to Detect Opaque Predicates that Disrupt Static Disassembly

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**).

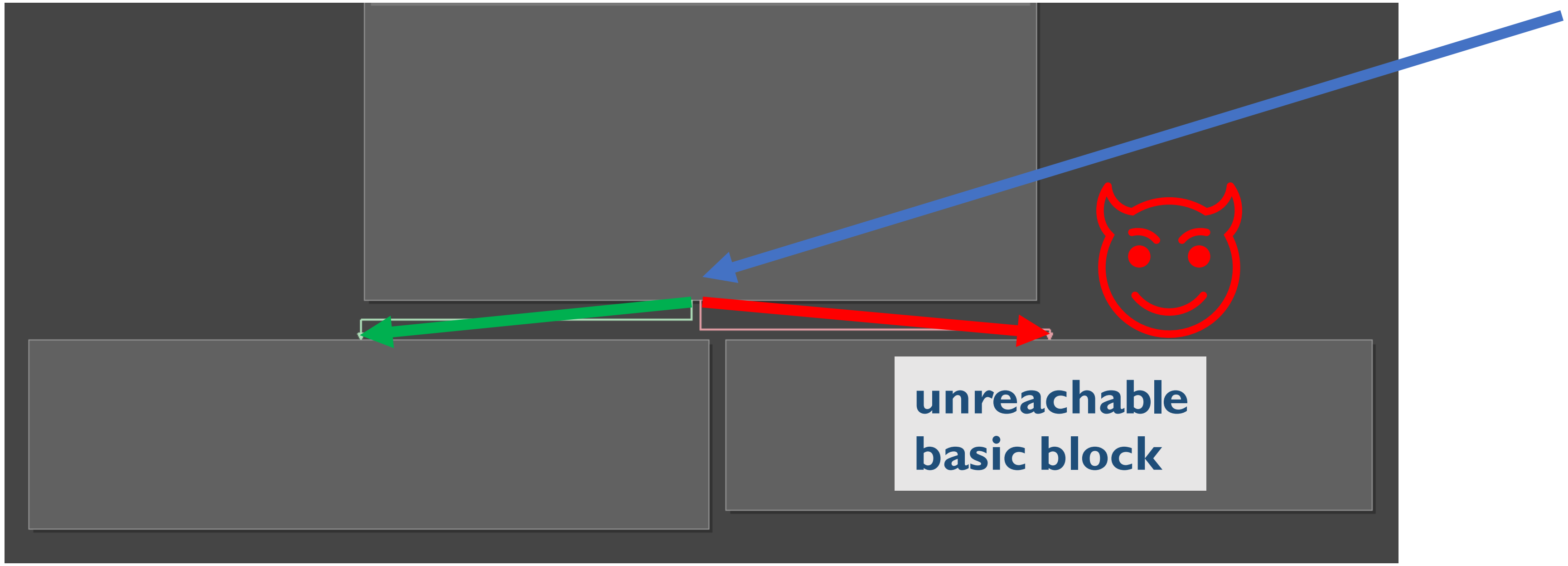


"Opaque Predicates"

Opaque Predicates

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

Invariant expression evaluates to True

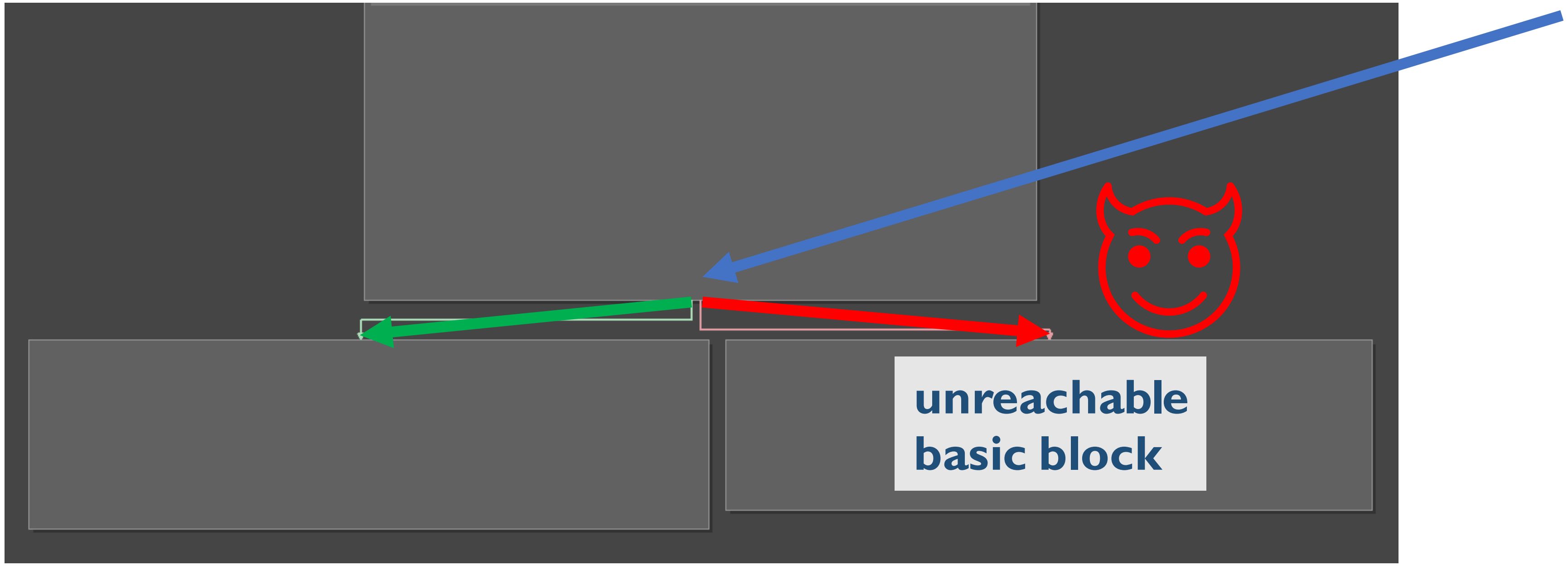


"Opaque Predicates"

Opaque Predicates' Damage

- Code Bloat
- Disassembly Desynchronization

Invariant expression evaluates to True



"Opaque Predicates"

Other Approaches

Dynamic Symbolic Execution

Machine Learning

Pattern Matching

Value-Set Analysis

Statistical Analysis

BINSEC

Vector35 / OpaquePredicatePatcher

Does the conditional branch contain an invariant expression?

Ref.: S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded dse: targeting infeasibility questions on obfuscated codes," in 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017, pp. 633–651.

Ref.: M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," in International Conference on Algebraic Methodology and Software Technology. Springer, 2006, pp. 81–95.

Ref.: P. LaFosse (2017) Automated opaque predicate removal. [Online]. Available: https://binary.ninja/2017/10/01/automated-opaque-predicate-removal.htm.

Ref.: R. Tofighi-Shirazi, I. Asăvoae, P. Elbaz-Vincent, and T.-H. Le, "Defeating opaque predicates statically through machine learning and binary analysis," in Proceedings of the 3rd ACM Workshop on Software Protection. ACM, 2019, pp. 15–26.

Ref.: J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 757–768.

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

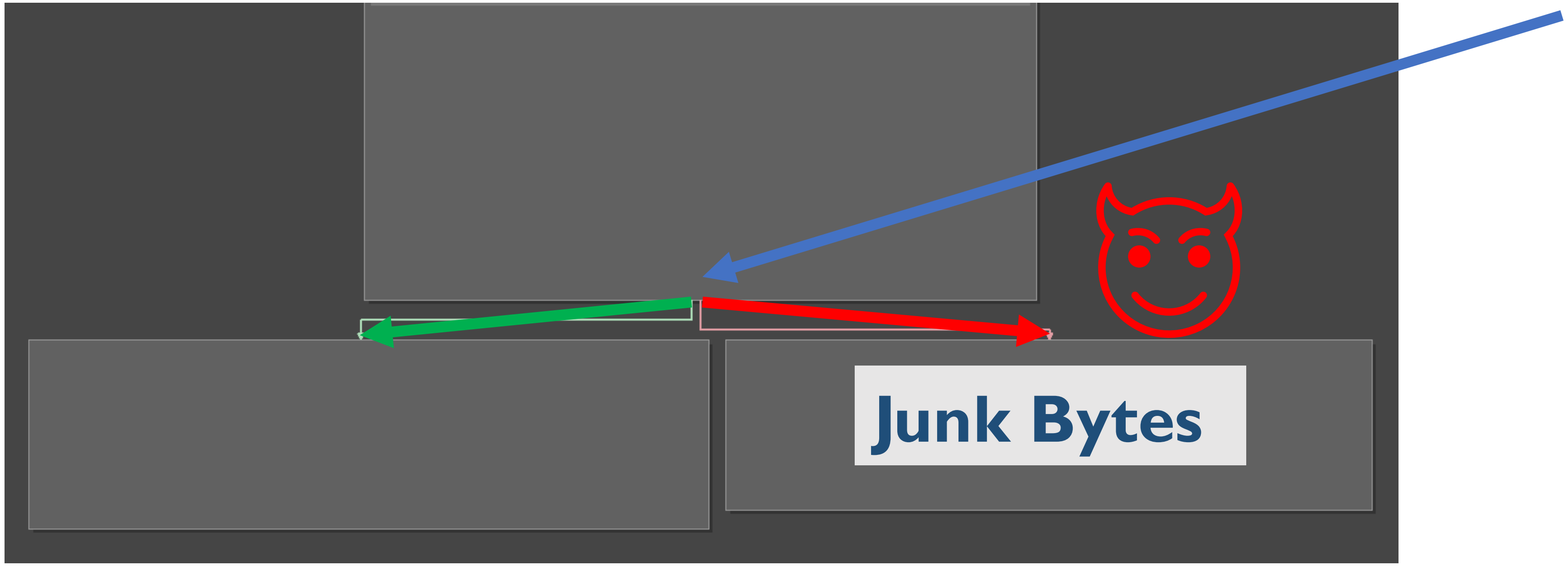
- Invariant expression is constructed across multiple processes.

Ref.: C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

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**.

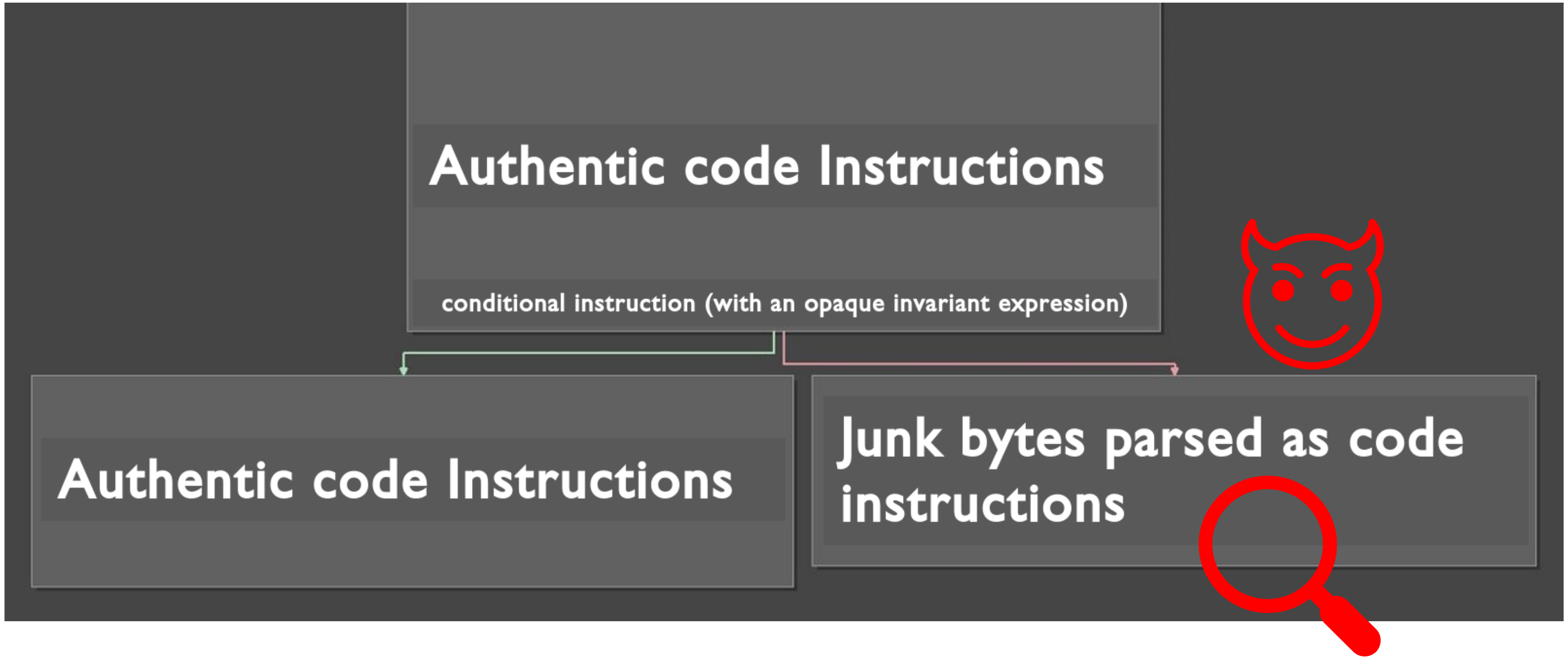
Invariant expression evaluates to True



"Opaque Predicates"

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

- nonexistence memory address
- unreasonable memory offset
- abrupt basic block end
- unimplemented BNILs percentage
- priviledge instruction usage
- memory pointer constraints
- defined but unused

```
B ←
set of basic blocks originating from a conditional branch
rules ← {
 nonexistence_memory_address,
 unreasonable_memory_offset,
 abrupt_basic_block_end,
 unimplemented_BNILs_percentage,
 privileged_instruction_usage,
 memory_pointer_constraints
 defined_but_unused,
}

for each b ∈ B do
    illogical_basic_block ← false
    for each r ∈ rules do
        if r(b) then
            illogical_basic_block ← true
            break
        end if
    end for
    if illogical_basic_block then
        print ”b’s origin is an opaque predicate”
    end if
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.

```
0804878c   a2c3cd16db          mov      byte [0xffffffffdb16cdc3], al  {0xdb16cdc3}
08048791   c6                  ??
```

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.

```
08048c3b   c7059c400001b800…   mov    dword [0x100409c], 0xb8
08048c45   00eb                add    bl, ch
08048c47   05b8000000          add    eax, 0xb8
08048c4c   008b7de46533        add    byte [ebx+0x3365e47d], cl
08048c52   3d14000000          cmp    eax, 0x14
```

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.

bytes in hex	corresponding disassembly
42	inc edx
d0afeb158b85	shr byte [edi-0x7a74ea15], 0x1
60	pushad
ff	??

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 executes a privileged instruction, or any instruction that can only be executed in the most privileged level.

```
08048612   ee                  out     dx, al
08048613   a5                  movsd   dword [edi], [esi]  {0x0}
08048614   dd81a2eb380f        fld     qword [ecx+0xf38eba2]
0804861a   b645                mov     dh, 0x45
0804861c   e30f                jecxz   0x804862d
```

"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 $\times$ and $\div$ in the set of primitive arithmetic operators $\{+, -, \times, \div\}$.
- 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.

```
080486e9   a9bf1cd221    test   eax, 0x21d21cbf
080486ee   92            xchg   edx, eax
080486ef   e945010000    jmp    0x8048839
```

"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

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

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

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

Accuracy of our tool on detecting *trivial*, *weak*, and *strong* opaque predicates without *defined_but_unused* rule.

Reason For FP Other Than Junk Code

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"

Basic Block A

```
2f            das
a029ab614b    mov al, byte [0x4b61ab29]
72eb          jb   0x8048752
```

Basic Block B

```
0a837d90010f    or   al, byte [ebx+0xf01907d]
8ed7            mov  ss, di
0100            add  dword [eax], eax
008b459099c1    add  byte [ebx-0x3e666fbb], cl
Ea1f01d083e001  jmp far 0x83d0011f, 0x1e0
```

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

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