A Heuristic Approach to Detect Opaque Predicates that Disrupt Static Disassembly

Yu-Jye Tung
University of California, Irvine
yujyet@uci.edu

Ian G. Harris
University of California, Irvine
harris@ics.uci.edu

***Abstract*—Opaque predicates are used to perform code obfuscation by injecting superfluous branches into the program. Superfluous branches are the gateways for junk bytes or unreachable code to inconspicuously mingle with authentic code instructions. In this paper, we focus on the case where opaque predicates introduce junk bytes, thus causing damage to the static disassembly process when junk bytes are also treated as code. Although introduced two decades ago, detecting opaque predicates is still an unsolved problem due to the flexibility in their constructions. Past works on detecting opaque predicates only detect opaque predicates with specific constructions. We propose a novel approach to opaque predicates detection that allows us to generically detect opaque predicates when the damage is the inserted junk bytes. Our proposed approach is the first to detect opaque predicates by identifying their corresponding superfluous branches through the damage caused by the obfuscation. Preliminary experiments show the potential of this novel approach by detecting opaque predicates of varied constructions.**

I. Introduction

A. Context

Code obfuscation is a software protection mechanism that transforms a program into a more complex yet semantically-equivalent program [36]. Although code obfuscation does not provide strong mathematical guarantees on the protection it offers, it is still widely used in scenarios where cryptographically secured methods cannot be practically realized. Such is the case for executable binaries that need to run on an end-user's machine, where the user has the privilege to scrutinize an executable binary in any way he or she wants. While code obfuscation is used to protect proprietary software's algorithmic intellectual property, it is also used by malware authors to harden their malware against reverse engineering attempts [26], [42], [45], [30] since reverse engineering is an effective method to uncover behaviors the malware can exhibit during program runtime.

B. Problem

Code obfuscation can harden an executable binary from reverse engineering by complicating the retrieval of accurate and complete disassembly from it. Inaccurate disassembly contains instructions that will never be executed during program runtime, whereas incomplete disassembly fails to include instructions that are executed during program runtime. Accurate and complete disassembly is a fundamental requirement for reverse engineering before any code reasoning can be done. Accurate and complete disassembly is vital for *manual* reverse engineering since the majority of the work consists of reading, manipulating, and commenting on the disassembly to understand program behaviors. *Manual* reverse engineering still accounts for the majority of a reverse engineer's workflow [39], [28], [27], [41], [4], [34]. Accurate and complete disassembly is even more vital for *automated* reverse engineering as it is the basis that binary analysis platforms [35], [5], [12], [33], [17] rely on.

Opaque predicates can result in both inaccurate and incomplete disassembly. They are also popular because stealthy, cheap, and resilient opaque predicates can be generated at scale [36]. In fact, many real-world obfuscation tools have support for them [9], [24], [18], [8].

Fundamentally, opaque predicates inject **superfluous branches** (a.k.a dead branches), or branches that are never taken during program runtime, into the disassembly. An opaque predicate injects a superfluous branch by syntactically disguising an unconditional branch as a conditional branch. This disguise is enabled by an **invariant expression** that always evaluates to true or false. The eventual disguised conditional branch is composed of an unconditional branch and a superfluous branch. A simple opaque predicate in x86 assembly is demonstrated below:

```
xor eax, eax
jz always_jump
```

The disguised `JZ` instruction will always jump to the label "always_jump" because prior to the jump the zero flag is always set by the `XOR` instruction. Here, the invariant expression evaluates to true. The disguised `JZ` instruction is composed of a unconditional branch whose target address is the "always_jump" label (true branch) and a superfluous branch whose target address is the location immediately following the `JZ` instruction (false branch).

There are two types of damage that can result from opaque predicates: code bloat [10] or disassembly desynchronization [22], [13]. Code bloat complexifies reverse engineering by inserting unreachable code, or code that will never be executed, into the instruction stream. On the other hand, disassembly desynchronization complexifies reverse engineering by

Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-62-2
https://dx.doi.org/10.14722/bar.2020.23004
www.ndss-symposium.org

```
.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
X .text:08049396 loc_8049396:                          ; CODE XREF: main+E4B↑j
.text:08049396                 add     al, bl
.text:08049398                 dec     eax
```

Fig. 1: An opaque predicate with an invariant expression that evaluates to false.

inserting junk bytes, or data bytes that are not meant to be parsed as code instructions, into the instruction stream. When the damage is disassembly desynchronization, junk bytes are inserted into superfluous branches' target basic blocks as they are unreachable during program runtime. This complicates the retrieval of accurate and complete disassembly since disassemblers that cannot identify the opaque predicate will parse the junk bytes as code instructions; note that junk bytes have high statistical chance of having corresponding legal instructions on compact instruction set architecture (ISA) such as x86 and ARM Thumb.

Our current method focuses on detecting opaque predicates when the damage is disassembly desynchronization. Disassembly desynchronization results in both inaccurate and incomplete disassembly.

Inaccurate disassembly leads to unintended red-herrings that increase overall reverse engineering time. Figure 1 shows the damage done by an opaque predicate with an invariant expression that evaluates to false. The bolded `JZ` instruction will never jump but modern disassemblers like IDA Pro [16] cannot statically determine that property, leading to IDA Pro parsing the junk bytes at `JZ` instruction's jump target (0x8049396) as instructions, resulting in inaccurate disassembly since instructions that will never be executed are also part of the disassembly. The "X" in Figure 1 implies program execution will never reach there.

On the other hand, incomplete disassembly, like a jigsaw puzzle without all the pieces, increases overall reverse engineering time since the whole picture to reason about program interactions is not present. Figure 2 shows the damage done by an opaque predicate with an invariant expression that evaluates to true. The bolded `JNZ` instruction will always jump but since the subsequent instructions overlap the instructions at `JNZ`'s jump target, IDA Pro cannot display both disassembly sequences. Here IDA Pro makes the choice to only disassemble the subsequent instructions following `JNZ`, but the subsequent instructions are the injected junk bytes. The "?" at Figure 2's jump target implies authentic instructions are not disassembled because of the mis-disassembled junk bytes. The havoc caused by this particular opaque predicate results in both incomplete and inaccurate disassembly.

```
.text:08049AD1                 jnz   short near ptr loc_8049AD3+4
.text:08049AD3
.text:08049AD3 loc_8049AD3:                            ; CODE XREF:
.text:08049AD1
.text:08049AD3                 and   [edx+458B30EBh], edx
? .text:08049AD9               cmp   byte ptr [ebx], 45h
.text:08049ADC                 loopne  near ptr loc_8049B52+1
.text:08049ADE                 adc   eax, 0FF08EC83h
```

Fig. 2: An opaque predicate with an invariant expression that evaluates to true.

C. Contributions

In this paper, we propose a novel approach to detect opaque predicates that cause disassembly desynchronization; from hereon forward, we will call this type of opaque predicates **desynchronizing opaque predicates**. Our approach generically detects an opaque predicate without identifying its invariant expression by identifying its superfluous branch instead. An opaque predicate's superfluous branch exists in a disguised conditional branch. We analyze the disassembly of each basic block originating from a conditional branch for illogical behaviors to identify superfluous branches since junk bytes exist in the target basic block of a desynchronizing opaque predicate's superfluous branch. In Section IV, we define a simple set of non-exhaustive heuristic-based rules that model illogical behaviors. In Section V, we show that our simple set of heuristic-based rules produce promising results in generically detecting desynchronizing opaque predicates. Our approach is the first to detect opaque predicates by identifying superfluous branches instead of invariant expressions.

Previous works that detect opaque predicates strictly with pattern matching [14], [40] avoid the need to determine opaque predicates' invariant expressions, but are restricted to detecting a small and specific subset of possible opaque predicates. In recent years, dynamic symbolic execution-based approaches to detection [25], [32], [3] have shown to be effective in detecting opaque predicates whose invariant expressions can be determined at the basic block level. For example, Backward-Bounded DSE [3] can detect 100% of the opaque predicates inserted by the OLLVM obfuscator [18]. However, dynamic symbolic execution-based approaches' detection accuracy decreases when encountering opaque predicates of varied constructions [37]. This is because their detection approaches are based on determining if a conditional branch contains an invariant expression. Depending on how the opaque predicate is constructed, it can be non-trivial or even undecidable to identify the invariant expression. Tofighi-Shirazi *et al.* [37] recently introduce a detection approach using machine learning based on a decision-tree classification model. Although it can detect opaque predicates of varied constructions, the performance of Tofighi-Shirazi's approach suffers when detecting an opaque predicate whose construction is not previously encountered in its training data.

We propose an approach that can generically detect desynchronizing opaque predicates across varied constructions. Our approach can also detect new or never-seen-before opaque predicate constructions since our detection approach identifies the damage done by an opaque predicate, which is independent of the specific construction.

The key contributions we present in this paper are the

following:

- We propose a novel approach to desynchronizing opaque predicates detection that is effective in detecting opaque predicates of any construction.
- We implement our approach as a BinaryNinja [1] plugin and release it on GitHub[1] to facilitate further research in this area.
- We present a preliminary experimental evaluation assessing the potential of our method.

Our current limitation in detecting desynchronizing opaque predicates is when the junk code insertion obfuscation technique is also present. Junk code insertion inserts carefully selected, but useless, code instructions into the instruction stream such that primary program functionalities will not be affected even if the junk code is executed. As explained in Section VI-A, our dataflow-based rule will detect junk code as code instructions manifested from junk bytes. Section VI-A also provides possible mitigation we will explore in the future to eliminate false positive identifications in the presence of junk code. In future work, we will also work on detecting opaque predicates when the damage is code bloat. Section V-D and VI-B discuss how our current method will perform when the damage is code bloat.

II. Background

A. Opaque Predicates

Collberg *et al.* introduce opaque predicates in 1997 [10]. A predicate is a conditional statement that evaluates to a boolean value, true or false. A predicate P is opaque at location p in the program if its boolean value is known during obfucation but with greater difficulty determined post-obfuscation. There are two main types of opaque predicates: P_p^F if P always evaluates to false at p and P_p^T if P always evaluates to true at p. The unique property of P_p^F and P_p^T is enabled by invariant expressions. An invariant expression allows an unconditional branch to syntactically acquire a conditional branch disguise. This disguise composes of an unconditional branch and a superfluous branch.

When Collberg introduces opaque predicates, he suggests the use of opaque predicates to increase program complexity. Linn [22] is the first to suggest using opaque predicates to damage the disassembly by inserting junk bytes into superfluous branches' unreachable basic blocks.

B. Disassembly Desynchronization

Desynchronizing opaque predicates fall under an obfuscation class called disassembly desynchronization [13]. The characterization of opaque predicates into disassembly desynchronization is not universally accepted, as the term is only used in *The IDA Pro Book* [13], but is useful for our purpose since all the obfuscation techniques that fall under it degrade quality of the disassembly. The following are all part of disassembly desynchronization: branch function [22], call conversion [22], [13], opaque predicates [10], [22], [13], jump table spoofing [22], and overlapping instructions [7], [13]. Of all the obfuscation techniques under disassembly desynchronization, the technique representing opaque predicates is of the utmost importance due to opaque predicates' prevalence in-the-wild and flexibility in their constructions.

Beside overlapping instructions, the obfuscation techniques listed above can all degrade retrieved disassembly's quality through the insertion of junk bytes into the instruction stream.

Branch function and call conversion place junk bytes after x86's `CALL` instruction; the function in the `CALL` operand, or callee, will alter its return address so at callee's function completion it will not return to the instruction following x86's `CALL` instruction, allowing the insertion of junk bytes instead. Branch function and call conversion exploit the traditional assumption made by disassemblers that the callee will return to the instruction following its invocation to mislead disassemblers to parse junk bytes following x86's `CALL` as instructions.

There has been research in deobfuscating transformations made by disassembly desynchronization, including Kruegel *et al.* [20]. Kruegel introduces disassembly strategies to handle disassembly desynchronization, particularly handling junk bytes added after x86's `CALL` instruction. Kruegel's work demonstrates high accuracy in eliminating the disassembly degradation effects caused by branch function and call conversion.

Overlapping instructions, jump table spoofing, and opaque predicates are not handled by Kruegel's disassembly strategies as he argues that they are impractical. Overlapping instructions' disassembly degradation effects are shown to be minimal and there are very few candidates satisfying the criteria for it [22]. Jump table spoofing is essentially a variant of opaque predicates specific to creating artificial jump tables. Kruegel argues that opaque predicates not easily recognizable by a disassembler are non-trivial to create [20]. However, time has shown that detecting opaque predicates is a pressing issue as most real-world obfuscation tools now have support for inserting opaque predicates [9], [18], [23], but modern disassemblers like IDA Pro struggle to identify them (Figure 1 and Figure 2).

C. Classification of Opaque Predicates

Collberg classifies opaque predicates based on their resiliency against automatic detection [10]. In order of increasing resiliency, the classification is: **trivial**, **weak**, **strong**, and **full**. A reason for opaque predicates' prevalence is in the flexibility of their constructions, which allows for the creation of opaque predicates with varying resiliency.

Trivial A *trivial* opaque predicate is constructed inside a basic block so its invariant expression can be identified at a basic block level.

Weak A *weak* opaque predicate is constructed throughout a function so it requires intra-procedural analysis to identify its invariant expression.

Strong A *strong* opaque predicate is constructed across multiple functions so it requires inter-procedural analysis to identify its invariant expression.

[1] https://github.com/yellowbyte/opaque-predicates-detective

Full A *full* opaque predicate is constructed across multiple processes so it requires inter-process analysis to identify its invariant expression.

By Collberg's classification, *trivial* opaque predicates are most prone to successful detection. An algebraic-based opaque predicate relies on a mathematical identity that will always evaluate to the same boolean value; it will fall in the category of *trivial* opaque predicates if not specially constructed since a mathematical identity's corresponding machine instructions are executed in sequence when directly translated, resulting in its entirety ending up in a single basic block. This is the case for the algebraic-based opaque predicates generated by the publicly available obfuscator OLLVM[2] [18]. The mathematical identity used by OLLVM's algebraic-based opaque predicates is the following:

$$\forall x, y \in \mathbb{Z}: y < 10 \lor (x \cdot (x - 1)) \bmod 2 == 0$$

Algebraic-based opaque predicates can also be classified as *weak* opaque predicates if the dependencies for the mathematical identity they rely on are constructed across multiple basic blocks in the same function.

D. Opaque Predicates Detection

Dynamic symbolic execution-based approaches [25], [32], [3] have mainly shown to effectively detect *trivial* opaque predicates. For example, beside effectively detecting *trivial* opaque predicates, Backward-Bounded DSE has also shown to detect *weak* opaque predicates found in the X-TUNNEL malware — although with more difficulties [3]. Yadegari *et al.* [44] propose a generic and semantic-preserving approach to deobfuscating any obfuscation technique. Theoretically, Yadegari's approach has the potential to detect across opaque predicates of different resiliency. Tofighi-Shirazi *et al.* [37] recently show that decision-tree based machine learning is effective in detecting opaque predicates of varied constructions, but Tofighi-Shirazi's approach can only detect opaque predicates whose constructions fall under *trivial* or *weak* resiliency since the training data for their machine learning model is generated using intra-procedural symbolic execution.

We are not aware of any previous work that evaluated against *strong* or *full* opaque predicates. We will be the first to evaluate against *strong* opaque predicates, but not *full* opaque predicates since an automatic method to construct them is yet to be implemented. Nevertheless, our approach should detect *full* opaque predicates as explained in Section III. *Full* opaque predicates cannot be ignored since they can still be constructed manually on an ad hoc basis. In Section V, we evaluate against *trivial*, *weak* and *strong* opaque predicates generated by the Tigress C Diversifier/Obfuscator [8].

III. Our Approach

Our approach to opaque predicates detection can detect desynchronizing opaque predicates across the whole classification spectrum. An opaque predicate is classified based on how an opaque predicate's invariant expression is constructed, but our approach does not have to identify the invariant expression to perform detection. Instead of detecting opaque predicates by identifying invariant expressions, we detect opaque predicates by reasoning on the disassembly of each basic block originating from a conditional branch as it is the site where junk bytes exist if the originating branch is a desynchronizing opaque predicate's superfluous branch. If the basic block's disassembly exhibits illogical behaviors either by itself or w.r.t. the other basic blocks in the same function, the conditional branch it originates from is a superfluous branch and its sibling basic block will always execute at program runtime. In Section V, we show that a simple, non-exhaustive set of heuristic-based rules defined in Section IV produce promising results in generically detecting desynchronizing opaque predicates.

IV. Algorithms

Our heuristic-based rules identify illogical behaviors, or code behaviors that should not happen during program runtime. We analyze each basic block that is a target destination for a conditional branch. If basic block's disassembly is illogical based on our heuristic-based rules, we conclude the conditional branch the basic block originates from is an opaque predicate's superfluous branch.

To statically reason on the disassembly of native machine instructions, we use BinaryNinja [1] to lift the native machine instructions to BinaryNinja's family of intermediate languages (BNILs). BNILs encode the behaviors of native machine instructions explicitly in their representations, allowing for a complete code analysis statically. Note that it is now customary for binary-level analysis to rely on intermediate representations (IRs) such as BNILs [33], [5], [12]. Algorithm 1 explains the overall opaque predicates detection process.

Algorithm 1 Detecting Opaque Predicates

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_BNILs_percentage,
7: privileged_instruction_usage,
8: memory_pointer_constraints
9: defined_but_unused,
10: $\}$
11:
12: **for each** $b \in B$ **do**
13: $illogical_basic_block \leftarrow false$
14: **for each** $r \in rules$ **do**
15: **if** $r(b)$ **then**
16: $illogical_basic_block \leftarrow true$
17: **break**
18: **end if**
19: **end for**
20: **if** $illogical_basic_block$ **then**
21: print "b's origin is an opaque predicate"
22: **end if**
23: **end for**

Lines 1 - 10 define two sets; the first set denotes all basic blocks where junk bytes can be inserted as a result of an

[2] https://github.com/obfuscator-llvm/obfuscator/wiki/Bogus-Control-Flow.

opaque predicate and the second set contains each heuristic-based rule. On line 12, we iterate through each basic block in the first set and on line 14 we iterate through each heuristic-based rule to pass each basic block as function argument to each heuristic-based rule (line 15). Each heuristic-based rule will have access to the basic block that is passed to it and the basic block's enclosing function. If any one of the heuristic-based rule identifies illogical behaviors based on the basic block itself or its interaction with the rest of the code in the same function, we set the "illogical_basic_block" variable to true (line 16). On line 20, we check the status of the "illogical_basic_block" variable after checking behaviors of the basic block with each heuristic-based rule. If the variable is set to true, we conclude the basic block originates from an opaque predicate's superfluous branch. The rest of this section describes the heuristic-based rules.

A. Heuristic-Based Rules.

a) nonexistence memory address (R1): The target address of a control-flow altering instruction must be in the executable section of mapped address space. Likewise, the memory location used to store written data must be in the writable section of mapped address space.

Formally, assume the mapped address space's range is a set represented by M, the executable section range is a set represented by E, and the writable section range is a set represented by W, then $E \subseteq M \wedge W \subseteq M$. If a control-flow altering instruction's target address A can be determined, this property must hold: $A \in E$. If A is an address that has data written to it during program runtime, this property must hold: $A \in W$.

b) unreasonable memory offset (R2): 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 down to native machine code. Typically a data structure is reasonably-sized so program complexity can be controlled in a maintainable manner. However, this is only considered to be a good coding practice as it is not always the case. We empirically determine the reasonable bound, with account for the extreme case, for a memory offset O to be the following: $0x100000 < O < 0x100000$.

c) abrupt basic block end (R3): 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 (e.g., an indirect control-flow altering instruction such as `RET`). Definition 1 defines an incomplete basic block.

Definition 1: A vector of bytes S contains an incomplete basic block if and only if an invalid instruction encoding byte exists at index i_1 in S and if a valid instruction encoding byte that decodes to a control-flow altering instruction exists at index i_2 in S, then this must hold: $i_1 < i_2$.

Figure 3 is an example of an incomplete basic block caused by the junk bytes sequence: '0x42 0xd0 0xaf'. Only the encoding of the first instruction, `inc edx`, is completely made up of bytes from the junk bytes sequence. The encoding of the next instruction is made up of the rest of the junk bytes in the sequence and arbitrary bytes that happen to neighbor it. Eventually, a byte not part of an instruction encoding, '0xff', is encountered, leading to an incomplete basic block. Junk bytes will not introduce an incomplete basic block if a control-flow altering instruction is disassembled before encountering an invalid instruction encoding byte, thus still forming a valid basic block (Figure 4). In Figure 4, although the byte '0xff' in the junk bytes sequence, '0x31 0xc0 0xeb 0x08 0xff', is not part of a valid x86 instruction encoding, it does not result in an incomplete basic block because the 2 bytes prior to '0xff' disassembled to a `JMP` instruction.

d) unimplemented BNILs percentage (R4): A basic block is illogical if it contains too many instructions that BinaryNinja's lifter cannot lift to its family of intermediate languages, BNILs. Binary lifting is not a straightforward task to perform since the documentations for native *instruction set architecture*s (ISAs) such as x86 and ARM are enormous and continue to increase in size [19]. Furthermore, the operational semantics for native ISA's instructions are informally defined or even completely left out in official documentations. As a result it is not surprising that BinaryNinja cannot lift all instructions in a native ISA to BNILs, but a copious amount of unliftable instructions concentrated in a single basic block is noteworthy. Therefore, this rule considers the numbers of unliftable instructions in a single basic block. Empirically, we identify illogical basic block based on the ratio of instructions in a single basic block B that intersects with the set of all unliftable instructions U in respective ISA: $\frac{|B \cap U|}{|B|} > 0.2$

e) privileged instruction usage (R5): A user space program, a program that has the least privilege, cannot executes a privileged instruction, or any instruction that can only be executed in the most privileged level. Note that this rule assumes the analysis target is an executable binary that is intended to execute in user space, which encompasses most programs, as only the operating system, device drivers, and hypervisors execute in a different privilege level.

In a computing system with a multitasking operating system, certain resources need to be restricted from direct access by any program to provide harmonious co-existence for all the programs that are running concurrently. This restriction is enforced by dividing instructions in an ISA into different privilege levels and assigning a specific privilege level to each program. In x86, there are 2 main privilege levels — ring 0 (most privilege) and ring 3 (least privilege). In ARM, there are 8 privilege levels with USR being the least privilege and SVC being the most privilege. In MIPS, the 2 main levels are User Mode and Kernel Mode. Other ISAs also have this concept of privilege isolation. A formal definition for this rule follows.

Definition 2: Let E represent the set of instructions found in the user space executable binary. Given that privileged instructions P is a subset of its corresponding ISA's complete set of instructions and a privileged instruction cannot execute in non-privileged mode, the following property must hold to avoid premature program termination: $|E \cap P| = 0$.

f) memory pointer constraints (R6): While a register contains a memory pointer, all subsequent usages of the register will exhibit distinctively restrictive behaviors. First, we identify registers containing memory pointers by the semantics

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

Fig. 3: Junk bytes that result in an incomplete basic block.

```
bytes in hex        corresponding disassembly
31c0                xor    eax, eax
eb08                jmp    <addr>
ff
```

Fig. 4: Junk bytes that do not result in an incomplete basic block.

of how a register is used. If a register is used to load data from memory or store data to memory, we identify it as containing a memory pointer. After identifying a register that contains a memory pointer, we place behavioral constraints that model behaviors an authentic memory pointer should not exhibit on all usages of the register while it contains the memory pointer. Identifying usages of a register while it is containing a particular value from all other usages of same register when its content could be different is made possible by representing the register in *single static assignment* (SSA) form. In SSA form, every variable is defined only once, meaning that if the same register is assigned multiple times that register will be represented by multiple variables in SSA form. The behavioral constraints for a memory pointer are the following:

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) since memory address size is the size of a full-length register.

A memory pointer is restricted from operation by and in the set of primitive arithmetic operators $f+,\ ,\ ,\ g$.

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.

Algorithm 2 contains the pseudocode used to describe the algorithm for this rule. The following helper functions are used in the pseudocode:

SIZE_OF(x, i): calculates the storage size, in bytes, for storage location x used in instruction i. x can be on the stack, heap, data section, bss section, or a register.

VALUE_OF(x, i): retrieves the data stored at storage location x in instruction i.

LEN(x): calculates the length, or number of bytes, used to represent x.

REGISTER_TYPE(x, i): retrieves the machine register x refers to in instruction i, assuming x is a register in SSA form.

ASSIGN_TO(x, y, i): return true if y is stored to storage location x in instruction i, else return false.

OPERATED_BY(x, y, i): return true if operation y is applied on value in storage location x in instruction i, else return false.

Algorithm 2 *memory pointer constraints* rule

Require: Beside sp, all other mentions of registers are in SSA form.
Require: elements in M exist in the same basic block
sp stack pointer register
n arbitrary element in the set of integers **Z**
a arbitrary element in the mapped address range
M set of registers containing a pointer
L_r set of instructions that accesses register $r\ 2\ M$
for each $r\ 2\ M$ **do**
  for each $i\ 2\ L_r$ **do**
    if SIZE_OF(r, i)≠LEN(a) **then**
      return true
    end if
    if OPERATED_BY(r, , i) **then**
      return true
    end if
    if OPERATED_BY(r, , i) **then**
      return true
    end if
    address = VALUE_OF(r, i)
    if ASSIGN_TO(address, address, i) **then**
      return true
    end if
    if REGISTER_TYPE(r, i) == sp **then**
      if ASSIGN_TO(r, n, i) **then**
        return true
      end if
    end if
  end for
end for
return false

Algorithm 3 *defined but unused* rule

$dead_return_value_seen \quad false$
for each $v\ 2\ V$ **do**
  if v 2/ A **then**
    if IS_RETURN_VALUE_VARIABLE(v) **then**
      if $dead_return_value_seen$ **then**
        return true
      else
        $dead_return_value_seen \quad true$
      end if
    else
      return true
    end if
  end if
end for
return false

g) defined but unused (R7): Every defined variable should have a subsequent instruction that uses it. Here, variable is an abstract entity that can represent registers, stack locations,

or status flags. For status flags, we only account for instructions that exclusively affect status flags, such as `TEST` and `CMP` in x86. Since we do not perform inter-procedural analysis and analyze the disassembly up to the function containing the variable, an exception to the rule is the variable that stores the return value since return value is assigned in the function but used in parent function. It is acceptable for the return value variable to have no subsequent usage in the function unless in addition to no subsequent usage the return value variable is also reassigned, indicating that the value previously assigned to the variable is discarded without use. Algorithm 3 presents the pseudocode that enforces this rule and the following are the definitions used in the algorithm:

> $dead_return_value_seen$: a boolean value that is set to true if it encounters a return value variable whose assigned value has no subsequent usage, otherwise it is set to false.
>
> V: set of variables in SSA form in a basic block. By representing the variable in SSA form, it is implied that the variable is defined once.
>
> A: subset of V where each variable is accessed in another instruction in the same function.
>
> IS_RETURN_VALUE_VARIABLE(x): assume x is a variable in SSA form, return true if x is the return value variable. Otherwise, return false.

V. Evaluation

For our evaluation, we are interested in answering the following research questions:

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?

A. Inserting Opaque Predicates

We evaluate our tool by inserting opaque predicates of different resiliency into a set of programs and measure the accuracy of our tool on detecting the inserted opaque predicates using the performance metric, F1 score. We choose F1 score as our performance metric because this metric takes into account of both correct and incorrect identifications. We evaluate against *trivial*, *weak* and *strong* opaque predicates generated by Tigress [8] used to insert junk bytes into opaque predicates' unreachable basic blocks.

Tigress can insert other types of bogus computation into the unreachable basic blocks and our current method's effectiveness in detecting them is discussed in Section V-D and VI-B. The main limitation with our approach in detecting desynchronizing opaque predicates is when the obfuscation technique junk code insertion is also present. In Section VI-A, we propose an approach that we will explore to mitigate false positive identifications in the presence of junk code.

B. Benchmark Programs

We use the obfuscation benchmark provided by Banescu [2] as our benchmark programs. A subset of the benchmark contains source code files that are randomly generated by Tigress. Randomly generated programs are unrealistic examples, so we do not present results for those programs.

Excluding the source code files that are randomly generated, Banescu's benchmark contains 99 total source code files. Each source code file is injected with 3 opaque predicates of the same resiliency, which gives us a total of 297 opaque predicates to evaluate against for each resiliency. However, the subset of Banescu's benchmark that contains multiple functions sums up to 11 source code files, so we only have 33 *strong* opaque predicates since their constructions require the traversal of multiple functions. For *trivial* and *weak* opaque predicates, we have 297 opaque predicates each. Of the combined 627 *trivial*, *weak*, and *strong* opaque predicates, there are 597 unique junk bytes sequences randomly generated by Tigress.

We use all 109 GNU core utilities' executable binaries compiled with GCC at optimization level O0, O1, O2, and O3 for our ground truth to accurately assess our tool's false positive identifications (**RQ2**). We choose version 8.31 of the GNU core utilities since it is the most recent version at the time of our evaluation. 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 analyzing executable binaries compiled at optimization level O0. This is because unoptimized executable binaries, such as those generated by GCC at optimization level O0, can naturally contain junk code. Section VI-A explains why junk code causes false positive identifications and how we plan to mitigate the problem in the future.

C. Evaluator's Setup

Tigress works by a source-to-source transformation. Here are the relevant command options we use to generate the obfuscated source:

> *InitOpaque Transform* adds the initialization code to generate opaque predicates to specified function.
>
> *AddOpaque Transform* adds opaque predicates to specified function.
>
> *InitOpaqueStructs* select how the invariant expression is generated. For consistency, we set this to *env* so the invariant expression is generated from entropy; this also means that the *InitEntropy Transform* needs to be specified.
>
> *UpdateEntropy Transform* updates the entropy variable the invariant expression relies on in the specified function.
>
> *AddOpaqueKinds* specifies what to add to the unreachable basic blocks. To insert junk bytes, we set *AddOpaqueKinds* to *junk*. The other options beside *junk* are *call*, *fake*, *true*, *bug*, and *question*. Details on the other options are explained in Section V-D.

AddOpaqueCount controls the numbers of opaque predicates to add. We set this to 3.

Seed is the randomization seed.

To generate *trivial* opaque predicates, we set the function for *InitOpaque Transform*, *AddOpaque Transform*, and *InitEntropy Transform* to "main." We do not set the *UpdateEntropy Transform*.

To generate *weak* opaque predicates, we have the same setup as generating *trivial* opaque predicates with the addition of setting the function for *UpdateEntropy Transform* to "main."

To generate *strong* opaque predicates, we have the same setup as generating *weak* opaque predicates with the exception that the function for *AddOpaque Transform* is set to a function other than "main."

We sets the command option *AddOpaqueKinds* to *junk* so Tigress will insert junk bytes into the unreachable basic blocks. By specifying *junk* it does not indicate that each inserted junk bytes sequence will be different. The exact junk bytes sequence inserted by the *junk* option is directly correlated to the value the randomization seed is set to (*Seed* option). We attempt to set the randomization seed to a different value each time we invoke Tigress by setting the randomization seed to Bash's internal pseudo-random number generator, accessed through the magic variable "RANDOM."

D. Other Types of Bogus Computation

Although placing junk bytes in opaque predicates' unreachable basic blocks to cause disassembly desynchronization has been popular since its introduction by Linn *et al.* [22], other bogus computations that lead to code bloat can be placed in the unreachable basic blocks instead. For example, beside inserting junk bytes, Tigress can also insert the following types of bogus computation:

Call: places a function call to a random function existing in executable binary in the unreachable basic block.

Fake: places a function call to a non-existing function in the unreachable basic block.

Bug: places buggified version of its sibling authentic basic block in the unreachable basic block.

Question: places copy of sibling authentic basic block in the unreachable basic block.

Our detection approach is focused on detecting opaque predicates when a junk bytes sequence is inserted into the unreachable basic block. Out of all the other types of bogus computation that can be inserted instead, inserting junk bytes does the most damage to the disassembly since only it can result in both incomplete and inaccurate disassembly. Future work will address detecting other types of bogus computation. Our current performance on detecting opaque predicates that introduced other types of bogus computation is discussed in Section VI-B.

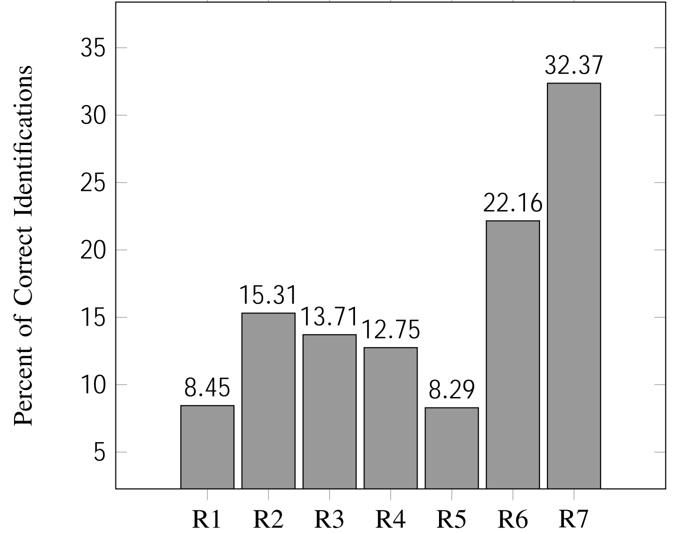

Fig. 5: Detection contribution of each heuristic-based rule on combined *trivial*, *weak*, and *strong* benchmark.

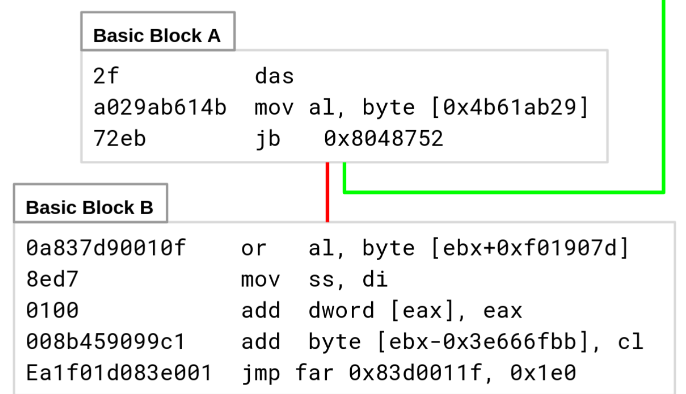

Fig. 6: Junk bytes creating multiple unreachable basic blocks.

E. Results

To answer **RQ1**, we show the performance of our heuristic-based rules on detecting *trivial*, *weak* and *strong* opaque predicates in Table I. In Figure 5, we show the effectiveness of each rule in correctly identifying opaque predicates. The percentage for each rule is obtained by dividing the numbers of correctly identified opaque predicates in respective rule over the total numbers of opaque predicates across the 3 classifications (297+297+33). Note that the percentage is not supposed to add up to 100 percent because multiple rules can detect the same opaque predicate. Also, since rule R7 (IV-A0g) will erroneously identify a basic block containing junk code as damage done by an opaque predicate, Table II displays our performance without rule R7 (IV-A0g).

A symbolic-based detection tool expects to detect 100% of trivial opaque predicates. Although our approach did not achieve detecting 100% of trivial opaque predicates, our approach shows promising and consistent results in detecting

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%

TABLE I: 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%

TABLE II: Accuracy of our tool on detecting *trivial*, *weak*, and *strong* opaque predicates without rule R7 (IV-A0g).

desynchronizing opaque predicates across different levels of resiliency as seen in Table I and II. An immediate improvement on our approach will be to identify a method to determine the basic block containing the start of a junk bytes sequence in the scenario that the junk bytes sequence creates multiple unreachable basic blocks; this will allow us to reduce our numbers of false positive identifications.

From Table I and II, we see that our rules produce non-negligible false positive identifications; this is because the inserted junk bytes can create multiple unreachable basic blocks and our rules detect junk bytes in an unreachable basic block that does not contain the start of the junk bytes sequence. For example, the inserted junk bytes sequence in Figure 6 is '0x2f 0xa0 0x29 0xab 0x61 0x4b 0x72' and it results in 2 unreachable basic blocks because the byte '0x72' happens to be the opcode for the conditional jump instruction `JB`, which leads BinaryNinja to create another unreachable basic block at `JB`'s destination ("Basic Block B"). Our rules detect "Basic Block B" as containing junk bytes, but it does not know that the branch "Basic Block B" originates from belongs to another unreachable basic block ("Basic Block A") and not the offending opaque predicate. The offending opaque predicate's superfluous branch is the branch that "Basic Block A" originates from.

VI. Discussion

A. Junk Code Insertion

Junk code insertion is an obfuscation technique that inserts carefully selected code into the instruction stream such that the inserted code will not affect program functionalities. Junk code insertion's primary purpose is to bloat the disassembly to complicate manual reverse engineering. A simple example is demonstrated below:

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

Here, the first instruction is junk code since register `EAX` is subsequently overwritten. One of our heuristic-based rules, rule R7 (IV-A0g), will erroneously identify a basic block containing junk code as the unreachable basic block resulting from an opaque predicate. For example, in the above code the first `MOV` instruction will trigger rule R7 (IV-A0g) since the value assigned to register `EAX` is discarded without usage.

In future works, we plan to identify approaches that can differentiate between junk code inserted by junk code insertion and the junk code that is introduced from an opaque predicate's superfluous branch. One approach we will experiment with is to take into account of other instructions that exist in the basic block containing the junk code rule R7 (IV-A0g) identified. If we can confidently determine that the other instructions in the same basic block as the junk code are authentic, we conclude that the basic block is not introduced by an opaque predicate's superfluous branch.

B. Our Performance on Other Types of Bogus Computation

Of the different types of code bloat-related bogus computation discussed in Section V-D, we should be able to detect bogus computation of type *fake* with rule R1 (IV-A0a). Theoretically, we also should be able to detect bogus computation of type *call* with rule R7 (IV-A0g) because the return value from the function call chosen by *call* is unlikely to be used by any subsequent authentic basic block. However, generally function that returns a value, as identified from its disassembly, does not always have its return value used in the caller function. As a result, we do not analyze the dataflow of a function's return value. Although our current approach will not be able to detect bogus computation of type *bug* or *question* since the bogus computation generated from those two types do not exhibit illogical behaviors, we can detect them by extending our approach with the approach taken by DoSE [38]. DoSE detects basic block clones in two-way opaque predicates ($P_p^?$ where predicate P at point p in the code can be evaluated to either true or false) by verifying sibling basic blocks' equivalence with a constraint solver. We can extend DoSE's approach to also detect invariant opaque predicates where the injected instructions sequence in the unreachable basic block is semantically equivalent to the authentic instructions sequence in its sibling basic block.

VII. Previous Works

Current approaches to identifying opaque predicates includes fuzzing [23], statistical analysis [10], abstract interpretation [11], value-set analysis [21], heuristic-based analysis [14], [40], machine learning [37], and dynamic symbolic execution [25], [32], [3] or other theorem proving-based approaches [31], [15], but none of the current approaches is sufficient in identifying all opaque predicate constructions effectively [36], [10], [6], [43], [18].

IDA Pro's approach pattern match against desynchronizing opaque predicate's damage where basic blocks originating from the same conditional branch overlap, but IDA Pro assumes junk bytes to always be inserted at false branch's target basic block. This assumption by IDA Pro has shown to have detrimental effect as it can be used against IDA Pro to instead create stealthy and incomplete disassembly [40]. Gabriel's approach [14] pattern match against algebraic-based opaque predicates utilized in OLLVM [18]; this tool-specific approach does not scale as it can only deobfuscate OLLVM-protected code.

Previous works also attempt to generically detect opaque predicates. Madou's [23] and Collberg's [10] methods of dynamically determining invariant expressions in conditional branches err on the side of high false negative rate [33], but statically determining invariant expressions in the general case is undecidable. As a result, previous fully static approaches to opaque predicate detection have to sacrifice completeness for practicality, resulting in those approaches to detecting a specific subset of opaque predicates instead. For example, Preda's implementation of abstract interpretation [11] only detects *trivial* algebraic-based opaque predicates. OpaquePredicatePatcher, a Binary Ninja plugin, uses BinaryNinja's value-set analysis to detect opaque predicates by identifying if a register used to determine a conditional expression contains constant value, but BinaryNinja's value-set analysis does not perform analysis on loops, writable segments, unmodeled data source (e.g. system call's return value), and unlifted instructions [21]. Therefore, OpaquePredicatePatcher will not detect an opaque predicate whose construction involves BinaryNinja's limitations. Yadegari *et al.* [44] propose a generic approach to deobfuscating any obfuscation technique. Theoretically, Yadegari's approach has the potential to generically detect opaque predicates.

In the past few years, it has been shown that dynamic symbolic execution-based approaches [25], [32], [3] are effective in detecting opaque predicates, but their performances decrease when detecting opaque predicates of different constructions [37]. Moreover, bi-opaque predicates [43], whose constructions exploit dynamic symbolic execution's limitation as their core constructs, are introduced recently to specifically weaken the performance of dynamic symbolic execution-based deobfuscators. Ultimately, all theorem proving-based approaches [25], [32], [3], [31], [15], whether the approach utilizes dynamic symbolic execution or not, are bottlenecked by current limitations of automated theorem proving [29].

Recently, Tofighi-Shirazi *et al.* show that a decision-tree classification model for machine learning performs effectively to detect opaque predicates across multiple constructions [37]. However, Tofighi-Shirazi's approach will only perform optimally when it is detecting opaque predicates constructed in the same manners as the opaque predicates previously seen in its training data. Any new opaque predicate construction will degrade the detection accuracy. Furthermore, the training data for Tofighi-Shirazi's approach is generated using intra-procedural symbolic execution. In other words, Tofighi-Shirazi's approach will not detect any opaque predicate whose construction crosses multiple functions' boundaries. By the classification discussed in Section II-C, it will not detect opaque predicates of *strong* or *full* resiliency.

VIII. Conclusion

By observing that an invariant expression in a conditional branch is not the only identifier for a desynchronizing opaque predicate and that its superfluous branch can also be used, we propose a detection approach based on identifying a desynchronizing opaque predicate's superfluous branch. Superfluous branches are identified by analyzing the disassembly of each conditional branch's target basic blocks for illogical behaviors as they are the sites where desynchronizing opaque predicates introduce junk bytes. This novel approach allows us to effectively detect opaque predicates that disrupt static disassembly regardless of how they are constructed — from constructions at the basic block level to constructions at the inter-process level — since determining invariant expressions is not required for our approach. We build an implementation of our approach as a plugin to the BinaryNinja disassembler. Finally, we provide an encouraging preliminary results by displaying promising and consistent detection accuracy across desynchronizing opaque predicates of varied resiliency.

Acknowledgment

This research was supported by a generous gift from the Herman P. & Sophia Taubman Foundation.

References

[1] V. 35, "Binaryninja," https://binary.ninja/.

[2] S. Banescu, M. Ochoa, and A. Pretschner, "A framework for measuring software obfuscation resilience against automated attacks," in *Proceedings of the 1st International Workshop on Software Protection*. IEEE Press, 2015, pp. 45–51.

[3] 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.

[4] B. Bellay and H. Gall, "A comparison of four reverse engineering tools," in *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE, 1997, pp. 2–11.

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[6] C.-F. Chen, T. Petsios, M. Pomonis, and A. Tang, "Confuse: Llvm-based code obfuscation," 2013.

[7] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[8] C. Collberg, "The tigress c diversifier/obfuscator," *Retrieved August*, vol. 14, p. 2015, 2015.

[9] C. Collberg, G. Myles, and A. Huntwork, "Sandmark-a tool for software protection research," *IEEE security & privacy*, vol. 1, no. 4, pp. 40–49, 2003.

[10] 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.

[11] 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.

[12] A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 212–217.

[13] C. Eagle, *The IDA pro book*. No Starch Press, 2011.

[14] F. Gabriel, "Deobfuscation: recovering an ollvm-protected program," *QuarkLabs*, vol. 4, p. 12, 2014.

[15] P. Garba and M. Favaro, "Saturn-software deobfuscation framework based on llvm," in *Proceedings of the 3rd ACM Workshop on Software Protection*. ACM, 2019, pp. 27–38.

[16] Hex-Rays, "Ida pro," https://www.hex-rays.com/products/ida/.

[17] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2r2: Building an efficient front-end for binary analysis," in *The NDSS Workshop on Binary Analysis Research*. Internet Society, 2019.

[18] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 3–9.

[19] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 353–364.

[20] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

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

[22] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.

[23] M. Madou, "Application security through program obfuscation," 2006.

[24] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de) obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2006, pp. 140–144.

[25] 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.

[26] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.

[27] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 47–60.

[28] H. A. Müller and H. M. Kienle, "A small primer on software reverse engineering," *University of Victoria, Tech. Rep*, 2009.

[29] R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Challenges in satisfiability modulo theories," in *International Conference on Rewriting Techniques and Applications*. Springer, 2007, pp. 2–18.

[30] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[31] R. K. R. Prakash, P. Amritha, and M. Sethumadhavan, "Opaque predicate detection by static analysis of binary executables," in *International Symposium on Security in Computing and Communication*. Springer, 2017, pp. 250–258.

[32] T. Rinsma, "Seeing through obfuscation: interactive detection and removal of opaque predicates," Ph.D. dissertation, Radboud University, 2017.

[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[34] R. Singh, "A review of reverse engineering theories and tools," *International Journal of Engineering Science Invention*, vol. 2, no. 1, pp. 35–38, 2013.

[35] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

[36] C. Thomborson, C. Collberg, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[37] 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.

[38] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent, and T.-H. Le, "Dose: Deobfuscation based on semantic equivalence," in *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*. ACM, 2018, p. 1.

[39] C. Treude, F. Figueira Filho, M.-A. Storey, and M. Salois, "An exploratory study of software reverse engineering in a security context," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 184–188.

[40] Y.-J. Tung. (2018) The return of disassembly desynchronization. [Online]. Available: https://github.com/yellowbyte/analysis-of-anti-analysis/blob/develop/research/the-return-of-disassembly-desynchronization/the-return-of-disassembly-desynchronization.md

[41] A. von Mayrhauser and A. M. Vans, "From code understanding needs to reverse engineering tool capabilities," in *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*. IEEE, 1993, pp. 230–239.

[42] B. Wanswett and H. K. Kalita, "The threat of obfuscated zero day polymorphic malwares: An analysis," in *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2015, pp. 1188–1193.

[43] H. Xu, Y. Zhou, Y. Kang, F. Tu, and M. Lyu, "Manufacturing resilient bi-opaque predicates against symbolic execution," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 666–677.

[44] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

[45] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.