On Manufacturing Resilient Opaque Constructs Against Static Analysis

Brendan Sheridan and Micah Sherr

Georgetown University, Washington DC 20057, USA

Abstract. Opaque constructs have developed into a commonly used primitive in obfuscation, watermarking, and tamper-proofing schemes. However, most prior work has based the resilience of these primitives on a poorly defined reduction to a known $\mathcal{NP}$-complete problem. There has been little scrutiny of the adversarial model and little discussion of how to generate instances that are always hard. In this paper, we offer what we believe to be the first complete algorithm for generating *resilient* opaque constructs against static analysis. We base their resilience on the complexity of 3SAT instances with cn clauses for $c = 6$ and n distinct variables. We draw on existing theoretical bounds to show that these instances always require exponential time to defeat under formal notions of resolution complexity.

This paper also explores in-depth the security of opaque constructs in real-world settings. We argue that the common theoretical model used in prior work (as well as our resilient opaque construction scheme) is too optimistic. It does not offer practical obfuscation against an adversary who tolerates some small false positive rate. We offer a heuristic-based attack to demonstrate this issue. Our results suggest that opaque constructs should be viewed with a high degree of skepticism until they can be proven secure under more useful theoretical models.

1 Introduction

Code obfuscation is the process of transforming source or machine code such that the original functionality is maintained, but is hard to discern from inspection of the transformed code. Traditionally, obfuscation was employed to confuse a human reader with the goal of preventing reverse-engineering or hiding certain functionality. This adversarial setting spawned an ecosystem of sophisticated automated obfuscation, and conversely, increasingly sophisticated de-obfuscation and analysis techniques. Currently, an effective obfuscation scheme must not only make the code unreadable, but also difficult to analyze for both targeted and generalized adversarial analysis.

Opaque predicates [9] were introduced to formalize the notion that an effective obfuscation scheme must be able to conceal at least one bit of information from an adversary. Informally, the runtime value of the opaque predicate should be known a priori by the obfuscater based on asymmetric information involved in its creation, but difficult for an adversary to determine without that same information. This primitive allows an obfuscation scheme to naturally obscure the control flow of a program by simply inserting opaque predicates into conditional branch tests. Since the obfuscater can predict the runtime value of the predicate, they can structure the program branching accordingly whereas an adversary must consider both possible values and their associated control

flow. This same primitive has similarly been used in software watermarking [20], the embedding of information that can later be used to identify the original author via public key cryptography, as well as tamper-proofing, the goal of obfuscation such that modifying the functionality is difficult [7].

Unfortunately, while these primitives have seen heavy use, their theoretical basis is very weak. Most commonly, they base their hardness on a reduction to a $\mathcal{NP}$-complete problem. This is an unfortunately common fallacy because these problems are only known to be hard in the worst case. Moreover, since these problems are being artificially generated, there is no inherent guarantee that any of them will be hard in practice. There is an extensive line of work originating from the study of satisfiability problems in artificial intelligence (AI) which suggests that natural choices for generation algorithms actually produce instances that can be solved in polynomial time on average [24,10,5]. However, this line of work has also established that it is possible to construct instances that are always hard with careful parameter selection. We seek to formally apply this line of work to the obfuscation context in order to strengthen the theoretical basis of opaque constructs.

Using opaque predicates to construct an actual obfuscation scheme is largely beyond the scope of this work. To simplify discussion and establish context, we primarily focus on formalizing and extending the work of Moser et al. [19] on obfuscation in the context of static analysis. The authors offer an impressive engineering contribution, fully implementing their x86 binary rewriting scheme and defeating state-of-the-art semantics-aware malware detectors with reasonable overhead. However, they claim their scheme is provably hard to analyze for any static code analyzer based only on an informal reduction from their obfuscation primitive to a 3-satisfiability problem (3SAT) [17]. We believe this assertion to be accurate, but seek to formalize it by more narrowly defining static analysis, giving an algorithm for picking appropriate 3SAT instances, and explicitly proving the reduction as well as the original theorem. In the Appendix, we also examine alternative problems on which to base the primitive, but it is currently unclear if any candidate is a fundamentally better choice than 3SAT due to open problems in cryptography and complexity.

While our main focus is the theoretical strength of the obfuscation primitive, we also offer extensions to the overall obfuscation scheme. Notably, we will show how the ability to obfuscate a single bit, consistent with the original obfuscation primitive, can be generalized to securely and efficiently encrypt the data section. This technique systematically defeats problems the original authors encountered when testing against commercial, regular expression based, malware detectors.

As a second major contribution, we offer a critique of this model and explore its efficacy in the real-world. We present a *heuristic* attack against an obfuscation scheme employing our resilient predicates. Analyzing its effectiveness and efficiency, we find that targeted heuristic approaches can defeat the theoretically resilient construct with high probability. Our results reveal weaknesses in the commonly used model for opaque constructs and suggest the need for increased skepticism over the use of opaque constructs in obfuscation schemes.

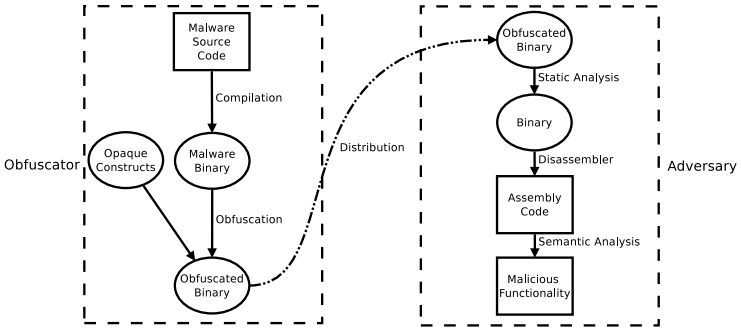

Fig. 1. Malware detection use-case where the obfuscator seeks to hide the malicious nature of the program using opaque constructs and the static adversary seeks to find malicious functionality.

2 Problem and Definitions

Code obfuscation is applicable in a wide variety of contexts. To simplify discussion, we adopt the use-case of Moser et al. [19]. The authors were interested in exploring the limits of static analysis for malware detection, where malware is simply defined as malicious code. The obfuscator is given a piece of (presumably malicious) machine code and must transform the program using opaque constructs such that its functionality remains the same and its runtime performance is not drastically altered. The transformed code should be difficult to identify as the original code. The static adversary, in turn, seeks to identify the obfuscated code given only the original code. For our purposes, it is unnecessary to restrict the behavior of the adversary. However, most prior work first seeks to remove binary obfuscation from the program to allow disassembly, then uses semantic analysis to identify known malicious functionality. Figure 1 gives a graphical representation of this use-case.

It should be noted that this overall problem more generally falls under black-box obfuscation and the general case is suspected to be impossible for formal hardness guarantees [2]. Our problem is, more simply, to generate resilient opaque primitives. More specifically, it is to generate opaque constructs in polynomial time that can be evaluated in linear time but are resilient with exponential resolution complexity.

Definition 1 *A construct is said be **opaque** at point p in a program if its value q is known at obfuscation time with exponentially high probability (w.h.p). We denote an **opaque predicate** P_p^q and an **opaque variable** V_p^q, and we drop the subscript of both when the execution point is obvious from context.*

This definition slightly weakens the original definition from Collberg et al. [9] for convenience purposes. The original contains no notion of probability. However, we show that the probability of one of our constructs failing is exponentially small with respect to n, so it should not be a concern in practice. We discuss the issue in more detail in Section A. We also discuss how obfuscation schemes should avoid compounding this probability in Section 4. We concern ourselves only with constant opaque constructs,

i.e., those for which q is independent of p. However, prior work has shown how to construct dynamic opaque predicates based on several constant opaque predicates [9,19].

Definition 2 *Given a program, P, an obfuscation transformation, $\mathcal{T}$, and a positive scalar-valued complexity measuring function, E, the* ***potency*** *of $\mathcal{T}$ measures the complexity increase in the result program, P'. Formally, the potency of $\mathcal{T}$ on the program P, $\mathcal{T}_{pot}(P)$, is given by $\mathcal{T}_{pot}(P) \equiv E(P')/E(P) - 1$.*

Abstractly, the potency of an obfuscation transformation measures how complex or unreadable the resulting program is compared to the original. By convention, E increases with the logical complexity of the input program so an obfuscator should seek to maximize potency. However, the concrete definition is dependent on E. It is natural to draw on various software engineering metrics to measure readability and complexity, but they are often context dependent and subjective, with no de facto standard. Collberg et al. [8] give a taxonomy of potential choices.

Fortunately, since we are primarily interested in the obfuscation primitive itself rather than any motivating obfuscation scheme, it suffices to assume that the inclusion of the primitive increases the complexity of the resulting program, $E(P')$. For ease of discussion, we assume that the primitive is used as a branch condition and the goal of the deobfuscator is to remove unreachable branches.

Definition 3 *The* ***resilience*** *of an opaque predicate is formally the time and space required by an automatic deobfuscater to effectively reduce the potency of an obfuscation transformation, $\mathcal{T}$.*

To distinguish our model from the more relaxed approximation-based methods for static analysis commonly employed in software verification research, we offer a more restricted definition of static analysis. However, we note that prior research in this context makes no such distinction.

Definition 4 *We define* ***complete static analysis*** *as any algorithm which takes program code as input and enumerates all possible execution paths of the given code unless it can prove via resolution strategy that a branch will never be taken.*

This definition is consistent with the traditional notion of a complete static analyzer, that is, one which returns no false-positives. We have only augmented it to include a notion of computational complexity that is dependent on resolution complexity. This is, to our knowledge, the most appropriate and formal notion of complexity available.

One might argue that a more appropriate goal is resilience against a **sound static analyzer**, that is, one which sacrifices false-positives but does not allow false-negatives (it is easy to see that soundness and completeness are competing goals since an analysis that is both sound and complete would cover the halting problem). In fact, we argue in Section 5 that the requirement of complete static analysis is unrealistic.

Unfortunately, it is difficult to formalize the notion of sound static analysis in this context. Any such notion would likely also be contingent on stealth (since detecting obfuscation is reasonable grounds to flag a program), and we argue informally that stealth is most likely unachievable.

Importantly, our definition is consistent with the prior work in this context, which assumes the analyzer must cover all possible execution paths unless it can prove a path will never be taken [4,19].

Definition 5 *A **resolution strategy** is any algorithm that proves unsatisfiability via resolution reduction. We call the size of the resulting proof the **resolution complexity**.*

Our assumption that resolution complexity is representative of computational complexity is common in AI research and consistent with prior work on random 3SAT hardness [5,10,24]. We are unaware of any more formal lower-bounds for the complexity of artificially generated $\mathcal{NP}$-complete instances.

Definition 6 *3-satifiability or **3SAT** is the boolean satisfiability problem, given a boolean expression in the form $\bigwedge_{i=1}^{m}(X_{i,1} \bigvee X_{i,2} \bigvee X_{i,3})$ and assignment of n boolean variables and their negation to the m clauses, $X_{i,j} \in \{x_1, x_2, ..., x_n, \overline{x_1}, \overline{x_2}, ..., \overline{x_n}\}$, of determining if the expression is satisfiable for some truth value assignment of each variable. It is known to be $\mathcal{NP}$-complete in the number of the variables [17]. By convention, we say that $m = c \times n$ so we can refer to the ratio of clauses to variables, or **density**, as c.*

For simplicity in our analysis, we also adopt the notion of a random 3SAT problem consistent with the works by Kamath et al. [16] and Chvátal and Szemerédi [5].

Definition 7 *A **random 3SAT** problem is defined by the random distribution used to create it. Each clause is chosen independently by choosing three distinct variables uniformly at random and independently negating each with probability $1/2$; the variables are then assigned to $X_{i,1}, X_{i,2}$, and $X_{i,3}$ accordingly.*

3 Generating Opaque Constructs

This section gives our first main contribution: algorithms for constructing theoretically strong opaque constructs and proofs of their resilience against complete static analysis. To start, we must first outline the intended usage of an opaque predicate instance. Algorithm 1 shows the intended inclusion of an opaque predicate in an obfuscation scheme. The 3SAT instance is naturally encoded into a boolean statement that is evaluated at runtime. Each variable in the statement is set randomly at runtime during the call to evaluate. For an opaque predicate, P^f, the obfuscater knows w.h.p. that this branch will never be followed. However, the adversary cannot discount the possibility without proving the statement's unsatisfiability. This formulation is taken directly from Moser et al. [19].

Lemma 1. *The runtime evaluation indicated by Algorithm 1 takes $O(n)$ time given a predicate with $O(n)$ clauses.*

Proof. The algorithm performs two loops with a constant number of operations in each body. This first is over n explicitly and the second is over the number of clauses which we have specified is $O(n)$. The entire algorithm therefore runs in $O(n)$.

Algorithm 1: Runtime evaluation of opaque predicate

```
/* index variables and their inverses                    */
S[] ← [x_1, x_2, ..., x_n, x̄_1, x̄_2, ..., x̄_n]
/* input X_{i,j}: constructed predicate                  */
satisfied ← evaluate(S, X_{i,j})
...
if (satisfied) then
  branch

Function evaluate(S, X_{i,j})
    Input: S− index variables
    Input: X_{i,j}− opaque predicate
    Output: satisfied− boolean indicating if predicate was satisfied
    /* assign boolean values randomly                    */
    for i in 0..n − 1 do
        S[i] ← rand(true, false)
        S[n + i] ← ¬V[i]
    /* record satisfaction in predicate                  */
    satisfied ← true
    for i in 0..X_{i,j}-1 do
        if ¬S[X_{i,1}] ∧ ¬S[X_{i,2}] ∧ ¬S[X_{i,3}] then
            satisfied ← false
    return satisfied
```

Algorithm 2 gives our method for producing opaque predicates. Special attention should be paid to the choice of density $c = 6$. Maintaining the appropriate ratio of clauses to variables allows us to directly apply known resolution complexity bounds. Moser et al. did not explicitly specify their algorithm for predicate generation, but we can reasonably assume they did not maintain this ratio because they discuss changing the number of clauses in their performance section without any mention of the number of variables.

We show that our formulation is efficient, correct, and resilient.

Lemma 2. *Algorithm 2 generates a valid opaque predicate, P^f, and runs in $O(n)$.*

Proof. The algorithm enumerates each clause in the expression and explicitly constructs it based on Definition 7. We can therefore say that it is a valid random 3SAT instance by construction. A valid opaque predicate P^f must also evaluate to false w.h.p. By construction, every set of variable assignments has uniform probability of satisfying a random 3SAT instance. Therefore, w.l.o.g. consider the specific assignment of true to each variable, $\{x_1 = ... = x_n = true\}$. The probability of a randomly chosen clause being satisfied by this assignment is the probability that at least one of the chosen variables is not negated, $1 - (1/2)^3 = 7/8$. Since all clauses are constructed independently, the probability of all clauses being satisfied is thus $(7/8)^{cn}$. Given our choice of $c = 6$, $(7/8)^{6n} \approx (1/2)^{1.16n}$ for any positive integer n, so the probability of the predicate being satisfied at runtime clearly grows exponentially small with respect to n.

Algorithm 2: 3SAT based method for generating strong opaque predicates

```
Function generate_predicate(n)
    Input: n— number of 3SAT variables
    Output: X_{i,j}— 2D array of variable assignments for each variable(j) in each clasue(i)
    /* maintain minimum clause/variable ratio                    */
    num_clauses ← n × 6
    /* randomly set each clause                                  */
    for i in 0..num_clauses-1 do
        /* chose boolean indices at random                       */
        choose x_1, x_2, and x_3 from 0..n-1
        /* negate each with Pr=1/2                               */
        for j in 1..3 do
            if rand(true, false) then
                X_{i,j} ← x_i
            else
                X_{i,j} ← x_i + n
    return X_{i,j}
```

Since our algorithm merely enumerates each clause, making a constant number of random decisions for each, we can conclude that it runs in $O(m)$. We have explicitly set $m = 6n$ so it runs in $O(m) = O(n)$.

Lemma 3. *Any complete static analysis of the opaque predicate controlled branch from Algorithm 1 must consider both execution paths unless it can prove that the boolean statement is unsatisfiable and will never be followed.*

Proof. This follows directly from our definition of complete static analysis. We require that a complete static analysis algorithm consider all possible execution paths. Unless the adversary can prove that the boolean statement will never evaluate to false, it must consider the possibility that the branch will be followed and include it in their analysis.

Lemma 4. *Opaque predicates generated by Algorithm 2 are resilient with exponential resolution complexity.*

Proof. From Definition 2, resilience is the time and space required to remove a predicate from the static analysis. Lemma 3 states that a branch cannot be eliminated without proving that the opaque predicate always evaluates to false. Therefore the problem of reducing branching complexity is equivalent to proving the unsatisfiability of our opaque predicate construction and the associated random 3SAT instance with $c > 6$. Here, we can draw on a lower bound from Chvátal and Szemerédi [5]. The authors proved that, for every choice of positive integers c and k such that $k \geq 3$ and $c2^{-k} \geq 0.7$, the unsatisfiability resolution proof for a randomly chosen family of cn clauses of size k over n variables generates at least $(1+\varepsilon)^n$ clauses. Since we are working with 3SAT, $k = 3$ and this theorem applies for $c \geq 5.6$. We have deliberately chosen $c = 6$ corresponding to $\lceil 5.6 \rceil$ so that this result can be applied directly. Because any resolution proof of our

predicates requires at least exponential space, we can say that they are resilient with exponential resolution complexity.

Next, we show how to use our opaque predicates to trivially generate opaque variables of arbitrary constant length and a given value, q. Note that q is not somehow encoded in the variable itself, but rather passed into the runtime evaluation. This may seem counterintuitive because our overall goal is to hide information from the adversary and the information is clearly human readable in this form. However, preventing things like human readability is the responsibility of the overall obfuscation scheme. Here, it suffices to provide the promised resilience against complete static analysis and trust that the calling obfuscation scheme uses the primitive appropriately. This is consistent with our abstract interpretation of potency from Section 2. We do show that these variables exhibit the same resilience and correctness guarantees as our predicates.

Algorithm 3: Generating opaque variables from opaque predicates

```
Function generate_variable(n, l)
    Input: l− desired bit-length
    Input: n− number of 3SAT variables
    Output: V− array of opaque predicates
    /* generate predicate for each bit                    */
    for i in 0..l-1 do
        V[i] ← generate_predicate(n)
    return V

Function evaluate_variable(q, l, V)
    Input: q− desired variable value
    Input: l− bit-length of variable
    Input: V− opaque variable
    Output: Q− evaluated value of variable
    /* bool array to represent bits of q                  */
    Q[]
    for i in 0..l-1 do
        Q[i] ←evaluate_predicate(V[i]) ⊕ q[i]
    return Q
```

Lemma 5. *Algorithm 3 generates valid opaque variables, V^q and runs in $O(n)$ given a constant bit-length l.*

Proof. Each bit of q is simply xor'ed with the evaluation of an opaque predicate; thus, it suffices to show that none of these predicates are satisfied w.h.p. Lemma 2 gives us that the probability of any single predicate being satisfied by a random assignment is $(7/8)^{6n}$. Therefore the probability that any of these predicates are satisfied is $\sum_{i=1}^{l} \left(\frac{7}{8}\right)^{6n} \approx l \cdot 2^{-1.16n}$, which clearly still grows exponentially small w.r.t. n.

The methods simply perform l different instances of generate_predicate and evaluate_predicate respectively. Lemmas 1 and 2 state that these both run in $O(n)$. We have required that l is constant so both methods run in $O(n)$.

Lemma 6. *Opaque variables generated by Algorithm 3 are resilient with exponential resolution complexity.*

Proof. Here, our measure of potency is the possible 2^l possible values of V^q that a static adversary must consider. Clearly, since each bit of V can take the value 0 or 1 based on the result of the xor with an opaque predicate, the adversary cannot reduce the potency of the variable without defeating one of the opaque predicates. We have from Lemma 4 that each predicate is resilient with exponential resolution complexity so we can say that these opaque variables also have exponential resolution complexity.

4 Obfuscation Scheme Extensions

We next offer a simple extension to the original obfuscation scheme given by Moser et al. [19]. We also discuss how one might intelligently scale the number of variables, n, and the density, c, based on the desired properties of the overall obfuscation scheme.

4.1 Encrypting data against complete static analysis

One notable shortcoming of the original obfuscation scheme is that it is potentially vulnerable to the simple data section pattern matching used by commercial virus scanners. Opaque variables cannot be readily used to hide data patterns from a heuristic-based adversary and the linear space increase associated with using an opaque variable for the entire data section is unappealing in practice.

Moser et al. contend that this is a non-issue because an obfuscater can simply encrypt the data section using a unique key stored in the binary, unpacking the data accordingly at runtime. We argue that this is inconsistent with their goal of defeating static analysis. Given a secret key naïvely stored in the binary as well as a known or unobfuscated encryption algorithm, even a static adversary can simply decrypt the data section before applying a pattern matching strategy. However, to defeat a static adversary, it suffices to hide the secret key with an opaque variable. Figure 4 gives a straightforward key generation algorithm based on this intuition. The key can be used in any stateless symmetric encryption scheme, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, such as CTR-C with AES [14]. The actual cryptography is interchangeable and the resilience of the scheme derives simply from the resilience of our opaque variables shown in Lemma 6.

Algorithm 4: $\mathcal{K}-$ Opaque key generation

Function *generate_key*(n)
- **Input**: $n-$ number of 3SAT variables
- **Output**: $K-$ opaque cryptographic key
- $K \leftarrow_{\$} \mathbb{Z}_p$
- $V \leftarrow$ generate_variable$(n, |K|)$
- $K \leftarrow$ evaluate_variable$(K, |K|, V)$
- **return** K

Lemma 7. *Algorithm 4 takes $O(n)$ time to generate a constant-length opaque key from a cryptographic key of the same length. The resulting key has exponential resolution complexity.*

Proof. We assume here that an appropriate constant-length key is supplied or is trivial to select. We have from Lemma 5 that generate_variable and evaluate_variable both run in $O(n)$ given a constant bit length variable. generate_key simply applies both to the constant length cryptographic key so it must also run in $O(n)$.

We also have from Lemma 6 that the resulting variable (in this case the key) is a valid opaque construct with exponential resolution complexity.

4.2 Choosing opaque construct parameters

There are several tradeoffs arising from our opaque construct generation algorithms that can be controlled via the number of 3SAT variables, n, and the ratio of clauses to variables, c.

First, we can exponentially increase the resilience of our constructs by increasing n. This comes at a linear $O(n)$ cost in the size of the transformed code, the runtime of the generation algorithm, and the runtime of the opaque evaluation.

The second is a very subtle property of random 3SAT refutation. Although we are guaranteed exponential resolution complexity for any positive integer $c \geq 5.6$ and increasing c decreases the probability that our construct will ever be incorrectly evaluated, there is a hidden drawback to allowing c to be much larger than 6. In practice, large values of c make the resulting 3SAT instance very over-constrained and easier to resolve. For example, Crawford and Auton [11] showed experimentally that the growth rate of their algorithm was approximately $2^{n/17}$ for $c = 4.3$ compared to only $2^{n/57}$ for $c = 10$. Selman et al. [24] later showed this is due to a monotonically decreasing behavior in the search space above the critical point of roughly $c = 4.3$. As such, we feel it is wise to choose c as close to the theoretically proven lower bound as possible.

4.3 Compounding effects

There is an error probability compounding effect resulting from the use of multiple opaque constructs. Since an error in any single construct affects the overall correctness of the transformed code, one should consider the probability of any single construct failing when bounding the error probability of their obfuscation scheme. Fortunately, this probability can be calculated explicitly as in Lemma 5 and the scaling is very favorable. It suffices to choose n sufficiently large to exercise the exponential scaling.

5 Heuristic Attacks

Given the strong formal guarantees described in the previous sections, we next take a different tack and explore the efficacy of opaque constructs in practice. That is, we ask a more fundamental question: are these theoretical constructs actually useful?

A major weakness of the formal model considered by Moser et al. [19] and adopted by us in the proceeding sections is that it envisions an unrealistic adversary. Malware

detectors (unlike compilers) are typically uninterested in precisely proving that a transformation is safe. They intentionally tolerate some small false positive rate, sacrificing completeness for soundness.

Such detectors often employ heuristic strategies against which our construction would not be provably resilient. In what follows, we highlight this potential problem by offering an effective attack against the predicates we previously proved were resilient with exponential resolution complexity against complete static analysis.

We start by giving a heuristic based algorithm designed to correctly identify predicates generated by Alg. 2 in polynomial time. We show that, assuming our heuristic can correctly identify instances of random 3SAT, we can identify our generated opaque predicates with perfect recall in polynomial time. We also show that the probability of incorrectly identifying a satisfiable predicate as opaque is bounded by a small constant, making the chance that we change the functionality of the program similarly small.

Algorithm 5 gives our heuristic-based detection strategy. First, the algorithm tests to see if the predicate is controlled by a random 3SAT instance: we naïvely verify that the predicate is a 3SAT instance and we also test whether the observed literals follow the expected uniform distribution; the details of which of are discussed in Section 5.1. If either test fails, we abandon analysis of the predicate since our generation algorithm only produces random 3SAT instances.

If the predicate has been determined to be a random 3SAT instance, the algorithm then tests the estimated value of c to determine if $\tilde{c} > 6$. If so, the tester can be extremely confident that the branch will never be followed because we know that random 3SAT instances with sufficient values of c are unsatisfiable w.h.p. Thus, the branch can be safely removed without altering the program's functionality.

Algorithm 5: Heuristic method for defeating opaque predicates generated by Algorithm 2

```
Function check_predicate(s)
    Input: s—boolean formula controlling branch
    /* check if 3SAT instance                                          */
    for each clause in s do
        if variables ≠ 3 then
            return
    /* check for uniform distribution using χ²-test for uniformity    */
    if χ²_test(s) > uniformity_threshold then
        return
    /* count the unique variables in s                                 */
    ñ ←count(s)
    /* check estimated value of c                                      */
    c̃ ←clauses(s)/ñ
    if c̃ < 6 then
        return
    /* if all tests pass, assume opaque                                */
    remove(branch(s))
    remove(s)
```

Lemma 8. *Algorithm 5 runs in polynomial time.*

Proof. The algorithm simply steps once through a series of tests and each test runs in polynomial time so the algorithm as a whole must also run in polynomial time.

Next, we would like to account for the potential skew in the observed value of $\tilde{n}$, and consequently skew in the estimated value of c, $\tilde{c}$. Since the attack sees only the generated predicate, it can potentially underestimate the actual number of variables in the generating distribution. Formally,

Definition 8 *We consider the result of Algorithm 5 to be a* ***true-positive*** *if the true value of $c = 6$ and the branch and predicate are appropriately removed.*

Given this definition, we can actually show that an opaque predicate will always be correctly identified.

Lemma 9. *The recall of Algorithm 5 is exactly 1, i.e. $TPR = 1$.*

Proof. By Definition 8, a false-negative can only occur when $c = 6$. By construction, the predicate and branch are only removed when $\tilde{c} < 6$. Since, also by construction, $\tilde{c} = cn \,/\, \tilde{n}$, they are removed when $\tilde{n} \leq n$. Clearly, we will never observe more than n variables because there are only n variables in the distribution. Thus, our algorithm can never mistakenly reject an opaque predicate generated by Algorithm 2 provided that it successfully passed the uniformity test.

We would also like to show that the possibility of incorrectly identifying a satisfiable predicate as opaque is appropriately small. To do so, we use the Satisfiability Threshold Conjecture for random 3SAT. It states that there exists a single density, c, such that generated instances with density $\leq c$ are satisfiable whereas generated instances with density $> c$ are unsatisfiable w.h.p. The best known upper bound for this conjecture is $c = 4.51$ [3] so we will conservatively consider the probability that a given predicate with $c = 4.51$ is determined to be opaque and incorrectly removed. We do not distinguish between our generated opaque predicates and predicates that coincidentally have $c > 4.51$ because, regardless, it is safe to remove a predicate and branch that will never be satisfied.

Lemma 10. *The probability that Algorithm 5 removes a satisfiable branch is guaranteed to be small, i.e., $P\{c \leq 4.51\} < 5.33 \times 10^{-6}$.*

Proof. From Algorithm 5, we remove a predicate when the estimated value of c, $\tilde{c} \geq 6$. We have assumed that $c = 4.51$ and $\tilde{c}$ is calculated with $\tilde{c} = cn \,/\, \tilde{n}$ so we would like to bound the probability that $\tilde{n} \,/\, n \leq 4.51 \,/\, 6$. Let Y be an indicator variable for the absence of the i-th variable from the generating distribution in the predicate. We can say that $\tilde{n} = n - \sum Y$, so we would like to bound the probability that $\sum Y \geq .25n$. Applying Markov's inequality gives us a very loose but sufficient bound, i.e.

$$Y_i = \begin{cases} 1 & \text{if } x_i \text{ does not appear} \\ 0 & \text{otherwise} \end{cases} \implies P\left\{\sum Y \geq .25n\right\} \leq \frac{E(\sum Y)}{.25n}$$

and we can calculate $E\left(\sum Y\right)$ directly by linearity of expectation. The probability that a particular variable, x_i, is left out of every clause is

$$P\{Y_i = 1\} = \left(\frac{n-3}{n}\right)^{cn} \implies E\left(\sum Y\right) = n\left(\frac{n-3}{n}\right)^{cn}$$

Substituting in, we are left with a monotonically increasing function for $n \geq 3$,

$$P\left\{\sum Y \geq .25n\right\} \leq 4\left(\frac{n-3}{n}\right)^{4.51n} < 5.33 \times 10^{-6}$$

5.1 Distribution testing

The testing of the distribution is, theoretically, the weak-point of this detection strategy. However, we will argue that it is reasonable to assume its effectiveness in practice. We first address the non-adversarial setting. That is, we assume all input boolean formulas are generated via some random distribution and we need only distinguish between uniform and non-uniform distributions. This is a standard use-case for a χ^2-test and fairly trivial given a sufficiently large sample.

In Figure 2, we compare average p-values as a function of n to truncated normal distributions with increasing variance. Although there is some skew due to the sampling without replacement in random 3SAT generation, we see, as expected, that there is a very drastic fall-off in all the non-uniform distributions.

To ensure a sufficiently large sample, it suffices to remove 3SAT instances that can be solved in small constant time. For example, we found experimentally that, given $c = 6$, we could consistently solve instances with $n < 58$ in under a minute using eight processors and a simple 3SAT solver. In contrast, all the distributions we tested had already suitably diverged by $n = 10$.

One could make this classification problem arbitrarily hard by using a generating distribution that more closely approximates uniformity. However, the resulting misclassified 3SAT instances effectively approximate the frequency distribution of a random 3SAT instance. Since proofs for the unsatisfiable threshold primarily rely on the frequency distribution, they should be fairly insensitive to such instances. We leave formally bounding this insensitivity as a future research direction.

5.2 Potential defenses

Of course, the assumption that all input instances are randomly generated is arguably optimistic. A simple defense against this heuristic attack might entail inserting artificially generated formulas that resemble random 3SAT but are known to be satisfiable. In the AI context, this could be considered the problem of generating hard satisfiable instances in the over-constrained region. Our context adds the additional constraint that these instances cannot be too tightly clustered (otherwise they can be identified and removed heuristically). We argue that there is no such known method.[1]

[1] Moser et al. [19] mention this strategy as part of a proposed heuristic-based defense, but they offered no specific construction scheme so we cannot make a direct comparison.

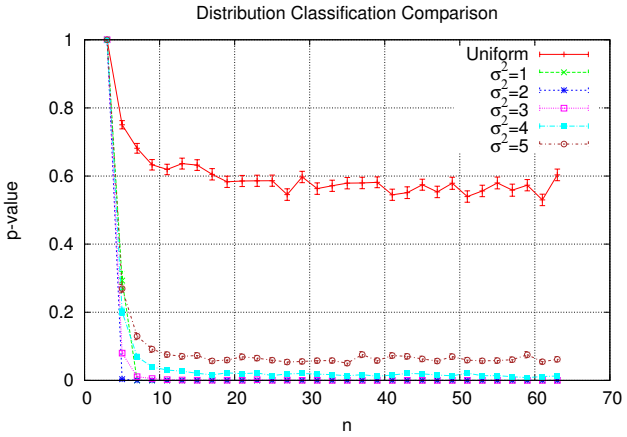

Fig. 2. Mean p-values for uniform distribution and truncated normal distributions with different variance values as a function of n, c=6. Error bars show standard error of the sample size r, std-dev/$\sqrt{r}$.

Finding satisfiable random 3SAT instances in the over-constrained region by brute-force is intractable since candidates are unsatisfiable with exponentially high probability. As such, it is typical to start with some truth assignment and select only clauses which satisfy the truth assignment. The resulting instances are called **planted 3SAT**.

Unfortunately, this skews the distribution of variables and their negation (since the form that agrees with the truth assignment is more likely to appear in viable clauses). Even in the general form, where the frequency distribution used to select clauses is allowed to vary arbitrarily, there are known algorithms for solving planted 3SAT in polynomial time w.h.p [6]. Moreover, experimental results show that all satisfiable instances in the over-constrained region are tightly clustered, suggesting that they could be easily solved even if they were not generated with a planted assignment.

Since there are no known algorithms that meet the resulting satisfiable 3SAT generation criteria and any such algorithm would represent a considerable breakthrough in AI community, we conjecture that any such defense against a targeted heuristic attack is infeasible.

5.3 Discussion

In summary, Algorithm 5 defeats our theoretically strong opaque construct generation algorithm by iteratively and w.h.p. removing unsatisfiable branches, effectively "unraveling" its opaqueness.

We suspect the effectiveness of our heuristic attack against a provably resilient opaque construct is indicative of inaccuracies in the commonly used assumptions for modeling static analysis. Specifically, the assumption that the static analyzer must prove unsatisfiability of a predicate in order to remove it is too strong. In practice, it is sufficient, and computationally much easier, to determine the value of the predicate with

some high probability. This issue seems fundamentally at odds with the correctness of a constant opaque predicate, which calls into question the practical utility of opaque constructs for obfuscation.

We conjecture that all opaque construct generation schemes are vulnerable to similar targeted heuristic attacks because generating instances of a hard problem with known solutions naturally limits the instances to some subset of the problem space with measurable properties. Recent experimental results, which we discuss next in Section 6, seem to support this conjecture.

6 Related Work

The concept of an opaque predicate was first introduced by Collberg et al. [9]. However, in addition to resilience and cost, the authors also design their obfuscation scheme around a poorly defined stealth metric. Consequently, they base their main primitive on the hardness of precise flow-sensitive alias analysis. More specifically, their scheme builds a set of complex dynamic structures with a set of implicit invariants. The invariants are known a priori but difficult to verify statically so they can be tested at runtime as an opaque predicate. Precise flow-sensitive alias analysis is known to be undecidable in general [23], but this formulation is only known to be $\mathcal{NP}$-hard in the worst case [15]. They argue informally that this is not dissimilar to data structures kept by real applications, achieving stealth. However, it is unclear how to scale this scheme since each opaque predicate requires an invariant. They offer a second scheme based on the potential interleaving of parallel regions, but it suffers from the same faults in addition to being architecture specific and potentially indeterminate on a loaded operating system. In contrast, we seek to achieve a scalable scheme that is proven to be $\mathcal{NP}$-complete in the average case. We have also abandoned the stealth metric because we suspect it is unachievable even against static analysis.

Unfortunately, the feasibility of stealth remains an open question, largely dependent on its formalization. Probably the most intuitive definition is that an obfuscated function should behave as a "virtual black box", meaning that an adversary cannot compute anything with the obfuscated function that they could not compute with oracle access to the same function. This definition implicitly includes both resilience and stealth, but was shown to be impossible in general [2]. The result is not necessarily applicable to obfuscation primitives which can be specialized functions lending themselves to obfuscation, but it does imply that there is no clever way to apply said primitives to achieve the virtual black box property for an arbitrary application. This work partially motivates our conservative focus on opaque predicates themselves as well as resilience only against static analysis. However, our work is only tangentially related since we intentionally avoid the impossible general case.

Heuristic-based approaches to stealthy opaque predicates have produced an academic arms race. Most continue to base their resiliency on pointer alias analysis, but offer no formal definition of stealth [22,20,7,13]. These techniques remain vulnerable to targeted detection [12], and were recently shown to be detectable in general with dynamic analysis by Ming et al. [18]. We diverge from this line of work by abandoning

the goal of stealth and focusing on resilience against static analysis, giving us arguably weaker properties, but ones that can be formally proven.

There are several notable exceptions to the trend of using pointer alias analysis as a basis for resilience. Ogiso et al. [21] and later Borello and Mé [4] both base their hardness on the related problem of inter-procedural analysis. Or, more specifically, the problem of determining if there exists an execution path such that a given function pointer points to a given function at a given point of the program. This formulation naturally reduces to 3SAT, making the problem $\mathcal{NP}$-complete and analysis of the entire program $\mathcal{NP}$-hard. Unfortunately, the inter-procedural focus does not lend itself to a scalable self-contained obfuscation primitive, nor is this formulation known to be $\mathcal{NP}$-complete in the average case. Our approach seeks to guarantee both of these properties. It is most similar to the work of Moser et al. [19], who similarly base their hardness on 3SAT but make the encoding explicit and self-contained. We seek mainly to improve on their theoretical contribution by proving that a deliberate 3SAT instance selection algorithm produces opaque predicates that are $\mathcal{NP}$-complete in the average case.

7 Conclusions

Opaque constructs are a commonly employed primitive in obfuscation, watermarking, and tamper-proofing schemes. However, their theoretical basis has historically been very weak. We have proven the resilience and correctness of random 3SAT based opaque constructs under formal notions of resolution complexity and complete static analysis. However, in doing so we have revealed some weaknesses in the commonly used model and potentially opaque constructs as an obfuscation primitive in general. We suggest that future research apply more skepticism to the use of opaque constructs in obfuscation schemes since their theoretical basis remains dubious.

Acknowledgments. We are grateful for the helpful comments and suggestions from the anonymous reviewers. This work is partially funded from National Science Foundation (NSF) grants CNS-1445967, CNS-1527401, and CNS-1149832. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

A Alternative Sources of Hardness

Random 3SAT may seem like a strange source of hardness given that our goal is simply to hide information from the static analyzer. However, the information must be known at runtime and the obfuscator cannot use traditional means to store a key without also making it available to the static analyzer. Therefore, more traditional means of encryption are inapplicable in this setting. Below, we briefly describe the relative merits and drawbacks of some alternate choices:

Integer factorization: As the basis of most modern cryptography, integer factorization was naturally one of our first considerations. Unfortunately, we found no natural way to incorporate the problem into an opaque predicate. We conjecture that trap-door functions in general are unsuitable because the opaque construct still needs to be evaluated at runtime without adding any additional knowledge to the system.

Primality testing: Collberg et al. [9] mention primality testing as a possible basis for opaque predicates The strategy being to pick a prime during obfuscation and have the runtime evaluation try to evenly divide the prime by a random number. Naturally, an adversary cannot guarantee the division will fail without proving that the number is prime. Unfortunately, it has since been proven that primality testing can always be done in polynomial time [1], making it too weak to serve as a hardness basis.

One-way functions: A one-way function is more natural than a trap-door function since we can apply it to a chosen input during obfuscation and compare that result to the result of a random input evaluated at runtime. However, if the generating value is included in the set of possible runtime inputs, there is at least one potential collision. Typically, the resulting correctness bound is weaker than our 3SAT based construction.

Flow-sensitive alias analysis: Alias analysis is the basis primarily employed by Collberg et al. [9]. It has the arguable advantage of naturally resembling normal code. This would make it a better candidate for meeting some formal notion of "stealth". However, since no one has proposed a usable metric of stealth and recent impossibility results suggest it is not obtainable, we do not feel stealth is an appropriate goal. Alias analysis also has the advantage of provable correctness but it comes at the cost of scalability since it's unclear how to generate an arbitrary number of the deliberately crafted invariants used to guarantee correctness.

Race conditions: Another possible basis briefly mentioned by Collberg et al. [8] takes advantage of concurrency and the intractability of precise race detection. Intuitively, an attacker might be able to insert a data race into a concurrent program and be fairly confident of the outcome on a particular platform. Static analysis, in contrast would not be able to reliably find the data race, let alone determine its outcome. This has (often unintentionally) been a source of hardness in reverse engineering programs for the purpose of porting them to a different platform. Unfortunately, this basis would require that the original program be concurrent and might violate correctness on platforms other than the particular one targeted. Even ignoring these problems, scaling would be problematic because its unclear how to reliably generate appropriate data races in general.

Random 3SAT: The main advantage of using random 3SAT for our hardness basis was the large body of existing work from the AI context on satisfiability and provably hard instance generation [3,5,24,10]. Resolution complexity is an arguably weak hardness conjecture because it states only that actually proving satisfiability is hard. As we showed in Section 5, a less restricted adversary can still make a very accurate guess. However, our assumptions were consistent with prior work and we failed to find any stronger hardness conjectures that were applicable in this context.

References

1. M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math*, 2:781–793, 2002.
2. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.
3. A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
4. J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.
5. V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. ACM*, 35(4):759–768, 1988.
6. A. Coja-Oghlan, M. Krivelevich, and D. Vilenchik. Why almost all satisfiable k-cnf formulas are easy. In *2007 Conference on Analysis of Algorithms, AofA 07*, pages 95–108. Discrete Mathematics and Theoretical Computer Science, 2007.
7. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *Transactions on Software Engineering*, 28(8):735–746, 2002.
8. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.
9. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM POPL*. ACM, 1998.
10. S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: A survey. pages 1–17. American Mathematical Society, 1997.
11. J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-sat. *Artificial Intelligence*, 81(1-2):31 – 57, 1996.
12. 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 (AMAST)*, pages 81–95, 2006.
13. S. Darwish, S. Guirguis, and M. Zalat. Stealthy code obfuscation technique for software security. In *International Conference on Computer Engineering and Systems (ICCES)*, pages 93–99, Nov 2010.
14. S. Goldwasser and M. Bellare. Lecture notes on cryptography, 2001.
15. S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, Jan. 1997.
16. A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. In *FOCS*, 1994.
17. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.
18. J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *CCS*, 2015.
19. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference (ACSAC)*, pages 421–430, Dec 2007.
20. G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, Apr. 2006.
21. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):176–186, 2003.
22. M. Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Software Engineering and Formal Methods (SEFM)*, pages 301–310, Sept 2005.
23. G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, Sept. 1994.
24. B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2), 1996.