LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code

Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
{jum310,dux103,lzw158,dwu}@ist.psu.edu

ABSTRACT

Opaque predicates have been widely used to insert superfluous branches for control flow obfuscation. Opaque predicates can be seamlessly applied together with other obfuscation methods such as junk code to turn reverse engineering attempts into arduous work. Previous efforts in detecting opaque predicates are far from mature. They are either ad hoc, designed for a specific problem, or have a considerably high error rate. This paper introduces *LOOP*, a Logic Oriented Opaque Predicate detection tool for obfuscated binary code. Being different from previous work, we do not rely on any heuristics; instead we construct general logical formulas, which represent the intrinsic characteristics of opaque predicates, by symbolic execution along a trace. We then solve these formulas with a constraint solver. The result accurately answers whether the predicate under examination is opaque or not. In addition, LOOP is obfuscation resilient and able to detect previously unknown opaque predicates. We have developed a prototype of LOOP and evaluated it with a range of common utilities and obfuscated malicious programs. Our experimental results demonstrate the efficacy and generality of LOOP. By integrating LOOP with code normalization for matching metamorphic malware variants, we show that LOOP is an appealing complement to existing malware defenses.

1. INTRODUCTION

In general, a predicate is a conditional expression that evaluates to true or false. A predicate is opaque when its result is known to the obfuscator *a priori*, but at runtime it still needs to be evaluated and is difficult to deduce by an adversary *afterwards*. Opaque predicates have been applied extensively in various areas of software security, such as software protection [14, 15], software watermarking [3, 39], software diversification [19, 29], securing mobile agents [35], metamorphism malware [9, 10], and obfuscation of Android applications [28].

Real-world obfuscation tools have already supported embedding opaque predicates into program at link time or binary level [16, 26, 34]. As a result, control flow graph is heavily cluttered with infeasible paths and software complexity increases as well [2]. Unlike other control flow graph obfuscation schemes such as call stack tampering or control flow flattening [46], opaque predicates are more covert as it is hard to distinguish opaque predicates from normal conditions. Furthermore, opaque predicates can be seamlessly woven together with other obfuscation methods, such as opaque constants [37] and metamorphic mutations [44] to further subvert reverse engineering efforts. Such an example can be found in a recent notorious "0 day" exploit (CVE-2012-4681), in which opaque predicates are used together with encrypted code [21]. Therefore, it has become more difficult to locate the exploit code of interest due to the use of opaque predicates.

Depending on the construction cost and resilience to deobfuscation, we classify previous work on opaque predicates into three categories. The first category is *invariant opaque predicates*. Such predicates always evaluate to the same value for all possible inputs. Invariant predicates are mainly constructed from well-known algebraic theorems [3, 39]. For example, predicate $(x^3 - x \equiv 0 \pmod 3)$ is opaquely true for all integers x. Since it is easy to construct invariant opaque predicates they are commonly used. The second category, *contextual opaque predicates*, is built on some program invariant under a specific context. That means only the obfuscator knows such a predicate is true (false) at a particular point, but could be false (true) if the context is not satisfied. The third category, *dynamic opaque predicates*, is the most complicated one. In this category, a set of *correlated* and *adjacent* predicates evaluate to the same value in any given run, but the value might be different in another run. In any case, the program produces the same output. To make matters worse, dynamic opaque predicates can be carefully crafted by utilizing the static intractability property of pointer aliasing [17].

A number of methods have been proposed to identify opaque predicates [17, 33, 42, 43, 45]. Unfortunately, none of them is sufficient to meet our requirements: generality, accuracy, and obfuscation-resilience. They are either heuristics based [17], limited to a specific type of already known opaque predicates [42], unable to work on highly obfuscated binary [45] (e.g, binary packing and virtualization obfuscation), or have a rather high error rate [33]. On the adversary's side, to defeat the pattern matching of commonly used opaque predicates, Arboit [3] introduces a construction method based on quadratic residues, which can be extended to a larger set of new opaque predicates. Furthermore, all existing detection approaches only focus on invariant opaque

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
 ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813617.

predicates. There has been little work on systematically modeling and solving contextual or dynamic opaque predicates.

To bridge the gap stated above, we introduce a novel logic-based, general approach to detecting opaque predicates progressively in obfuscated binary code. We first perform symbolic execution on an execution trace to build path condition formulas, on which we detect invariant opaque predicates (the first category) by verifying tautologies with a constraint solver. In the next step, we identify an implication relationship to detect possible contextual opaque predicates (the second category). Finally, with input generation and semantics-based binary diffing techniques, we further identify correlated predicates to detect dynamic opaque predicates. Our method is based on formal logic that captures the intrinsic semantics of opaque predicates. Hence, LOOP can detect previously unknown opaque predicates. A benefit of LOOP's trace oriented detection is that it is resilient to most of the known attacks that impede static analysis, ranging from indirect jump, pointer alias analysis [8], opaque constants [37], to function obfuscation [36]. Our results can be fed back to security analysts to further de-obfuscate the cluttered control flow graph incurred by opaque predicates.

We have implemented LOOP to automate opaque predicates detection on top of the BAP platform [7] and conducted the evaluation with a set of common utilities and obfuscated malicious programs. The experimental results show that LOOP is effective and general in detecting opaque predicates with zero false negatives. Several optimizations such as taint propagation and "short cut" strategy offer enhanced performance gains. To confirm the merit of our approach, we also test LOOP in the task of code normalization for metamorphic malware [9, 10]. This kind of malware often uses opaque predicates to mutate the code during propagations to evade signature-based malware detection. The result indicates that LOOP can greatly speed up control flow graph matching by a factor of up to 2.0.

In summary, we make the following contributions.

- We study the common limitations of existing work in detecting opaque predicates and propose LOOP, an effective and general approach that identifies opaque predicates in the obfuscated binary code. Our approach captures the intrinsic semantics of opaque predicates with formal logic, so that LOOP can detect previously unknown opaque predicates.
- Our method is based on strong principles of program semantics and logic, and can detect known and unknown, simple invariant, intermediate contextual, and advanced dynamic opaque predicates.
- LOOP is developed based on symbolic execution and theorem proving techniques. Our evaluation shows that our approach automatically diagnoses opaque predicates in an execution trace with zero false negatives.
- To the best of our knowledge, our approach is the first solution towards solving both contextual and dynamic opaque predicates.

The rest of the paper is organized as follows. Section 2 presents the background information about three categories of opaque predicates. Section 3 illustrates our core method with a motivating example. Section 4 describes each step of our approach in detail. Section 5 introduces our implementation. We evaluate our approach in Section 6. Discussions and future work are presented in Section 7. Related work is discussed in Section 8. We conclude the paper in Section 9.

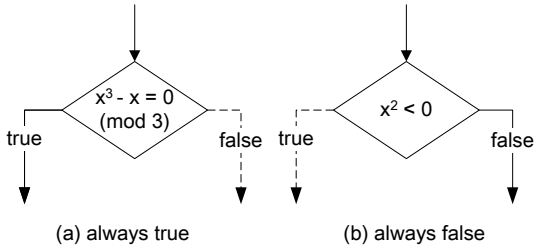

Figure 1: Examples of two invariant opaque predicates for all integers x.

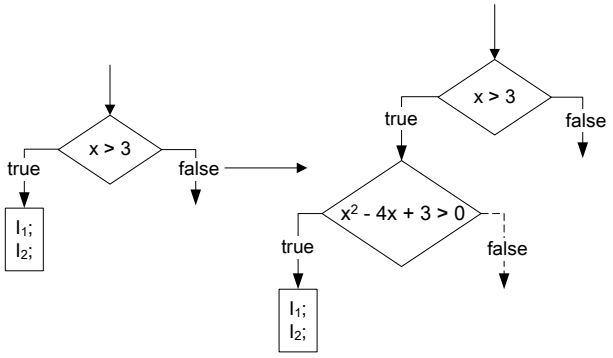

Figure 2: Example of a contextual opaque predicate for all integers satisfying $x > 3$.

2. BACKGROUND

In this section, we introduce the three types of opaque predicates we try to solve:

Invariant Opaque Predicates.

An opaque predicate is invariant when its value always evaluates to true or false for all possible inputs, but only obfuscator knows the value in advance. Figure 1 shows two cases of invariant opaque predicates: always true and always false. The dashed line indicates that the path will never be executed. Due to the simplicity, this kind of opaque predicates have a large set of candidates. Most of them are derived from well-known algebraic theorems [39] or quadratic residues [3]. However, the invariant property also becomes the drawback of this category. For example, we can identify possible invariant opaque predicates by observing the branches that never change at run time with fuzzing testing [33].

Contextual Opaque Predicates.

To avoid an opaque predicate always produces the same value for all inputs, Drape [22] proposes a more covert opaque predicate that is always true (false) under a specific precondition, but could be false (true) when precondition does not hold. We call this kind as contextual opaque predicates, which can be carefully constructed based on program invariants under a particular context. Figure 2 shows an example of contextual opaque predicate, in which $x^2 - 4x + 3 > 0$

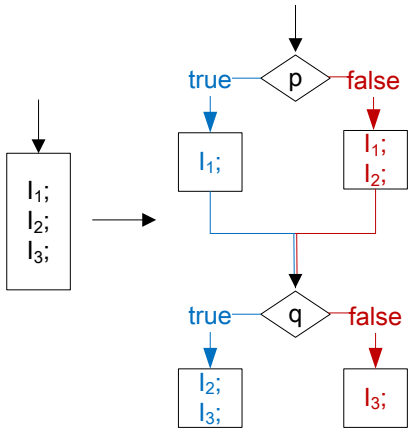

Figure 3: Example of a dynamic opaque predicate.

is always true if the precondition $x > 3$ holds. Note that the constant value in the precondition can be further obfuscated [37] to hide the context relationship.

Dynamic Opaque Predicates.

Palsberg et al. [41] introduce the idea of dynamic opaque predicates, which are a family of *correlated* and *adjacent* predicates that all present the same value in any given run, but the value may vary in another run. That means the values of such opaque predicates switch dynamically. Combined with code clone, dynamic opaque predicates can always produce the same output. The term "correlated" is used to describe that dynamic opaque predicates contain a set of mutually related predicates, and "adjacent" means these opaque predicates execute one after another strictly. Figure 3 illustrates an example of dynamic opaque predicates. Two correlated predicates, p and q, meet the requirement of evaluating to true (false) in any given run. The original three instructions $\{I_1; I_2; I_3;\}$ execute one after another. After transformation, each run either follows the path $p \wedge q$ (blue path) or $\neg p \wedge \neg q$ (red path). In any case, the same instructions will be executed. Look carefully at Figure 3, we can find another common feature. Since predicate q divides both blue path and red path into different segments (i.e., $\{I_1;\}$ vs. $\{I_1; I_2;\}$ and $\{I_2; I_3;\}$ vs. $\{I_3\}$), p and q must be strictly adjacent; or else the transformation is not semantics-persevering. The correlated predicates can be crafted by utilizing pointer aliasing, which is well known for its static intractability property [17].

Existing efforts in identifying opaque predicates mainly focus on invariant opaque predicates and they are unable to detect more covert opaque predicates such as contextual and dynamic opaque predicates. A general and accurate approach to opaque predicate detection is still missing. Our research aims to fill in this gap.

3. OVERVIEW

3.1 Method

The core of our approach is an opaque predicate detector, whose overall detection flow is shown in Figure 4. There are three rounds in our system to detect three kinds of opaque predicates progressively. Here we present an overview of our core method.

```
1 int opaque(int x)
2 {
3    int *p = &x;
4    int *q = &x;
5    int y = 0;
6    if (x*x < 0)        // invariant opaque predicate
7       x = x+1;
8    if (x > 3)
9    {                   // contextual opaque predicate
10      if (x*x-4x+3 > 0)
11      x = x<<1;
12   }
13   if ((*p)%2 == 0) // dynamic opaque predicate
14      y = x+1;
15   else
16   {
17      y = x+1;
18      y = y+2;
19   }
20   if ((*q)%2 == 0)
21   {
22      y = y+2;
23      x = y+3;
24   }
25   else
26      x = y+3;
27   return x;
28 }
```

Figure 5: A motivating example.

Since embedding opaque predicates into a program is a semantics-preserving transformation, deterministic programs before and after opaque predicate obfuscation should produce the same output. Let us assume the program P is obfuscated by opaque predicates and the resulting program is denoted as P_o. The logic of an execution of P_o is expressed as a formula Ψ, which is the conjunction of all branch conditions executed, including the following opaque predicates.

$$\Psi = \psi_1 \wedge ... \wedge \psi_{i-1} \wedge \psi_i \wedge ... \wedge \psi_n$$

Formula Ψ represents the conditions that an input must satisfy to execute the same path. Supposing constraint ψ_i is derived from an opaque predicate, we call ψ_i a *culprit branch*. The key to our approach is to locate all culprit branches in Ψ. Similar to dynamic symbolic execution [25] on binary code, we first characterize the logic of an execution in terms of symbolic path conditions, by performing a symbolic execution on the concrete execution trace.

Then our approach carries out three rounds of scanning. In the first round, we diagnose whether ψ_i is derived from an invariant opaque predicate by proving whether ψ_i is always true; that is, it is a *tautology*. Note that the false branch conditions have already been negated in the recorded trace. After that, we remove identified culprit branches from Ψ and continue to detect possible contextual opaque predicates in the second round. Our key insight is that a contextual opaque predicate does not enforce any *further* constraint on its prior path condition. Based on this observation, diagnosing whether a path constraint ψ_i $(1 \leq i \leq n)$ is a contextual opaque predicate boils down to answering an implication query, namely

$$\psi_1 \wedge ... \wedge \psi_{i-1} \Rightarrow \psi_i$$

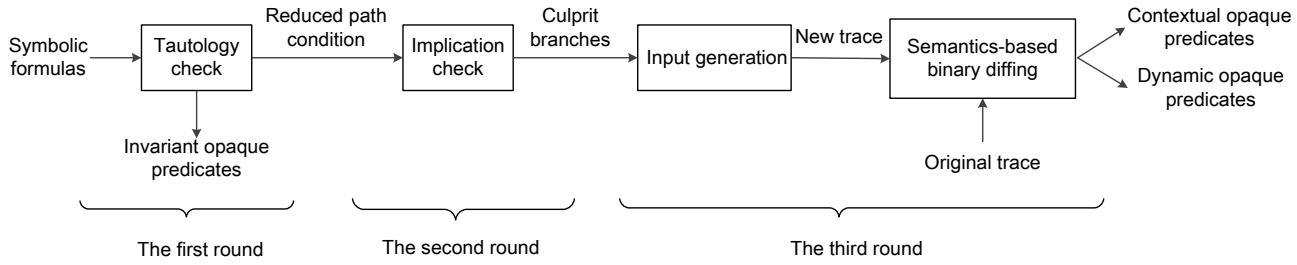

Figure 4: Opaque predicate detector.

```
 3  int *p = &x;
 4  int *q = &x;
 5  int y = 0;
 6  if (x*x < 0)        // invariant opaque predicate
 8  if (x > 3)
 9  {                   // contextual opaque predicate
10     if (x*x-4x+3 > 0)
11        x = x<<1;
12  }
13  if ((*p)%2 == 0) // dynamic opaque predicate
14     y = x+1;
20  if ((*q)%2 == 0)
21  {
22     y = y+2;
23     x = y+3;
24  }
27  return x;
28 }
```

Figure 6: An execution trace given x=4.

Note that dynamic opaque predicates satisfy such implication check as well. For example, the combination of path condition for Figure 3 is either $p \wedge q$ or $\neg p \wedge \neg q$. It is straightforward to infer the following implication relationship.

$$(p \Rightarrow q) \wedge (\neg p \Rightarrow \neg q)$$

Assume we have detected $p \Rightarrow q$ in the second round of scanning. To further clarify p and q are correlated dynamic opaque predicates, we go one step further in the third round to verify whether $\neg p \Rightarrow \neg q$ holds as well. To this end, we automatically generate a new input that follows the path of $\neg p \wedge \neg q$. If $\neg p \Rightarrow \neg q$ is also true, we continue to compare trace segments guided by both $p \wedge q$ and $\neg p \wedge \neg q$ to make sure two paths are semantically equivalent. Further details about the detection process are discussed in Section 4.

3.2 Example

We create a motivating example (shown in Figure 5) to illustrate our core method. Figure 5 contains three different kinds of opaque predicates. Note that the two dynamic opaque predicates are constructed using pointer deference (line 13 $\sim$ line 27). The predicates in line 13 and line 20 are *correlated*, since they evaluate to the same value at any given run. In any case, the same instruction sequence $\{y = x + 1;\ y = y + 2;\ x = y + 3;\}$ will be executed. Consider an execution of the code snippet given $x = 4$ as input. Figure 6 shows a source-level view of such execution trace. We perform backward slicing and symbolic execution to calculate path condition formula Ψ. In our example, the predicates are represented as follows.

$$\begin{aligned}
\psi_1 &: \ x * x >= 0 \\
\psi_2 &: \ x > 3 \\
\psi_3 &: \ x * x - 4x + 3 > 0 \\
\psi_4 &: \ (*p)\%2 == 0 \\
\psi_5 &: \ (*q)\%2 == 0 \\
\Psi &: \ \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5
\end{aligned}$$

We present the three rounds step by step.

1. At the first round, we verify whether a predicate satisfies invariant property; i.e., it is a *tautology*. In our example, we prove that ψ_1 $(x * x >= 0)$ is always true and therefore conclude that ψ_1 is an invariant opaque predicate. After that, we remove ψ_1 from path condition Ψ to reduce the formula size and pass the new path condition to the next round.

2. We start the second round to detect possible contextual opaque predicates by performing implication check cumulatively from the first predicate. We identify two cases that satisfy the implication check in our example: $\psi_2 \Rightarrow \psi_3$ (ψ_1 has been removed), i.e., $(x > 3) \Rightarrow (x * x - 4x + 3 > 0)$ and $\psi_2 \wedge \psi_3 \wedge \psi_4 \Rightarrow \psi_5$, i.e., $(x > 3) \wedge (x * x - 4x + 3 > 0) \wedge ((*p)\%2 == 0) \Rightarrow ((*q)\%2 == 0)$. Note that ψ_5 in the second case is corresponding to the second culprit branch of the dynamic opaque predicates.

3. In the third round, we trace back from the culprit branches identified in the second step and further verify whether their prior predicates are correlated or not. Recall that another property of dynamic opaque predicates is being *adjacent*. In our example, we first negate each prior predicate as $\neg\psi_2$ and $\psi_2 \wedge \psi_3 \wedge \neg\psi_4$ and automatically generate inputs to satisfy such new path conditions. Here we generate two new inputs respectively, namely, $x = 0$ and $x = 5$. With the new traces, we perform implication check for $\neg\psi_2 \Rightarrow \neg\psi_3$ and $\psi_2 \wedge \psi_3 \wedge \neg\psi_4 \Rightarrow \neg\psi_5$. It is evident that $\neg\psi_2 \Rightarrow \neg\psi_3$ fails under the counterexample of $x = 0$. At last, we compare trace segments controlled by $\psi_4 \wedge \psi_5$ and $\neg\psi_4 \wedge \neg\psi_5$ to make sure they are semantically equivalent. As a result, we conclude that ψ_3 is a contextual opaque predicate and ψ_4 and ψ_5 consist of dynamic opaque predicates.

For the presentation purpose, all the examples in this section are shown as C code and the predicates are presented as

the "if" conditional statements. LOOP works at the binary level, in which the conditional statements such as "if" and "switch" are compiled as conditional jump instructions like je/jne/jg. LOOP identifies the conditional jump instructions that are opaquely true or false.

4. APPROACH

In this section, we present each step of our approach in detail. Figure 7 illustrates the architecture of LOOP, which includes two main parts: online trace logging and offline analysis. The online part, as shown in the left side of Figure 7, is built on top of a dynamic binary instrumentation (DBI) platform, enabling LOOP to work with unmodified binary code. To analyze packed malware, our online part includes two tools: generic unpacking and trace logging.

The logged trace is passed to the second part of LOOP for offline analysis (the right component of Figure 7). We first lift x86 instructions to BAP IL [7], a RISC-like intermediate language. Then starting from each predicate (branch), we perform backward slicing to determine the instructions that contribute to the value of the predicate. Then we perform symbolic execution along the slice to calculate a symbolic expression for each predicate. Based on that, our opaque predicate detector will construct formulas to represent the semantics of opaque predicates. We then solve them with a constraint solver. As discussed in Section 3, the detector conducts three rounds of scanning. The net result is a set of culprit branches corresponding to the opaque predicates.

4.1 Online Logging

As shown in Figure 7, LOOP's online part is built on a dynamic binary instrumentation framework. To undermine anti-malware detection, most malware developers apply different packers to compress or encrypt malware binaries. As a result, when a packed sample starts running, unpacking routines will first restore the original payload (e.g., decompress or decrypt) and then jump to the original entry point (OEP) to continue the execution. One of our implementation choices is we only detect opaque predicates within malware real payload. To this end, when a malware sample starts running, we first invoke our generic unpacking tool to monitor memory write operations. If a memory region pointed by the `eip` register is "written and then execute" [31], it indicates that we have identified the newly generated code. Then we activate our trace logging tool to start trace recording.

In general, analyzing all the instructions could be a tedious job. To keep the logged trace compact, LOOP supports on-demand logging to optionally record instructions of interest. In addition, our logging tool can perform dynamic taint tracking so that only the tainted instructions are collected.

4.2 Slicing and Symbolic Execution

Taking the logged trace as input, LOOP's offline analysis first lifts x86 instructions to BAP IL, which is a RISC-like intermediate language without side effect. In addition, the property of static single assignment (SSA) format facilitates tracing the use-def chain when we perform slicing. The symbolic execution is carried out on BAP IL as well.

Given a predicate (or branch), LOOP first identifies all the instructions that contribute to the calculation of this predicate. Note that x86 control transfer instructions typically depend on certain bits of the `eflags` register (e.g., `jz` and `jnz`). Therefore, the slicing criteria look like $\langle eip, zf \rangle$, where eip is the instruction pointer and zf is the abbreviation of the zero flag bit. Starting from the slicing criteria, we perform dynamic slicing [1] to backtrack a chain of instructions with data and control dependencies. We terminate our backward slicing when the source of the slice criteria satisfy one of the following conditions: constant values, static strings, user defined value (e.g., function return value) or input. Besides, we also observe a case that the conditional logic is implemented without `eflags` register: `jecxz` jumps if register `ecx` is zero. LOOP handles this exception as well.

By labeling inputs or user defined values as symbols, we conduct symbolic execution along the slice to compute a symbolic expression for each predicate. The result will be passed to the opaque predicate detector, the core of LOOP's offline analysis. Figure 4 shows the components of our opaque predicate detector, which consists of three rounds of scanning to detect invariant, contextual and dynamic opaque predicates progressively. We will present each round of detection in the following subsections.

4.3 Invariant Opaque Predicates

Invariant opaque predicates refer to those predicates that are always true or false for all possible inputs. This kind of opaque predicates mainly relies on well-known algebraic theorems [3, 39]. Since they are easy to construct, invariant opaque predicates are the most frequently used ones. The detection method is straightforward by exploiting the invariant property. Tautology check in Figure 4 is used to prove whether the symbolic expression for each predicate in the trace always evaluates to true. For instance, one symbolic expression may be expressed as $\forall x \in Z.\ (x^3 - x)\%3 = 0$. We feed this formula to a constraint solver to prove its validity. Another characteristic of this category of predicates is that they are independent from each other, and therefore it is natural to parallelize the detection. We remove the identified invariant opaque predicates from the path condition to reduce its size. After that, the reduced path condition is sent to the second round of scanning.

4.4 Contextual Opaque Predicates

Different from the invariant property of the first category of predicates, a contextual opaque predicate relies on some program invariants so that it always produces the same value when certain precondition holds. At other places, such a predicate may evaluate to a different value. The precondition can be further obfuscated to camouflage the context information. Our detection method is based on the observation that a contextual opaque predicate (ψ_i) does not impose additional constraint on its prior path condition. Therefore, ψ_i should be *logically implied* by its prior path condition: $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1}$; that is, $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$. The process of testing whether a predicate ψ_i and its preceding predicates satisfy the relation $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$ is called *implication check*. Without knowing which predicate is a potential contextual opaque predicate, we have to perform the implication check cumulatively starting from the first predicate to the last one. One challenge here is, subject to computing resources (e.g., memory and CPU), a complicated formula may be hard or infeasible to solve. To address this issue, we adopt a "short cut" strategy. When the generated formula becomes too complicated, we divide it into each single predicate and detect whether $\psi_j \Rightarrow \psi_i, j < i$. If this check

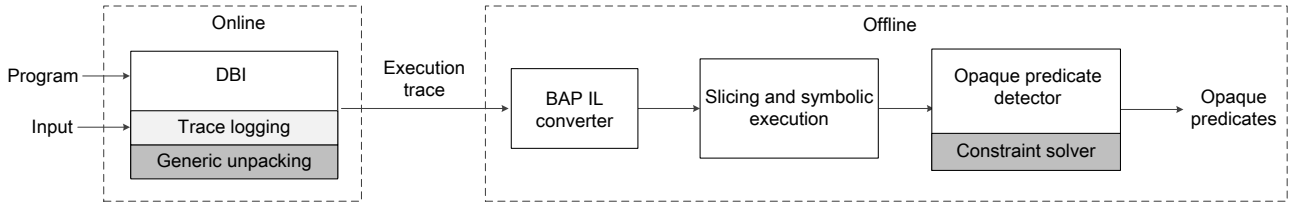

Figure 7: The architecture of LOOP.

Algorithm 1 Testing dynamic opaque predicates

P: a set of predicates which pass the implication check
T: the execution trace from which P is derived

function TESTDOP(T, P)
  for each ψ_i in P **do**
    $T' \leftarrow$ GenTrace($\neg\psi_{i-1}$)
    $\psi_i' \leftarrow$ Next(T', $\neg\psi_{i-1}$)
    if $\psi_i' \neq \neg\psi_i$ **then**
      return False
    end if
    if $\neg\psi_{i-1} \Rightarrow \neg\psi_i$ **then**
      if $T_{i-1..i} \approx T'_{i-1..i}$ **then**
        // semantically equivalent
        return True
      else
        return False
      end if
    else
      return False
    end if
  end for
end function

passes, we do not need to prove $\psi_1 \wedge \psi_2 ... \wedge \psi_{i-1} \Rightarrow \psi_i$ any more, as the standard implication check will pass certainly.

The results of the second round of scanning are culprit branches that pass the implication check. Note that at this point, we cannot conclude whether these culprit branches are corresponding to real contextual opaque predicates. Recall the correlation property of dynamic opaque predicates discussed in Section 2, most dynamic opaque predicates (from the second to the last one in a family of related dynamic opaque predicates) satisfy the implication check as well. In the third round of detection, LOOP will distinguish these two categories and identify all correlated dynamic opaque predicates.

4.5 Dynamic Opaque Predicates

Dynamic opaque predicates consist of a family of *correlated* and *adjacent* predicates which produce the same boolean value in any given run. However, the value may switch to a different result in another run. Locating correlated predicates is the key to dynamic opaque predicate detection. A set of predicates are correlated means they are implied by each other, or in other words, they are *logically equivalent*. Therefore, predicates $\psi_1, \psi_2, ..., \psi_n$ are correlated iff

$$\psi_1 \Leftrightarrow \psi_2 \Leftrightarrow ... \Leftrightarrow \psi_n.$$

Assume we have two correlated predicates ψ_{i-1} and ψ_i. Recall that in the detection of contextual opaque predicates, we have proved that $\psi_{i-1} \Rightarrow \psi_i$. Now we have to go one

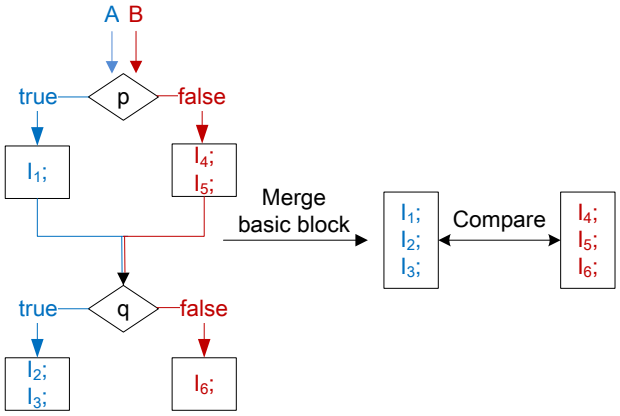

Figure 8: Trace segment comparison (A and B represent different inputs).

step further to verify whether $\psi_i \Rightarrow \psi_{i-1}$. In logic, we have the following equation.

$$\psi_i \Rightarrow \psi_{i-1} \equiv \neg\psi_{i-1} \Rightarrow \neg\psi_i$$

Therefore, we alternatively verify whether $\neg\psi_{i-1}$ implies $\neg\psi_i$ by generating a new input. Another property of dynamic opaque predicates is that they utilize code clone to make sure the same instructions are executed in any case. As a result, we have to compare the two traces to ensure they are semantically equivalent. Similar to the trace-oriented binary diffing tool [36], our comparison approach relies on symbolic execution and theorem proving techniques so that we can find that two code pairs with ostensibly different syntax are semantically equivalent.

Suppose ψ_{i-1} and ψ_i are dynamic opaque predicates and ψ_i has been identified as a culprit branch satisfying implication check. The procedure of our third round of detection is shown in Algorithm 1. First, we negate the predicate ψ_{i-1} and use constraint solver to generate a new input to follow the path $\psi_1 \wedge \psi_2 ... \wedge \neg\psi_{i-1}$. Given the new trace, the predicate following $\neg\psi_{i-1}$ is ψ_i'. Then we verify ψ_i' is equivalent to $\neg\psi_i$. Next, we test whether $\neg\psi_{i-1} \Rightarrow \neg\psi_i$. If it succeeds, we will compare the trace segments controlled by $\psi_{i-1} \wedge \psi_i$ and $\neg\psi_{i-1} \wedge \neg\psi_i$. Different from previous semantics-based binary diffing tools [36], whose comparison unit is a single basic block, LOOP merges all the basic blocks which have control flow dependence with $\psi_{i-1} \wedge \psi_i$ or $\neg\psi_{i-1} \wedge \neg\psi_i$ as a trace segment (as shown in Figure 8). LOOP carries out symbolic execution for the trace segment and represents its input-output relation with a set of formulas. At last, LOOP uses a constraint solver to compare the output formula pairs and then finds a bijective equivalent mapping among them.

Table 1: Common mathematical formulas and their STP solving time.

Formulas	Solving time (s)
$\forall x \in Z.\ x^2 \geq 0$	0.003
$\forall x \in Z.\ 2 \mid x(x+1)$	0.008
$\forall x \in Z.\ 3 \mid x(x+1)(x+2)$	0.702
$\forall x, y \in Z.\ 7y^2 - 1 \neq x^2$	0.008
$\forall x \in Z.\ (x^2+1)\%7 \neq 0$	17.762
$\forall x \in Z.\ (x^2+x+7)\%81 \neq 0$	22.657
$\forall x \in Z.\ (4x^2+4)\%19 \neq 0$	15.392
$\forall x \in Z.\ 4 \mid x^2(x+1)(x+1)$	0.012
$\forall x \in Z.\ 2 \mid x \vee 8 \mid (x^2-1)$	0.022
$\forall x \in Z.\ 2 \mid \lfloor \frac{x^2}{2} \rfloor$	0.015

If so, we conclude that these two trace segments are semantically equivalent and therefore ψ_{i-1} and ψ_i are dynamic opaque predicates. If $\neg\psi_{i-1} \Rightarrow \neg\psi_i$ fails or new trace segment is different to the original one, we conclude ψ_i is a contextual opaque predicate.

Following the similar idea, we can identify the correlated predicates containing more than two predicates and then check the equivalence of the trace segments which have control dependence on these predicates.

4.6 Solving Opaque Predicate Formulas

It is evident that the efficacy of LOOP depends on the capability of constraint solvers in proving the validity of a formula. State-of-the-art constraint solvers (e.g., STP [23], Z3 [38]) have supported all arithmetic operators found in the C programming language (e.g., bitwise operations, multiplication, division, and modular arithmetics). Table 1 shows ten mathematical formulas that are commonly used as invariant opaque predicates, including integer division, modulo and remainder operations. We solve these formulas with STP in our testbed and present the solving time in the second column. Most formulas only need less than 0.1 seconds, which are almost negligible. However, we notice that formulas involving modular arithmetic increase the solving time considerably. For example, the solving time for three complicated modular arithmetic formulas in Table 1 varies from 15 to 23 seconds. Even so, compared with other methods which need to run a large set of test inputs (e.g., statistical analysis [17] and fuzz testing [33]), our approach achieves better performance and accuracy.

4.7 Whole Program Deobfuscation

After three rounds of detection, LOOP returns all possible opaque predicates in an execution trace. Our result can be fed back to security analysts to further deobfuscate the cluttered control flow graph of a whole program. For example, the unreachable paths introduced by invariant opaque predicates are discarded; the redundant contextual opaque predicates are cut off as well. To reverse the effect of dynamic opaque predicates, we remove the set of correlated dynamic opaque predicates and replace the corresponding multiple paths with a single straight-line path. Since our approach is trace-oriented, we deobfuscate a part of the control flow graph each time. To increase path coverage, we can leverage automatic input generation techniques [13, 25].

5. IMPLEMENTATION

LOOP's online logging part consists of two tools, generic unpacking and trace logging, which are implemented based on the Pin DBI framework [32] (version 2.12) with 1,752 lines of code in C/C++. To make the logged trace compact, we start trace logging when unpacking routine finishes and support on-demand logging for instructions of interest. LOOP's offline analysis part is implemented on top of BAP [7] (version 0.8) with 2,588 lines of OCaml code. We rely on BAP to lift up x86 instructions to the BAP IL and convert BAP IL to CVC formulas. LOOP's backward slicing is performed on BAP IL; the opaque predicate detector and trace segment comparison tool are built on BAP's symbolic execution engine. We use STP [23] as our constraint solver. Besides, we write 644 lines of Perl scripts to glue all components together to automate the detection. To facilitate future research, we have made LOOP code available at `https://github.com/s3team/loop`.

Table 2: Contextual opaque predicates used in common utilities evaluation.

1	$\forall x \in Z.\ x > 5 \Rightarrow x > 0$
2	$\forall x \in Z.\ x > 3 \Rightarrow x^2 - 4x + 3 > 0$
3	$\forall x \in Z.\ x\%4 = 0 \Rightarrow x\%2 = 0$
4	$\forall x \in Z.\ x\%9 = 0 \Rightarrow x\%3 = 0$
5	$\forall x \in Z.\ x\%10 = 0 \Rightarrow x\%5 = 0$
6	$\forall x \in Z.\ 3 \mid (7x-5) \Rightarrow 9 \mid (28x^2 - 13x - 5)$
7	$\forall x \in Z.\ 5 \mid (2x-1) \Rightarrow 25 \mid (14x^2 - 19x - 19)$
8	$\forall x, y, z \in Z.\ (2 \nmid x \wedge 2 \nmid y) \Rightarrow x^2 + y^2 \neq z^2$

6. EVALUATION

We evaluate LOOP's effectiveness to automatically detect various opaque predicates. We test LOOP with a set of Linux common utilities and highly obfuscated malicious programs. We make sure we have the ground truth so that we can accurately assess false positives and false negatives. Also, since LOOP is strongly motivated by its application, we evaluate the impact of LOOP in the task of code normalization for metamorphic malware. The experiments are performed on a machine with a Intel Core i7-3770 processor (Quad Core, 3.40GHz) and 8GB memory, running both Ubuntu 12.04 and Windows XP SP3.

6.1 Evaluation with Common Utilities

This experiment is designed to evaluate the effectiveness and efficiency of our method. First, we select ten widely used utility programs in Linux as our test cases. To ensure the samples' variety, those candidates are picked from different areas, including data compression, core utilities, regular expression search, hash computing, web file transfer, and HTTP server. Since all of those programs are open source, we can easily verify our detection results. We implement automatic opaque predicate insertion as an LLVM pass, based on Obfuscator-LLVM [26]. For each program, we insert seven opaque predicates, including three invariant, three contextual and one dynamic opaque predicates. The three invariant opaque predicates are randomly selected from Table 1; the three contextual opaque predicates are chosen from Table 2. We use the example of dynamic opaque predicates shown in Figure 5. At the same time, we make sure that all the opaque predicates can be reached by the test inputs.

We first label the inputs of test cases as tainted and record tainted instructions. Then we run LOOP's offline analysis to detect the opaque predicates. Table 3 shows the experimental results. The second column shows the number of

Table 3: Evaluation results on linux common utilities.

Program	#Predicates (no-taint, taint)	Invariant			Contextual			Dynamic			(#FP, #FN)
		#OP	SE (s)	STP (s)	#OP	SE (s)	STP (s)	#OP	SE (s)	STP (s)	
bzip	(6,313, 13)	4	0.9	1.4	6	1.2	2.3/1.9	2	1.8	2.4	(5, 0)
grep	(4,969, 16)	5	0.9	1.5	6	1.8	1.6/1.3	3	2.4	0.3	(6, 0)
ls	(5,867, 21)	13	3.3	3.9	3	22.4	3.5/2.8	1	12.5	2.1	(10, 0)
head	(1,496, 11)	6	0.7	2.5	7	1.1	1.3/1.3	1	1.9	1.0	(1, 0)
md5sum	(3,450, 14)	3	2.8	4.5	3	1.5	25.0/20.3	1	1.0	2.2	(0, 0)
thttpd	(6,605, 124)	3	8.7	16.8	3	3.6	5.4/2.2	1	1.4	0.7	(0, 0)
boa	(4,718, 131)	3	6.8	18.7	3	2.5	6.2/1.8	1	1.7	0.8	(0, 0)
wget	(3,230, 36)	5	3.0	7.6	3	2.0	2.2/1.2	1	1.5	0.7	(2, 0)
scp	(2,402, 30)	5	3.4	5.8	4	2.8	4.1/2.4	1	1.5	0.7	(3, 0)
libpng	(25,377, 446)	7	51.4	351.6	4	14.6	33.2/11.6	2	10.4	5.2	(6, 0)

predicates before and after taint. It is evident that forward taint propagation reduces the number of predicates significantly. For each category of opaque predicate, we report the number of opaque predicate detected (#OP), the time of symbolic execution (SE) and the time of running STP solver (STP). Note that in the column of STP time for contextual opaque predicates, we list the different time before and after "short cut" optimization (see Section 4.4).

For the majority of our test cases, the symbolic execution and STP solver only take several seconds. Because libpng's trace size is large and its invariant opaque predicates inserted involve modulus arithmetic, the corresponding STP solving time is the longest. The data presented in the eighth column indicate that the effect of "short cut" optimization is encouraging, especially for the cases with large path formulas (e.g., libpng). Furthermore, we manually verify each logged traces and find that our approach successfully diagnoses all the opaque predicates if there is any; that is, we have zero false negatives (#FN in the last column).

To test false positives of our approach, namely whether LOOP mistakes a normal condition as an opaque predicate, we conduct a similar evaluation on our test cases' clean version (no opaque predicate insertion). Contrary to our expectation, we notice that seven out of ten cases detect opaque predicates but all of them are false positives (#FP in the last column). We look into the factors leading to the false positives and find that one major reason is "under tainting" [27], which is a common problem in taint analysis. Generally, under tainting means instructions that should be tainted are not recorded. As a result, under tainting will mistakenly replace some symbols with concrete values. For instance, supposing `y` with a concrete value of `2` is not labeled as a symbol in the predicate `y > 1`, the predicate in the trace would be `2 > 1`, which is a tautology, and LOOP will issue a false alarm.

6.2 Evaluation with Obfuscated Malware

Opaque predicates are also widely used by malware developers. Moreover, opaque predicates are typically integrated with other obfuscation methods to impede reverse engineering attempts. To evaluate the resilience of LOOP against various obfuscation methods, we collect 15 malware binaries from VX Heavens.[1] These malware samples are chosen for two reasons: 1) they are representative in obfuscation techniques; 2) we have either source code (e.g., QQThief and KeyLogger) or detailed reverse engineering reports (e.g., Bagle and Mydoom). Hence, we can accurately evaluate our detection results.

```
call near ptr GetFontData ; eax = 0xFFFFFFFF
                          ; ebp, esp are even numbers
and eax, ebp              ; eax = eax & ebp
                          ; eax is an even number
inc eax                   ; eax += 1, eax is an odd number
and eax, 01h
cmp eax, 0                ; eax = 01h
jne   new_target          ; always true
```

Figure 9: Example of an opaque predicate in malware.

Table 4 shows a variety of obfuscation techniques with different purposes. Column 4 $\sim$ 9 represent the methods to obfuscate code and data, such as binary compression and encryption packers, junk code, code reorder, and opaque constant. Column 10 $\sim$ 12 denote the control flow obfuscation methods in addition to opaque predicates, including call-stack tampering, CFG flatten, and obfuscated control transfer target. The methods in column 13 and 14 are used to detect the debugging and virtual machine (VM) environment. The "# OP" column presents the number of opaque predicates detected by LOOP. The triple such as (5,3,1) represents the number of invariant, contextual, and dynamic opaque predicates, respectively. We find that most malware samples (12 out of 15) are embedded with opaque predicates and invariant opaque predicates are the most frequently used. The high number (90) of invariant opaque predicates detected in Hunatcha is caused by loop unrolling. With the help of source code and reverse engineering reports, we count false positives and false negatives (shown in column 16). Similar with common utilities' results, LOOP achieves zero false negatives. That is, LOOP does not miss any opaque predicates. The last column lists the total offline analysis time. Note that our generic unpacking cannot handle virtualization obfuscators [18] such as VMProtect[2] and Themida[3]. In our test cases, we find two malware samples (BullMoose and Branko) are obfuscated with virtualization obfuscation. As a result, the logged trace mixes the code of virtualization interpreter with the code of malicious payload. LOOP nevertheless detects opaque predicates successfully.

Figure 9 shows an opaque predicate we detected in KeyLogger Trojan. This opaque predicate utilizes the fact of stack memory alignment, and therefore both `ebp` and `esp` are even numbers. After several arithmetic operations, the

[1] http://vxheaven.org/src.php

[2] http://vmpsoft.com/

[3] http://www.oreans.com/themida.php

Table 4: Various obfuscation methods applied by malicious program.

Sample	Type	Size (kb)	Packer	Compres. packer	Encrypt. packer	Junk code	Code reorder	Opaque constant	Call-stack tampe.	CFG flatten	Obfuscat. transf.	Anti-debugging	Anti-VM	# OP	(#FP, #FN)	Time (s)
Bube	Virus	12	FSG	✓		✓	✓	✓	✓					(0,0,0)	(0, 0)	3.4
Tefuss	Virus	29	UPX	✓		✓	✓		✓			✓	✓	(2,0,0)	(0, 0)	4.2
Champ	Virus	13	PECompact	✓				✓	✓			✓	✓	(0,0,0)	(0, 0)	3.0
BullMoose	Trojan	30	VMProtect		✓	✓	✓					✓		(5,3,1)	(2, 0)	7.8
QQThief	Trojan	32	Yoda's Protector	✓	✓			✓	✓		✓	✓		(3,0,0)	(1, 0)	15.4
KeyLogger	Trojan	33	ASPack	✓	✓				✓		✓			(8,1,1)	(0, 0)	22.3
Autocrat	Backdoor	276	PECompact	✓				✓	✓			✓	✓	(12,0,0)	(4, 0)	244.5
Codbot	Backdoor	30	ASPack	✓	✓				✓		✓			(2,0,0)	(0, 0)	8.2
Loony	Backdoor	20	ASPack	✓	✓				✓		✓			(0,0,0)	(1, 0)	5.3
Branko	Worm	17	Themida		✓					✓	✓	✓	✓	(10,0,0)	(0, 0)	8.5
Hunatcha	Worm	61	PolyEnE		✓	✓		✓		✓	✓			(90,3,1)	(0, 0)	36.2
Bagle	Worm	47	UPack	✓	✓	✓	✓							(6,0,0)	(2, 0)	24.6
Sasser	Worm	60	UPack	✓	✓	✓	✓					✓		(4,0,0)	(0, 0)	18.2
Mydoom	Worm	41	ASProtect	✓	✓	✓	✓	✓			✓			(6,0,0)	(1, 0)	132.3
Zeynep	Worm	85	Yoda's Protector	✓	✓			✓	✓		✓	✓		(8,1,0)	(3, 0)	192.3

Table 5: Speed up metamorphic malware variants matching

Family	Basic blocks reduction (%)	Isomorphism speedup (X)
Metaphor	26	2.0
Lexotan32	20	1.6
Win32.Evol	16	1.2

last branch is always true. In summary, our experiments show that LOOP is effective in detecting opaque predicates in obfuscated binary code with a zero false negative rate. Considering that LOOP aims to provide a general and automatic de-obfuscation solution, which usually involves tedious manual work, the false positive rate is tolerable.

6.3 Metamorphic Malware Matching

To confirm the value of our approach in malware defenses, we also test LOOP in the task of code normalization for metamorphic malware [9, 10]. Metamorphic malware mutates its code during infection so that each variant bears little resemblance to another one in syntax. It is well known that metamorphism can undermine the signature-based anti-malware solutions [44]. Bruschi et al. [9, 10] propose code normalization to reverse the mutation process. To test whether an instance of metamorphic malware is present inside an infected host program, they compare malicious code and normalized program by inter-procedural control flow subgraph isomorphism. The drawback is that they do not handle opaque predicates, although opaque predicates are one of the commonly used mutation methods. Opaque predicates can seriously thwart normalized control flow graph comparison [9]. We re-implement their code normalization based on BAP and test the speedup of normalized control flow graph isomorphism under the preprocess of LOOP on three famous metamorphic malware families.

Since the three metamorphic malware samples are all file-infecting, we first force each malware to infect 20 Cygwin utilities.[4] For each family, we follow similar steps as in Bruschi et al. to normalize infected programs and compare their control flow graphs (CFG) with malicious code's CFG, leveraging VFLIB library [20]. In addition, we apply LOOP to preprocess infected programs to remove the corresponding superfluous branches and infeasible paths. Table 5 shows the improvements introduced by our approach on average. Compared with the results without applying LOOP, we remove redundant basic blocks as much as 26% and speed up subgraph isomorphism by a factor of up to 2.0 (e.g., Metaphor).

7. DISCUSSIONS AND FUTURE WORK

We further discuss about our design choices, limitations and future work in this section.

Dynamic Approach.

Our approach bears the similar limitations as dynamic analysis in general. For example, LOOP can only detect opaque predicates executed at run time. Static analysis might explore all the possible paths in the program. However, even static disassembly of stripped binaries is still a challenge [30, 47]. Moreover, the various obfuscation techniques listed in Table 4 will undoubtedly deter extracting accurate control flow graph from binary code. We believe our approach, based on the test cases that execute opaque predicates, is practical in analyzing a malicious program. A possible way to increase path coverage is to leverage test-generation techniques [13, 25] to automatically explore new paths. Another concern we want to discuss is scalability issue. The size of a slice may become significant for a program with high workload, and our detection approach is linearly dependent on the size of a slice. In that case, LOOP has to analyze a large number of predicates, resulting in a substantial performance slowdown. One way to alleviate the high overhead is to detect opaque predicates in parallel. We plan to explore this direction in our future work. Detect-

[4] www.cygwin.com

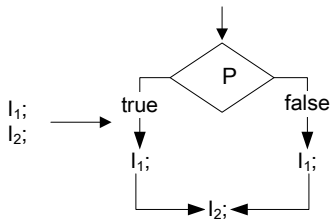

Figure 10: Example of two-way opaque predicates.

```
x = input ( );
I1;
I2;
```

```
x = input ( );
I1;
y = x;
while (y > 1)
{
   if (y%2 != 0)
       y = 3*y+1;
   else y = y/2;
}
I2;
```

Figure 11: Example of 3x+1 conjecture.

ing repetitive opaque predicates due to loop unrolling leads to performance penalty. We will extend our tool with loop identification in the next version.

Floating Point.

The effect of LOOP is restricted by the capability of the constraint solver and symbolic execution tools. One example is floating point. Since precisely defining the rounding semantics of floating point is challenging, current binary symbolic execution does not support floating point instructions (e.g., fdiv and fmul) [7]. As a result, currently LOOP is unable to detect opaque predicates involving floating point operations. One might argue that attackers can easily get around LOOP by using floating point equations. However, using floating point equations in malware increases the possibility of detection as well, because both floating point instructions and related numerics library API calls (e.g., log, exp and sqrt) are rarely seen in malicious code.

Two-way Opaque Predicates.

Figure 10 shows a subtle case of opaque predicate, in which two possible directions will sometimes be taken and $I_1; I_2$ are executed in any case. P does not belong to the categories we summarized in Section 2 and therefore we cannot ensure P is an opaque predicate in a single trace. One way to detect such case is to check semantic equivalence of P's two jump targets [49].

Unsolved Conjectures.

Recently Wang et al. [48] proposed a more stealthy obfuscation scheme by incorporating linear unsolved conjectures, which appear to be correct but without proof. Figure 11 presents an example of embedding the well-known 3x+1 conjecture into a program. This conjecture asserts that given any positive integer y, the loop will always terminate. In principle, we could treat the conjunction of branch conditions derived from the unroll loop as a single opaque predicate, which always evaluates to be true for any positive integer. However, different from dynamic opaque predicates, the number of conditions is various under different inputs and conditions themselves are not correlated as well. To detect such unsolved conjectures, we observe that all the examples [48] will eventually converge to a fixed value regardless of the initial value. We could automatically generate test cases to explore different paths and observe whether the multiple inputs cover the same value when the conjecture loop ends. We leave it as our future work.

8. RELATED WORK

In this section, we first present previous work on opaque predicates detection, which is close in spirit to our work. Then, we introduce related work on concolic testing, a hybrid approach to perform symbolic execution. In principle, our trace-oriented detection is also a hybrid approach that applies symbolic execution in de-obfuscation. At last we introduce previous work on infeasible paths identification, which is related to our work in that LOOP can be applied to detect infeasible paths in binary.

Opaque Predicate Detection.

The concept of opaque predicates is first proposed by Collberg et al. [17] to prevent malicious reverse engineering attempts. Collberg et al. [17] also provide some ad-hoc detection methods. One of them is called "statistical analysis"; that is, a predicate that always produces the same result over a larger number of test cases has a great chance to be an opaque predicate. Due to the limited set of inputs, statistical analysis could lead to high false positive rates. Preda et al. [42] propose to detect opaque predicates by abstract interpretation. However, their approach only detects a specific type of known invariant opaque predicates such as $\forall x \in Z.n|f(x)$. Madou [33] first identifies candidate branches that never change at run time, and then verifies such predicates by fuzz testing with a considerably high error rate. Udupa et al. [45] utilize static path feasibility analysis to determine whether an execution path is feasible. However, their approach cannot work on a highly obfuscated binary with, for example, complicated opaque predicates based on pointer aliasing, which is known to be statically intractable. OptiCode [43] has a similar idea in using theorem prover to decide if a desired branch is always true or false, but it can only deal with invariant opaque predicates. Our work is different from the previous work in that LOOP is both general and accurate. We are able to detect previously unknown opaque predicates in obfuscated binary, including more sophisticated ones such as contextual and dynamic opaque predicates.

Concolic Testing.

Our logic-based approach is inspired by the active research in concolic testing [13, 12, 24, 25], a hybrid software verification method combining concrete execution with symbolic execution. Similar to SAGE [25], LOOP first maps symbols to inputs and then collects constraints of these symbolic inputs along a recorded execution trace. The difference is our primary purpose is not for path exploration; instead we construct formulas representing the characteristics of opaque predicates and solve these formulas with a constraint solver. In addition to automatic input generation, we have seen applications of concolic testing in discovery of deviations in binary [6], software debugging with golden implementation [4], and alleviating under-tainting problem [27]. Our

approach adopts part of the concolic testing idea in software de-obfuscation and malware analysis.

Infeasible Path Identification.

The effect of opaque predicates is to obfuscate control flow graph with infeasible paths. In software testing, eliminating infeasible paths saves efforts to generate redundant test cases. Previous work identifies infeasible paths in source code, either by branch correlation analysis [5], pattern matching [40], or monitoring the search for test data [11]. However, these work cannot be directly used to detect opaque predicates in an adversary environment, in which the program source code under examination is typically absent. Therefore, we believe LOOP has compelling application in identifying infeasible paths in binary.

9. CONCLUSION

Opaque predicates have been widely used in software protection and malicious program to obfuscation program control flow. Existing efforts to detect opaque predicates are either heuristics-based or work only on specific categories. In this paper, we have presented LOOP, a program logic-based and obfuscation resilient approach to the opaque predicate detection in binary code. Our approach represents the characteristics of various opaque predicates with logical formulas and verifies them with a constraint solver. LOOP detects not only simple invariant opaque predicates, but also advanced contextual and dynamic opaque predicates. Our experimental results show that LOOP is effective in detecting opaque predicates in a range of benign and obfuscated binary programs. By diagnosing culprit branches derived from opaque predicates in an execution trace, LOOP can help analysts for further de-obfuscation. The experiment of speeding up code normalization for matching metamorphic malware variants confirms the value of LOOP in malware defenses.

10. ACKNOWLEDGMENTS

We thank the ACM CCS 2015 anonymous reviewers and Bill Harris for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175.

11. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990.

[2] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of Protection (QoP'07)*, 2007.

[3] G. Arboit. A method for watermarking Java programs via opaque predicates. In *Proceedings of 5th International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[4] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, 2010.

[5] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'97)*, 1997.

[6] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*, 2011.

[8] D. Brumley and J. Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180R, School of Computer Science, Carnegie Mellon University, 2006.

[9] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, 2006.

[10] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2), 2007.

[11] P. M. S. Bueno and M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, 2000.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[13] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS'06)*, 2006.

[14] J. Cappaert and B. Preneel. A general model for hiding control flow. In *Proceedings of the 10th Annual ACM Workshop on Digital Rights Management (DRM'10)*, 2010.

[15] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*, 2009.

[16] C. Collberg, G. Myles, and A. Huntwork. Sandmark–a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, July 2003.

[17] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[18] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[19] B. Coppens, B. De Sutter, and J. Maebe. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), Jan. 2013.

[20] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[21] DefenseCode. Diving into recent 0day Javascript obfuscations. `http://blog.defensecode.com/2012/10/diving-into-recent-0day-javascript.html`, last reviewed, 04/27/2015.

[22] S. Drape. Intellectual property protection using obfuscation. Technical Report RR-10-02, Oxford University Computing Laboratory, 2010.

[23] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07)*, 2007.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, 2005.

[25] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[26] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM - software protection for the masses. In *Proceedings of the 1st International Workshop on Software PROtection (SPRO'15)*, 2015.

[27] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[28] A. Kovacheva. Efficient code obfuscation for Android. Master's thesis, University of Luxembourg, 2013.

[29] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*, 2014.

[30] 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 (CCS'03)*, 2003.

[31] L. Liu, J. Ming, Z. Wang, D. Gao, and C. Jia. Denial-of-service attacks on host-based generic unpackers. In *Proceedings of the 11th International Conference on Information and Communications Security (ICICS'09)*, 2009.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[33] M. Madou. *Application Security through Program Obfuscation*. PhD thesis, Ghent University, 2007.

[34] 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 (PEPM'06)*, 2006.

[35] A. Majumdar and C. Thomborson. Securing mobile agents control flow using opaque predicates. In *Proceedings of the 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, 2005.

[36] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, 2012.

[37] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23th Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

[38] L. D. Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[39] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155 – 171, April 2006.

[40] M. N. Ngo and H. B. K. Tan. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*, 2007.

[41] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000.

[42] M. D. Preda, M. Madou, K. D. Bosschere, and R. Giacobazzi. Opaque predicate detection by abstract interpretation. In *Proceedings of 11th International Conference on Algebriac Methodology and Software Technology (AMAST'06)*, 2006.

[43] N. A. Quyn. OptiCode: Machine code deobfuscation for malware analysis. In *Proceedings of the 2013 SyScan*, 2013.

[44] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[45] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, 2005.

[46] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[47] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[48] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Proceedings of the 2011 European Symposium on Research in Computer Security (ESORICS'11)*, 2011.

[49] F. Zhang, D. Wu, P. Liu, and S. Zhu. Program logic based software plagiarism detection. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE'14)*, 2014.